PGF/Ti*k*Z(3.1.9.a) 笔记

感谢网友指出笔记中的错误！

我的水平是业余的初学者，笔记中的错误肯定有很多，请参考这个笔记的读者自己鉴别。

对于熟悉 Ti*k*Z 的人，或者有能力熟悉 Ti*k*Z 的人，这个笔记没有多大用处。对于尚不熟悉 Ti*k*Z 或者希望去熟悉 Ti*k*Z 的业余初学者，这个笔记不友好，用处也不大。总之，请参考这个笔记的读者自己斟酌。

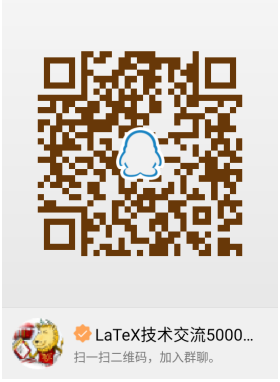
我不懂编程，也不会写作，所以很多内容基本上没用，只是记录我的学习过程而已。

当这个笔记完成后，它就属于“过去”了，建议读者尽早抛弃这个笔记，转向更好的参考资料。

Thank you, all!

https://github.com/pgf-tikz/pgf

http://www.latexstudio.net



目录

第一部分

通用命令

第一章　一些命令

1.1　来自文件《pgfutil-common.tex》

```
\catcode`\@=11\relax
```

\pgfutil@trimspaces{⟨*something*⟩}

这个命令将其参数 ⟨*something*⟩ 的首尾的空格去掉。

本命令的定义是：

```
\catcode`\Q=3
\def\pgfutil@trimspaces#1{%
  \romannumeral-`0\pgfutil@trimspaces@\noexpand#1Q Q}
\def\pgfutil@trimspaces@#1 Q{\pgfutil@trimspaces@@#1Q}
\def\pgfutil@trimspaces@@#1Q#2{#1}
\catcode`\Q=11
```

参数 ⟨*something*⟩ 开头的空格由 \romannumeral 去掉，结尾的空格由 \pgfutil@trimspaces@@ 去掉。

在执行本命令前，字符 Q 的类代码最好不要设置为 3(数学模式)。

\pgfutil@ifx{⟨*tok 1*⟩}{⟨*tok 2*⟩}{⟨*true coed*⟩}{⟨*false code*⟩}

本命令用 \ifx 比较 ⟨*tok 1*⟩ 与 ⟨*tok 2*⟩, 如果二者相同就执行 ⟨*true code*⟩; 否则执行 ⟨*false code*⟩.

本命令的定义是：

```
\long\def\pgfutil@ifx#1#2{%
  \ifx#1#2%
    \expandafter\pgfutil@firstoftwo
  \else
    \expandafter\pgfutil@secondoftwo
  \fi}
```

注意本命令的定义有前缀 \long.

\pgfutil@ifstrequal{⟨*tok 1*⟩}{⟨*tok 2*⟩}{⟨*true coed*⟩}{⟨*false code*⟩}

本命令用 \pgfsys@strcmp 比较字符串 ⟨*tok 1*⟩ 与 ⟨*tok 2*⟩, 如果二者相同就执行 ⟨*true code*⟩; 否则执行 ⟨*false code*⟩.

本命令的定义是：

```
\long\def\pgfutil@ifstrequal#1#2{%
  \ifnum\pgfsys@strcmp{\pgfutil@unexpanded{#1}}{\pgfutil@unexpanded{#2}}=0
    \expandafter\pgfutil@firstoftwo
  \else
    \expandafter\pgfutil@secondoftwo
  \fi
}
```

注意本命令的定义有前缀 \long.

\pgfutil@ifxempty⟨*macro*⟩{⟨*true coed*⟩}{⟨*false code*⟩}

本命令用 \ifx 检查 ⟨*macro*⟩ 是否等于 \pgfutil@empty, 即空的。如果是就执行 ⟨*true code*⟩; 否则执行 ⟨*false code*⟩.

\pgfutil@ifundefined{⟨*command name*⟩}{⟨*code 1*⟩}{⟨*code 2*⟩}

本命令用 \ifx 检查控制序列 \csname ⟨*command name*⟩\endcsname 是否等于 \relax:

- 若是，则执行

  \pgfutil@firstoftwo{⟨*code 1*⟩}{⟨*code 2*⟩}

  导致执行 ⟨*code 1*⟩.
- 若不是，则执行

  \pgfutil@secondoftwo{⟨*code 1*⟩}{⟨*code 2*⟩}

  导致执行 ⟨*code 2*⟩.

\pgfutil@IfUndefined{⟨*command name*⟩}{⟨*code 1*⟩}{⟨*code 2*⟩}

本命令执行 \ifcsname ⟨*command name*⟩\endcsname, 检查控制序列 \csname ⟨*command name*⟩\endcsname 是否等于 \relax.
若不是，则执行 ⟨*code 1*⟩; 若是，则 ⟨*code 2*⟩.

\pgfutil@firstoftwo{⟨*code 1*⟩}{⟨*code 2*⟩}

本命令把 ⟨*code 1*⟩, ⟨*code 2*⟩ 吃掉，但会执行 ⟨*code 1*⟩.

```
\long\def\pgfutil@firstoftwo#1#2{#1}
```

\pgfutil@secondoftwo{⟨*code 1*⟩}{⟨*code 2*⟩}

本命令把 ⟨*code 1*⟩, ⟨*code 2*⟩ 吃掉，但会执行 ⟨*code 2*⟩.

```
\long\def\pgfutil@secondoftwo#1#2{#2}
```

\pgfutil@empty

这个宏保存空内容。

```
\def\pgfutil@empty{}
```

\pgfutil@gobble@until@relax⟨*something*⟩\relax

```
\long\def\pgfutil@gobble@until@relax#1\relax{}
```

本命令将之后、直到 \relax 的记号全部吃掉。

\pgfutil@gobble{⟨*something*⟩}

```
\long\def\pgfutil@gobble#1{}
```

本命令将其参数 ⟨*something*⟩, 或者说将其后面的一个记号吃掉。

\pgfutil@gobbletwo{⟨*arg 1*⟩}{⟨*arg 2*⟩}

```
\long\def\pgfutil@gobbletwo#1#2{}
```

本命令将其 2 个参数，或者说将其后面的 2 个记号吃掉。

\pgfutil@namedef{⟨*name*⟩}⟨*arg list*⟩{⟨*code*⟩}

```
\def\pgfutil@namedef#1{\expandafter\def\csname #1\endcsname}
```

本命令 (非全局地) 定义控制序列 \csname ⟨*name*⟩\endcsname, 例如

10^3
```
\catcode`\@=11\relax
\pgfutil@namedef{a b c}#1#2{$#1^{#2}$}
\csname a b c\endcsname{10}{3}
\catcode`\@=12\relax
```

\pgfutil@namelet{⟨*name 1*⟩}{⟨*name 2*⟩}

本命令使得控制序列 \csname ⟨*name 1*⟩\endcsname 等于控制序列 \csname ⟨*name 2*⟩\endcsname, 但不是全局地等于。

```
\def\pgfutil@namelet#1{\expandafter\pgfutil@@namelet\csname#1\endcsname}
\def\pgfutil@@namelet#1#2{\expandafter\let\expandafter#1\csname#2\endcsname}
```

\pgfutil@g@addto@macro{⟨*macro*⟩}{⟨*code*⟩}

```
\long\def\pgfutil@g@addto@macro#1#2{%
  \begingroup
    \pgfutil@toks@\expandafter{#1#2}%
    \xdef#1{\the\pgfutil@toks@}%
 \endgroup}
```

本命令全局地重定义宏 ⟨*macro*⟩, 将 ⟨*code*⟩ 作为记号添加到宏 ⟨*macro*⟩ 的定义内容中。

AB
```
\def\aaa{A}
\def\bbb{B}
\pgfutil@g@addto@macro\aaa\bbb
\aaa
```

\pgfutil@ifnextchar⟨*char*⟩{⟨*first*⟩}{⟨*second*⟩}⟨t_0⟩⟨t_1⟩$\cdots$

参数 ⟨*char*⟩ 是个记号。⟨t_0⟩⟨t_1⟩$\cdots$ 是记号序列。

本命令先用 \ifx 检查记号 ⟨t_0⟩ 与 \pgfutil@sptoken 是否相同，即检查 ⟨t_0⟩ 保存的是不是空格 (blank space)，

- 如果不是空格，就再用 \ifx 检查记号 ⟨t_0⟩ 与 ⟨*char*⟩ 是不是相同，
  - 如果相同，就执行 ⟨*first*⟩⟨t_0⟩⟨t_1⟩$\cdots$
  - 如果不相同，就执行 ⟨*second*⟩⟨t_0⟩⟨t_1⟩$\cdots$
- 如果 ⟨t_0⟩ 是空格，就吃掉 ⟨t_0⟩, 然后再对 ⟨t_1⟩ 重复本命令的操作。

```
\long\def\pgfutil@ifnextchar#1#2#3{%
  \let\pgfutil@reserved@d=#1%
  \def\pgfutil@reserved@a{#2}%
  \def\pgfutil@reserved@b{#3}%
  \futurelet\pgfutil@let@token\pgfutil@ifnch}
\def\pgfutil@ifnch{%
  \ifx\pgfutil@let@token\pgfutil@sptoken
    \let\pgfutil@reserved@c\pgfutil@xifnch
  \else
    \ifx\pgfutil@let@token\pgfutil@reserved@d
      \let\pgfutil@reserved@c\pgfutil@reserved@a
    \else
      \let\pgfutil@reserved@c\pgfutil@reserved@b
    \fi
  \fi
  \pgfutil@reserved@c}
```

\pgfutil@sptoken

这个宏是全局定义的，它保存一个空格 (blank space)。

```
{%
  \def\:{\global\let\pgfutil@sptoken= } \:
  \def\:{\pgfutil@xifnch} \expandafter\gdef\: {\futurelet\pgfutil@let@token
  ↪  \pgfutil@ifnch}
}
```

\pgfutil@ignorespaces⟨t_0⟩⟨t_1⟩ ...

这个命令的用法一般是：

```
\let\pgfutil@next\⟨macro⟩
\futurelet\pgfutil@let@token\pgfutil@ignorespaces⟨t0⟩⟨t1⟩ ...
```

其中 ⟨t_0⟩⟨t_1⟩ ... 是记号序列，本命令先用 \ifx 检查 ⟨t_0⟩ 是否等于 \pgfutil@sptoken(空格)：

- 如果是，就吃掉 ⟨t_0⟩, 再检查 ⟨t_1⟩, 重复本命令的操作。
- 如果不是，就执行 \pgfutil@next.

本命令的定义是：

```
\def\pgfutil@ignorespaces
  {\ifx\pgfutil@let@token\pgfutil@sptoken
     \expandafter\pgfutil@ignorespaces@helper
   \else
     \expandafter\pgfutil@next
   \fi}

{
  \def\:{\pgfutil@ignorespaces@helper}
  \expandafter\gdef\: {\futurelet\pgfutil@let@token\pgfutil@ignorespaces}
}
```

\pgfutil@in@{⟨*list 1*⟩}{⟨*list 2*⟩}

本命令的定义是：

```
\newif\ifpgfutil@in@
\def\pgfutil@in@#1#2{%
    \def\pgfutil@in@@##1#1##2##3\pgfutil@in@@{%
        \ifx\pgfutil@in@##2\pgfutil@in@false\else\pgfutil@in@true\fi}%
    \pgfutil@in@@#2#1\pgfutil@in@\pgfutil@in@@}
```

本命令判断符号序列 ⟨*list 1*⟩ 是否 ⟨*list 2*⟩ 的一部分，如果是就设置真值 \pgfutil@in@true, 如果不是就设置真值 \pgfutil@in@false.

```
\pgfutil@in@{one}{three two one}
导致
\def\pgfutil@in@@#1one#2#3\pgfutil@in@@{%
  \ifx\pgfutil@in@#2\pgfutil@in@false\else\pgfutil@in@true\fi%
}%
\pgfutil@in@@ three two oneone\pgfutil@in@\pgfutil@in@@
导致
\ifx\pgfutil@in@ o\pgfutil@in@false\else\pgfutil@in@true\fi
导致
\pgfutil@in@true
```

注意本命令不对参数 ⟨*list 1*⟩, ⟨*list 2*⟩ 做展开。

```
\def\pgfutil@nnil{\pgfutil@nil}
\def\pgfutil@fornoop#1\@@#2#3{}
```

\pgfutil@iforloop⟨*arg 1*⟩,⟨*arg 2*⟩\@@\⟨*macro*⟩{⟨*code*⟩}

参数 \⟨*macro*⟩ 是以反斜线开头的宏形式的一串符号。⟨*code*⟩ 外围的花括号是参数定界符号，不是组符号。注意 ⟨*arg 1*⟩ 与 ⟨*arg 2*⟩ 之间用逗号分隔。

本命令的处理是：

1. 定义 \def\⟨*macro*⟩{⟨*arg 1*⟩}
2. 检查 ⟨*arg 1*⟩ 是否 \pgfutil@nil,
   - 如果是，则执行 \pgfutil@fornoop⟨*arg 2*⟩\@@\⟨*macro*⟩{⟨*code*⟩}, 按定义，这什么也不做。
   - 如果不是，则
     (a) 执行 ⟨*code*⟩, 若在 ⟨*code*⟩ 中使用宏 \⟨*macro*⟩, 则它的值是 ⟨*arg 1*⟩
     (b) 执行 \pgfutil@iforloop⟨*arg 2*⟩\@@\⟨*macro*⟩{⟨*code*⟩}

```
\long\def\pgfutil@iforloop#1,#2\@@#3#4{\def#3{#1}\ifx #3\pgfutil@nnil
      \expandafter\pgfutil@fornoop \else
     #4\relax\expandafter\pgfutil@iforloop\fi#2\@@#3{#4}}
```

\pgfutil@forloop⟨*arg 1*⟩,⟨*arg 2*⟩,⟨*arg 3*⟩\@@\⟨*macro*⟩{⟨*code*⟩}

参数 \⟨*macro*⟩ 是以反斜线开头的宏形式的一串符号。本命令的处理是：

1. 定义 \def\⟨*macro*⟩{⟨*arg 1*⟩}
2. 检查 ⟨*arg 1*⟩ 是否 \pgfutil@nil,
   - 如果是，则什么也不做
   - 如果不是，则
     (a) 执行 ⟨*code*⟩, 若在 ⟨*code*⟩ 中使用宏 \⟨*macro*⟩, 它的值是 ⟨*arg 1*⟩
     (b) 定义 \def\⟨*macro*⟩{⟨*arg 2*⟩}
     (c) 检查 ⟨*arg 2*⟩ 是否 \pgfutil@nil,
     – 如果是，则什么也不做
     – 如果不是，则
     ◇ 执行 ⟨*code*⟩, 若在 ⟨*code*⟩ 中使用宏 \⟨*macro*⟩, 它的值是 ⟨*arg 2*⟩
     ◇ 执行 \pgfutil@iforloop⟨*arg 3*⟩\@@\⟨*macro*⟩{⟨*code*⟩}

```
\long\def\pgfutil@forloop#1,#2,#3\@@#4#5{\def#4{#1}\ifx #4\pgfutil@nnil \else
      #5\def#4{#2}\ifx #4\pgfutil@nnil \else#5\pgfutil@iforloop #3\@@#4{#5}\fi\fi
      ↪  }
```

\pgfutil@for\⟨*macro*⟩:=⟨*arg list*⟩\do{⟨*code*⟩}

参数 \⟨*macro*⟩ 是以反斜线开头的宏形式。⟨*arg list*⟩ 是一个 (用逗号分隔的) 列表，也可以是保存这种列表的宏。

本命令先用 \expandafter 将 ⟨*arg list*⟩ 展开一次，这个展开应该得到一个列表。本命令逐个读取列表中的各项，只要读取的列表项不是 \pgfutil@nil, 就执行一次 ⟨*code*⟩ (不是在组中执行的)，从而实现 for 循环的效果。参数 \⟨*macro*⟩ 用作循环变量，临时保存读取的列表项，所以在 ⟨*code*⟩ 中可以使用宏 \⟨*macro*⟩. 如果读取的列表项是 \pgfutil@nil, 就终止循环，未被处理的列表项会被吃掉。

```
\long\def\pgfutil@for#1:=#2\do#3{%
  \expandafter\def\expandafter\pgfutil@fortmp\expandafter{#2}%
  \ifx\pgfutil@fortmp\pgfutil@empty \else
    \expandafter\pgfutil@forloop#2,\pgfutil@nil,\pgfutil@nil\@@#1{#3}\fi}
```

1,2,4,
4

```
\def\aaa{0}
\def\bbb{1,2,3}
\pgfutil@for\tempvar:=\bbb\do{%
    \pgfmathsetmacro{\aaa}{int(2^(\aaa))}\aaa,}\par \aaa
```

1,2,4,
4

```
\def\aaa{0}
\pgfutil@for\tempvar:=a,b,c\do{%
    \pgfmathsetmacro{\aaa}{int(2^(\aaa))}\aaa,}\par \aaa
```

2,4,8,
8

```
\def\aaa{0}
\pgfutil@for\tempvar:={1,2,3}\do{%
    \pgfmathsetmacro{\aaa}{int(2^(\tempvar))}\aaa,}\par \aaa
```

\pgfutil@tforloop⟨*arg 1*⟩⟨*arg 2*⟩\@@\⟨*macro*⟩{⟨*code*⟩}

参数 \⟨*macro*⟩ 是以反斜线开头的宏形式的一串符号。⟨*code*⟩ 外围的花括号是参数定界符号，不是组符号。注意 ⟨*arg 1*⟩ 与 ⟨*arg 2*⟩ 之间没有分隔符号。

本命令的处理是：

1. 定义 \def\⟨*macro*⟩{⟨*arg 1*⟩}
2. 检查 ⟨*arg 1*⟩ 是否 \pgfutil@nil，
   - 如果是，则执行 \pgfutil@fornoop⟨*arg 2*⟩\@@\⟨*macro*⟩{⟨*code*⟩}，按定义，这什么也不做。
   - 如果不是，则
     (a) 执行 ⟨*code*⟩，若在 ⟨*code*⟩ 中使用宏 \⟨*macro*⟩，它的值是 ⟨*arg 1*⟩
     (b) 执行 \pgfutil@tforloop⟨*arg 2*⟩\@@\⟨*macro*⟩{⟨*code*⟩}，这是循环处理步骤

```
\long\def\pgfutil@tforloop#1#2\@@#3#4{\def#3{#1}\ifx #3\pgfutil@nnil
      \expandafter\pgfutil@fornoop \else
     #4\relax\expandafter\pgfutil@tforloop\fi#2\@@#3{#4}}
```

\pgfutil@tf@r\⟨*macro*⟩⟨*token list*⟩\do{⟨*code*⟩}

参数 ⟨*token list*⟩ 是一个记号序列 (一个无需分隔符号的记号列表)。每当读取列表中的一个记号后，只要这个记号不是 \pgfutil@nil，就执行一次 ⟨*code*⟩ (不是在组中执行的)，从而实现 for 循环的效果。参数 \⟨*macro*⟩ 用作循环变量，临时保存读取的记号，所以在 ⟨*code*⟩ 中可以使用宏 \⟨*macro*⟩. 如果读取的记号是 \pgfutil@nil，就终止循环，未被处理的记号会被吃掉。

本命令的处理是：

- 如果 ⟨*token list*⟩ 是空格 (即 \pgfutil@space 保存的内容)，则什么也不做
- 如果 ⟨*token list*⟩ 不是空格，则执行

  \pgfutil@tforloop⟨*token list*⟩\pgfutil@nil\pgfutil@nil\@@\⟨*macro*⟩{⟨*code*⟩}

  这是循环处理

注意，本命令不会对参数 ⟨*token list*⟩ 做展开。

```
\long\def\pgfutil@tf@r#1#2\do#3{\def\pgfutil@fortmp{#2}\ifx\pgfutil@fortmp
↪   \pgfutil@space\else
    \pgfutil@tforloop#2\pgfutil@nil\pgfutil@nil\@@#1{#3}\fi}
```

1,3,27,
27

```
\def\aaa{0}
\pgfutil@tf@r\tempvar{a}{b}{c}\do{%
    \pgfmathsetmacro{\aaa}{int(3^(\aaa))}\aaa,}\par \aaa
```

1,3,27,
27

```
\def\aaa{0}
\pgfutil@tf@r\tempvar abc\do{%
    \pgfmathsetmacro{\aaa}{int(3^(\aaa))}\aaa,}\par \aaa
```

```
3,9,27,
   27 \def\aaa{0}
      \pgfutil@tf@r\tempvar{123}\do{%
          \pgfmathsetmacro{\aaa}{int(3^(\tempvar))}\aaa,}\par \aaa
```

\pgfutil@tfor\⟨*macro*⟩:=⟨*token list*⟩\do{⟨*code*⟩}

本命令实际上执行 \pgfutil@tf@r\⟨*macro*⟩⟨*token list*⟩\do{⟨*code*⟩}

```
\def\pgfutil@tfor#1:={\pgfutil@tf@r#1 }
```

\pgfutil@IfFileExists{⟨*file name*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

本命令检查文件 ⟨*file name*⟩ 是否能 (以读取的方式，用命令 \openin) 打开 (能打开就存在)。若能打开，则执行 ⟨*true code*⟩; 否则执行 ⟨*false code*⟩. 然后 (用 \closein) 关闭文件。

```
\chardef\pgfutil@inputcheck0
\def\pgfutil@IfFileExists#1#2#3{%
  \openin\pgfutil@inputcheck=#1 %
  \ifeof\pgfutil@inputcheck
     #3\relax
  \else
    #2\relax
  \fi
  \closein\pgfutil@inputcheck}
```

\pgfutil@InputIfFileExists{⟨*file name*⟩}{⟨*extr true code*⟩}{⟨*false code*⟩}

本命令检查文件 ⟨*file name*⟩ 是否能 (以读取的方式，用命令 \openin) 打开 (能打开就存在)。若能打开，则载入文件，并执行 ⟨*extr true code*⟩; 否则执行 \false code. 然后关闭文件。

```
\def\pgfutil@InputIfFileExists#1#2#3{\pgfutil@IfFileExists{#1}{\input #1\relax#2
↪  }{#3}}%
```

\pgfutil@loop⟨*body*⟩\pgfutil@repeat

这是一个循环，⟨*body*⟩ 是循环体。在 ⟨*body*⟩ 中写出 \if ..., 但不写出与之匹配的 \fi, 那么这个 \if ... 就决定了循环条件。例如，

```
2„ 4„ 6„ \def\aaa{0}
         \pgfutil@loop%
             \ifnum 5>\aaa%
                \pgfmathsetmacro{\aaa}{int(\aaa+2)}
                \aaa,,
         \pgfutil@repeat%
```

```
\def\pgfutil@loop#1\pgfutil@repeat{\def\pgfutil@body{#1}\pgfutil@iterate}
\def\pgfutil@iterate{\pgfutil@body \let\pgfutil@next\pgfutil@iterate
↪  \else\let\pgfutil@next\relax\fi \pgfutil@next}
\let\pgfutil@repeat=\fi % this makes \loop...\if...\repeat skippable
```

\pgfutil@switch\⟨*comparison function*⟩{⟨*tok*⟩}{⟨*tokens pair list*⟩}{⟨*match code*⟩}{⟨*no match code*⟩}

参数 \⟨*comparison function*⟩ 是一个比较命令，其参数格式为：

\⟨*comparison function*⟩{⟨*tok A*⟩}{⟨*tok B*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

在这个格式中，\⟨*comparison function*⟩ 对 ⟨*tok A*⟩ 与 ⟨*tok B*⟩ 做某种比较，若比较结果为 true, 则执行 ⟨*true code*⟩; 否则执行 ⟨*false code*⟩.

参数 ⟨*tok*⟩ 是将被 \⟨*comparison function*⟩ 比较的记号.

⟨*tokens pair list*⟩ 是形式为

```
{⟨tokens⟩0}{⟨something⟩0}
{⟨tokens⟩1}{⟨something⟩1}
......
```

这样的“用花括号构造的对”。

本命令会按次序读取一个 $\langle tokens\rangle_i$, $i = 0, 1, \cdots$, 并用 \⟨*comparison function*⟩ 比较 ⟨*tok*⟩ 与 $\langle tokens\rangle_i$, 如果比较的结果为真，则执行 $\langle something\rangle_i$ 以及 ⟨*match code*⟩, 并终止本命令。如果所有的比较结果为假，则执行 ⟨*no match code*⟩.

本命令在做比较前，不会先对参数 ⟨*tok*⟩ 与 ⟨*tokens pair list*⟩ 做展开。

1match
```
\pgfutil@switch\pgfutil@ifx{x}{%
  {x}{1}
  {y}{2}
  {x}{3}
}{match}{nomatch}
```

```
\let\pgfutil@exp\romannumeral
\chardef\pgfutil@exp@end=0

\let\pgfutil@scan@mark\relax
\let\pgfutil@scan@stop\relax

\long\def\pgfutil@switch#1#2#3#4#5{%
  \pgfutil@exp
  \pgfutil@switch@@{#1}%
  {#2}#3% user-defined cases
  {#2}{}% default case
  \pgfutil@scan@mark{#4}% true code
  \pgfutil@scan@mark{#5}% false code
  \pgfutil@scan@stop
}
\long\def\pgfutil@switch@@#1#2#3#4{%
  #1{#2}{#3}%
    {\pgfutil@switch@end{#4}}%
    {\pgfutil@switch@@{#1}{#2}}%
}
\long\def\pgfutil@switch@end#1#2#3\pgfutil@scan@mark#4#5\pgfutil@scan@stop{
↪   \pgfutil@exp@end#1#4}
```

可见本命令会使用 \romannumeral 吃掉不必要的空格。

\pgfutil@packageerror{⟨*package name*⟩}{⟨*error message*⟩}{⟨*error help*⟩}

```
\def\pgfutil@packageerror#1#2#3{\errhelp{#3}\errmessage{Package #1 Error: #2}}
```

\pgfutil@packagewarning{⟨*package name*⟩}{⟨*waring message*⟩}

```
\def\pgfutil@packagewarning#1#2{\immediate\write17{Package #1: Warning! #2.}}
```

\pgferror{⟨*error message*⟩}

```
\def\pgferror#1{\pgfutil@packageerror{pgf}{#1}{}}
```

\pgfwarning{⟨*waring message*⟩}

```
\def\pgfwarning#1{\pgfutil@packagewarning{pgf}{#1}}
```

\usepgflibrary[⟨*library list*⟩] 或 {⟨*library list*⟩}

⟨*library list*⟩ 是 PGF 的库列表。本命令载入列出的库。

\usepgfmodule[⟨*module list*⟩] 或 {⟨*module list*⟩}

⟨*module list*⟩ 是 PGF 的模块列表。本命令载入列出的库。

\pgfutilensuremath{⟨*math code*⟩}

本命令确保 ⟨*math code*⟩ 处于数学模式下。

```
\def\pgfutilensuremath#1{%
  \ifmmode#1\else$#1$\fi
}
```

\pgfutilpreparefilename{⟨*string*⟩}

本命令是被全局定义的。

本命令的作用是处理文件名称。把参数 ⟨*string*⟩ 看作是文件名：

- 如果 ⟨*string*⟩ 中不含双引号，则定义 \pgfretval 保存 ⟨*string*⟩; 还定义 \pgfretvalquoted 保存 "⟨*string*⟩"
- 如果 ⟨*string*⟩ 中包含双引号，则定义 \pgfretval 和 \pgfretvalquoted 都保存 "⟨*string*⟩".

\pgfutil@command@to@string⟨*macro 1*⟩⟨*macro 2*⟩

由 \meaning 得到的结果是类代码为 12 的一些符号。

macro:->
macro:->\pgfpoint {1cm}{2cm}

```
\meaning\pgfutil@empty\par
\def\aaa{\pgfpoint{1cm}{2cm}}
\meaning\aaa
```

本命令把 \meaning⟨*macro 1*⟩ 所得结果中“macro:->”之后的那些符号保存到 ⟨*macro 2*⟩ 中 (注意这些符号的类代码为 12)。

\pgfpoint {1cm}{2cm}

```
\def\aaa{\pgfpoint{1cm}{2cm}}
\pgfutil@command@to@string\aaa\bbb
\bbb
```

如例子所示，本命令将 \aaa 的定义内容 (类代码为 12 的符号) 保存到 \bbb 中。

```
\def\pgfutil@command@to@string#1#2{%
    \expandafter\pgfutil@command@to@string@@\meaning#1\pgfutil@EOI{#2}%
}%
\xdef\pgfutil@glob@TMPa{\meaning\pgfutil@empty}%
\expandafter\def\expandafter\pgfutil@command@to@string@@\pgfutil@glob@TMPa#1
↪   \pgfutil@EOI#2{%
    \def#2{#1}%
}%
```

\pgfutil@backslash@as@other

这个宏全局地保存类代码为 12 (其他符号) 的反斜线“\”。

```
\begingroup
\catcode`\|=0
\catcode`\\=12
|gdef|pgfutil@backslash@as@other{\}%
|endgroup
```

\pgfutilifcontainsmacro{⟨*tokens*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

本命令先把 ⟨*tokens*⟩ 做成类代码为 12 的一些符号，然后检查其中是否含有类代码为 12 的反斜线“\”。如果含有 (说明原来的 ⟨*tokens*⟩ 中含有宏)，则执行 ⟨*true code*⟩, 否则执行 ⟨*false code*⟩.

yes `\pgfutilifcontainsmacro{abc\mmm}{yes}{no}`

```
\def\pgfutilifcontainsmacro#1#2#3{%
    \def\pgf@marshal{#1}%
    \pgfutil@command@to@string\pgf@marshal\pgf@marshal
    \edef\pgf@marshal{\noexpand\pgfutil@in@ →P.30{\pgfutil@backslash@as@other}{
    ↪  \pgf@marshal}}%
    \pgf@marshal
    \ifpgfutil@in@
        \def\pgf@marshal{#2}%
    \else
        \def\pgf@marshal{#3}%
    \fi
    \pgf@marshal
}%
```

\pgfutilifstartswith⟨*tokens 1*⟩⟨*tokens 2*⟩{⟨*true code*⟩}{⟨*false code*⟩}

本命令检查 ⟨*tokens 1*⟩ 是不是 ⟨*tokens 2*⟩ 开头的一部分，如果是，那么 ⟨*tokens 2*⟩ 就是由 ⟨*tokens 1*⟩ 和另一部分——记为 ⟨*suf-tokens*⟩ 组成的。如果是，本命令将 ⟨*suf-tokens*⟩ 保存到 \pgfretval 中，再执行 ⟨*true code*⟩; 如果不是，则执行 ⟨*false code*⟩.

yes„ macro:->c \ddd
```
\pgfutilifstartswith{\aaa b}{\aaa bc \ddd}{yes}{no},,
\meaning\pgfretval
```

```
\def\pgfutilifstartswith#1#2#3#4{%
    \def\pgfutilifstartswith@ ##1#1##2\pgf@EOI{%
        \def\pgfutil@tmp{##1}%
        \ifx\pgfutil@tmp\pgfutil@empty
            % Ah - a hit!
            %
            % define \pgfretval to be the suffix...
            \def\pgfutil@tmp#1####1\pgf@EOI{%
                \def\pgfretval{####1}%
            }%
            \pgfutil@tmp#2\pgf@EOI%
            %
            % ... and execute the <true> code:
            #3\relax%
        \else
            % hm. No such prefix.
            #4\relax%
        \fi
    }%
    \pgfutilifstartswith@#2--#1\pgf@EOI%
}%
```

\pgfutilifstartswith{\aaa b}{\aaa bc \ddd}{⟨*true code*⟩}{⟨*false code*⟩} 的处理过程是：

1. 定义 \def\pgfutilifstartswith@ #1\aaa b#2\pgf@EOI{...}
2. 执行 \pgfutilifstartswith@ \aaa bc \ddd--\aaa b\pgf@EOI, 导致
3. 定义 \def\pgfutil@tmp{}
4. 用 \ifx 检查，导致
5. 定义 \def\pgfutil@tmp \aaa b#1\pgf@EOI{...}
6. 执行 \pgfutil@tmp\aaa bc \ddd\pgf@EOI, 导致

7. 定义 \def\pgfretval{c \ddd}
8. 执行 ⟨*true code*⟩.

\pgfutilstrreplace{⟨*tokens 1*⟩}{⟨*tokens 2*⟩}{⟨*tokens 3*⟩}

把参数 ⟨*tokens 1*⟩, ⟨*tokens 2*⟩, ⟨*tokens 3*⟩ 都看作是符号串，本命令的作用是：

- 如果 ⟨*tokens 1*⟩ 不是 ⟨*tokens 3*⟩ 的“子串”，则把 ⟨*tokens 2*⟩ 保存到 \pgfretval 中。
- 如果 ⟨*tokens 1*⟩ 是 ⟨*tokens 3*⟩ 的“子串”，则把 ⟨*tokens 3*⟩ 中的子串 ⟨*tokens 1*⟩ 都替换为 ⟨*tokens 2*⟩, 将替换后的结果保存到 \pgfretval 中。

```
\long\def\pgfutilstrreplace#1#2#3{%
    \def\pgfretval{}%
    \long\def\pgfutil@search@and@replace@@##1#1##2\pgf@EOI{%
        \expandafter\def\expandafter\pgfretval\expandafter{\pgfretval ##1#2}%
        \pgfutil@search@and@replace@loop{#1}{##2}%
    }%
    \pgfutil@search@and@replace@loop{#1}{#3}%
}
\long\def\pgfutil@search@and@replace@loop#1#2{%
    \pgfutil@in@{#1}{#2}%
    \ifpgfutil@in@
        \def\pgf@loc@TMPa{\pgfutil@search@and@replace@@ #2\pgf@EOI}%
    \else
        \expandafter\def\expandafter\pgfretval\expandafter{\pgfretval #2}%
        \let\pgf@loc@TMPa=\relax
    \fi
    \pgf@loc@TMPa
}%
```

\pgfutilsolvetwotwoleq{{⟨a_a⟩}{⟨a_b⟩}{⟨b_a⟩}{⟨b_b⟩}}{{⟨r_a⟩}{⟨r_b⟩}}

本命令用于解方程组

$$\begin{cases} a_a x + a_b y = r_a \\ b_a x + b_b y = r_b \end{cases} \quad \text{其解是} \quad \begin{cases} x = \dfrac{r_b \cdot a_b - r_a \cdot b_b}{a_b \cdot b_a - a_a \cdot b_b} \\ y = \dfrac{r_a \cdot b_a - r_b \cdot a_a}{a_b \cdot b_a - a_a \cdot b_b} \end{cases}$$

给矩阵 $\begin{pmatrix} a_a & a_b & r_a \\ b_a & b_b & r_b \end{pmatrix}$ 的诸元素加上单位 pt, 就等价于方程组

$$\begin{cases} \texttt{\textbackslash aa}\ x + \texttt{\textbackslash ab}\ y = \texttt{\textbackslash ra} \\ \texttt{\textbackslash ba}\ x + \texttt{\textbackslash bb}\ y = \texttt{\textbackslash rb} \end{cases} \quad \text{其解是} \quad \begin{cases} x = \dfrac{\texttt{\textbackslash rb} \cdot \texttt{\textbackslash ab} - \texttt{\textbackslash ra} \cdot \texttt{\textbackslash bb}}{\texttt{\textbackslash ab} \cdot \texttt{\textbackslash ba} - \texttt{\textbackslash aa} \cdot \texttt{\textbackslash bb}} \\ y = \dfrac{\texttt{\textbackslash ra} \cdot \texttt{\textbackslash ba} - \texttt{\textbackslash rb} \cdot \texttt{\textbackslash aa}}{\texttt{\textbackslash ab} \cdot \texttt{\textbackslash ba} - \texttt{\textbackslash aa} \cdot \texttt{\textbackslash bb}} \end{cases}$$

矩阵 $\begin{pmatrix} \texttt{\textbackslash aa} & \texttt{\textbackslash ab} & \texttt{\textbackslash ra} \\ \texttt{\textbackslash ba} & \texttt{\textbackslash bb} & \texttt{\textbackslash rb} \end{pmatrix}$ 由 TEX 的尺寸寄存器构成。矩阵 $\begin{pmatrix} a_a & a_b \\ b_a & b_b \end{pmatrix}$ 的内部表示是 $\begin{pmatrix} \texttt{\textbackslash maa} & \texttt{\textbackslash mab} \\ \texttt{\textbackslash mba} & \texttt{\textbackslash mbb} \end{pmatrix}$.
如果 |\aa| ⩽ |\ba|, 就交换方程组中两个方程的次序，但方程组的符号表示仍然不变，所以不妨就认为 |\aa| > |\ba|. 方程组的解（不带长度单位的数值）保存在 \pgfmathresult 中，即“{⟨x 的解⟩}{⟨y 的解⟩}”，注意数值解是分别处于花括号内的，也就是说，\pgfmathresult 中保存的是两个“被花括号包裹的数值”。宏 \pivot 的值是 $\frac{1}{\texttt{\textbackslash maa}}$, 宏 \factor 的值是 $\frac{\texttt{\textbackslash mba}}{\texttt{\textbackslash maa}}$.
当 $|\texttt{\textbackslash aa}| < 0.0001\text{pt}$ 时，或者当 $|\texttt{\textbackslash bb} - \frac{\texttt{\textbackslash ba}\cdot\texttt{\textbackslash ab}}{\texttt{\textbackslash aa}}| < 0.0001\text{pt}$ 时，认为系数矩阵是奇异矩阵，将方程组的解定义为 \pgfutil@empty, 即空的。
本命令利用 TEX 的寄存器做计算。

\pgfutilsolvetwotwoleqfloat{{⟨a_a⟩}{⟨a_b⟩}{⟨b_a⟩}{⟨b_b⟩}}{{⟨r_a⟩}{⟨r_b⟩}}

本命令用于解方程组

$$\begin{cases} a_a x + a_b y = r_a \\ b_a x + b_b y = r_b \end{cases} \quad \text{其解是} \quad \begin{cases} x = \dfrac{r_b \cdot a_b - r_a \cdot b_b}{a_b \cdot b_a - a_a \cdot b_b} \\ y = \dfrac{r_a \cdot b_a - r_b \cdot a_a}{a_b \cdot b_a - a_a \cdot b_b} \end{cases}$$

本命令的计算过程利用 fpu 库的函数。在本命令的计算过程中限制了阈值 1.0×10^{-4}, 这个阈值保存在 `\thresh` 中，当求倒数或做除法时，若分母的绝对值小于这个阈值，就认为方程组无解，即 `\pgfmathresult` 等于 `\pgfutil@empty`。也设置了界限 1.6×10^{4}, 当所得数值的绝对值大于这个界限时，也认为方程组无解。

`\pgfutil@shellescape{`⟨*system command*⟩`}`

本命令的定义是：

```
\def\pgfutil@shellescape#1{%
    \immediate\write18{#1}%
}%
```

参数 ⟨*system command*⟩ 是能够调用其他系统命令 (非 TEX 命令) 的代码。本命令需要获得调用权限才能执行 ⟨*system command*⟩, 通常需要给编译器使用 `--shell-escape` 选项。

1.2 来自文件《pgfmathutil.code.tex》

`\pgfmath@ifregister@unguarded{`⟨*token type*⟩`}{`⟨*a token*⟩`}{`⟨*true code*⟩`}{`⟨*false code*⟩`}`

参数 ⟨*token type*⟩ 可以是 `count`, `dimen`, `skip`, `muskip`, `toks` 之一，代表的是记号的类型。

参数 ⟨*a token*⟩ 是一个记号。

本命令检查记号 ⟨*a token*⟩ 是否属于 ⟨*token type*⟩ 类型；如果属于，就执行 ⟨*true code*⟩; 如果不属于，就执行 ⟨*false code*⟩.

本命令的定义利用了 `\meaning` (它返回类代码为 12 的记号)。

`\pgfmath@ifregister{`⟨*token type*⟩`}{`⟨*a token*⟩`}{`⟨*true code*⟩`}{`⟨*false code*⟩`}`

参数 ⟨*token type*⟩ 可以是 `count`, `dimen`, `skip`, `muskip`, `toks` 之一，代表的是记号的类型。

参数 ⟨*a token*⟩ 是一个记号。

本命令：

- 如果 ⟨*token type*⟩ 不是 `count`, `dimen`, `skip`, `muskip`, `toks` 之一，否则报错；
- 检查记号 ⟨*a token*⟩ 是否属于 ⟨*token type*⟩ 类型；如果属于，就执行 ⟨*true code*⟩; 如果不属于，就执行 ⟨*false code*⟩.

`\pgfmath@ensureregister{`⟨*token type*⟩`}{`⟨*a token*⟩`}`

参数 ⟨*token type*⟩ 可以是 `count`, `dimen`, `skip`, `muskip`, `toks` 之一，代表的是记号的类型。

参数 ⟨*a token*⟩ 是一个记号。

本命令检查记号 ⟨*a token*⟩ 是否属于 ⟨*token type*⟩ 类型；如果属于，就什么也不做；如果不属于，就把 ⟨*a token*⟩ 声明为一个 ⟨*token type*⟩ 类型的记号。

`\pgfmathloop`⟨*code*⟩`\repeatpgfmathloop`

这个循环命令的定义是：

```
\newif\ifpgfmathcontinueloop
\def\pgfmathloop#1\repeatpgfmathloop{%
    \def\pgfmathcounter{1}%
    \def\pgfmath@iterate{%
```

```
        #1\relax%
        {% Do this inside a group, just in case...
            \c@pgfmath@counta\pgfmathcounter%
            \advance\c@pgfmath@counta1\relax%
            \xdef\pgfmathloop@temp{\the\c@pgfmath@counta}%
        }%
        \edef\pgfmathcounter{\pgfmathloop@temp}%
        \expandafter\pgfmath@iterate\fi}%
    \pgfmath@iterate\let\pgfmath@iterate\relax}
\let\repeatpgfmathloop\fi
\def\pgfmathbreakloop{\let\pgfmath@iterate\relax}%
```

这个循环的开头定义计数宏 \pgfmathcounter(初始值是 1), 用它规定循环条件，每循环一次，它的值就自动加 1. 循环内容被保存在 \pgfmath@iterate 中，它至少被执行一次，执行它导致：

```
⟨code⟩\relax%
{%
  \c@pgfmath@counta\pgfmathcounter%
  \advance\c@pgfmath@counta1\relax%
  \xdef\pgfmathloop@temp{\the\c@pgfmath@counta}%
}%
\edef\pgfmathcounter{\pgfmathloop@temp}%
\expandafter\pgfmath@iterate\fi
```

所以参数 ⟨*code*⟩ 中应当包含以 \if 开头的条件判断代码，与定义中的 \fi 配合。判断条件应该用 \pgfmathcounter 来构造，并确保能在有限步骤内结束循环。

如果在 ⟨*code*⟩ 中使用宏 \pgfmathbreakloop, 而这个宏在某个循环步骤中被执行，那么在结束这个循环步骤后就不再继续循环。所有循环步骤都不被放在组中。

\aaa is 10
\aaa is 5
\aaa is 1

```
\def\aaa{10}
\pgfmathloop%
  \ifnum\pgfmathcounter<\aaa\relax%
    \pgfmathsetmacro{\aaa}{int(\aaa/\pgfmathcounter)}%
    \string\aaa{} is \aaa\\
\repeatpgfmathloop%
```

下面代码利用“条件判断为假”来执行循环：

\aaa is 10
\aaa is 5
\aaa is 1

```
\def\aaa{10}
\pgfmathloop%
  \ifnum\pgfmathcounter>\aaa\relax%
  \else
    \pgfmathsetmacro{\aaa}{int(\aaa/\pgfmathcounter)}%
    \string\aaa{} is \aaa\\
\repeatpgfmathloop%
```

\pgfmath@returnone{⟨*dimension*⟩}\endgroup

参数 ⟨*dimension*⟩ 可以是尺寸表达式、保存尺寸的宏、尺寸寄存器名称。本命令将尺寸 ⟨*dimension*⟩ 的数值部分保存到 \pgfmathresult 中。

```
\def\pgfmath@returnone#1\endgroup{%
  \pgfmath@x#1%
  \edef\pgfmath@temp{\pgfmath@tonumber{\pgfmath@x}}%
  \expandafter\endgroup\expandafter\def\expandafter\pgfmathresult\expandafter{
  ↪  \pgfmath@temp}%
}

\let\pgfmathreturn=\pgfmath@returnone
```

使用本命令时注意，在本命令之前的某个地方应该有 \begingroup, 本命令会吃掉与 \begingroup 对应的 \endgroup, 还会再给补上一个 \endgroup.

\pgfmath@smuggleone⟨*macro*⟩\endgroup

参数 ⟨*macro*⟩ 是个宏。

```
\def\pgfmath@smuggleone#1\endgroup{%
  \expandafter\endgroup\expandafter\def\expandafter#1\expandafter{#1}}

\let\pgfmathsmuggle=\pgfmath@smuggleone
```

在本命令之前的某个地方应该有 \begingroup, 本命令会吃掉与这个 \begingroup 对应的 \endgroup, 再补上一个 \endgroup.

本命令将宏 ⟨*macro*⟩ 的定义推到 \endgroup 之后，不受这个组的限制。

\pgfmathsmuggle

等于 \pgfmath@smuggleone.

```
\let\pgfmathsmuggle=\pgfmath@smuggleone
```

1.3 来自文件《pgfutil-latex.def》

\ifpgfutil@format@is@latex

这个 TEX-if 标志者当前的格式是否 LATEX.

文件《pgfutil-common.tex》声明：\newif\ifpgfutil@format@is@latex

\pgfutil@auxout

这个宏代表编译过程中的 .aux 文件。

\let\pgfutil@auxout=\@auxout

\pgfutil@writetoaux{⟨*something*⟩}

当有真值 \@fileswtrue 时，这个命令可以向 .aux 文件中写入 ⟨*something*⟩.

```
\def\pgfutil@writetoaux#1{\if@filesw\write\pgfutil@auxout{#1}\fi}
```

\if@filesw 的默认值是 true.

\pgfutil@definecolor[⟨*type*⟩]{⟨*name*⟩}{⟨*model-list*⟩}{⟨*spec-list*⟩}

这个命令等于 xcolor 宏包的命令 \definecolor.

```
\def\pgfutil@definecolor{\definecolor}
```

\pgfutil@color[⟨*model*⟩]{⟨*color*⟩}

这个命令等于 xcolor 宏包的命令 \color.

```
\def\pgfutil@color{\color}
```

\pgfutil@colorlet{⟨*name*⟩}{⟨*color*⟩}

本命令等效于 xcolor 宏包的命令 \colorlet{⟨*name*⟩}{⟨*color*⟩}, 令 ⟨*name*⟩ 等于 ⟨*color*⟩.

\pgfutil@extractcolorspec{⟨*color*⟩}\⟨*macro*⟩

本命令等效于 xcolor 宏包的命令 \extractcolorspec{⟨*color*⟩}\⟨*macro*⟩, 将颜色 ⟨*color*⟩ 所属的模式以及颜色数据保存到宏 \⟨*macro*⟩ 中。相当于

\def\⟨*macro*⟩{{⟨*model*⟩}{⟨*spec*⟩}}.

\pgfutil@convertcolorspec{⟨model⟩}{⟨spec⟩}{⟨target model⟩}\⟨macro⟩

本命令等效于 xcolor 宏包的命令 \convertcolorspec, 将属于模式 ⟨model⟩ 的颜色数据 ⟨spec⟩ 转换为目标模式 ⟨target model⟩ 的颜色数据并保存到宏 \⟨macro⟩ 中。相当于 \def\⟨macro⟩{⟨target model spec⟩}.

\pgfutil@minipage

```
\let\pgfutil@minipage=\minipage
```

\pgfutil@endminipage

```
\let\pgfutil@endminipage=\endminipage
```

\pgfutil@doifcolorelse{⟨color⟩}{⟨true code⟩}{⟨false code⟩}

如果颜色 ⟨color⟩ 是有效的颜色，则执行 ⟨true code⟩, 否则执行 ⟨false code⟩.

\pgfutil@font@normalsize

```
\def\pgfutil@font@normalsize{\normalsize}
```

\pgfutil@font@itshape

```
\def\pgfutil@font@itshape{\itshape}
```

\pgfutil@selectfont

```
\let\pgfutil@selectfont=\selectfont
```

第二章　Key 机制

宏包 pgfkeys 不依赖其他宏包，可以单独使用。本宏包的实现是文件《pgfkeys.code.tex》。

一个完整的键 (full key) 的形式类似 `/a/b/c` 这样，由斜线和字符组成，并且以斜线开头，其中的 `c` 叫做“键名称”(key name)，`/a/b` 叫做“键的路径”(key path)。例如，键 `/tikz/intersection/sort by` 是一个完整的键，`sort by` 是键名称。注意 `tikz/intersection/sort by` 不是一个完整的键，因为它不以斜线开头。

Key 机制的基本思路是：用键来控制一些代码。定义一个键的意思是：使得这个键对应某些代码。执行一个键的意思是：将这个键对应的代码拿出来加以执行或处理。

例如，在定义键 `/a/b/c` 时，可以让键 `/a/b/c` 对应控制序列

```
\csname pgfk@/a/b/c\endcsname
```

并且令这个控制序列保存一些代码：

```
\pgfkeyssetvalue{/a/b/c}{$x^{y^z}$}
```

上一个命令把 `$x^{y^z}$` 保存到这个控制序列中。在执行键 `/a/b/c` 时，调用这个控制序列，从而执行其中的代码：

x^{y^z} `\pgfkeysvalueof{/a/b/c}`

所谓“键值对”就是键及其值的组合，例如 `/tikz/draw=red` 是一个键值对，其中用 `=red` 为键 `/tikz/draw` 赋值，`red` 是键 `draw` 的值。而 `/tikz/draw` 也可以看作是一个键值对，因为它有默认值 `black`。“键值对列表”就是由键值对和逗号组成的列表，例如 `/tikz/draw=red, /tikz/fill=green`。

当说“某个键的初始值”时，指的是这样的情况：用户没有使用这个键，也没有明确程序禁止使用这个键，而程序总会自动使用这个键，并在使用时为这个键取一个值，这个值就是初始值。

当说“某个键的默认值”时，指的是这样的情况：程序并不自动使用这个键，当用户使用这个键，但没有明确这个键的值时，程序就为这个键取一个值，这个值就是默认值。

键分为几种类型：

1. 有的键对应一个控制序列，而这个控制序列只是保存某些记号，并不处理参数，这样的键是“变量”，它只保存某些记号。

2. 有的键对应一个控制序列，这个控制序列能够处理参数，这样的键是“函数”。

3. “手柄”(key handlers)，这种键是一种函数，但比较常用，所以单列出来。

4. “手柄键”。

5. 首字符句法，这种键是一种函数。

变量与函数的区分是从“能否处理参数”的角度看的，但有时这种区分依赖展开的层次：

```
\def\aaa{#1} % 变量
\def\bbb#1{x} % 函数
\def\ccc{\bbb} % 通常是函数，起到 \bbb 的作用；但如果只允许一次展开，则 \ccc 是变量
```

2.1 作为变量的键

```
\newtoks\pgfkeys@pathtoks
\def\pgfkeyscurrentpath{\the\pgfkeys@pathtoks}
\newtoks\pgfkeys@temptoks
```

\pgfkeyssetvalue{⟨*full key*⟩}{⟨*tokens*⟩}

此命令的定义是：

```
\long\def\pgfkeyssetvalue#1#2{%
  \pgfkeys@temptoks{#2}\expandafter\edef\csname pgfk@#1\endcsname{
  ↪  \the\pgfkeys@temptoks}%
}
```

- 参数 #1 代表完整的键 (full key)，可以是保存完整键的宏。
- 参数 #2 代表一些记号，或者说代码。本命令有前缀 \long，所以参数 ⟨*tokens*⟩ 中可以含有 \par。

本命令工作时，先将 ⟨*tokens*⟩ 保存到记号寄存器 \pgfkeys@temptoks 中，然后再转存到控制序列

\csname pgfk@⟨*full key*⟩\endcsname

中。注意 \the\pgfkeys@temptoks 得到的字符的类代码通常是 12(其他字符) 或者 10(空格)。

\pgfkeyssetevalue{⟨*full key*⟩}{⟨*tokens*⟩}

此命令的定义是：

```
\long\def\pgfkeyssetevalue#1#2{%
  \expandafter\edef\csname pgfk@#1\endcsname
  {\pgfkeys@unexpanded\expandafter{\pgfkeys@expanded{#2}}}%
}
```

- 参数 #1 代表完整的键 (full key)，可以是保存完整键的宏。
- 参数 #2 代表一些记号，或者说代码。本命令有前缀 \long，所以参数 ⟨*tokens*⟩ 中可以含有 \par。

\pgfkeysifdefined{⟨*full key*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

此命令的定义是：

```
\long\def\pgfkeysifdefined#1{%
  \ifcsname pgfk@#1\endcsname
    \expandafter\pgfkeys@firstoftwo
  \else
    \expandafter\pgfkeys@secondoftwo
  \fi
}
```

参数 #1 代表完整的键 (full key)，可以是保存完整键的宏。

本命令检查 \csname pgfk@⟨*full key*⟩\endcsname 是否不等于 \relax，即是否已经被定义。如果是，执行 ⟨*true code*⟩；如果不是，执行⟨*false code*⟩。

\pgfkeysifassignable{⟨*full key*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

此命令的定义是：

```
\long\def\pgfkeysifassignable#1#2#3{%
  \pgfkeysifdefined{#1}%
    {#2}
```

```
  {\pgfkeysifdefined{#1/.@cmd}%
    {#2}%
    {#3}}%
}%
```

参数 #1 代表完整的键 (full key)，可以是保存完整键的宏。

- 本命令先检查 \csname pgfk@⟨*full key*⟩\endcsname 是否不等于 \relax，即是否已经被定义。
  - 如果是，执行 ⟨*true code*⟩，结束本命令；如果不是，则再：
  - 检查 \csname pgfk@⟨*full key*⟩/.@cmd\endcsname 是否不等于 \relax，即是否已经被定义。如果是，执行 ⟨*true code*⟩；如果不是，执行⟨*false code*⟩。

\pgfkeyslet{⟨*full key*⟩}\⟨*macro*⟩

本命令的定义是：

```
\def\pgfkeyslet#1#2{%
  \expandafter\let\csname pgfk@#1\endcsname#2%
}
```

参数 #1 代表完整的键 (full key)，可以是保存完整键的宏。

本命令使得控制序列 \csname pgfk@⟨*full key*⟩\endcsname 等于 \⟨*macro*⟩，即定义或重定义键 ⟨*full key*⟩ 所保存的代码。

\pgfkeysgetvalue{⟨*full key*⟩}\⟨*macro*⟩

本命令的定义是：

```
\def\pgfkeysgetvalue#1#2{\expandafter\let\expandafter#2\csname pgfk@#1\endcsname}
```

参数 ⟨*full key*⟩ 代表完整的键 (full key)，可以是保存完整键的宏。

本命令使得 \⟨*macro*⟩ 等于控制序列 \csname pgfk@⟨*full key*⟩\endcsname，也就是把键 ⟨*full key*⟩ 所保存的代码复制到 \⟨*macro*⟩。

\pgfkeysvalueof{⟨*full key*⟩}

本命令的定义是：

```
\def\pgfkeysvalueof#1{\csname pgfk@#1\endcsname}
```

参数 #1 代表完整的键 (full key)，可以是保存完整键的宏。

本命令将键 ⟨*full key*⟩ 对应的控制序列 \csname pgfk@⟨*full key*⟩\endcsname 插入到当前位置。

\pgfkeysaddvalue{⟨*full key*⟩}{⟨*prefix code*⟩}{⟨*post code*⟩}

参数 ⟨*full key*⟩ 代表完整的键 (full key)，可以是保存完整键的宏。

本命令重定义键 ⟨*full key*⟩ 对应的控制序列 \csname pgfk@⟨*full key*⟩\endcsname。假设这个控制序列原来保存的是 ⟨原来的记号⟩，本命令重定义这个控制序列，使之保存 ⟨*prefix code*⟩⟨原来的记号⟩⟨*post code*⟩。

2.2 作为函数的键

\pgfkeysdef{⟨*full key*⟩}{⟨*code with #1*⟩}

本命令的定义是：

```
\long\def\pgfkeysdef#1#2{%
  \long\def\pgfkeys@temp##1\pgfeov{#2}%
  \pgfkeyslet{#1/.@cmd}{\pgfkeys@temp}%
  \pgfkeyssetvalue{#1/.@body}{#2}%
}
```

本命令有前缀 \long，所以它的参数中可以含有 \par。

- 参数 #1 代表完整的键 (full key)，可以是保存完整键的宏。
- 参数 ⟨*code with #1*⟩ 是一些代码，其中可以含有参数符号 #1。如果 ⟨*code with #1*⟩ 中含有定义命令，则定义命令中的参数符号 # 要双写。

本命令定义 2 个控制序列

- \csname pgfk@⟨*full key*⟩/.@cmd\endcsname，它是个函数，它的参数格式是：

  \csname pgfk@⟨*full key*⟩/.@cmd\endcsname#1\pgfeov

  它的替换文本是 ⟨*code with #1*⟩。
- \csname pgfk@⟨*full key*⟩/.@body\endcsname，它是个变量，它保存 ⟨*code with #1*⟩。

\pgfkeysedef{⟨*full key*⟩}{⟨*code with #1*⟩}

本命令的定义是：

```
\long\def\pgfkeysedef#1#2{%
  \long\edef\pgfkeys@temp##1\pgfeov{#2}%
  \pgfkeyslet{#1/.@cmd}{\pgfkeys@temp}%
  \pgfkeyssetevalue{#1/.@body}{#2}%
}
```

本命令类似 \pgfkeysdef，只是在定义控制序列 \csname pgfk@⟨*full key*⟩/.@cmd\endcsname 时，它的替换文本 ⟨*code with #1*⟩ 是被 \edef 展开的。但在定义控制序列 \csname pgfk@⟨*full key*⟩/.@body\endcsname 时，替换文本 ⟨*code with #1*⟩ 没有被展开。

\pgfkeysdefargs{⟨*full key*⟩}{⟨*argument pattern*⟩}{⟨*code with arguments*⟩}

本命令的定义是：

```
\long\def\pgfkeysdefargs#1#2#3{%
  \long\def\pgfkeys@temp#2\pgfeov{#3}%
  \pgfkeyslet{#1/.@cmd}{\pgfkeys@temp}%
  \pgfkeyssetvalue{#1/.@args}{#2\pgfeov}%
  \pgfkeyssetvalue{#1/.@body}{#3}%
}
```

本命令有前缀 \long，所以它的参数中可以含有 \par。

- 参数 #1 代表完整的键 (full key)，可以是保存完整键的宏。
- ⟨*argument pattern*⟩ 是定义命令时的参数格式。
- 参数 ⟨*code with arguments*⟩ 是一些代码，其中可以含有 ⟨*argument pattern*⟩ 中列出的参数符号。如果 ⟨*code with arguments*⟩ 中含有定义命令，则定义命令中的参数符号 # 要双写。

本命令定义 3 个控制序列

- \csname pgfk@⟨*full key*⟩/.@cmd\endcsname，它是个函数，它的参数格式是：

  \csname pgfk@⟨*full key*⟩/.@cmd\endcsname⟨*argument pattern*⟩\pgfeov

  它的替换文本是 ⟨*code with arguments*⟩。
- \csname pgfk@⟨*full key*⟩/.@body\endcsname，它是个变量，它保存 ⟨*code with arguments*⟩
- \csname pgfk@⟨*full key*⟩/.@args\endcsname，它是个变量，它保存 ⟨*argument pattern*⟩\pgfeov

\pgfkeysedef{⟨*full key*⟩}{⟨*argument pattern*⟩}{⟨*code with arguments*⟩}

本命令的定义是：

```
\long\def\pgfkeysedefargs#1#2#3{%
  \long\edef\pgfkeys@temp#2\pgfeov{#3}%
  \pgfkeyslet{#1/.@cmd}{\pgfkeys@temp}%
  \pgfkeyssetvalue{#1/.@args}{#2\pgfeov}%
  \pgfkeyssetevalue{#1/.@body}{#3}%
```

```
}
```

本命令类似 \pgfkeysdefargs，只是在定义控制序列 \csname pgfk@⟨full key⟩/.@cmd\endcsname 时，它的替换文本 ⟨code with arguments⟩ 是被 \edef 展开的。

\pgfkeysdefnargs{⟨full key⟩}{⟨arguments number⟩}{⟨code with arguments⟩}

本命令有前缀 \long，所以它的参数中可以含有 \par。

- 参数 ⟨full key⟩ 代表完整的键 (full key)，可以是保存完整键的宏。
- ⟨argument number⟩ 是参数个数，最多 9 个参数。
- 参数 ⟨code with arguments⟩ 是一些代码，其中可以含有最多 ⟨argument number⟩ 个参数符号。如果 ⟨code with arguments⟩ 中含有定义命令，则定义命令中的参数符号 # 要双写。

命令 \pgfkeysdefnargs{⟨full key⟩}3{⟨code⟩} 的处理是：

1. 先定义 2 个控制序列：
   - \csname pgfk@⟨full key⟩/.@args\endcsname，这是个变量，它保存 #1#2#3
   - \csname pgfk@⟨full key⟩/.@@body\endcsname，这是个函数，它的参数格式是 #1#2#3，它的替换文本是 ⟨code with arguments⟩。使用命令 \long\def 定义这个控制序列，所以这个函数的参数中可以含有 \par。
2. 调用 \pgfkeysdef →P. 44 定义 2 个控制序列
   - \csname pgfk@⟨full key⟩/.@cmd\endcsname，它是个函数，它的参数格式是：

     \csname pgfk@⟨full key⟩/.@cmd\endcsname#1\pgfeov

     其中的 #1 对应“一次展开的 \pgfkeyscurrentvalue”，它的替换文本是：
     - \csname pgfk@⟨full key⟩/.@@body\endcsname{#1} (对于只有 1 个参数的情况)
     - \csname pgfk@⟨full key⟩/.@@body\endcsname#1 (对于多个参数的情况)

     其中的 #1 对应“一次展开的 \pgfkeyscurrentvalue”。
   - \csname pgfk@⟨full key⟩/.@body\endcsname，它是个变量，它保存上述替换文本。
3. 调用 \pgfkeyssetevalue →P. 43 重定义变量 \csname pgfk@⟨full key⟩/.@body\endcsname，使之保存 ⟨code with arguments⟩

当执行 \pgfkeys{⟨full key⟩={⟨arg1⟩}{⟨arg2⟩}{⟨arg3⟩}} 时，导致

```
\csname pgfk@⟨full key⟩/.@cmd\endcsname{⟨arg1⟩}{⟨arg2⟩}{⟨arg3⟩}\pgfeov
导致
\csname pgfk@⟨full key⟩/.@@body\endcsname{⟨arg1⟩}{⟨arg2⟩}{⟨arg3⟩}
导致执行 ⟨code with arguments⟩
```

\pgfkeysedefnargs{⟨full key⟩}{⟨arguments number⟩}{⟨code with arguments⟩}

本命令类似 \pgfkeysdefnargs,只是使用命令 \long\edef 来定义 \csname pgfk@⟨full key⟩/.@@body\endcsname

文件中给出的例子：

1=‘1’, 2=‘ 2’

```
\pgfkeysdefargs{/b}{#1#2}{1=`#1', 2=`#2'}
\pgfkeys{
  /b=
  {1}
  {2}
}
```

注意上面例子中“‘ 2’”里的空格，对比下面的

1=‘1’, 2=‘2’

```
\pgfkeysdefargs{/b}{#1#2}{1=`#1', 2=`#2'}
\pgfkeys{
  /b=
  {1}%
  {2}
}
```

```
\def\aaaa#1#2{#1}
\aaaa{a} {b}\par% 空格被吃掉
\def\aaaa#1#2\bbbb{#1#2}
\aaaa{a} {b}\bbbb% 空格被保留
```

a
a b

2.3 处理键的命令

2.3.1 \pgfkeys

命令 \pgfkeys 处理键值对列表

\pgfkeys{⟨*key1*⟩=⟨*value1*⟩, ⟨*key2*⟩=⟨*value2*⟩, ...}

注意：

- 如果写出的键 ⟨*key*⟩ 不是完整的键，那么本命令会给它的开头加上默认路径 (通常是一个斜线)，使之成为完整的键。
- 如果写出的 ⟨*value*⟩ 中含有逗号，则必须用花括号将 ⟨*value*⟩ 括起来，因为本命令用逗号来分隔键值对。

```
\def\pgfkeys@root{/}
\let\pgfkeysdefaultpath\pgfkeys@root
```

\pgfkeys@sptoken

这个宏保存一个空格。

```
{\def\:{\global\let\pgfkeys@sptoken= } \: }
```

\pgfkeys@spdef\⟨*macro*⟩{⟨*tokens*⟩}

文件中有下面的代码：

```
\long\def\pgfkey@argumentisspace#1{%
  \long\def\pgfkeys@spdef##1##2{%
    \futurelet\pgfkeys@possiblespace\pgfkeys@sp@a##2\pgfkeys@stop\pgfkeys@stop#1
    ↪  \pgfkeys@stop\relax##1}%
  \def\pgfkeys@sp@a{%
    \ifx\pgfkeys@possiblespace\pgfkeys@sptoken%
      \expandafter\pgfkeys@sp@b%
    \else%
      \expandafter\pgfkeys@sp@b\expandafter#1%
    \fi}%
  \long\def\pgfkeys@sp@b#1##1 \pgfkeys@stop{\pgfkeys@sp@c##1}%
}
\pgfkey@argumentisspace{ }
\long\def\pgfkeys@sp@c#1\pgfkeys@stop#2\relax#3{\pgfkeys@temptoks{#1}\edef#3{
↪  \the\pgfkeys@temptoks}}
```

本命令可以将 ⟨*tokens*⟩ 开头与结尾的空格去掉，把剩下的字符 (包括空格) 保存到 \⟨*macro*⟩ 中。

\pgfkeys{⟨*key-value list*⟩}

这个命令可以定义键，也可以执行键对应的代码。

参数 ⟨*key-value list*⟩ 是键值对列表。

本命令的定义是：

```
\def\pgfkeys{\expandafter\pgfkeys@@set\expandafter{\pgfkeysdefaultpath}}%
\long\def\pgfkeys@@set#1#2{%
  \let\pgfkeysdefaultpath\pgfkeys@root%
```

```
  \pgfkeys@parse#2,\pgfkeys@mainstop%
  \def\pgfkeysdefaultpath{#1}}
```

可见，\pgfkeys@@set 有前缀 \long，所以参数 ⟨*key-value list*⟩ 中可以含有 \par。

本命令的处理是：

1. 获取 \pgfkeysdefaultpath 的当前值，记为 ⟨*origin default key path*⟩。
2. \let\pgfkeysdefaultpath\pgfkeys@root

   在后面的处理中，宏 \pgfkeysdefaultpath 用作默认路径，也就是说，如果 ⟨*key-value list*⟩ 中的某个键值对不是以斜线 / 开头，就认为这个键不是完整的键，此时把宏 \pgfkeysdefaultpath 保存的字符添加到这个键的开头，使之成为完整的键。

   由于 \pgfkeys@root 的初始值被定义为 /，所以 \pgfkeysdefaultpath 的初始值也是 /。如果事先改变了 \pgfkeys@root 的值，在此就可以改变 \pgfkeysdefaultpath 的值。

   ```
   在默认下，
   \pgfkeys{a/b/c}
   等价于
   \pgfkeys{/a/b/c}
   ```

3. \pgfkeys@parse⟨*key-value list*⟩,\pgfkeys@mainstop

   这是主要处理步骤。注意在 ⟨*key-value list*⟩ 后面添加了一个逗号，这个逗号导致命令 \pgfkeys@normal 至少被执行一次。

4. \def\pgfkeysdefaultpath{⟨*origin default key path*⟩}

   恢复 \pgfkeysdefaultpath 当初的值，至此结束 \pgfkeys。

\pgfkeys@parse⟨*a token*⟩

如果 ⟨*a token*⟩ 是 \pgfkeys@mainstop，则执行 \pgfkeys@cleanup，停止处理；否则执行 \pgfkeys@normal。

\pgfkeys@cleanup\pgfkeys@mainstop

此命令什么也不做。

```
\def\pgfkeys@cleanup\pgfkeys@mainstop{}
```

```
\newif\ifpgfkeys@syntax@handlers
%...
\def\pgfkeys@mainstop{\pgfkeys@mainstop} % equals only itself
\def\pgfkeys@novalue{}
\def\pgfkeysnovalue{\pgfkeys@novalue} % equals only itself
\def\pgfkeysnovalue@text{\pgfkeysnovalue}
\def\pgfkeysvaluerequired{\pgfkeysvaluerequired} % equals only itself
```

\pgfkeys@normal

如果有真值 \pgfkeys@syntax@handlerstrue，则执行 \pgfkeys@syntax@handlers → P. 58。

如果有真值 \pgfkeys@syntax@handlersfalse，则执行 \pgfkeys@@normal。

\pgfkeys@@normal⟨*key-value*⟩,

参数 ⟨*key-value*⟩ 是一个键值对，不是多个键值对的列表。由于这个命令以逗号为参数定界标志，所以，如果在 ⟨*key-value*⟩ 的 ⟨*value*⟩ 中含有逗号，则必须用花括号将 ⟨*value*⟩ 括起来。

本命令的定义是：

```
\long\def\pgfkeys@@normal#1,{%
  \pgfkeys@unpack#1=\pgfkeysnovalue=\pgfkeys@stop%
  \pgfkeys@parse%
```

```
}
```

可见，\pgfkeys@@normal 有前缀 \long，所以参数 ⟨*key-value*⟩ 中可以含有 \par。

本命令调用 \pgfkeys@unpack 处理 ⟨*key-value*⟩，并吃掉一个逗号，然后再调用 \pgfkeys@parse 处理之后的键值对。

\pgfkeys@add@path@as@needed

本命令检查 ⟨*key*⟩=⟨*value*⟩ 中的 ⟨*key*⟩，即 \pgfkeyscurrentkey 是否以斜线开头：

```
\ifx\pgfkeys@possibleslash/%
%...
```

- 如果是，则

  ```
  \pgfkeysaddeddefaultpathfalse
  \let\pgfkeyscurrentkeyRAW\pgfkeyscurrentkey
  ```

  即设置真值，定义 \pgfkeyscurrentkeyRAW 为 ⟨*key*⟩ 的展开值。

- 如果不是，则

  ```
  \pgfkeysaddeddefaultpathtrue
  \def\pgfkeyscurrentkeyRAW{\pgfkeyscurrentkey}%
  \edef\pgfkeyscurrentkey{\pgfkeysdefaultpath\pgfkeyscurrentkey}%
  ```

  即设置真值，定义 \pgfkeyscurrentkeyRAW 为 \pgfkeyscurrentkey，再为 \pgfkeyscurrentkey 添加默认路径，将 \pgfkeyscurrentkey 做成完整的键。

\pgfkeyscurrentkey

在 \pgfkeys@add@path@as@needed 之后，这个宏保存完整的键。在后续的处理中，它的值可能会被改变。

\pgfkeyscurrentkeyRAW

如上。

\ifpgfkeysaddeddefaultpath

这个 TEX-if 的真值被 \pgfkeys@add@path@as@needed 设置。如果给出的 ⟨*key*⟩ 不是完整键，则需要添加默认路径，此时设置真值 \pgfkeysaddeddefaultpathtrue；否则设置真值 \pgfkeysaddeddefaultpathfalse。

\pgfkeys@unpack#1=#2=#3\pgfkeys@stop

按 \pgfkeys@@normal →P. 48 的处理，本命令的参数实际可能是：

(i) \pgfkeys@unpack⟨*key*⟩=⟨*value*⟩=\pgfkeysnovalue=\pgfkeys@stop
例如 \pgfkeys{⟨*key*⟩=⟨*value*⟩} 是这种情况。

(ii) \pgfkeys@unpack⟨*key*⟩==\pgfkeysnovalue=\pgfkeys@stop
例如 \pgfkeys{⟨*key*⟩=} 是这种情况。

(iii) \pgfkeys@unpack⟨*key*⟩=\pgfkeysnovalue=\pgfkeys@stop
例如 \pgfkeys{⟨*key*⟩} 是这种情况。

(iv) \pgfkeys@unpack⟨*key1*⟩={⟨⟨*key2*⟩=⟨*value*⟩⟩}=\pgfkeysnovalue=\pgfkeys@stop
例如 \pgfkeys{⟨*key1*⟩={⟨*key2*⟩=⟨*value*⟩}} 是这种情况。

本命令有前缀 \long，所以本命令的参数中可以含有 \par。

本命令的第一个参数 #1 应该对应一个键 ⟨*key*⟩。键 ⟨*key*⟩ 可以是完整的键 (以斜线开头)，可以是不完整的键 (不以斜线开头)，可以是保存 (完整或不完整) 键的宏。

本命令的处理是：首先，键 ⟨*key*⟩ 被 \edef 展开后，保存到 \pgfkeyscurrentkey。然后检查：

- 如果 \pgfkeyscurrentkey 保存的内容是空的，则什么也不做；

- 如果 \pgfkeyscurrentkey 保存的内容非空，则
  1. 执行 \pgfkeys@add@path@as@needed，将 \pgfkeyscurrentkey 做成完整的键。
  2. 定义 \pgfkeyscurrentvalue，这个宏保存的可能是：
     (i) ⟨value⟩，例如 \pgfkeys{⟨key⟩=⟨value⟩} 是这种情况。
     (ii) 空的内容，例如 \pgfkeys{⟨key⟩=} 是这种情况。
     (iii) \pgfkeysnovalue，例如 \pgfkeys{⟨key⟩} 是这种情况。
     (iv) ⟨key2⟩=⟨value⟩，例如 \pgfkeys{⟨key1⟩={⟨key2⟩=⟨value⟩}} 是这种情况。
     (v) \pgfkeysvaluerequired，如果之前有 \pgfkeys{⟨key⟩/.value required}，那么之后执行 \pgfkeys{⟨key⟩} 时会是这个情况 (其中缺少 ⟨value⟩)。
  3. 检查 \pgfkeyscurrentvalue 的值，如果它的值是 \pgfkeysnovalue，例如 \pgfkeys{⟨key⟩} 这种没有提供键值的情况，则再检查控制序列 pgfk@\pgfkeyscurrentkey/.@def 是否已定义，即键 \pgfkeyscurrentkey 是否有默认值，如果有就把默认值保存到 \pgfkeyscurrentvalue；如果没有默认值，则什么也不做。
  4. 再次检查 \pgfkeyscurrentvalue 的值，如果它的值是 \pgfkeysvaluerequired，则执行键 /errors/value required/.@cmd；如果不是，则执行 \pgfkeys@case@one。

  \pgfkeyscurrentvalue

  注意此时 \pgfkeyscurrentvalue 的值可能是键 \pgfkeyscurrentkey 的默认值，或者提供的键值 ⟨value⟩，或者是空的，或者是 \pgfkeysnovalue(等于 \pgfkeysnovalue@text)。

\pgfkeys@case@one

这个命令检查键 \pgfkeyscurrentkey 是否是一个函数，也就是检查控制序列 (函数) pgfk@\pgfkeyscurrentkey/ 是否已定义。如果已定义，就用这个函数处理参数 ⟨一次展开的 *\pgfkeyscurrentvalue*⟩\pgfeov；如果未定义，就执行 \pgfkeys@case@two。
此命令的定义是：

```
\def\pgfkeys@case@one{%
  \pgfkeysifdefined{\pgfkeyscurrentkey/.@cmd}%
  {\pgfkeysgetvalue{\pgfkeyscurrentkey/.@cmd}{\pgfkeys@code}%
   \expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov}
  {\pgfkeys@case@two}%
}
```

\pgfkeys@case@two

这个命令检查键 \pgfkeyscurrentkey 是否是一个变量，也就是检查控制序列 (函数) pgfk@\pgfkeyscurrentkey 是否已定义。

- 如果已定义，再检查 \pgfkeyscurrentvalue 的当前值，
  – 如果是 \pgfkeysnovalue，即 \pgfkeys{⟨key⟩} 这种情况，就把键 \pgfkeyscurrentkey 保存的代码插入到当前位置。
  – 如果不是 \pgfkeysnovalue，即 \pgfkeys{⟨key⟩=⟨value⟩}，或者 \pgfkeys{⟨key⟩=} 的情况，或者 \pgfkeyscurrentvalue 的值是键 \pgfkeyscurrentkey 的默认值，就重定义这个键：

    ```
    \pgfkeyslet{\pgfkeyscurrentkey}\pgfkeyscurrentvalue%
    ```

- 如果未定义，则执行 \pgfkeys@case@three →P. 51

```
\def\pgfkeys@case@two{%
  \pgfkeysifdefined{\pgfkeyscurrentkey}%
  {\pgfkeys@case@two@extern}%
  {\pgfkeys@case@three}%
}
```

\pgfkeys@case@two@extern

```
\def\pgfkeys@case@two@extern{%
  \ifx\pgfkeyscurrentvalue\pgfkeysnovalue@text%
    \pgfkeysvalueof{\pgfkeyscurrentkey}%
  \else%
    \pgfkeyslet{\pgfkeyscurrentkey}\pgfkeyscurrentvalue%
  \fi%
}
```

\pgfkeys@split@path

这个命令将当前的键 \pgfkeyscurrentkey 分解，得到它的键名称 \pgfkeyscurrentname 和键路径 \pgfkeyscurrentpath。

\pgfkeyscurrentname

如上。

\pgfkeyscurrentpath

如上。

\pgfkeys@case@three

这个键检查当前键 \pgfkeyscurrentkey 是否是一个“手柄键”。本命令：

1. 执行 \pgfkeys@split@path 分解 \pgfkeyscurrentkey，
2. 检查 \pgfkeyscurrentname 是否一个手柄，即检查控制序列 (函数)

   pgfk@/handlers/\pgfkeyscurrentname/.@cmd

   是否有定义，
   - 如果有定义,则用这个控制序列 (函数) 处理参数 ⟨展开一次的 *\pgfkeyscurrentvalue*⟩\pgfeov
   - 如果未定义，则执行 \pgfkeys@unknown。

本命令的定义是：

```
\def\pgfkeys@case@three{%
  \pgfkeys@split@path%
  \pgfkeysifdefined{/handlers/\pgfkeyscurrentname/.@cmd}%
  {\pgfkeysgetvalue{/handlers/\pgfkeyscurrentname/.@cmd}{\pgfkeys@code}%
    \expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov}
  {\pgfkeys@unknown}%
}
```

\pgfkeys@unknown

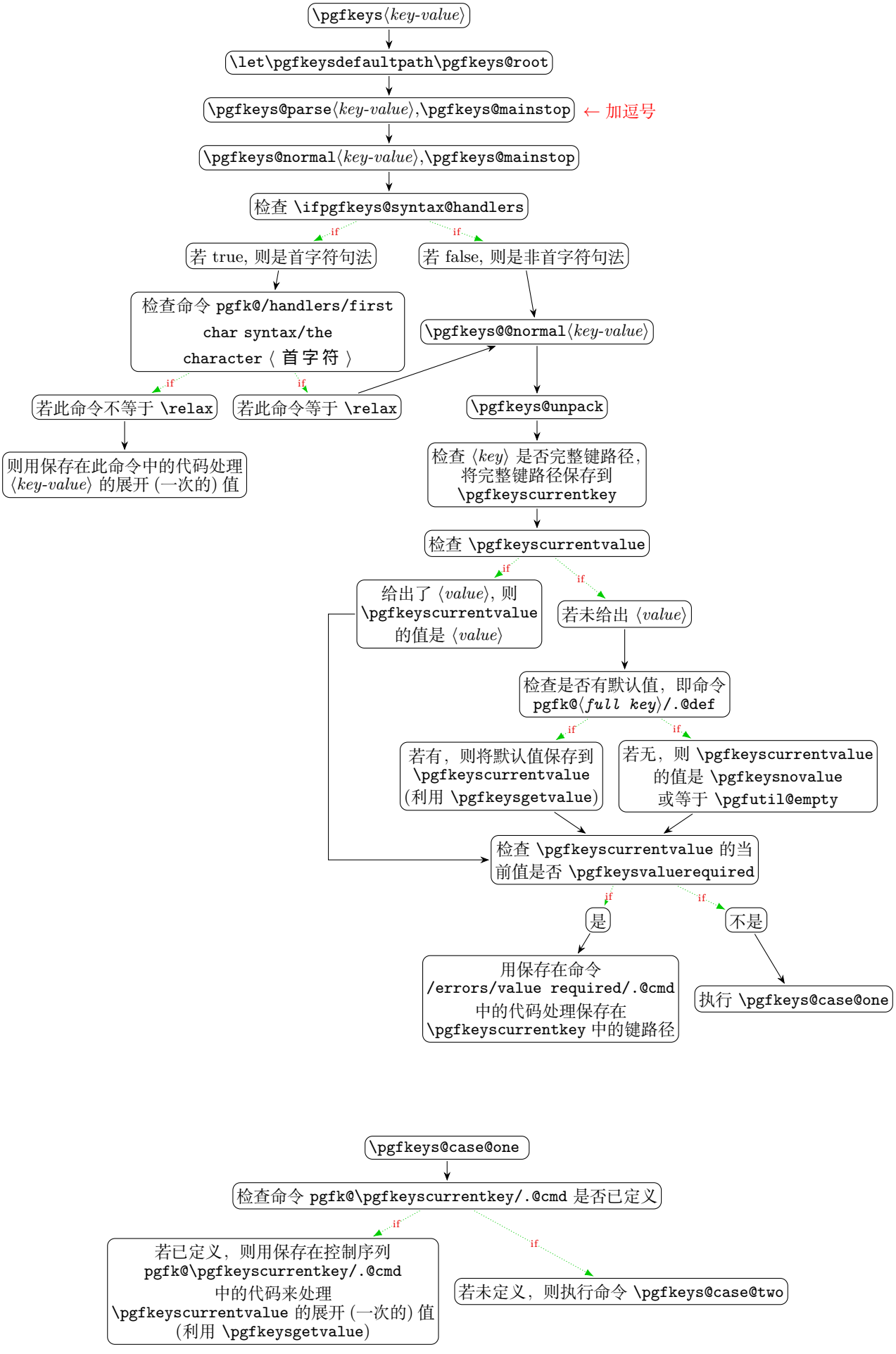
本命令检查控制序列 (函数) pgfk@\pgfkeyscurrentpath/.unknown/.@cmd 是否已定义，

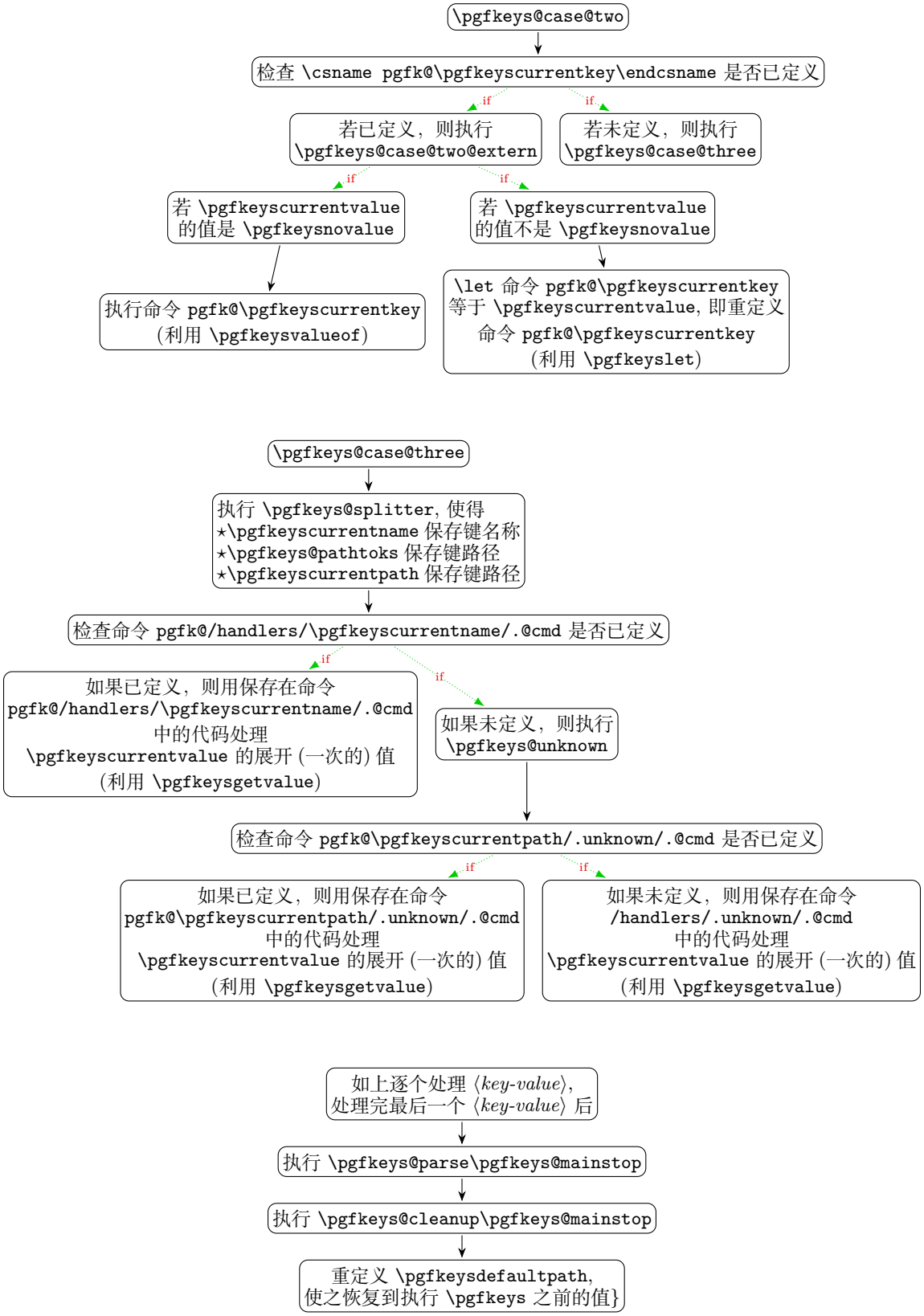
- 如果有定义，则用这个控制序列 (函数) 处理参数 ⟨展开一次的 *\pgfkeyscurrentvalue*⟩\pgfeov
- 如果未定义，则用键 /handlers/.unknown/.@cmd，也就是用控制序列 (函数)

  pgfk@/handlers/.unknown/.@cmd

  处理参数 ⟨展开一次的 *\pgfkeyscurrentvalue*⟩\pgfeov

下面的图形大致展示了命令 \pgfkeys 的处理过程：


\pgfkeys⟨key-value⟩
\let\pgfkeysdefaultpath\pgfkeys@root
\pgfkeys@parse⟨key-value⟩,\pgfkeys@mainstop
← 加逗号
\pgfkeys@normal⟨key-value⟩,\pgfkeys@mainstop
检查 \ifpgfkeys@syntax@handlers
if
if
若 true, 则是首字符句法
若 false, 则是非首字符句法
检查命令 pgfk@/handlers/first char syntax/the character ⟨ 首字符 ⟩
\pgfkeys@@normal⟨key-value⟩
if
if
若此命令不等于 \relax
若此命令等于 \relax
\pgfkeys@unpack
则用保存在此命令中的代码处理 ⟨key-value⟩ 的展开 (一次的) 值
检查 ⟨key⟩ 是否完整键路径，将完整键路径保存到 \pgfkeyscurrentkey
检查 \pgfkeyscurrentvalue
if
if
给出了 ⟨value⟩, 则 \pgfkeyscurrentvalue 的值是 ⟨value⟩
若未给出 ⟨value⟩
检查是否有默认值，即命令 pgfk@⟨full key⟩/.@def
if
if
若有，则将默认值保存到 \pgfkeyscurrentvalue (利用 \pgfkeysgetvalue)
若无，则 \pgfkeyscurrentvalue 的值是 \pgfkeysnovalue 或等于 \pgfutil@empty
检查 \pgfkeyscurrentvalue 的当前值是否 \pgfkeysvaluerequired
if
if
是
不是
用保存在命令 /errors/value required/.@cmd 中的代码处理保存在 \pgfkeyscurrentkey 中的键路径
执行 \pgfkeys@case@one
\pgfkeys@case@one
检查命令 pgfk@\pgfkeyscurrentkey/.@cmd 是否已定义
if
if
若已定义，则用保存在控制序列 pgfk@\pgfkeyscurrentkey/.@cmd 中的代码来处理 \pgfkeyscurrentvalue 的展开 (一次的) 值 (利用 \pgfkeysgetvalue)
若未定义，则执行命令 \pgfkeys@case@two



2.3.2 \pgfkeys 的变化形式

\pgfqkeys{⟨*default key path*⟩}{⟨*key-value list*⟩}

这是 \pgfkeys 变化形式。参数 ⟨*default key path*⟩ 是 /a/b/c 这样的以斜线开头 (不以斜线结尾) 的一串符号，作为 ⟨*key-value list*⟩ 中的键的默认路径。

本命令首先将 \pgfkeysdefaultpath 定义为“⟨*default key path*⟩/”，然后处理 ⟨*key-value list*⟩，最后再把 \pgfkeysdefaultpath 恢复为之前的值。

```
\def\pgfqkeys{\expandafter\pgfkeys@@qset\expandafter{\pgfkeysdefaultpath}}%
\long\def\pgfkeys@@qset#1#2#3{\def\pgfkeysdefaultpath{#2/}\pgfkeys@parse#3,
↪  \pgfkeys@mainstop\def\pgfkeysdefaultpath{#1}}
```

\pgfkeysalso{⟨*key-value list*⟩}

本命令的定义是：

```
\long\def\pgfkeysalso#1{\pgfkeys@parse#1,\pgfkeys@mainstop}
```

本命令只是处理 ⟨*key-value list*⟩。

\pgfqkeysalso{⟨*default key path*⟩}{⟨*key-value list*⟩}

这是 \pgfkeys 变化形式。参数 ⟨*default key path*⟩ 是 /a/b/c 这样的以斜线开头 (不以斜线结尾) 的一串符号，作为 ⟨*key-value list*⟩ 中的键的默认路径。

本命令首先将 \pgfkeysdefaultpath 定义为“⟨*default key path*⟩/”，然后处理 ⟨*key-value list*⟩，但不会再把 \pgfkeysdefaultpath 恢复为之前的值。

```
\long\def\pgfqkeysalso#1#2{\def\pgfkeysdefaultpath{#1/}\pgfkeys@parse#2,
↪  \pgfkeys@mainstop}
```

2.4 手柄

2.4.1 手柄的基本思路

当把控制序列 \csname pgfk@/handlers/⟨*handler name*⟩/.@cmd\endcsname 定义为一个函数后，键 ⟨*handler name*⟩ 就可以看作是一个“手柄”。在 \pgfkeys 的处理过程中，\pgfkeys@case@three→P. 51 处理这种手柄。例如：

1. 先定义函数 pgfk@/handlers/.my handler name/.@cmd，

```
\pgfkeysdef{/handlers/.my handler name}{\tikz\path[#1]circle[radius=2mm];}
```

2. 然后用函数 pgfk@/handlers/.my handler name/.@cmd 处理参数 fill=red\pgfeov

```
\pgfkeys{/my test/.my handler name={fill=red}}
```

不过上面的图形实际上可以这样得到：

```
\pgfkeys{/handlers/.my handler name={fill=red}}
```

对比下面的例子：

1. 先定义函数 pgfk@/handlers/.my handler name/.@cmd，

```
\pgfkeysdef{/handlers/.my handler name}{
  \pgfkeysdef{\pgfkeyscurrentpath}{\tikz{#1}}
}
```

2. 然后利用手柄 .my handler name

```
\pgfkeys{/my test/.my handler name={\path[#1]circle[radius=2mm];}}
```

3. 然后利用键 /my test 处理参数 fill=red\pgfeov

如上面的例子，键 /handlers/.my handler name 或者 .my handler name 或者 /.my handler name 叫做手柄，手柄一般带有点号。键 /my test 或者 /my test/.my handler name 叫做手柄键。手柄键借助手柄起作用。通常，定义、使用“手柄”的步骤如上面例子所示：

(i) 执行

```
\pgfkeysdef{/handlers/.⟨handler name⟩}{\pgfkeysdef{\pgfkeyscurrentpath}{f(#1)}}
或者执行 (参考 /.code)
\pgfkeys{/handlers/.⟨handler name⟩/.code={f(#1)}}
```

这样就把 ⟨*handler name*⟩ 做成一个手柄，也就是定义了函数

\csname pgfk@/handlers/⟨*handler name*⟩/.@cmd\endcsname

这个函数的参数格式是 #1\pgfeov，它的替换文本是

```
\pgfkeysdef{\pgfkeyscurrentpath}{f(#1)}
```

(ii) 执行

```
\pgfkeys{/⟨key⟩/.⟨handler name⟩={g(#1)}}
```

把 /⟨*key*⟩ 做成一个手柄键，也就是以 $g(\#1)$ 为参数来执行手柄 ⟨*handler name*⟩ 的替换文本

```
\pgfkeysdef{\pgfkeyscurrentpath}{f(g(#1))}
```

其中的 \pgfkeyscurrentpath 是 /⟨*key*⟩，这就定义了作为函数的键 /⟨*key*⟩，即控制序列 (函数)

\csname pgfk@/⟨*key*⟩/.@cmd\endcsname

它的参数格式是 #1\pgfeov，它的替换文本是 $f(g(\#1))$

(iii) 执行

```
\pgfkeys{/⟨key⟩=⟨value⟩}
```

也就是求函数值 $f(g(\langle value\rangle))$.

2.4.2 调整 \pgfkeys@case@three 的处理方式

\pgfkeys{⟨*key-value*⟩} 对 ⟨*key*⟩ 的处理次序是：

- 检查 ⟨*key*⟩ 是否首字符句法，如果不是，则
- 检查 ⟨*key*⟩ 是否完整的键，如果不是就添加默认路径，得到 \pgfkeyscurrentkey，然后
- 检查 ⟨*key*⟩ 的值 \pgfkeyscurrentvalue，如果正常，则
- 检查 ⟨*key*⟩ 是否函数 (\pgfkeys@case@one)，如果不是，则
- 检查 ⟨*key*⟩ 是否变量 (\pgfkeys@case@two)，如果不是，则
- 检查 ⟨*key*⟩ 是否手柄键 (\pgfkeys@case@three)

在初始之下，对于 ⟨*key*⟩/.⟨*handler name*⟩，命令 \pgfkeys@case@three 只是检查手柄 ⟨*handler name*⟩，并不会检查键 ⟨*key*⟩ 是否已定义。这一点是可以改变的。

```
\let\pgfkeys@case@three@handleall=\pgfkeys@case@three
%...
\def\pgfkeys@ifexecutehandler#1#2{#1}%
\let\pgfkeys@ifexecutehandler@handleall=\pgfkeys@ifexecutehandler
%...
```

```
\def\pgfkeysaddhandleonlyexistingexception#1{\expandafter\def\csname pgfk@excpt@#1
↪  \endcsname{1}}%
%...
\pgfkeysaddhandleonlyexistingexception{.cd}%
\pgfkeysaddhandleonlyexistingexception{.try}%
\pgfkeysaddhandleonlyexistingexception{.retry}%
\pgfkeysaddhandleonlyexistingexception{.lastretry}%
\pgfkeysaddhandleonlyexistingexception{.unknown}%
\pgfkeysaddhandleonlyexistingexception{.expand once}%
\pgfkeysaddhandleonlyexistingexception{.expand twice}%
\pgfkeysaddhandleonlyexistingexception{.expanded}%
```

上面的命令 \pgfkeysaddhandleonlyexistingexception 限定了一些手柄，(在初始下) 它们是 /.cd, /.try, /.retry, /.lastretry, /.unknown, /.expand once, /.expand two once, /.expanded。

\pgfkeys@case@three@handle@restricted

这个命令的处理是：

1. 执行 \pgfkeys@split@path →P. 51，分解 \pgfkeyscurrentkey
2. 检查手柄 \pgfkeyscurrentname 是否已定义，即检查函数

   \csname pgfk@/handlers/\pgfkeyscurrentname/.@cmd\endcsname

   是否已定义，

   - 如果已定义,则执行 \pgfkeys@ifexecutehandler{⟨*first code*⟩}{⟨*second code*⟩},命令 \pgfkeys@ifexecut 可能会被 \let 为不同的命令，如
     - \pgfkeys@ifexecutehandler@handleonlyexisting
     - \pgfkeys@ifexecutehandler@handlefullorexisting →P. 57

     这里的 ⟨*first code*⟩ 是

     ```
     \pgfkeysgetvalue{/handlers/\pgfkeyscurrentname/.@cmd}{\pgfkeys@code}%
     \expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
     ```

     这里的 ⟨*second code*⟩ 是

     ```
     \let\pgfkeys@tempa=\pgfkeyscurrentkey
     \let\pgfkeys@tempb=\pgfkeyscurrentname
     \edef\pgfkeyscurrentkey{\pgfkeyscurrentpath}%
     \pgfkeys@split@path%
     \let\pgfkeyscurrentkey=\pgfkeys@tempa
     \edef\pgfkeyscurrentname{\pgfkeyscurrentname/\pgfkeys@tempb}%
     \pgfkeys@unknown
     ```

   - 如果未定义，则执行 \pgfkeys@unknown

\pgfkeys@ifexecutehandler@handleonlyexisting{⟨*first code*⟩}{⟨*second code*⟩}

这个命令的处理是：检查控制序列

\csname pgfk@excpt@\pgfkeyscurrentname\endcsname

是否已定义，

- 如果已定义，则执行 ⟨*first code*⟩；
- 如果未定义,则检查作为变量的键 \pgfkeyscurrentpath 是否已定义,即控制序列 (变量) \csname pgfk@\pgfkeyscurrentpath\endcsname 是否已定义，
  - 如果已定义，则执行 ⟨*first code*⟩；
  - 如果未定义，则检查作为函数的键 \pgfkeyscurrentpath 是否已定义，即控制序列 (函数) \csname pgfk@\pgfkeyscurrentpath/.@cmd\endcsname 是否已定义，
    - 如果已定义，则执行 ⟨*first code*⟩；

    * 如果未定义，则执行 ⟨*second code*⟩。

\pgfkeys@ifexecutehandler@handlefullorexisting{⟨*first code*⟩}{⟨*second code*⟩}

这个命令的处理是：检查 \ifpgfkeysaddeddefaultpath 的真值，

- 如果真值是 true，则检查控制序列

  \csname pgfk@excpt@\pgfkeyscurrentname\endcsname

  是否已定义，
  - 如果已定义，则执行 ⟨*first code*⟩；
  - 如果未定义，则检查作为变量的键 \pgfkeyscurrentpath 是否已定义，即控制序列 (变量) \csname pgfk@\pgfkeyscurrentpath\endcsname 是否已定义，
    * 如果已定义，则执行 ⟨*first code*⟩；
    * 如果未定义，则检查作为函数的键 \pgfkeyscurrentpath 是否已定义，即控制序列 (函数) \csname pgfk@\pgfkeyscurrentpath/.@cmd\endcsname 是否已定义，
      · 如果已定义，则执行 ⟨*first code*⟩；
      · 如果未定义，则执行 ⟨*second code*⟩。
- 如果真值是 false，则执行 ⟨*first code*⟩。

2.4.2.1 情况一

假设

```
\let\pgfkeys@case@three=\pgfkeys@case@three@handleall
\let\pgfkeys@ifexecutehandler=\pgfkeys@ifexecutehandler@handleall
```

这等价于原始的 \pgfkeys@case@three。

2.4.2.2 情况二

假设

```
\let\pgfkeys@case@three=\pgfkeys@case@three@handle@restricted
\let\pgfkeys@ifexecutehandler=\pgfkeys@ifexecutehandler@handleonlyexisting
```

在这个情况下，\pgfkeys{⟨*key*⟩/.⟨*handler name*⟩=⟨*value*⟩} 的处理是：

- 如果 ⟨*handler name*⟩ 是未定义的手柄，则执行 \pgfkeys@unknown → P. 51
- 如果 ⟨*handler name*⟩ 是已定义的手柄，再检查 ⟨*handler name*⟩ 是否属于“限定的手柄”，
  - 如果 ⟨*handler name*⟩ 属于“限定的手柄”，则执行 ⟨*first code*⟩，即

    ```
    \pgfkeysgetvalue{/handlers/\pgfkeyscurrentname/.@cmd}{\pgfkeys@code}%
    \expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
    ```

  - 如果 ⟨*handler name*⟩ 不属于“限定的手柄”，则再检查键 \pgfkeyscurrentpath 是否已定义的变量或者函数，
    * 如果是，则执行 ⟨*first code*⟩，
    * 如果不是，则执行 ⟨*second code*⟩，即

      ```
      \let\pgfkeys@tempa=\pgfkeyscurrentkey
      \let\pgfkeys@tempb=\pgfkeyscurrentname
      \edef\pgfkeyscurrentkey{\pgfkeyscurrentpath}%
      \pgfkeys@split@path%
      \let\pgfkeyscurrentkey=\pgfkeys@tempa
      \edef\pgfkeyscurrentname{\pgfkeyscurrentname/\pgfkeys@tempb}%
      \pgfkeys@unknown
      ```

2.4.2.3 情况三

假设

```
\let\pgfkeys@case@three=\pgfkeys@case@three@handle@restricted
\let\pgfkeys@ifexecutehandler=\pgfkeys@ifexecutehandler@handlefullorexisting
```

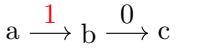
在这个情况下，\pgfkeys{⟨*key*⟩/.⟨*handler name*⟩=⟨*value*⟩} 的处理是：

- 如果 ⟨*handler name*⟩ 是未定义的手柄，则执行 \pgfkeys@unknown→P. 51
- 如果 ⟨*handler name*⟩ 是已定义的手柄，再检查 \ifpgfkeysaddeddefaultpath 的真值，即检查 ⟨*key*⟩ 是否完整的键，
  - 如果是完整的键，则执行 ⟨*first code*⟩，
  - 如果不是完整的键，则再检查 ⟨*handler name*⟩ 是否属于“限定的手柄”，
    - 如果 ⟨*handler name*⟩ 属于“限定的手柄”，则执行 ⟨*first code*⟩，
    - 如果 ⟨*handler name*⟩ 不属于“限定的手柄”，则再检查键 \pgfkeyscurrentpath 是否已定义的变量或者函数，
      - 如果是，则执行 ⟨*first code*⟩，
      - 如果不是，则执行 ⟨*second code*⟩。

2.5 首字符句法

当 \ifpgfkeys@syntax@handlers 的真值是 true 时，开启首字符句法检查，见 \pgfkeys@normal→P. 48。

“首字符句法”的意思是，将某个特殊字符与某个命令对应起来；当 \pgfkeys→P. 47 处理键 ⟨*key*⟩ (或键值对 ⟨*key*⟩=⟨*value*⟩) 时，首先检查 ⟨*key*⟩ 的第一个字符是否那个特殊字符；如果是，就用对应的命令处理 ⟨*key*⟩ (或键值对 ⟨*key*⟩=⟨*value*⟩)；如果不是，就按通常的流程处理。例如




```
%\usetikzlibrary {graphs,quotes}
\tikz \graph { a ->["1" red] b ->["0"] c };
```

上面例子中的引号就是首字符句法。

\ifpgfkeys@syntax@handlers

当这个 TEX-if 的真值是 true 时，开启首字符检查。

\pgfkeys@syntax@handlers

本命令调用 \pgfkeys@syntax@@handlers。

\pgfkeys@syntax@@handlers

本命令将 ⟨*key*⟩ 的第一个字符复制到 \pgfkeys@first@char，然后调用 \pgfkeys@syntax@handlers@test。

\pgfkeys@syntax@handlers@test

本命令 let 宏 \pgfkeys@the@handler 等于控制序列

```
\csname pgfk@/handlers/first char syntax/\meaning\pgfkeys@first@char\endcsname
```

然后检查这个宏是否等于 \relax，如果等于，就是它还未被定义，则执行 \pgfkeys@@normal→P. 48；如果不等于，就是它已被定义，则执行 \pgfkeys@use@handler→P. 59。

注意在上述控制序列的名称中有 \meaning\pgfkeys@first@char，它的展开是“the character ⟨字符⟩”这种形式。

本命令的定义是：

```
\def\pgfkeys@syntax@handlers@test{%
  \pgfkeysgetvalue{/handlers/first char syntax/\meaning\pgfkeys@first@char}
  ↪  \pgfkeys@the@handler%
```

```
  \ifx\pgfkeys@the@handler\relax%
    \expandafter\pgfkeys@@normal%
  \else%
    \expandafter\pgfkeys@use@handler%
  \fi%
}
```

注意上面定义中使用了 \pgfkeysgetvalue→P. 44.

\pgfkeys@use@handler⟨*tokens*⟩,

本命令调用前述的 \pgfkeys@the@handler 处理 ⟨*tokens*⟩，注意这个 ⟨*tokens*⟩ 是整个的键 ⟨*key*⟩ (或键值对 ⟨*key*⟩=⟨*value*⟩, 包含首字符)。

本命令吃掉一个逗号。然后执行 \pgfkeys@parse→P. 48，继续解析后面的键值对。

本命令的定义是：

```
\long\def\pgfkeys@use@handler#1,{%
  \pgfkeys@the@handler{#1}%
  \pgfkeys@parse%
}
```

本命令有前缀 \long，所以本命令的参数中可以含有 \par。

注意 \pgfkeys@the@handler 所等于的控制序列

```
\csname pgfk@/handlers/first char syntax/\meaning\pgfkeys@first@char\endcsname
```

应当能把键 ⟨*key*⟩ (或键值对 ⟨*key*⟩=⟨*value*⟩) 作为参数。

使用首字符句法功能至少需要 2 个准备：

1. 设置真值 \pgfkeys@syntax@handlerstrue，可以使用下面的键：

   /handlers/first char syntax=true|false (initially false)

   这个键是布尔键，它的定义是：

   ```
   \pgfkeys{/handlers/first char syntax/.is if=pgfkeys@syntax@handlers}
   ```

   \pgfkeys{/handlers/first char syntax=true} 会设置真值 \pgfkeys@syntax@handlerstrue，开启首字符句法检查。

2. 定义控制序列

   ```
   \csname pgfk@/handlers/first char syntax/the character ⟨ 字符⟩\endcsname
   ```

   如果把这个控制序列定义为一个变量，那么它应当保存一个函数，这个函数应当能处理之后的 {#1}. 此时可以使用 \pgfkeyssetvalue→P. 43, 或者手柄 /.initial 把键

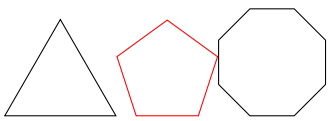
   /handlers/first char syntax/the character ⟨ 字符⟩

   定义为一个变量。

   如果把这个控制序列定义为一个函数，那么它应当处理一个参数，参数格式就简单的是 {#1}. 但是注意此时不能用 \pgfkeysdef→P. 44 或者手柄 /.code 来把键

   /handlers/first char syntax/the character ⟨ 字符⟩

   定义为一个函数，因为这样的定义与 \pgfkeysgetvalue→P. 44 不匹配。



```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character '/.initial=\mysinglechar,
```

```
  /tikz/mysinglecharstyle/.style={}
}
\makeatletter
\def\mysinglechar#1{%
  \begingroup
  \mysinglecharcont#1%
  \tikz\node[regular polygon, draw, inner sep=3mm,
             regular polygon sides=\mysinglecharnum, mysinglecharstyle]{};%
  \endgroup
}
\def\mysinglecharcont'#1'{%
  \def\mysinglecharnum{#1}%
  \pgfutil@ifnextchar[{\mysinglecharoptA}%
  {\pgfutil@ifnextchar\bgroup\mysinglecharoptB\relax}%
}

\def\mysinglecharoptA[#1]{%
  \tikzset{%
    mysinglecharstyle/.append style={#1}%
  }%
}
\def\mysinglecharoptB#1{%
  \mysinglecharoptA#1%
}
\makeatother
\pgfkeys{'3','5'{[inner sep=4mm,draw=red]},'8'[inner sep=5mm]}
```

2.6 预定义的手柄

2.6.1 设置键路径的手柄

Key handler ⟨*key*⟩`/.cd`

例如

```
\pgfkeys{⟨key⟩/.cd, ⟨key 1⟩, ⟨key 2⟩,...}
```

将 ⟨*key*⟩ 设为 ⟨*key 1*⟩, ⟨*key 2*⟩ 的默认前缀路径。

此手柄的定义是

```
\pgfkeys{/handlers/.cd/.code=\edef\pgfkeysdefaultpath{\pgfkeyscurrentpath/}}
```

这会导致

```
\pgfkeysdef{/handlers/.cd}{\edef\pgfkeysdefaultpath{\pgfkeyscurrentpath/}}
```

所以执行 `\pgfkeys{/my key/.cd}` 就导致

```
\edef\pgfkeysdefaultpath{/my key/}
```

Key handler ⟨*key*⟩`/.is family`

此手柄的作用是，例如：

```
\pgfkeys{/tikz/.is family}
\pgfkeys{tikz,line width=1cm,line cap=round}
等效于
\pgfkeys{tikz/.cd,line width=1cm,line cap=round}
```

此手柄的定义是：

```
\pgfkeys{/handlers/.is family/.code=\pgfkeys{\pgfkeyscurrentpath/.ecode=
↪  \edef\noexpand\pgfkeysdefaultpath{\pgfkeyscurrentpath/}}}
```

这会导致

```
\pgfkeysdef{/handlers/.is family}{\pgfkeys{\pgfkeyscurrentpath/.ecode=\edef
↪  \noexpand\pgfkeysdefaultpath{\pgfkeyscurrentpath/}}}
```

所以执行 `\pgfkeys{/my key/.is family}` 就导致

```
\pgfkeys{/my key/.ecode=\edef\noexpand\pgfkeysdefaultpath{/my key/}}
导致
\pgfkeysedef{/my key}{\edef\noexpand\pgfkeysdefaultpath{/my key/}}
```

所以执行 `\pgfkeys{/my key}` 就导致

```
\edef\pgfkeysdefaultpath{/my key/}
```

2.6.2 设置键的默认值的手柄

Key handler ⟨*key*⟩`/.default=`⟨*value*⟩

将 ⟨*key*⟩ 的默认值设为 ⟨*value*⟩.

此手柄的定义是：

```
\pgfkeys{/handlers/.default/.code=\pgfkeyssetvalue{\pgfkeyscurrentpath/.@def}{#1}}
```

这会导致

```
\pgfkeysdef{/handlers/.default}{\pgfkeyssetvalue{\pgfkeyscurrentpath/.@def}{#1}}
```

所以执行 `\pgfkeys{/my key/.default=`⟨*value*⟩`}` 就导致

```
\pgfkeyssetvalue{/my key/.@def}{⟨value⟩}
```

这会定义 `\csname pgfk@/my key/.@def\endcsname` 的值为 ⟨*value*⟩.

Key handler ⟨*key*⟩`/.value required`

此手柄将 ⟨*key*⟩ 的默认值，也就是变量 `pgfk@\pgfkeyscurrentpath/.@def` 的值设置为 `\pgfkeysvaluerequired`。当执行 ⟨*key*⟩ 但这没有为 ⟨*key*⟩ 赋值时，就引起 `/erros/value required` 被执行，产生一个错误提示。例如

```
\pgfkeys{/width/.value required}
\pgfkeys{/width}
```

就会得到错误提示：！Package pgfkeys Error: The key '/width' requires a value.

此手柄的定义是：

```
\pgfkeys{/handlers/.value required/.code=\pgfkeyssetvalue{
↪  \pgfkeyscurrentpath/.@def}{\pgfkeysvaluerequired}}
```

Key handler ⟨*key*⟩`/.value forbidden`

这个手柄禁止为 ⟨*key*⟩ 赋值。当以 ⟨*key*⟩=⟨*value*⟩ 的形式执行时，这个手柄引起函数 `/errors/value forbidden/.@cmd` 被执行，产生一个错误提示。

2.6.3 定义键所储存的代码

Key handler ⟨*key*⟩`/.code=`⟨*code*⟩

手柄 `/.code` 的定义是：

```
\pgfkeysdef{/handlers/.code}{\pgfkeysdef{\pgfkeyscurrentpath}{#1}}
```

这个定义展开如下 (参考`\pgfkeysdef` → P. 44)：

```
\long\def\pgfkeys@temp#1\pgfeov{\pgfkeysdef{\pgfkeyscurrentpath}{#1}}%
\pgfkeyslet{/handlers/.code/.@cmd}{\pgfkeys@temp}%
\pgfkeyssetvalue{/handlers/.code/.@body}{\pgfkeysdef{\pgfkeyscurrentpath}{#1}}%
```

所以执行 \pgfkeys{⟨key⟩/.code={⟨code⟩}} 导致 (参考 \pgfkeys@case@three→P. 51)：

```
\pgfkeysgetvalue{/handlers/.code/.@cmd}{\pgfkeys@code}%
\expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
导致
\pgfkeysdef{⟨key⟩}{⟨code⟩}
```

在 ⟨code⟩ 中最多能用一个变量 #1, 这会定义两个控制序列：

- 函数 \csname pgfk@⟨full key⟩/.@cmd\endcsname, 它的参数格式是 #1\pgfeov
- 变量 \csname pgfk@⟨full key⟩/.@body\endcsname

此后，使用 \pgfkeys{/my key} 或 \pgfkeys{/my key=⟨参数⟩} 可以导致代码 ⟨code⟩ 被执行，执行时以 ⟨参数⟩ 替换 ⟨code⟩ 中的变量 #1. 也可以执行

\pgfkeysvalueof{⟨my key⟩/.@cmd}{⟨ 参数⟩}

来处理 ⟨参数⟩。

按照 \pgfkeys 的处理流程，手柄 /.default 与 /.code 可以配合，例如

2^n

```
\pgfkeys{/a/.default=n}
\pgfkeys{/a/.code={$2^{#1}$}}
\pgfkeys{/a}
```

上面例子中执行 \pgfkeys{/a} 时使用了 /a 的默认值 n.

但是手柄 /.initial 与 /.code 一般不能配合，其原因参考 /.initial 的例子。

Key handler ⟨key⟩/.code 2 args=⟨code⟩

手柄 /.code 2 args 的定义是：

```
\pgfkeysdef{/handlers/.code 2 args}{\pgfkeysdefargs{\pgfkeyscurrentpath}{##1##2
↪  }{#1}}
```

此定义展开为：

```
\long\def\pgfkeys@temp#1\pgfeov{\pgfkeysdefargs{\pgfkeyscurrentpath}{##1##2}{#1}}%
\pgfkeyslet{/handlers/.code 2 args/.@cmd}{\pgfkeys@temp}%
\pgfkeyssetvalue{/handlers/.code 2 args/.@body}{\pgfkeysdefargs{
↪  \pgfkeyscurrentpath}{##1##2}{#1}}%
```

所以执行 \pgfkeys{/my key/.code 2 args={⟨code with ⟨value1⟩ and ⟨value2⟩⟩}} 就得到

```
\pgfkeysgetvalue{/handlers/.code 2 args/.@cmd}{\pgfkeys@code}%
\expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
导致
\pgfkeys@code{⟨code with ⟨value1⟩ and ⟨value2⟩⟩}\pgfeov
导致
\pgfkeysdefargs{/my key}{#1#2}{{⟨code with ⟨value1⟩ and ⟨value2⟩⟩}}
```

这导致定义 3 个控制序列 (参考 \pgfkeysdefargs→P. 45)：

- 函数 \csname pgfk@⟨full key⟩/.@cmd\endcsname，它的参数格式是 #1#2\pgfeov
- 变量 \csname pgfk@⟨full key⟩/.@args\endcsname
- 变量 \csname pgfk@⟨full key⟩/.@body\endcsname

所以，执行

\pgfkeys{⟨full key⟩/.code 2 args={⟨code with ⟨value1⟩ and ⟨value2⟩⟩}}

会导致执行 \pgfkeysdefargs→P. 45

\pgfkeysdefargs{⟨full key⟩}{#1#2}{⟨code with ⟨value1⟩ and ⟨value2⟩⟩}

Key handler ⟨key⟩/.ecode=⟨code⟩

手柄 /.ecode 的定义是：

```
\pgfkeysdef{/handlers/.ecode}{\pgfkeysedef{\pgfkeyscurrentpath}{#1}}
```

这是手柄 /.code 的 \edef 版本。

Key handler ⟨*key*⟩/.ecode 2 args=⟨*code*⟩

手柄 /.ecode 2 args 定义是：

```
\pgfkeysdef{/handlers/.ecode 2 args}{\pgfkeysedefargs{\pgfkeyscurrentpath}{##1##2
↪  }{#1}}
```

这是手柄 /.code 2 args 的 \edef 版本。

Key handler ⟨*key*⟩/.code args={⟨*argument pattern*⟩}{⟨*code with* ⟨*value1*⟩*...*⟩}

手柄 /.code args 的定义是：

```
\pgfkeysdefnargs{/handlers/.code args}{2}{\pgfkeysdefargs{\pgfkeyscurrentpath}{#1
↪  }{#2}}
```

此定义导致控制序列 (参考 \pgfkeysdefnargs→P. 46)：

- 函数 \csname pgfk@/handlers/.code args/.@cmd\endcsname，它的参数格式是 #1\pgfeov
- 变量 \csname pgfk@/handlers/.code args/.@args\endcsname，这是个变量，它保存 #1#2
- 变量 \csname pgfk@/handlers/.code args/.@body\endcsname，它保存

  ```
  \pgfkeysdefargs{\pgfkeyscurrentpath}{#1}{#2}
  ```

当执行 \pgfkeys{/my key/.code args={⟨*argument pattern*⟩}{⟨*code with* ⟨*value1*⟩*...*⟩}} 时，导致

```
\pgfkeysgetvalue{/handlers/.code args/.@cmd}{\pgfkeys@code}%
\expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
```

导致

```
\csname pgfk@/handlers/.code args/.@cmd\endcsname
↪  {⟨argument pattern⟩}{⟨code with ⟨value1⟩...⟩}\pgfeov
```

导致

```
\csname pgfk@/handlers/.code args/.@@body\endcsname
↪  {⟨argument pattern⟩}{⟨code with ⟨value1⟩...⟩}
```

导致

```
\pgfkeysdefargs{/my key}{⟨argument pattern⟩}{⟨code with ⟨value1⟩...⟩}
```

这又导致定义 3 个控制序列 (参考 \pgfkeysdefargs→P. 45)：

- 函数 \csname pgfk@⟨*full key*⟩/.@cmd\endcsname，它的参数格式是 #1\pgfeov，其中 #1 对应键值
- 变量 \csname pgfk@⟨*full key*⟩/.@args\endcsname
- 变量 \csname pgfk@⟨*full key*⟩/.@body\endcsname

所以，执行

\pgfkeys{/my key/.code args={⟨*argument pattern*⟩}{⟨*code with* ⟨*value1*⟩*...*⟩}}

会导致执行

\pgfkeysdefargs{/my key}{⟨*argument pattern*⟩}{⟨*code with* ⟨*value1*⟩*...*⟩}

在 ⟨*argument pattern*⟩ 中最多使用 9 个参数。

Key handler ⟨*key*⟩/.ecode args={⟨*argument pattern*⟩}{⟨*code with* ⟨*value1*⟩*...*⟩}

手柄 /.ecode args 的定义是：

```
\pgfkeysdefnargs{/handlers/.ecode args}{2}{\pgfkeysedefargs{\pgfkeyscurrentpath
↪  }{#1}{#2}}
```

这是手柄 /.code args 的 \edef 版本。

Key handler ⟨*key*⟩/.code n args={⟨*argument count*⟩}{⟨*code with* ⟨*value1*⟩...⟩}

手柄 /.code n args 的定义是：

```
\pgfkeysdefnargs{/handlers/.code n args}{2}{\pgfkeysdefnargs{\pgfkeyscurrentpath
↪ }{#1}{#2}}
```

执行 \pgfkeys{/my key/.code args={⟨*argument count*⟩}{⟨*code with* ⟨*value1*⟩...⟩}} 会导致执行

\pgfkeysdefnargs{/my key}{⟨*argument count*⟩}{⟨*code with* ⟨*value1*⟩...⟩}

在 ⟨*argument count*⟩ 中最多使用 9 个参数，这导致定义 3 个控制序列 (参考 \pgfkeysdefnargs→P. 46)：

- 函数 \csname pgfk@⟨*full key*⟩/.@cmd\endcsname，它的参数格式是 #1#2...\pgfeov
- 变量 \csname pgfk@⟨*full key*⟩/.@args\endcsname
- 变量 \csname pgfk@⟨*full key*⟩/.@body\endcsname

Key handler ⟨*key*⟩/.ecode n args={⟨*argument count*⟩}{⟨*code with* ⟨*value1*⟩...⟩}

手柄 /.ecode n args 的定义是：

```
\pgfkeysdefnargs{/handlers/.ecode n args}{2}{\pgfkeysedefnargs{\pgfkeyscurrentpath
↪ }{#1}{#2}}
```

这是手柄 /.code n args 的 \edef 版本。

Key handler ⟨*key*⟩/.add code={⟨*prefix code*⟩}{⟨*append code*⟩}

当用 ⟨*key*⟩/.code=⟨*code*⟩ 定义 ⟨*key*⟩ 之后，可以用这个手柄将 ⟨*prefix code*⟩ 添加到 ⟨*key*⟩/.@cmd 的开头，将 ⟨*append code*⟩ 添加到 ⟨*key*⟩/.@cmd 的结尾，二者任何一个都可以空置。也就是说此手柄能重定义键 ⟨*key*⟩, 使得保存在键 ⟨*key*⟩ 中的内容由三部分构成：第一是 ⟨*prefix code*⟩, 再是 ⟨*key*⟩/.@cmd, 再是 ⟨*append code*⟩. ⟨*prefix code*⟩ 和 ⟨*append code*⟩ 中可以带有变量，但变量不能超出 ⟨*code*⟩ 中的变量，且变量的序号与 ⟨*code*⟩ 中的变量序号一致。

比较 1 与 2 „, < „, is it?

```
\pgfkeys{/ps/.code args={#1 and #2}{\ifnum #1>#2 > \else < \fi}}
\pgfkeys{/ps/.add code={比较 #1 与 #2 ,,, }{ ,,, is it?}}
\pgfkeys{/ps=1 and 2}
```

Key handler ⟨*key*⟩/.prefix code={⟨*prefix code*⟩}

这是 ⟨*key*⟩/.add code={⟨*prefix code*⟩}{} 的简捷用法。

Key handler ⟨*key*⟩/.append code={⟨*append code*⟩}

这是 ⟨*key*⟩/.add code={}{⟨*append code*⟩} 的简捷用法。

2.6.4 定义样式的手柄

Key handler ⟨*key*⟩/.style={⟨*key list*⟩}

在执行 \pgfkeys{⟨*key*⟩/.style={⟨*key list*⟩}} 后，通常就把键 ⟨*key*⟩ 称为一个“样式”(style)。⟨*key list*⟩ 是一系列键值的列表，其中的键应当是已经定义的，并且都有相同的默认的前缀路径。在 ⟨*key list*⟩ 中至多可以使用 1 个变量 #1.

文件《pgfkeys.code.tex》中有定义：

```
\pgfkeys{/handlers/.style/.code=\pgfkeys{\pgfkeyscurrentpath/.code=\pgfkeysalso{#1
↪ }}}
```

按 /.code，此定义展开为

```
\pgfkeysdef{/handlers/.style}{\pgfkeys{\pgfkeyscurrentpath/.code=\pgfkeysalso{#1}}
↪ }
```

按 \pgfkeysdef 的定义，进一步展开为

```
\long\def\pgfkeys@temp#1\pgfeov{\pgfkeys{\pgfkeyscurrentpath/.code=\pgfkeysalso{#1
↪ }}}%
\pgfkeyslet{/handlers/.style/.@cmd}{\pgfkeys@temp}%
\pgfkeyssetvalue{/handlers/.style/.@body}{\pgfkeys{\pgfkeyscurrentpath/.code=
↪ \pgfkeysalso{#1}}}%
```

当执行 \pgfkeys{/my key/.style={⟨*key list*⟩}} 时，导致执行

```
\pgfkeys@temp ⟨key list⟩\pgfeov
```

导致

```
\pgfkeys{/my key/.code=\pgfkeysalso{⟨key list⟩}}
```

导致

```
\pgfkeysdef{/my key}{\pgfkeysalso{⟨key list⟩}}
```

在 ⟨*key list*⟩ 中最多能用一个变量 #1, 这会定义两个控制序列：

- \csname pgfk@⟨*key*⟩/.@cmd\endcsname
- \csname pgfk@⟨*key*⟩/.@body\endcsname

所以执行 \pgfkeys{/my key} 导致执行 \pgfkeysalso{⟨*key list*⟩}.

red box

blue box

```
\begin{tikzpicture}[outline/.style={draw=#1,fill=#1!20}]
  \node [outline=red] {red box};
  \node [outline=blue] at (0,-1) {blue box};
\end{tikzpicture}
```

按照 \pgfkeys 的处理流程，手柄 /.default 与 /.style 可以配合，例如

```
\pgfkeys{/a/.default=red}
\pgfkeys{/a/.style={draw=#1}}
\tikz\draw [/a,very thick](0,0)--(1,1);
```

但是手柄 /.initial 与 /.style 一般不能配合。

可以用一个样式定义其他的样式，例如

$x^{\ln y}$

```
\pgfkeysdef{/a/b/c}{$x^{#1}$}
\pgfkeys{/a/.style={/a/b/.style={/a/b/c={##1}}}}
\pgfkeys{/a}% 此命令定义样式 /a/b
\pgfkeys{/a/b={\ln y}}
```

Key handler ⟨*key*⟩/.estyle=⟨*key list*⟩

手柄 /.style 的定义是：

```
\pgfkeys{/handlers/.estyle/.code=\pgfkeys{\pgfkeyscurrentpath/.ecode=
↪ \noexpand\pgfkeysalso{#1}}}
```

这是 /.style 的 \edef 版本。

Key handler ⟨*key*⟩/.style 2 args=⟨*key list*⟩

手柄 /.style 2 args 的定义是：

```
\pgfkeys{/handlers/.style 2 args/.code=\pgfkeys{\pgfkeyscurrentpath/.code 2 args=
↪ \pgfkeysalso{#1}}}
```

此定义展开为

```
\pgfkeysdef{/handlers/.style 2 args}{\pgfkeys{\pgfkeyscurrentpath/.code 2 args=
↪ \pgfkeysalso{#1}}}
```

执行 \pgfkeys{/my key/.style 2 args={⟨*key list*⟩}} 就导致执行

```
\pgfkeys{/my key/.code 2 args=\pgfkeysalso{⟨key list⟩}}
```

导致定义

```
\pgfkeysdefargs{/my key}{#1#2}{\pgfkeysalso{⟨key list⟩}}
```

在 ⟨*key list*⟩ 中至多使用两个变量 #1, #2, 这导致定义 3 个控制序列 (参数 \pgfkeysdefargs→P. 45):

- \csname pgfk@⟨*full key*⟩/.@cmd\endcsname
- \csname pgfk@⟨*full key*⟩/.@args\endcsname
- \csname pgfk@⟨*full key*⟩/.@body\endcsname

Key handler ⟨*key*⟩/.style args={⟨*argument pattern*⟩}{⟨*key list*⟩}

手柄 /.style args 的定义是:

```
\pgfkeys{/handlers/.style args/.code 2 args=\pgfkeys{\pgfkeyscurrentpath/.code
↪  args={#1}{\pgfkeysalso{#2}}}}
```

此定义展开为

```
\pgfkeysdef{/handlers/.style args}{\pgfkeys{\pgfkeyscurrentpath/.code args={#1}{
↪  \pgfkeysalso{#2}}}}
```

执行 \pgfkeys{/my key/.style args={⟨*argument pattern*⟩}{⟨*key list*⟩}} 就导致执行

```
\pgfkeys{/my key/.code args={⟨argument pattern⟩}{\pgfkeysalso{⟨key list⟩}}}
```

导致定义

```
\pgfkeysdefargs{/my key}{⟨argument pattern⟩}{\pgfkeysalso{⟨key list⟩}
```

这导致定义 3 个控制序列 (参数 \pgfkeysdefargs→P. 45):

- \csname pgfk@⟨*full key*⟩/.@cmd\endcsname
- \csname pgfk@⟨*full key*⟩/.@args\endcsname
- \csname pgfk@⟨*full key*⟩/.@body\endcsname

Key handler ⟨*key*⟩/.estyle args={⟨*argument pattern*⟩}{⟨*key list*⟩}

手柄 /.estyle args 的定义是:

```
\pgfkeys{/handlers/.estyle args/.code 2 args=\pgfkeys{\pgfkeyscurrentpath/.ecode
↪  args={#1}{\noexpand\pgfkeysalso{#2}}}}
```

这是 /.style args 的 \edef 版本。

Key handler ⟨*key*⟩/.style n args={⟨*argument count*⟩}{⟨*key list*⟩}

手柄 /.style n args 的定义是:

```
\pgfkeys{/handlers/.style n args/.code 2 args=\pgfkeys{\pgfkeyscurrentpath/.code n
↪  args={#1}{\pgfkeysalso{#2}}}}
```

此定义展开为

```
\pgfkeysdef{/handlers/.style n args}{\pgfkeys{\pgfkeyscurrentpath/.code n args={#1
↪  }{\pgfkeysalso{#2}}}}
```

执行 \pgfkeys{/my key/.style n args={⟨*argument count*⟩}{⟨*key list*⟩}} 就导致执行

```
\pgfkeys{/my key/.code n args={⟨argument count⟩}{\pgfkeysalso{⟨key list⟩}}}
```

导致定义

```
\pgfkeysdefnargs{/my key}{⟨argument count⟩}{\pgfkeysalso{⟨key list⟩}
```

这导致定义 3 个控制序列 (参数 \pgfkeysdefnargs→P. 46):

- \csname pgfk@⟨*full key*⟩/.@args\endcsname
- \csname pgfk@⟨*full key*⟩/.@body\endcsname
- \csname pgfk@⟨*full key*⟩/.@cmd\endcsname, 这是可以处理参数的命令

Key handler ⟨*key*⟩/.add style={⟨*prefix key list*⟩}{⟨*append key list*⟩}

这个手柄重定义键 ⟨*key*⟩, 使得保存在键 ⟨*key*⟩ 中的内容是：先是 ⟨*prefix key list*⟩, 再是键 ⟨*key*⟩ 原来保存的 ⟨*key list*⟩, 再是 ⟨*append key list*⟩.

手柄 /.add style 的定义是：

```
\pgfkeys{/handlers/.add style/.code 2 args=\pgfkeys{\pgfkeyscurrentpath/.add code=
↪  {\pgfkeysalso{#1}}{\pgfkeysalso{#2}}}}%
```

此定义展开为

```
\pgfkeysdef{/handlers/.add style}{\pgfkeys{\pgfkeyscurrentpath/.add code={
↪  \pgfkeysalso{#1}}{\pgfkeysalso{#2}}}}
```

执行 \pgfkeys{/my key/.add style={⟨*prefix key list*⟩}{⟨*append key list*⟩}} 就导致执行

```
\pgfkeys{/my key/.add code={\pgfkeysalso{⟨prefix key list⟩}}{\pgfkeysalso{
↪  ⟨append key list⟩}}}
```

```
\pgfkeys{/a/b/c/.style={circle},
        /a/b/c/.add style={draw=red}{fill=green}}
\tikz\draw node[/a/b/c] {\color{gray!80}\rule{1em}{1em}};
```

Key handler ⟨*key*⟩/.prefix style={⟨*prefix key list*⟩}

这是 ⟨*key*⟩/.add style={⟨*prefix key list*⟩}{} 的简捷形式。

Key handler ⟨*key*⟩/.append style={⟨*append key list*⟩}

这是 ⟨*key*⟩/.add style={}{⟨*append key list*⟩} 的简捷形式。

2.6.5 Defining Value-, Macro-, If- and Choice-Keys

Key handler ⟨*key*⟩/.initial=⟨*value*⟩

设置 ⟨*key*⟩ 的初始值，如果不需要这个初始值，可以用 ⟨*key*⟩=⟨*Value*⟩ 修改键的值为 ⟨*Value*⟩. 也可以用 ⟨*key*⟩= 修改键的值为空的。

手柄 /.initial 的定义是：

```
\pgfkeys{/handlers/.initial/.code=\pgfkeyssetvalue{\pgfkeyscurrentpath}{#1}}
```

此定义展开为

```
\pgfkeysdef{/handlers/.initial}{\pgfkeyssetvalue{\pgfkeyscurrentpath}{#1}}
```

所以执行 \pgfkeys{/my key/.initial=⟨*value*⟩} 就导致

```
\pgfkeyssetvalue{/my key}{⟨value⟩}
```

这会把 ⟨*value*⟩ 作为记号保存到 \csname pgfk@/my key\endcaname 中，在此之后：

- 若是执行 \pgfkeys{/my key} 就导致 (参考命令 \pgfkeys@case@two)

  ```
  \pgfkeysvalueof{/my key}
  即执行
  \csname pgfk@/my key\endcaname
  输出 ⟨value⟩
  ```

- 若是执行 \pgfkeys{/my key=⟨*Value*⟩} 就导致

  ```
  \pgfkeyslet{/my key}{\pgfkeyscurrentvalue}
  ```

  也就是把 \csname pgfk@/my key\endcaname 重定义为 ⟨*Value*⟩, 并不输出 ⟨*Value*⟩.

注意，手柄 /.initial 设置的值一般不能被手柄 /.code 设置的键利用：

```
\pgfkeys{/a/.initial=A,/a/.code={x #1}}
\pgfkeys{/a}\par% 输出 x
\makeatletter
\csname pgfk@/a\endcsname% 输出 A
\makeatother\par
\pgfkeysvalueof{/a}% 输出 A
```

x
A
A

对于上面的例子，执行 \pgfkeys{/a} 时，首先遇到的是命令 \csname pgfk@/a/.@cmd\endcsname (在 \pgfkeys@case@one→P. 50 那里), 而不是 \csname pgfk@/a\endcsname (在 \pgfkeys@case@two→P. 50 那里)。可以使用命令 \pgfkeysvalueof→P. 44{/a} 或者 \pgfkeysgetvalue→P. 44{/a}{⟨*macro*⟩} 来获取键 /a 保存的内容。

与上面的例子类似，手柄 /.initial 与 /.store in 也有类似的问题需要注意。

手柄 /.initial 通常会定义一个变量，但如果 \pgfkeys{/my key=⟨*function*⟩}，那么键 /my key 也可以像函数那样处理参数。

Key handler ⟨*key*⟩/.get=⟨*macro*⟩

手柄 /.get 的定义是：

```
\pgfkeys{/handlers/.get/.code=\pgfkeysgetvalue{\pgfkeyscurrentpath}{#1}}
```

此定义展开为

```
\pgfkeysdef{/handlers/.get}{\pgfkeysgetvalue{\pgfkeyscurrentpath}{#1}}
```

所以执行 \pgfkeys{/my key/.get=⟨*macro*⟩} 就导致

```
\pgfkeysgetvalue{/my key}{⟨macro⟩}
```

即使用 \let 把保存在键 /my key 中的代码转存到 ⟨*macro*⟩ 中。

```
\pgfkeys{/my key/.initial=red}
\pgfkeys{/my key/.get=\mymacro}\mymacro\par
\pgfkeys{/my key=blue}
\pgfkeys{/my key/.get=\mymacro}\mymacro
```

red
blue

Key handler ⟨*key*⟩/.add={⟨*prefix value*⟩}{⟨*append value*⟩}

手柄 /.add 会重定义 ⟨*key*⟩ 所保存的值，使得它的值包含三部分：第一部分是 ⟨*prefix value*⟩, 第二部分是该键原来的值，第三部分是 {⟨*append value*⟩}.

手柄 /.add 的定义是：

```
\pgfkeys{/handlers/.add/.code 2 args=\pgfkeysaddvalue{\pgfkeyscurrentpath}{#1}{#2
↪  }}
```

命令 \pgfkeysaddvalue→P. 44 针对的是 \csname pgfk@/⟨*full key*⟩\endcsname

Key handler ⟨*key*⟩/.prefix={⟨*prefix value*⟩}

这是 ⟨*key*⟩/.add={⟨*prefix value*⟩}{} 的简捷形式。

Key handler ⟨*key*⟩/.append={⟨*append value*⟩}

这是 ⟨*key*⟩/.add={}{⟨*append value*⟩} 的简捷形式。

Key handler ⟨*key*⟩/.link=⟨*another key*⟩

将 \pgfkeysvalueof{⟨*another key*⟩} 的值保存到 ⟨*key*⟩ 中，在展开 ⟨*key*⟩ 时，展开的是 ⟨*another key*⟩ 的值。

手柄 /.link 的定义是：

```
\pgfkeys{/handlers/.link/.code=\pgfkeyssetvalue{\pgfkeyscurrentpath}{
↪  \pgfkeysvalueof{#1}}}
```

Key handler ⟨*key*⟩/.store in=⟨*macro*⟩

当执行 ⟨*key*⟩=⟨*value*⟩ 时，会自动执行 \def⟨*macro*⟩{⟨*value*⟩}.

手柄 /.store in 的定义是：

```
\pgfkeys{/handlers/.store in/.code=\pgfkeysalso{\pgfkeyscurrentpath/.code=\def#1
↪  {##1}}}
\pgfkeys{/handlers/.estore in/.code=\pgfkeysalso{\pgfkeyscurrentpath/.code=\edef#1
↪  {##1}}}
```

此定义的展开定义两个控制序列：

- \csname pgfk@/handlers/.store in/.@cmd\endcsname, 这个命令能处理一个参数
- \csname pgfk@/handlers/.store in/.@body\endcsname

当执行 \pgfkeys{⟨*key*⟩}/.store in=⟨*macro*⟩ 时，导致

```
\pgfkeysgetvalue{/handlers/.store in/.@cmd}{\pgfkeys@code}%
\expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
```

导致

```
\pgfkeysalso{⟨key⟩/.code=\def⟨\macro⟩{#1}}
```

这又导致定义两个控制序列：

- \csname pgfk@⟨*full key*⟩/.@cmd\endcsname, 这个命令能处理一个参数
- \csname pgfk@⟨*full key*⟩/.@body\endcsname

再执行 \pgfkeys{{⟨*key*⟩}=⟨*value*⟩} 时，就导致

```
\def⟨\macro⟩{⟨value⟩}
```

Hello Gruffalo!

```
\pgfkeys{/text/.store in=\mytext}
\def\a{world}
\pgfkeys{/text=Hello \a!}
\def\a{Gruffalo}
\mytext
```

当用 /.initial 和 /.store in 定义同一个键名时，可能出现下面的情况：

macro:->\pgfkeysnovalue

macro:->XXX

A

```
\pgfkeys{/test/.initial=A, /test/.store in=\mytest}
\pgfkeys{/test}% 没有 =<value>, 所以键值是 \pgfkeysnovalue
%\csname pgfk@/test/.@cmd\endcsname\pgfkeysnovalue\pgfeov
% 即 \def\mytest{\pgfkeysnovalue}
\meaning\mytest\par
\pgfkeys{/test=XXX}%\def\mytest{XXX}
\meaning\mytest\par
\makeatletter
\csname pgfk@/test\endcsname% 输出 A
\makeatother
```

其中的原因是，执行 \pgfkeys{/test} 时，首先遇到的是命令 \csname pgfk@/test/.@cmd\endcsname (在 \pgfkeys@case@one →P. 50 那里), 而不是 \csname pgfk@/test\endcsname (在 \pgfkeys@case@two →P. 50 那里)。对比 /.initial 的例子。

Key handler ⟨*key*⟩/.estore in=⟨*macro*⟩

当执行 ⟨*key*⟩=⟨*value*⟩ 时，会自动执行 \edef⟨*macro*⟩{⟨*value*⟩}.

Key handler ⟨*key*⟩/.is if=⟨*TEX-if name*⟩

Round?
```
\newif\iftheworldisflat
\pgfkeys{/flat world/.is if=theworldisflat}
\pgfkeys{/flat world=false}
\iftheworldisflat
  Flat
\else
  Round?
\fi
```

当执行 ⟨*key*⟩=⟨*value*⟩ 时 (⟨*value*⟩ 是 true 或 false)，将 ⟨*value*⟩ 赋予 ⟨*TEX-if name*⟩. 如果只写出 ⟨*key*⟩ 就默认 ⟨*value*⟩ 的真值是 true. 如果 ⟨*TEX-if name*⟩ 没有被事先声明，或者 ⟨*value*⟩ 不是 true 或 false, 就执行 `/errors/boolean expected/.@cmd` 给出错误提示。

手柄 `/.is if` 的定义是：

```
\pgfkeys{/handlers/.is if/.code=\pgfkeysalso{%
    \pgfkeyscurrentpath/.code=\pgfkeys@handle@boolean{#1}{##1},
    \pgfkeyscurrentpath/.default=true%
  }%
}
```

此定义展开为

```
\pgfkeysdef{/handlers/.is if}{\pgfkeysalso{%
    \pgfkeyscurrentpath/.code=\pgfkeys@handle@boolean{#1}{##1},
    \pgfkeyscurrentpath/.default=true%
  }%
}
```

所以执行 `\pgfkeys{/my key/.is if=`⟨*TEX-if name*⟩`}` 就导致

```
\pgfkeysalso{%
    /my key/.code=\pgfkeys@handle@boolean{⟨TEX-if name⟩}{#1},
    /my key/.default=true%
  }%
```

导致

```
\pgfkeysdef{/my key}{\pgfkeys@handle@boolean{⟨TEX-if name⟩}{#1}}
\pgfkeyssetvalue{/my key/.@def}{true}
```

此后：

- 如果执行 `\pgfkeys{/my key=false}` 就导致

  ```
  \pgfkeys@handle@boolean{⟨TEX-if name⟩}{false}
  ```

  `\pgfkeys@handle@boolean{`⟨*if tokens*⟩`}{`⟨*true or false*⟩`}`

  本命令的定义是：

  ```
  \def\pgfkeys@handle@boolean#1#2{%
    \ifcsname#1#2\endcsname%
      \csname#1#2\endcsname%
    \else%
      \def\pgf@marshal{\pgfkeysvalueof{/errors/boolean expected/.@cmd}}%
      \expandafter\pgf@marshal\expandafter{\pgfkeyscurrentkey}{#2}\pgfeov%
    \fi
  }
  ```

  由于 ⟨*if tokens*⟩ 处于 `\ifcsname...\endcsname` 之内，所以 ⟨*if tokens*⟩ 可以是 TEX-if 名称，或者是展开为 TEX-if 名称的命令。

yes
```
\expandafter\newif\csname ifa b\endcsname
\def\abtf#1{#1}
\ifcsname\abtf{a b}true\endcsname%
  yes\else%
  no\fi%
```

此时 \pgfkeys@handle@boolean 执行一个条件语句，检查 ⟨*TEX-if name*⟩ 是否已被声明：

– 如果名称为 ⟨*TEX-if name*⟩ 的 if 命令已经被“声明”，则执行

```
\csname ⟨TEX-if name⟩false\endcsname
```

– 如果名称为 ⟨*TEX-if name*⟩ 的 if 命令尚未被“声明”，则执行

```
\def\pgf@marshal{\pgfkeysvalueof{/errors/boolean expected/.@cmd}}%
\expandafter\pgf@marshal\expandafter{\pgfkeyscurrentkey}{#2}\pgfeov%
```

- 如果执行 \pgfkeys{/my key=true} 或 \pgfkeys{/my key} 就导致

```
\pgfkeys@handle@boolean{⟨TEX-if name⟩}{true}
```

Key handler ⟨*key*⟩/.is choice

这个手柄的作用是，当执行 ⟨*key*⟩=⟨*value*⟩ 时，会自动执行 ⟨*key*⟩/⟨*value*⟩, 这要求键 ⟨*key*⟩/⟨*value*⟩ 已经存在，否则，若找不到这个键就给出错误提示。

若键值 ⟨*key*⟩/⟨*value*⟩=⟨*Value*⟩ 可执行，那也可以执行 \pgfkeys{⟨*key*⟩={⟨*value*⟩=⟨*Value*⟩}}, 注意其中一定要用花括号把 ⟨*value*⟩=⟨*Value*⟩ 括起来 (原因在于命令 \pgfkeys@unpack → P. 49 的定义格式)。

c3
```
\pgfkeys{/a/b/.code={b}}
\pgfkeys{/a/c/.code={c#1}}
\pgfkeys{/a/.is choice}
\pgfkeys{/a={c=3}}
```

2.6.6 键值的展开，多重键值

当写出 \expandafter\aaaa\bbbb 时，命令 \expandafter 先把 \bbbb 展开一次，得到其定义内容 ⟨*contents*⟩, 然后再顺次执行 \aaaa⟨*contents*⟩. 命令 \expandafter 只对 \bbbb 做“一个层次”的展开，而不是把 \bbbb “彻底展开”。例如

AAAA
```
\def\aaaa\bbbb{AAAA}
\def\bbbb{BBBB}
\def\cccc{\bbbb}
\expandafter\aaaa\cccc
```

命令 \expandafter 把 \cccc 展开为 \bbbb, 而不是彻底展开为 BBBB. 用命令 \expandafter 把 \cccc 展开两次才会得到 BBBB.

AAAA
```
\def\aaaa\bbbb{AAAA}
\def\cccc{\bbbb}
\ifnum 1<2  \expandafter \aaaa \else \fi \bbbb
```

上面代码中命令 \expandafter 作用于 \else, 使得 TEX 去寻找 \fi 从而结束 \if...\fi 句子，但是会留下 \aaaa.

当处理 ⟨*key*⟩=⟨*value*⟩ 时，可以选择先将 ⟨*value*⟩ 展开一次或两次，再利用展开后的值。

Key handler ⟨*key*⟩/.expand once=⟨*value*⟩

这个手柄用命令 \expandafter 作用于 ⟨*value*⟩ 的第一个记号 (只是第一个)，得到 ⟨*Value*⟩, 再对 ⟨*Value*⟩ 加以利用，也就是相当于处理 ⟨*key*⟩=⟨*Value*⟩. 注意，如果 ⟨*key*⟩ 中含有手柄，则按通常的方式调用该手柄。

Key 1: \c
Key 2: \b
Key 3: \a
Key 4: bottom

```
\def\a{bottom}
\def\b{\a}
\def\c{\b}
\pgfkeys{/key1/.initial=\c}
\pgfkeys{/key2/.initial/.expand once=\c} % 将 \c 展开一次，/key2=\b
\pgfkeys{/key3/.initial/.expand twice=\c} % 将 \c 展开两次，/key3=\a
\pgfkeys{/key4/.initial/.expanded=\c} % 将 \c 完全展开，/key4=bottom
\def\a{{\ttfamily\string\a}} % 重定义 \a
\def\b{{\ttfamily\string\b}}
\def\c{{\ttfamily\string\c}}
Key 1:\quad \pgfkeys{/key1} \\
Key 2:\quad \pgfkeys{/key2} \\
Key 3:\quad \pgfkeys{/key3} \\
Key 4:\quad \pgfkeys{/key4}
```

手柄 /.expand once 的定义是：

```
\pgfkeys{/handlers/.expand once/.code=\expandafter\pgfkeys@exp@call\expandafter{#1
↪ }}
```

这导致

```
\pgfkeysdef{/handlers/.expand once}{\expandafter\pgfkeys@exp@call\expandafter{#1}}
```

所以执行 \pgfkeys{/my key/.expand once=⟨*value*⟩} 就导致执行

```
\expandafter\pgfkeys@exp@call\expandafter{⟨value⟩}
```

导致

```
\pgfkeysalso{/my key={⟨Value⟩}}
```

其中的 ⟨*Value*⟩ 是把 ⟨*value*⟩ 的第一个记号展开一次后的 (整个 ⟨*value*⟩ 的) 情况。这个命令就是给 \pgfkeyscurrentvalue 定义内容。

Key handler ⟨*key*⟩/.expand twice=⟨*value*⟩

这个手柄将 ⟨*value*⟩ 展开两次再加以利用，等效于 ⟨*key*⟩/.expand once/.expand once=⟨*value*⟩.

手柄 /.expand twice 的定义是：

```
\pgfkeys{/handlers/.expand twice/.code=
↪ \expandafter\expandafter\expandafter\pgfkeys@exp@call
↪ \expandafter\expandafter\expandafter{#1}}
```

Key handler ⟨*key*⟩/.expanded=⟨*value*⟩

使用命令 \edef 将 ⟨*value*⟩ 完全展开再加以利用。

手柄 /.expanded 的定义是：

```
\pgfkeys{/handlers/.expanded/.code=\edef\pgfkeys@temp{#1}
↪ \expandafter\pgfkeys@exp@call\expandafter{\pgfkeys@temp}}
```

Key handler ⟨*key*⟩/.expand two once={⟨*value1*⟩}{⟨*value2*⟩}

手柄 /.expand two once 的定义是：

```
\pgfkeys{/handlers/.expand two once/.code 2 args=
↪ \expandafter\expandafter\expandafter\pgfkeys@exp@call
↪ \expandafter\expandafter\expandafter{\expandafter#1\expandafter}\expandafter
↪ {#2}}
```

Key handler ⟨*key*⟩/.list=⟨*comma-separated list of values*⟩

⟨*comma-separated list of values*⟩ 是用逗号分隔的值列表，其中的每个值都可以作为键值，整个列表要用花括号括起来。⟨*comma-separated list of values*⟩ 会被 \foreach 处理，其中可以使用省略号。

(a)(b)(0)(1)(2)(3)(4)(5)

```
\pgfkeys{/foo/.code=(#1)}
\pgfkeys{/foo/.list={a,b,0,1,...,5}}
```

2.6.7 键路径的转换

Key handler ⟨*key*⟩/.forward to=⟨*another key*⟩

观察下面的例子：

(a:1), (b)(a:), (a:3)

```
\pgfkeys{
  /a/.code=(a:#1),
  /b/.code=(b),
  /b/.forward to=/a,
  /c/.forward to=/a,
}
\pgfkeys{/a=1}, \pgfkeys{/b}, \pgfkeys{/c=3}
```

可见手柄 /.forward to 会把键 ⟨*key*⟩ 与键 ⟨*another key*⟩ 联系起来。如果键 ⟨*key*⟩ 之前已经有定义，那么执行 ⟨*key*⟩=⟨*value*⟩ 时，先执行键 ⟨*key*⟩ 自己原来该执行的内容，然后执行 ⟨*another key*⟩=⟨*value*⟩. 如果键 ⟨*key*⟩ 之前没有定义，那么直接执行 ⟨*another key*⟩=⟨*value*⟩.

注意这里的 ⟨*another key*⟩ 最好是完整的键。

手柄 /.forward to 的定义是：

```
\pgfkeys{/handlers/.forward to/.code=%
  \pgfkeys{\pgfkeyscurrentpath/.add code={}{\pgfkeys{#1={##1}}}}
}
```

这会导致

```
\pgfkeysdef{/handlers/.forward to}{\pgfkeys{\pgfkeyscurrentpath/.add code={}{
↪   \pgfkeys{#1={##1}}}}}
```

所以执行 \pgfkeys{/my key/.forward to=/another full key} 就导致

```
\pgfkeys{/my key/.add code={}{\pgfkeys{/another full key={#1}}}}
```

这导致重定义 /my key, 使得 /my key 所保存的代码包含两部分：第一部分是 /my key 原来保存的代码，第二部分是 \pgfkeys{/another full key={#1}}. 所以执行 \pgfkeys{/my key={⟨*code*⟩}} 时，就导致这两部分代码被依次执行，并且执行时用 ⟨*code*⟩ 替换代码中的 #1.

Key handler ⟨*key*⟩/.search also={⟨*path list*⟩}

观察下面的例子：

Invoking /secondary path/option with ‘value’

```
\pgfkeys{/secondary path/option/.code={Invoking /secondary path/option with ‘#1’ }}
\pgfkeys{/main path/.search also={/secondary path}}
\pgfkeys{/main path/.cd, option=value}
```

上面例子的第一行定义键 /secondary path/option. 第二行将键路径 /main path 关联到 /secondary path. 在执行第三行时，命令会发现 /main path/option 是未定义的键，因此就用 /secondary path 替换 /main path, 执行键 /secondary path/option.

这个手柄的定义是：

```
\pgfkeys{%
  /handlers/.search also/.code={%
    \pgfkeys@searchalso@prepare@unknown@handler{#1}%
%\message{I prepared the '\pgfkeyscurrentpath/.unknown' handler \meaning\pgfkeys@global@temp\s
    \pgfkeyslet{\pgfkeyscurrentpath/.unknown/.@cmd}{\pgfkeys@global@temp}%
  }
}%
```

这导致

```
\pgfkeysdef{/handlers/.search also/}{
  \pgfkeys@searchalso@prepare@unknown@handler{#1}%
  \pgfkeyslet{\pgfkeyscurrentpath/.unknown/.@cmd}{\pgfkeys@global@temp}%
}
```

其中 \pgfkeys@searchalso@prepare@unknown@handler→P. 74 的最终作用是全局地定义命令

\pgfkeys@global@temp#1\pgfeov

而 \pgfkeyslet 又使得控制序列

\csname pgfk@\pgfkeyscurrentpath/.unknown/.@cmd\endcsname

等于 \pgfkeys@global@temp.

当执行 \pgfkeys{/key path 0/.search also={⟨*key path list*⟩}} 时，导致

```
\pgfkeys@searchalso@prepare@unknown@handler{⟨key path list⟩}%
\pgfkeyslet{/key path 0/.unknown/.@cmd}{\pgfkeys@global@temp}%
```

这导致如下处理：

1. \pgfkeys@searchalso@prepare@unknown@handler→P. 74{⟨*key path list*⟩}
2. \pgfkeyslet{/key path 0/.unknown/.@cmd}{\pgfkeys@global@temp}

在执行 \pgfkeys{/key path 0/key name 0={⟨*value*⟩}} 时，如果 /key path 0/key name 0 未定义，就执行

\pgfkeys@global@temp⟨*value*⟩\pgfeov

注意下面的代码导致错误：

```
\pgfkeys{/secondary path/option/.code={Invoking /secondary path/option with ‘#1’ }
↪  }
\pgfkeys{/main path/.search also={/secondary path}}
\pgfkeys{/main path/option=value}
```

```
! Package pgfkeys Error: I do not know the key '/main path/option', to which yo u
passed 'value', and I am going to ignore it. Perhaps you misspelled it.
```

导致错误的原因是：其中写出的 /main path/option 是以 / 开头的完整路径，此时有 \pgfkeysaddeddefaultpathfalse。

参见 \pgfkeys@searchalso@prepare@unknown@handler→P. 74 的处理结果。

下面看一下命令 \pgfkeys@searchalso@prepare@unknown@handler.

\pgfkeys@searchalso@prepare@unknown@handler{⟨*key path 1*⟩,⟨*key path 2*⟩}

本命令执行一个递归处理，大体上是：在一个组内，

1. 保存记号

```
\toks0={%
%
  \ifpgfkeyssuccess
  \else
    \pgfqkeys{⟨key path 1⟩}{%
      \pgfkeys@searchalso@name/.try/.expand once=
      ↪  \pgfkeys@searchalso@temp@value}%
  \fi
%
  \ifpgfkeyssuccess
  \else
    \pgfqkeys{⟨key path 2⟩}{%
      \pgfkeys@searchalso@name/.try/.expand once=
      ↪  \pgfkeys@searchalso@temp@value}%
```

```
  \fi
}
```

2. 全局定义

```
\gdef\pgfkeys@global@temp#1\pgfeov{%
  \def\pgfkeys@searchalso@temp@value{#1}%
  \ifpgfkeysaddeddefaultpath
    \expandafter\pgfkeys@firstoftwo
  \else
    \expandafter\pgfkeys@secondoftwo
  \fi{%
    \pgfkeyssuccessfalse
    \let\pgfkeys@searchalso@name=\pgfkeyscurrentkeyRAW
    \the\toks0 %
    }{%
       \pgfkeysgetvalue{/handlers/.unknown/.@cmd}{\pgfkeys@code}%
       \expandafter\pgfkeys@code\pgfkeys@searchalso@temp@value\pgfeov
    }%
}
```

- \ifpgfkeysaddeddefaultpath 的真值如下决定 (\pgfkeys@add@path@as@needed → P. 49)：
  - 如果命令 \pgfkeys 处理的键是以 / 开头的，那么就认为这个键是完整的，此时设置真值 \pgfkeysaddeddefaultpathfalse
  - 如果命令 \pgfkeys 处理的键不是以 / 开头的，那么就认为这个键不是完整的，需要确定这个键的前缀路径，此时设置真值 \pgfkeysaddeddefaultpathtrue
- \ifpgfkeyssuccess 是手柄 /.try，/.retry 使用的条件。当命令 \pgfkeys 利用手柄 /.try，/.retry 处理键时,如果处理成功,一般会有 \pgfkeyssuccesstrue, 否则有 \pgfkeyssuccessfalse
- 宏 \pgfkeyscurrentkeyRAW 保存的是用户写出的键。

\ifpgfkeyssuccess

如上述，这个 TEX-if 是手柄 /.try，/.retry 使用的条件。

2.6.8 测试键的手柄

Key handler ⟨*key*⟩/.try=⟨*value*⟩

有如下定义：

```
\newif\ifpgfkeyssuccess
\pgfkeys{/handlers/.try/.code=\pgfkeys@try}
```

这会导致

```
\pgfkeysdef{/handlers/.try}{\pgfkeys@try}
```

所以执行 \pgfkeys{/my key/.try} 或 \pgfkeys{/my key/.try=⟨*value*⟩} 就是执行 \pgfkeys@try.

\pgfkeys@try

命令 \pgfkeys@try 的处理过程是：

1. 定义

   ```
   \edef\pgfkeyscurrentkey{\pgfkeyscurrentpath}
   ```

   也就是把 /my key 作为 \pgfkeyscurrentkey, 以下针对 /my key.

2. 用 \ifx 检查 \pgfkeyscurrentvalue 与 \pgfkeysnovalue@text 的定义是否相同：
   - 如果 \pgfkeyscurrentvalue 与 \pgfkeysnovalue@text 的定义相同，即

\pgfkeys{/my key/.try} 这样，就用命令 \pgfkeysifdefined 检查名称为 pgfk@\pgfkeyscurrentpath/.@def 的命令是否已定义：

– 如果已定义就利用默认值

```
\pgfkeysgetvalue{\pgfkeyscurrentpath/.@def}{\pgfkeyscurrentvalue}
```

– 如果未定义，就什么也不做。

- 如果 \pgfkeyscurrentvalue 与 \pgfkeysnovalue@text 的定义不相同，就什么也不做。

此时 \pgfkeyscurrentvalue 保存的可能是：

(i) ⟨*value*⟩, 例如 \pgfkeys{/my key/.try=⟨*value*⟩} 是这种情况。
(ii) 空的内容，例如 \pgfkeys{/my key/.try=} 是这种情况。
(iii) \pgfkeysnovalue，例如 \pgfkeys{/my key/.try} 是这种情况。
(iv) 默认值。
(v) \pgfkeysvaluerequired。

3. 用 \pgfkeysifdefined 检查名称为 pgfk@\pgfkeyscurrentpath/.@cmd 的命令是否已定义：

- 如果已定义就执行

```
\pgfkeysgetvalue{\pgfkeyscurrentpath/.@cmd}{\pgfkeys@code}%
\expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov%
\pgfkeyssuccesstrue%
```

- 如果未定义，则用 \pgfkeysifdefined 检查名称为 pgfk@\pgfkeyscurrentpath 的命令是否已定义：

– 如果已定义，则

(a) 用 \ifx 检查 \pgfkeyscurrentvalue 与 \pgfkeysnovalue@text 的定义是否相同：

* 如果相同，则执行

```
\pgfkeysvalueof{\pgfkeyscurrentpath}
```

* 如果不同，则执行

```
\pgfkeyslet{\pgfkeyscurrentpath}\pgfkeyscurrentvalue
```

(b) 设置 \pgfkeyssuccesstrue.

– 如果未定义，则

(a) 执行 \pgfkeys@split@path

(b) 用 \pgfkeysifdefined 检查名称为 pgfk@/handlers/\pgfkeyscurrentname/.@cmd 的命令是否已定义：

▷ 如果已定义，则执行 \pgfkeys@ifexecutehandler,

\pgfkeys@ifexecutehandler

此命令的定义是：

```
\def\pgfkeys@ifexecutehandler#1#2{#1}%
```

此时这个命令有两个参数：第一个参数是

```
\pgfkeysgetvalue{/handlers/\pgfkeyscurrentname/.@cmd}{
↪  \pgfkeys@code}%
\expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
\pgfkeyssuccesstrue%
```

第二个参数是 \pgfkeyssuccessfalse. 在默认之下，执行其第一个参数。

▷ 如果未定义，则设置 \pgfkeyssuccessfalse.

从以上处理过程看出，命令 \pgfkeys@try 其实是

- \pgfkeys@unpack → P. 49
- \pgfkeys@case@one → P. 50
- \pgfkeys@case@two → P. 50
- \pgfkeys@case@three → P. 51

这几个命令的修改版。执行 \pgfkeys{/my key/.try} 或 \pgfkeys{/my key/.try=⟨*value*⟩} 会导致执行 \pgfkeys{/my key} 或 \pgfkeys{/my key=⟨*value*⟩}.

对于 \pgfkeys{/⟨*key path*⟩/⟨*key name*⟩/.try=⟨*value*⟩}, 以下情况之一：

- 名称为 pgfk@/⟨*key path*⟩/⟨*key name*⟩/.@cmd 的命令存在
- 名称为 pgfk@/⟨*key path*⟩/⟨*key name*⟩ 的命令存在
- 名称为 pgfk@/handlers/⟨*key name*⟩/.@cmd 的命令存在，即 ⟨*key name*⟩ 是手柄

设置 \pgfkeyssuccesstrue, 其它情况设置 \pgfkeyssuccessfalse.

注意在 \newif\ifpgfkeyssuccess 之后，自动默认 \pgfkeyssuccessfalse.

Key handler ⟨*key*⟩/.retry=⟨*value*⟩

有定义：

```
\pgfkeys{/handlers/.retry/.code=\ifpgfkeyssuccess\else\pgfkeys@try\fi}
```

即仅在 \pgfkeyssuccessfalse 的情况下执行 \pgfkeys@try.

2.6.9 解释键的手柄

Key handler ⟨*key*⟩/.show value

这个手柄用 \show 来显示保存在 ⟨*key*⟩ 中值。

有定义

```
\pgfkeys{/handlers/.show value/.code=\pgfkeysgetvalue{\pgfkeyscurrentpath}{
↪  \pgfkeysshower}\show\pgfkeysshower} % inspect the value
```

可见显示的是保存在 \csname pgfk@⟨*full key*⟩\endcsname 中的内容。

Key handler ⟨*key*⟩/.show code

这个手柄用 \show 来显示保存在 ⟨*key*⟩ 中的代码。

有定义

```
\pgfkeys{/handlers/.show code/.code=\pgfkeysgetvalue{\pgfkeyscurrentpath/.@cmd}{
↪  \pgfkeysshower}\show\pgfkeysshower} % inspect the body of the command
```

可见显示的是保存在 `\csname pgfk@⟨full key⟩/.@cmd\endcsname` 中的内容。

/utils/exec=⟨code⟩ (no default)

本选项直接执行 ⟨code⟩, 其定义是：

```
\pgfkeys{/utils/exec/.code=#1} % simply execute the given code directly.
```

2.7 提示错误的键

目前，error keys 只是被简单地执行，将来可能设计一些子键，来提供更丰富的信息。

/errors/value required={⟨offending key⟩}{⟨value⟩} (no default)

当这个键被执行时，编译会中断并产生一个错误提示，提示需要赋值的键 ⟨offending key⟩ 没有被赋值。⟨value⟩ 是提示信息，但实际上没有用处。

这个键的定义是：

```
\pgfkeys{/errors/value required/.code 2 args={%
    \toks1={#1}
    \pgfkeys@error{%
      The key '\the\toks1' requires a value. I am going to ignore this
      key%
    }}}
```

/errors/value forbidden={⟨offending key⟩}{⟨value⟩} (no default)

当这个键被执行时，编译会中断并产生一个错误提示，提示不需要赋值的键 ⟨offending key⟩ 被赋值。⟨value⟩ 是提示信息。

这个键的定义是：

```
\pgfkeys{/errors/value forbidden/.code 2 args={%
    \toks1={#1}
    \toks2={#2}
    \pgfkeys@error{%
      You may not specify a value for the key '\the\toks1'. I am going to ignore
      the value '\the\toks2' that you provided%
    }}}
```

/errors/boolean expected={⟨offending key⟩}{⟨value⟩} (no default)

使用手柄 `/.is if` 定义的键会自动关联此选项（参考手柄 `/.is if` 的定义）。如果 ⟨offending key⟩ 的值不是 `true` 或 `false`, 就给出错误信息 ⟨value⟩.

这个键的定义是：

```
\pgfkeys{/errors/boolean expected/.code 2 args={%
    \toks1={#1}
    \toks2={#2}
    \pgfkeys@error{%
      Boolean parameter of key '\the\toks1' must be 'true' or 'false', not
      '\the\toks2'. I am going to ignore it%
    }}}
```

/errors/unknown choice value={⟨offending key⟩}{⟨value⟩} (no default)

使用手柄 `/.is choice` 定义的键会自动关联此选项（参考手柄 `/.is choice` 的定义）。如果 ⟨value⟩ 没有出现在 ⟨offending key⟩ 之下的名单中，就给出错误信息.

这个键的定义是：

```
\pgfkeys{/errors/unknown choice value/.code 2 args={%
    \toks1={#1}
    \toks2={#2}
    \pgfkeys@error{%
      Choice '\the\toks2' unknown in choice key '\the\toks1'. I am
      going to ignore this key%
    }}}
```

/errors/unknown key={⟨*offending key*⟩}{⟨*value*⟩} (no default)

当该键被执行时，编译会中断并产生一个错误提示，提示键 ⟨*offending key*⟩ 是未定义的。

这个键的定义是：

```
\pgfkeys{/errors/unknown key/.code 2 args={%
    \toks1={#1}
    \toks2={#2}
    \def\pgf@temp{#2}%
    \pgfkeys@error{%
      I do not know the key '\the\toks1'\ifx\pgf@temp\pgfkeysnovalue@text
      ↪  \space\else, to which you passed
      '\the\toks2', \fi and I am going to ignore it. Perhaps you
      misspelled it%
    }}}
```

Key handler ⟨*key*⟩/.unknown

手柄 /.unknown 的定义是：

```
\pgfkeys{/handlers/.unknown/.code=%
  {%
    \def\pgf@marshal{\pgfkeysvalueof{/errors/unknown key/.@cmd}}%
    {\expandafter\expandafter\expandafter\pgf@marshal
    ↪  \expandafter\expandafter\expandafter{
    ↪  \expandafter\pgfkeyscurrentkey\expandafter}\expandafter{
    ↪  \pgfkeyscurrentvalue}\pgfeov}%
  }%
}
```

2.8 键筛选

2.8.1 简介

Key filtering, 以更多样的方式来利用 PGF 的键。本节内容主要面向 package (or library) authors.

本节介绍文件《pgfkeysfiltered.code.tex》提供的工具。注意 PGF 会先载入《pgfkeys.code.tex》，再载入《pgfkeysfiltered.code.tex》。

本节使用下面的键作为例子：

```
\pgfkeys{
    /my group/A1/.code=(A1:#1),
    /my group/A2/.code=(A2:#1),
    /my group/A3/.code=(A3:#1),
    /my group/B/.code=(B:#1),
    /my group/C/.code=(B:#1),
}
```

注意这些键都是用手柄 /.code 定义的，所以它们对应的 pgfk@.../.@cmd 命令都是有的。

“键筛选”或者“键过滤”(filtering) 的意思是，在处理键值对列表时，先设置某种条件，符合条件的键值对被执行，不符合条件的键值对被忽略或被另外保存。执行筛选、过滤工作的是命令 \pgfkeysfiltered→P. 84, \pgfqkeysfiltered→P. 86, 这两个命令实际上“改造过的” \pgfkeys@parse→P. 48:
当遇到 \pgfkeys@case@one 时就改为执行 \pgfkeys@case@one@filtered→P. 92, 由此进入筛选分析阶段，此时的

- \pgfkeyscurrentkey→P. 49
- \pgfkeyscurrentkeyRAW→P. 49
- \pgfkeyscurrentname→P. 51
- \pgfkeyscurrentpath→P. 51
- \pgfkeyscurrentvalue→P. 50

都是已定义的。筛选键的标准通常是围绕键的名称、路径、键值来设计的，所以会用到以上几个宏。

文件《pgfkeysfiltered.code.tex》中有

```
\pgfkeys{%
  /pgf/key filters/true/.code={\pgfkeysfiltercontinuetrue},%
  /pgf/key filters/true/.install key filter,
}
```

这导致

```
\def\pgfkeys@key@predicate{\pgfkeysfiltercontinuetrue}%
```

这使得：在初始之下，命令 \pgfkeysfiltered, \pgfqkeysfiltered 等效于 \pgfkeys, \pgfqkeys.

2.8.1.1 例子

筛选键的一般步骤是：

1. 首先应保证那些被筛选的键是已定义的，否则将导致错误。
2. 设置“过滤器”(filter)，过滤器的作用有 2 个：一是给出筛选标准，二是给出其他的一些操作。例如 /pgf/key filters/equals→P. 106 是一个过滤器。通常，过滤器是一个能处理参数的键，即函数，并且它的参数格式应当是 ⟨*argument format*⟩\pgfeov, 所以一般用手柄 /.code, /.code n args, /.code args 来定义过滤器，例如：

   ```
   \pgfkeys{⟨filter as key⟩/.code=⟨replace text⟩}
   ```

   在过滤器 ⟨*filter as key*⟩ 的替换文本 ⟨*replace text*⟩ 中，可以使用 \pgfkeyscurrentkey, \pgfkeyscurrentkeyRAW, \pgfkeyscurrentname, \pgfkeyscurrentpath, \pgfkeyscurrentvalue 这几个宏，在替换文本 ⟨*replace text*⟩ 中的参数符号 #1, #2, ... 等也可以代表这几个宏。

   过滤器分 2 种：

   (i) 被 \pgfkeys@key@predicate→P. 83 调用的过滤器，它主要给出筛选标准以及其他操作。这种过滤器可以用 /.install key filter 引入，也可以用命令 \pgfkeysinstallkeyfilter→P. 83 引入，如

   ```
   \pgfkeys{/pgf/key filters/equals/.install key filter={/my group/A1}}
   ```

   在这种过滤器的替换文本 ⟨*replace text*⟩ 中应该对 \ifpgfkeysfiltercontinue→P. 96 的真值做出规定：如果当前键值对符合筛选标准，就设置真值 \pgfkeysfiltercontinuetrue, 否则应设置真值 \pgfkeysfiltercontinuefalse.

   (ii) 被 \pgfkeys@filtered@handler→P. 84 调用的过滤器，这种过滤器可以用 /.install key filter handler 引入，也可以用命令 \pgfkeysinstallkeyfilterhandler→P. 84 引入，如

```
\pgfkeys{/pgf/key filter handlers/append filtered to/.install key filter
↪  handler=\remainingoptions}
```

在当前键不符合筛选标准的情况下，会使用这个过滤器做处理。

这两种过滤器的作用参考 \pgfkeys@case@one@filtered →P. 92.

3. 执行命令 \pgfkeysfiltered →P. 84 或 \pgfqkeysfiltered →P. 86 做筛选。

注意：一般情况下，不能在命令 \pgfkeysfiltered 或 \pgfqkeysfiltered 的内部再套嵌使用命令 \pgfkeysfiltered 或 \pgfqkeysfiltered, 也就是说，不能在一个筛选过程内部再使用筛选过程。如果你想使用多个筛选标准、过滤条件，可参考 /pgf/key filters/equals, /pgf/key filters/not, /pgf/key filters/and, /pgf/key filters/or, /pgf/key filters/true, /pgf/key filters/false.

下面是一个筛选键的例子，筛选标准是：如果被筛选的完整键名称中含有单词 nice, 就执行这个键值对；否则将这个键值对添加到 \NoNiceSave 中保存起来。

1 2 3

macro:->/four/out=no 'nice'

```
\makeatletter
% 定义筛选标准，过滤器 /a/b/c
\pgfkeysdefargs{/a/b/c}{!#1!#2!}{
  \edef\CheckIfInTemp{\noexpand\pgfutil@in@{#1}{#2}}
  \CheckIfInTemp
  \ifpgfutil@in@
    \pgfkeysfiltercontinuetrue
  \else
    \pgfkeysfiltercontinuefalse
  \fi
}
% 在不符合筛选标准时使用的过滤器 /x/y/z
\def\NoNiceSave{\pgfutil@gobble}
\pgfkeysdefargs{/x/y/z}{#1#2}{
  \edef\NoNiceSave{\NoNiceSave,#1=#2}
}
\makeatother
% 定义被筛选的键
\pgfkeys{
  /one/nice/.code={1#1},
  /two/nice/.code={2#1},
  /three/nice/.code={3#1},
  /four/out/.code={4#1},
}
% 引入筛选标准
\pgfkeysinstallkeyfilter{/a/b/c}{!nice!\pgfkeyscurrentkey!}
% 引入不符合筛选标准时使用的过滤器
\pgfkeysinstallkeyfilterhandler{/x/y/z}{\pgfkeyscurrentkey\pgfkeyscurrentvalue}
% 过滤
\pgfkeysfiltered{/one/nice,/two/nice,/three/nice,/four/out=no `nice'}\par
% 展示不符合筛选标准的键值对
\meaning\NoNiceSave
```

2.8.1.2 针对 family 的筛选

文件《pgfkeysfiltered.code.tex》对“族”(family) 这种筛选方式有专门的支持，也就是说，可以指定某些键属于同一个 family, 当筛选键时，在激活这个 family 的情况下，只有属于这个 family 的键才被执行。使用此方法的一般步骤是：

1. 先要有被筛选的键，如果被筛选的键是未定义的，就会导致错误。定义键的命令有数个，例如
   - \tikzset

- \pgfkeys
- \pgfkeyssetvalue
- \pgfkeyslet
- \pgfkeysdef
- \pgfkeysdefnargs

可用于定义键的手柄也有不少，如 /.code, 参考前文。

2. 声明一个名称为 ⟨*family name*⟩ 的 family, 例如

```
\pgfkeys{/my group/A/.is family}
```

把 /my group/A 声明为一个 family 的名称。

实际上，把 ⟨*full key as family name*⟩ 声明为一个 family 的意思是使得控制序列

\csname ifpgfe@⟨*full key as family name*⟩/familyactive\endcsname

有定义，即不等于 \relax. 参考 \pgfkeys@non@outer@newif→P. 95, /.is family.

3. 指定属于 /my group/A 的键，

```
\pgfkeys{
    /my group/A1/.belongs to family=/my group/A,
    /my group/A2/.belongs to family=/my group/A,
    /my group/A3/.belongs to family=/my group/A,
}
```

4. 激活 /my group/A

```
\pgfkeys{/my group/A/.activate family}
```

激活 ⟨*full key as family name*⟩ 这个 family 的意思是使得控制序列

\csname ifpgfe@⟨*full key as family name*⟩/familyactive\endcsname

等于 \iftrue. 参考 \pgfkeysactivatefamily→P. 87, /.activate family.

类似地，抑制 ⟨*full key as family name*⟩ 这个 family 的意思是使得控制序列

\csname ifpgfe@⟨*full key as family name*⟩/familyactive\endcsname

等于 \iffalse. 参考 \pgfkeysdeactivatefamily→P. 87, /.deactivate family.

5. 引入一个过滤器，例如，

```
\pgfkeys{/pgf/key filters/active families/.install key filter}
```

6. 执行命令 \pgfkeysfiltered 做筛选，

(A1:a1)(A2:a2)(A3:a3)

```
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
   /my group/B=b, /my group/C=c, /tikz/color=blue, /my group/A3=a3}
```

2.8.2 基本命令

在文件《pgfkeysfiltered.code.tex》的开始有以下：

```
\let\pgfkeys@orig@case@one=\pgfkeys@case@one
\let\pgfkeys@orig@@set=\pgfkeys@@set
\let\pgfkeys@orig@@qset=\pgfkeys@@qset
\let\pgfkeys@orig@try=\pgfkeys@try
\let\pgfkeys@orig@unknown=\pgfkeys@unknown

\newif\ifpgfkeysfilteringisactive
\newif\ifpgfkeysfiltercontinue
```

```
\let\pgfkeys@key@predicate=\pgfkeys@empty
\let\pgfkeys@filtered@handler=\pgfkeys@empty
\newtoks\pgfkeys@tmptoks
```

\pgfkeysinstallkeyfilter{⟨*full key*⟩}{⟨*argu list*⟩}

参数 ⟨*full key*⟩ 是完整的键，或者是保存完整键的宏，这个键应当是事先定义好的、能处理参数的“函数”。键 ⟨*full key*⟩ 对应的控制序列 pgfk@⟨*full key*⟩/.@cmd 应当是具有参数格式 ⟨*argument pattern*⟩\pgfeov 的函数，其参数以 \pgfeov 为定界标志。

函数 pgfk@⟨*full key*⟩/.@cmd 的替换文本中可以使用

- \pgfkeyscurrentkey
- \pgfkeyscurrentkeyRAW
- \pgfkeyscurrentname
- \pgfkeyscurrentpath
- \pgfkeyscurrentvalue

这几个宏，在替换文本中的参数符号 #1，#2，... 等也可以代表这几个宏。并且替换文本应当对 \ifpgfkeysfiltercontinue →P. 96 的真值做出规定。

参数 ⟨*argu list*⟩ 是将由命令 pgfk@⟨*full key*⟩/.@cmd 处理的实际参数 (不是 #1，#2，... 这样的形参)，⟨*argu list*⟩ 的格式要符合这个命令的参数格式。在 ⟨*argu list*⟩ 中可以使用上面列出的几个宏。

```
% #1: a full key name; may be a macro
% #2: optional arguments for the key. If the key expects more than one
% argument, supply '{{first}{second}}'
\def\pgfkeysinstallkeyfilter#1#2{%
    \pgfkeysifdefined{#1/.@cmd}{%
        \edef\pgfkeyscurrentkeyfilter{#1}%
        \def\pgfkeyscurrentkeyfilterargs{#2}%
        \pgfkeysgetvalue{#1/.@cmd}{\pgfkeys@key@predicate@}%
        \def\pgfkeys@key@predicate{\pgfkeys@key@predicate@#2\pgfeov}%
    }{%
        \pgfkeysvalueof{/errors/no such key filter/.@cmd}{#1}{#2}\pgfeov%
    }%
}
```

本命令检查命令 pgfk@⟨*full key*⟩/.@cmd 是否已定义，

- 如果已经定义，则
  1. 定义 \edef\pgfkeyscurrentkeyfilter{⟨*full key*⟩}
  2. 定义 \def\pgfkeyscurrentkeyfilterargs{⟨*argu list*⟩}
  3. 执行 \pgfkeysgetvalue{⟨*full key*⟩/.@cmd}{\pgfkeys@key@predicate@}
  4. 定义 \def\pgfkeys@key@predicate{\pgfkeys@key@predicate@⟨*argu list*⟩\pgfeov}
- 如果未定义，则执行

  \pgfkeysvalueof{/errors/no such key filter/.@cmd}{⟨*full key*⟩}{⟨*argu list*⟩}\pgfeov

  中断编译，给出错误信息。

1^a,2^2

```
\pgfkeys{/a/b/.code args={#1 and #2}{$1^#1$,$2^#2$}}
\def\ab{/a/b}
\pgfkeysinstallkeyfilter{\ab}{a and 2}
\makeatletter \pgfkeys@key@predicate \makeatother
```

\pgfkeys@key@predicate

如前述，在执行 \pgfkeysinstallkeyfilter 时定义此命令。它调用函数 pgfk@⟨*full key*⟩/.@cmd 处理 ⟨*argu list*⟩. 按 \pgfkeys@case@one@filtered →P. 92 处理过程,本命令的主要作用是检查当前

键值对是否符合筛选标准，也可以做一些其他的工作。在函数 pgfk@⟨*full key*⟩/.@cmd 的替换文本中应当给出筛选标准，而且：如果当前键值对符合筛选标准，就设置真值 \pgfkeysfiltercontinuetrue，否则应设置真值 \pgfkeysfiltercontinuefalse. 除了给出筛选标准，这个函数也可以做一些其他的事情。

\pgfkeysinstallkeyfilterhandler{⟨*full key*⟩}{⟨*argu list*⟩}

类似上一命令。

```
% #1: a full key name; may be a macro
% #2: optional arguments for the handler. If the handler expects more than one
% argument, supply '{{first}{second}}'
\def\pgfkeysinstallkeyfilterhandler#1#2{%
    \pgfkeysifdefined{#1/.@cmd}{%
        \edef\pgfkeyscurrentkeyfilterhandler{#1}%
        \def\pgfkeyscurrentkeyfilterhandlerargs{#2}%
        \pgfkeysgetvalue{#1/.@cmd}{\pgfkeys@filtered@handler@}%
        \def\pgfkeys@filtered@handler{\pgfkeys@filtered@handler@#2\pgfeov}%
    }{%
        \pgfkeysvalueof{/errors/no such key filter handler/.@cmd}{#1}{#2}\pgfeov%
    }%
}
```

\pgfkeys@filtered@handler

如上，在执行命令 \pgfkeysinstallkeyfilterhandler 时定义此命令。

执行命令 \pgfkeysactivatefamilies@impl →P. 88 时也会定义此命令。

本命令调用函数 pgfk@⟨*full key*⟩/.@cmd 处理 ⟨*argu list*⟩ (作为函数的实际参数)。

按 \pgfkeys@case@one@filtered →P. 92 处理过程，当检查 \ifpgfkeysfiltercontinue →P. 96 的真值为 false 时，会执行这个命令。也就是说，在当前键值对不符合筛选标准的情况下，会执行本命令。

\pgfkeyssavekeyfilterstateto{⟨*\macro*⟩}

使用本命令前应该先执行 \pgfkeysinstallkeyfilter →P. 83 和 \pgfkeysinstallkeyfilterhandler. 本命令定义宏 ⟨*\macro*⟩, 保存在这个宏中的内容是：

```
\pgfkeysinstallkeyfilter
{⟨exp 展开的 \pgfkeyscurrentkeyfilter⟩}
{⟨exp 展开的 \pgfkeyscurrentkeyfilterargs⟩}
\pgfkeysinstallkeyfilterhandler
{⟨exp 展开的 \pgfkeyscurrentkeyfilterhandler⟩}
{⟨exp 展开的 \pgfkeyscurrentkeyfilterhandlerargs⟩}
```

\pgfkeysfiltered{⟨*key-value-list*⟩}

本命令对 ⟨*key-value-list*⟩ 做筛选，其定义是：

```
\def\pgfkeysfiltered{%
    \expandafter\pgfkeysfiltered@@install\expandafter{\pgfkeysdefaultpath}%
}
```

其中的 \pgfkeysdefaultpath 是当前的默认前缀路径，将其展开值记为 ⟨*default path (string)*⟩ (一串符号)。此定义依赖下面的两个命令：

\pgfkeysfiltered@@install{⟨*default path (string)*⟩}{⟨*key-value-list*⟩}

本命令调用 \pgfkeys@install@filter@and@invoke.

```
% #1: old value of default path.
% #2: key-value-list.
\long\def\pgfkeysfiltered@@install#1#2{%
    \pgfkeys@install@filter@and@invoke{%
        \let\pgfkeysdefaultpath\pgfkeys@root%
        \pgfkeys@parse#2,\pgfkeys@mainstop%
        \def\pgfkeysdefaultpath{#1}%
    }%
}
```

\pgfkeys@install@filter@and@invoke

本命令实际执行筛选工作。

```
% #1 the code to invoke after init and before cleanup
\long\def\pgfkeys@install@filter@and@invoke#1{%
    \ifpgfkeysfilteringisactive
        \pgfkeys@error{Sorry, nested calls to key filtering routines are not
        ↪  allowed. (reason: It is not possible to properly restore the
        ↪  previous filtering state after returning from the nested call)}%
    \fi
    \pgfkeysfilteringisactivetrue
    \let\pgfkeys@case@one=\pgfkeys@case@one@filtered
    \let\pgfkeys@try=\pgfkeys@try@filtered
    \let\pgfkeys@unknown=\pgfkeys@unknown@filtered
    #1%
    \let\pgfkeys@case@one=\pgfkeys@orig@case@one
    \let\pgfkeys@try=\pgfkeys@orig@try
    \let\pgfkeys@unknown=\pgfkeys@orig@unknown
    \pgfkeysfilteringisactivefalse
}
```

执行 \pgfkeysfiltered{⟨*key-value-list*⟩} 导致

```
\ifpgfkeysfilteringisactive
    \pgfkeys@error{Sorry, nested calls to key filtering routines are not allowed.
    ↪  (reason: It is not possible to properly restore the previous filtering
    ↪  state after returning from the nested call)}%
\fi
\pgfkeysfilteringisactivetrue
\let\pgfkeys@case@one=\pgfkeys@case@one@filtered
\let\pgfkeys@try=\pgfkeys@try@filtered
\let\pgfkeys@unknown=\pgfkeys@unknown@filtered
\let\pgfkeysdefaultpath\pgfkeys@root%
\pgfkeys@parse⟨key-value-list⟩,\pgfkeys@mainstop%
\def\pgfkeysdefaultpath{⟨default path (string)⟩}%
\let\pgfkeys@case@one=\pgfkeys@orig@case@one
\let\pgfkeys@try=\pgfkeys@orig@try
\let\pgfkeys@unknown=\pgfkeys@orig@unknown
\pgfkeysfilteringisactivefalse
```

其主要处理是：

1. 如果 \ifpgfkeysfilteringisactive 的真值是 true, 则执行 \pgfkeys@error, 中断编译，给出错误信息，否则继续。也就是说，

   不能在命令 \pgfkeysfiltered 的内部套嵌使用 \pgfkeysfiltered;

   或者说，不能在命令 \pgfkeys@install@filter@and@invoke 的内部套嵌使用 \pgfkeys@install@filter@and@invoke;

除非能处理好 \ifpgfkeysfilteringisactive 的真值，使之不引起错误。

2. 设置 \pgfkeysfilteringisactivetrue
3. 重定义 \pgfkeys@case@one 等命令
4. let \pgfkeysdefaultpath = \pgfkeys@root, 即斜线 “/”
5. 执行 \pgfkeys@parse 解析键值列表
6. 重定义 \pgfkeysdefaultpath 的值，恢复为 ⟨*default path(string)*⟩
7. 恢复 \pgfkeys@case@one 等命令的原来的定义
8. 设置 \pgfkeysfilteringisactivefalse

\pgfkeysalsofrom{⟨\macro⟩}

参数 ⟨\macro⟩ 是保存 ⟨*key-value-list*⟩ 的宏。

```
\long\def\pgfkeysalsofrom#1{%
    \expandafter\pgfkeysalso\expandafter{#1}%
}
```

参考 \pgfkeysalso→P. 54.

\pgfkeysalsofilteredfrom{⟨\macro⟩}

参数 ⟨\macro⟩ 是保存 ⟨*key-value-list*⟩ 的宏。本命令对 ⟨*key-value-list*⟩ 做筛选。

```
\long\def\pgfkeysalsofilteredfrom#1{%
    \expandafter\pgfkeysalsofiltered\expandafter{#1}%
}
```

其中用到下面的命令：

\pgfkeysalsofiltered{⟨*key-value-list*⟩}

调用 \pgfkeys@install@filter@and@invoke→P. 85 做筛选。

```
\long\def\pgfkeysalsofiltered#1{%
    \pgfkeys@install@filter@and@invoke{\pgfkeysalso{#1}}%
}%
```

\pgfqkeysfiltered{⟨*default path set (string)*⟩}{⟨*key-value-list*⟩}

本命令对 ⟨*key-value-list*⟩ 做筛选，是 quick 版的筛选。

```
% #1: default path
% #2: key-value-pairs
\long\def\pgfqkeysfiltered#1{%
    \expandafter\pgfqkeysfiltered@@install\expandafter{\pgfkeysdefaultpath}{#1}%
}
```

参数 ⟨*default path set (string)*⟩ 是设置的默认前缀路径 (一串符号)。用到下面的：

\pgfqkeysfiltered@@install{⟨*old default path (string)*⟩}{⟨*default path set (string)*⟩}{⟨*key-value-list*⟩}

调用 \pgfkeys@install@filter@and@invoke→P. 85 做筛选。

```
% #1: old value of default path.
% #2: default path
% #3: key-value-list.
\long\def\pgfqkeysfiltered@@install#1#2#3{%
    \pgfkeys@install@filter@and@invoke{%
        \def\pgfkeysdefaultpath{#2/}\pgfkeys@parse#3,\pgfkeys@mainstop
        ↪  \def\pgfkeysdefaultpath{#1}%
    }%
```

```
}
```

\pgfkeysiffamilydefined{⟨*family name*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

参数 ⟨*family name*⟩ 是一串符号，或者是保存一串符号的宏。本命令的处理是：检查控制序列

\csname ifpgfk@⟨*family name*⟩/familyactive\endcsname

是否已定义; 如果已定义，则执行 ⟨*true code*⟩; 否则执行 ⟨*false code*⟩.

```
\long\def\pgfkeysiffamilydefined#1#2#3{\pgfkeys@ifcsname ifpgfk@#1/familyactive
↪  \endcsname#2\else#3\fi}
```

在执行

\pgfkeys@non@outer@newif →P. 95{pgfk@⟨*family name*⟩/familyactive}

后，控制序列 \csname ifpgfk@⟨*family name*⟩/familyactive\endcsname 就是已定义的。

而导致 \pgfkeys@non@outer@newif 被执行的是手柄 /.is family.

这个控制序列只可能等于 \iftrue 或 \iffalse. 若等于 \iftrue, 则标志着 ⟨*family name*⟩ 处于激活状态；若等于 \iffalse, 则标志着 ⟨*family name*⟩ 处于抑制状态。

\pgfkeysactivatefamily{⟨*family name*⟩}

参数 ⟨*family name*⟩ 是一串符号，或者是保存一串符号的宏。

```
% #1 maybe a macro.
\def\pgfkeysactivatefamily#1{%
    \pgfkeysiffamilydefined
        {#1}%
        {\csname pgfk@#1/familyactivetrue\endcsname}%
        {\pgfkeysvalueof{/errors/family unknown/.@cmd}{#1}\pgfeov}%
%\message{[ACTIVATING FAMILY #1]}%
}
```

本命令的处理是：

1. 检查命令 \csname ifpgfk@⟨*family name*⟩/familyactive\endcsname 是否已定义
   - 如果已定义，则执行

     ```
     \csname pgfk@⟨family name⟩/familyactivetrue\endcsname
     ```

     将控制序列

     \csname ifpgfk@⟨*family name*⟩/familyactive\endcsname

     let 为 \iftrue, 这样才算是激活 ⟨*family name*⟩ 这个 family, 参考 \pgfkeys@non@outer@newif →P. 95.
   - 如果未定义，则执行

     ```
     \pgfkeysvalueof{/errors/family unknown/.@cmd}{⟨family name⟩}\pgfeov
     ```

     也就是用保存在 /errors/family unknown/.@cmd 中的命令处理 ⟨*family name*⟩.

\pgfkeysdeactivatefamily{⟨*family name*⟩}

参数 ⟨*family name*⟩ 是一串符号，或者是保存一串符号的宏。

```
% #1 maybe a macro.
\def\pgfkeysdeactivatefamily#1{%
    \pgfkeysiffamilydefined
        {#1}%
        {\csname pgfk@#1/familyactivefalse\endcsname}%
        {\pgfkeysvalueof{/errors/family unknown/.@cmd}{#1}\pgfeov}%
%\message{[DEACTIVATING FAMILY #1]}%
}
```

本命令的处理是：

1. 检查命令 \csname ifpgfk@⟨*family name*⟩/familyactive\endcsname 是否已定义
   - 如果已定义，则执行

     \csname pgfk@⟨*family name*⟩/familyactivefalse\endcsname

     将控制序列

     \csname ifpgfk@⟨*family name*⟩/familyactive\endcsname

     let 为 \iffalse, 也就是抑制 ⟨*family name*⟩ 这个 family, 参考 \pgfkeys@non@outer@newif → P. 95.
   - 如果未定义，则执行

     \pgfkeysvalueof{/errors/family unknown/.@cmd}{⟨*family name*⟩}\pgfeov

     也就是用保存在 /errors/family unknown/.@cmd 中的命令处理 ⟨*family name*⟩.

\pgfkeysactivatefamilies{⟨*family names list*⟩}{⟨\macro *D*⟩}

本命令：

1. 保存当前的筛选标准和筛选器，
2. 改变筛选标准和筛选器，
3. 将 ⟨*family names list*⟩ 中列出的 family name 逐个激活，
4. 将抑制各个 family name 的命令保存到 ⟨\macro *D*⟩，
5. 恢复之前保存的筛选标准和筛选器。

```
% #1: a comma-separated list of fully qualified family names.
% #2: a command which will be filled with a deactivate-all command.
\def\pgfkeysactivatefamilies#1#2{%
    \pgfkeyssavekeyfilterstateto\pgfkeys@cur@state
    \expandafter\pgfkeysactivatefamilies@impl\expandafter{\pgfkeys@cur@state}{#1}
    ↪  {#2}%
}
```

\pgfkeysactivatefamilies@impl{⟨\macro *S*⟩}{⟨*family name list*⟩}{⟨\macro *D*⟩}

本命令：

1. 改变筛选标准和筛选器，
2. 将 ⟨*family names list*⟩ 中列出的 family name 逐个激活，
3. 将抑制各个 family name 的命令保存到 ⟨\macro *D*⟩，
4. 恢复之前保存的筛选标准和筛选器。

```
% #1: commands needed to restore the old filtering state
% #2: family name list
% #3: macro name for de-activate command
\def\pgfkeysactivatefamilies@impl#1#2#3{%
    \pgfkeysinstallkeyfilter{/pgf/key filters/false}{}%
    \let#3=\pgfkeys@empty%
    \def\pgfkeys@filtered@handler{\pgfkeys@family@activate@handler{#3}}%
    \pgfkeysalsofiltered{#2}%
    #1%
}
```

\pgfkeys@family@activate@handler{⟨\macro *D*⟩}

本命令：

1. 执行 \pgfkeysactivatefamily{\pgfkeyscurrentkey}, 即激活这个 family name,
2. 定义 (重定义) 宏 ⟨\macro *D*⟩, 将抑制 ⟨*exp* 展开的 *\pgfkeyscurrentkey*⟩ 这个 family name 的命令

   \pgfkeysdeactivatefamily{⟨*exp* 展开的 *\pgfkeyscurrentkey*⟩}

添加到这个宏保存中。

```
\def\pgfkeys@family@activate@handler#1{%
    \pgfkeysactivatefamily{\pgfkeyscurrentkey}%
    % produce
    %   old list '\pgfkeysdeactivatefamily{' current key '}'
    \pgfkeys@tmptoks=\expandafter\expandafter\expandafter{\expandafter#1
    ↪  \expandafter\pgfkeysdeactivatefamily\expandafter{\pgfkeyscurrentkey}}%
    \edef#1{\the\pgfkeys@tmptoks}%
}
```

\pgfkeysisfamilyactive{⟨*family name*⟩}

参数 ⟨*family name*⟩ 可以是一串符号，或者是保存一串符号的宏。

```
% #1 the family name. Maybe a macro.
\def\pgfkeysisfamilyactive#1{%
    \pgfkeysiffamilydefined{#1}{%
        \expandafter\let\expandafter\ifpgfkeysfiltercontinue\csname
        ↪  ifpgfk@#1/familyactive\endcsname
    }{%
        \pgfkeysvalueof{/errors/family unknown/.@cmd}{#1}\pgfeov%
        \expandafter\expandafter\expandafter\let\csname ifpgfkeysfiltercontinue
        ↪  \endcsname\csname iffalse\endcsname
    }%
}%
```

本命令检查命令 \csname ifpgfk@⟨*family name*⟩/familyactive\endcsname 是否已定义：

- 如果已定义，则执行

  ```
  \expandafter\let\expandafter\ifpgfkeysfiltercontinue\csname ifpgfk@
  ↪  ⟨family name⟩/familyactive\endcsname
  ```

  将 \ifpgfkeysfiltercontinue 这个条件 let 为 \iftrue.

- 如果未定义，则执行

  ```
  \pgfkeysvalueof{/errors/family unknown/.@cmd}{#1}\pgfeov%
  \expandafter\expandafter\expandafter\let\csname ifpgfkeysfiltercontinue
  ↪  \endcsname\csname iffalse\endcsname
  ```

  将 \ifpgfkeysfiltercontinue 这个条件 let 为 \iffalse.

参考：

- 命令 \pgfkeys@non@outer@newif →P. 95{pgfk@⟨*family name*⟩/familyactive} 定义
  \csname ifpgfk@⟨*family name*⟩/familyactive\endcsname
- 命令 \pgfkeysactivatefamily →P. 87{⟨*family name*⟩} 激活 ⟨*family name*⟩
- 手柄 /.is family

\pgfkeyssetfamily{⟨*full key*⟩}{⟨*family name*⟩}

参数 ⟨*family name*⟩ 可以是代表 family name 的完整键，或者是保存这种键的宏。

这个命令将键 ⟨*full key*⟩ 添加到 ⟨*family name*⟩ 这个 family 中。实际上是使得控制序列

\csname pgfk@⟨*full key*⟩/family\endcsname

保存 ⟨*family name*⟩. 可见本命令只能让 ⟨*full key*⟩ 属于一个 family.

```
% Equivalent to \pgfkeys{#1/.belongs to family=#2}
\def\pgfkeyssetfamily#1#2{%
    \pgfkeysiffamilydefined{#2}{%
        \pgfkeyssetvalue{#1/family}{#2}%
    }{%
```

```
        \pgfkeysalso{/errors/family unknown=#2}%
    }%
}%
```

本命令检查 \csname ifpgfk@⟨*family name*⟩/familyactive\endcsname 是否已定义，

- 如果已定义，则执行

  ```
  \pgfkeyssetvalue{⟨full key⟩/family}{⟨family name⟩}
  ```

  从而定义命令

  \csname pgfk@⟨*full key*⟩/family\endcsname

  此命令保存 ⟨*family name*⟩ (一个完整的键), 也就是把 ⟨*full key*⟩ 与 ⟨*family name*⟩ 这个 family name 相关联。
- 如果未定义，则执行

  ```
  \pgfkeysalso{/errors/family unknown=#2}
  ```

本命令被手柄 /.belongs to family 利用。

在执行

\pgfkeys@non@outer@newif →P.95{pgfk@⟨*family name*⟩/familyactive}

后，控制序列 \csname ifpgfk@⟨*family name*⟩/familyactive\endcsname 就是已定义的。

而导致 \pgfkeys@non@outer@newif 被执行的是手柄 /.is family.

\pgfkeysgetfamily{⟨*full key*⟩}{⟨*macro*⟩}

参数 ⟨*full key*⟩ 可以是完整的键，或者是保存完整键的宏。

本命令检查 ⟨*full key*⟩ 是否属于某个 family, 如果是，就将所属的 family name 保存到宏 ⟨*macro*⟩ 中，并设置真值 \pgfkeyssuccesstrue; 否则设置真值 \pgfkeyssuccessfalse.

```
\def\pgfkeysgetfamily#1#2{%
    \pgfkeysifdefined{#1/family}{\pgfkeysgetvalue{#1/family}{#2}
    ↪  \pgfkeyssuccesstrue}{\pgfkeyssuccessfalse}%
}
```

本命令检查命令 \csname pgfk@⟨*full key*⟩/family\endcsname 是否已定义，也就是检查 ⟨*full key*⟩ 是否已经与某个 family name 相关联 (属于这个 family)，

- 如果已定义，则执行

  ```
  \pgfkeysgetvalue{⟨full key⟩/family}{⟨\macro⟩}
  ```

  并设置真值 \pgfkeyssuccesstrue.
- 如果未定义，则设置真值 \pgfkeyssuccessfalse

定义命令 \csname pgfk@⟨*full key*⟩/family\endcsname 的是 \pgfkeyssetfamily →P.89.

\pgfkeysinterruptkeyfilter

```
\def\pgfkeysinterruptkeyfilter{%
    \ifpgfkeysfilteringisactive
        \let\pgfkeys@case@one=\pgfkeys@orig@case@one
        \let\pgfkeys@try=\pgfkeys@orig@try
        \let\pgfkeys@unknown=\pgfkeys@orig@unknown
    \fi
}
```

这个命令把筛选命令换成通常的处理命令。当需要在筛选过程中暂时中断筛选机制，以通常的处理方式来处理键值对时，可以使用这个命令，之后可以使用命令 \endpgfkeysinterruptkeyfilter 再引入筛选命令，继续筛选。

\endpgfkeysinterruptkeyfilter

```
\def\endpgfkeysinterruptkeyfilter{%
    \ifpgfkeysfilteringisactive
        \let\pgfkeys@case@one=\pgfkeys@case@one@filtered
        \let\pgfkeys@try=\pgfkeys@try@filtered
        \let\pgfkeys@unknown=\pgfkeys@unknown@filtered
    \fi
}
```

\pgfkeysactivatefamiliesandfilteroptions{⟨*family name list*⟩}{⟨*key-value-list*⟩}

本命令是 \pgfkeysactivatefamilies→P. 88, \pgfkeysfiltered→P. 84 的结合。

```
% #1: comma separated family list
% #2: key-value pairs
%
% @see \pgfkeysactivatefamiliesandfilteroptions
\def\pgfkeysactivatefamiliesandfilteroptions#1#2{%
    \pgfkeysactivatefamilies{#1}{\pgfkeys@family@deactivation}%
    \pgfkeysfiltered{#2}%
    \pgfkeys@family@deactivation
}
```

\pgfqkeysactivatefamiliesandfilteroptions{⟨*family name list*⟩}{⟨*default path set (string)*⟩}{⟨*key-value-list*⟩}

这是 quick 版的 \pgfkeysactivatefamiliesandfilteroptions

```
% #1: comma separated family list
% #2: default path
% #3: key-value pairs
\def\pgfqkeysactivatefamiliesandfilteroptions#1#2#3{%
    \pgfkeysactivatefamilies{#1}{\pgfkeys@family@deactivation}%
    \pgfqkeysfiltered{#2}{#3}%
    \pgfkeys@family@deactivation
}
```

\pgfkeysactivatesinglefamilyandfilteroptions{⟨*family name*⟩}{⟨*key-value-list*⟩}

这是 \pgfkeysactivatefamily→P. 87, \pgfkeysfiltered→P. 84, \pgfkeysdeactivatefamily→P. 87 的结合。

```
% #1: family (maybe a macro)
% #2: key-value pairs
\def\pgfkeysactivatesinglefamilyandfilteroptions#1#2{%
    \pgfkeysactivatefamily{#1}%
    \pgfkeysfiltered{#2}%
    \pgfkeysdeactivatefamily{#1}%
}
```

\pgfqkeysactivatesinglefamilyandfilteroptions{⟨*family name*⟩}{⟨*default path set (string)*⟩}{⟨*key-value-list*⟩}

这是 quick 版的 \pgfkeysactivatesinglefamilyandfilteroptions

```
% #1: family (maybe a macro)
% #2: default path
% #3: key-value pairs
\def\pgfqkeysactivatesinglefamilyandfilteroptions#1#2#3{%
```

```
    \pgfkeysactivatefamily{#1}%
    \pgfqkeysfiltered{#2}{#3}%
    \pgfkeysdeactivatefamily{#1}%
}
```

\pgfkeys@cur@is@descendant@of@errors

此命令的作用是，检查当前的 \pgfkeyscurrentkey 所保存的键，

- 如果它保存的键以 /errors 开头，或者它是空的，就设置真值 \pgfkeysfiltercontinuetrue
- 否则，就设置真值 \pgfkeysfiltercontinuefalse

\pgfkeys@case@one@filtered

在 \pgfkeys@parse 的处理过程中使用这个命令。使用这个命令时，

- \pgfkeyscurrentkey→P. 49
- \pgfkeyscurrentkeyRAW→P. 49
- \pgfkeyscurrentname→P. 51
- \pgfkeyscurrentpath→P. 51
- \pgfkeyscurrentvalue→P. 50

都是已定义的。

此命令的处理是：

1. 执行 \pgfkeys@cur@is@descendant@of@errors 检查当前的 \pgfkeyscurrentkey 的值
2. 检查 \ifpgfkeysfiltercontinue 的真值
   - 如果有真值 true, 则执行 \pgfkeys@orig@case@one
   - 如果有真值 false, 则
     (a) 设置真值 \pgfkeysfiltercontinuetrue
     (b) 执行 \pgfkeysifdefined→P. 43 检查命令

     pgfk@\pgfkeyscurrentkey/.@cmd

     是否已定义
     – 如果已定义，则
       i. 定义 \def\pgfkeyscasenumber{1}
       ii. 执行 \pgfkeys@key@predicate→P. 83
       iii. 检查 \ifpgfkeysfiltercontinue 的真值
         ▶ 如果有真值 true, 则执行

```
\pgfkeysgetvalue{\pgfkeyscurrentkey/.@cmd}{\pgfkeys@code}%
\expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
```

         ▶ 如果有真值 false, 则执行 \pgfkeys@filtered@handler→P. 84

     – 如果未定义，则
       i. 执行 \pgfkeysifdefined 检查命令 pgfk@\pgfkeyscurrentkey 是否已定义
         ▶ 如果已定义，则
           ⟨1.⟩ 定义 \def\pgfkeyscasenumber{2}
           ⟨2.⟩ 执行 \pgfkeys@key@predicate→P. 83
           ⟨3.⟩ 检查 \ifpgfkeysfiltercontinue 的真值：
             ▷ 如果有真值 true, 则执行 \pgfkeys@case@two@extern→P. 51
             ▷ 如果有真值 false, 则执行 \pgfkeys@filtered@handler→P. 84

         ▶ 如果未定义，则

⟨1.⟩ 执行 \pgfkeys@split@path→P. 51

⟨2.⟩ 执行 \pgfkeysifdefined 检查手柄命令

pgfk@/handlers/\pgfkeyscurrentname/.@cmd

是否已定义

▷ 如果已定义，执行 \pgfkeys@ifexecutehandler→P. 77，
此时这个命令的两个参数：第一个参数是

```
\def\pgfkeyscasenumber{3}%
\pgfkeys@key@predicate%
\ifpgfkeysfiltercontinue
%\message{PROCESSING KEY \pgfkeyscurrentkey!}%
    \pgfkeysgetvalue{/handlers/\pgfkeyscurrentname/.@cmd}
    ↪  {\pgfkeys@code}%
    \expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
\else
%\message{FILTERED OUT  KEY \pgfkeyscurrentkey!}%
    \pgfkeys@filtered@handler%
\fi
```

第二个参数是 \pgfkeys@unknown

▷ 如果未定义，执行 \pgfkeys@unknown

以上处理的要点是：

- 若有 pgfk@\pgfkeyscurrentkey/.@cmd, 则宏 \pgfkeyscasenumber 保存 1; 否则
- 若有 pgfk@\pgfkeyscurrentkey, 则宏 \pgfkeyscasenumber 保存 2; 否则
- 若有 pgfk@/handlers/\pgfkeyscurrentname/.@cmd, 则宏 \pgfkeyscasenumber 保存 3;

只要这 3 个控制序列有一个是存在的，就会：

1. 执行 \pgfkeys@key@predicate→P. 83
2. 然后检查 \ifpgfkeysfiltercontinue 的真值，
   - 如果它的真值是 true, 执行某些操作 (执行存在的控制序列)；
   - 如果它的真值是 false, 执行 \pgfkeys@filtered@handler→P. 84.

   所以 \pgfkeys@key@predicate 应该能设置 \ifpgfkeysfiltercontinue 的真值。

如果这 3 个控制序列都不存在,就执行 \pgfkeys@unknown→P. 51, 此时它等于 \pgfkeys@unknown@filtered→P. 96.

\pgfkeys@try@filtered

此命令的处理是：

1. 检查 \ifpgfkeysfiltercontinue 的真值
   - 如果有真值 true, 则执行 \pgfkeys@orig@try
   - 如果有真值 fase, 则
     (a) 设置 \pgfkeysfiltercontinuetrue
     (b) 定义 \edef\pgfkeyscurrentkey{\pgfkeyscurrentpath}
     (c) 检查 \pgfkeyscurrentvalue 与 \pgfkeysnovalue@text 的值是否相同，
     – 如果相同，就是没有 ⟨=⟨*value*⟩⟩ 这一部分，则执行 \pgfkeysifdefined 检查命令

     pgfk@\pgfkeyscurrentpath/.@def

     是否已定义

     * 如果已定义，则执行

     ```
     \pgfkeysgetvalue{\pgfkeyscurrentpath/.@def}{\pgfkeyscurrentvalue}
     ```

     获取默认值

* 如果未定义，则什么也不做

– 如果不同，则什么也不做

(d) 执行 \pgfkeysifdefined 检查命令 pgfk@\pgfkeyscurrentpath/.@cmd 是否已定义

– 如果已定义，则

i. 定义 \def\pgfkeyscasenumber{1}

ii. 执行 \pgfkeys@key@predicate →P. 83

iii. 检查 \ifpgfkeysfiltercontinue 的真值

▶ 如果为 true, 则执行

```
\pgfkeysgetvalue{\pgfkeyscurrentkey/.@cmd}{\pgfkeys@code}%
\expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov%
```

▶ 如果为 false, 则执行 \pgfkeys@filtered@handler →P. 84

iv. 设置 \pgfkeyssuccesstrue

– 如果未定义，则执行 \pgfkeysifdefined 检查命令 pgfk@\pgfkeyscurrentpath 是否已定义

* 如果已定义，则

⟨1.⟩ 定义 \def\pgfkeyscasenumber{2}

⟨2.⟩ 执行 \pgfkeys@key@predicate

⟨3.⟩ 检查 \ifpgfkeysfiltercontinue 的真值

▷ 如果为 true, 则检查 \pgfkeyscurrentvalue 与 \pgfkeysnovalue@text 的定义是否相同，

◇ 如果相同，则执行 \pgfkeysvalueof{\pgfkeyscurrentpath}

◇ 如果不同,则执行 \pgfkeyslet{\pgfkeyscurrentpath}\pgfkeyscurrentvalue

▷ 如果为 false, 则执行 \pgfkeys@filtered@handler →P. 84

⟨4.⟩ 设置 \pgfkeyssuccesstrue

* 如果未定义，则

⟨1.⟩ 执行 \pgfkeys@split@path →P. 51

⟨2.⟩ 执行 \pgfkeysifdefined 检查手柄命令

pgfk@/handlers/\pgfkeyscurrentname/.@cmd

是否已定义

◇ 如果已定义，则执行 \pgfkeys@ifexecutehandler →P. 77, 此命令的两个参数：第一个是：

```
\def\pgfkeyscasenumber{3}%
\pgfkeys@key@predicate%
\ifpgfkeysfiltercontinue
    \pgfkeysgetvalue{/handlers/\pgfkeyscurrentname/.@cmd}{
    ↪  \pgfkeys@code}%
    \expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
\else
    \pgfkeys@filtered@handler%
\fi
\pgfkeyssuccesstrue
```

第二个是：\pgfkeyssuccessfalse

◇ 如果未定义，则设置 \pgfkeyssuccessfalse

\pgfkeysevalkeyfilterwith{⟨*full key*⟩⟨=⟨*value*⟩⟩}

```
\long\def\pgfkeysevalkeyfilterwith#1{%
    \pgfkeys@eval@key@filter@subroutine@unpack#1=\pgfkeysnovalue=\pgfkeys@stop
}%
```

\pgfkeys@eval@key@filter@subroutine@unpack{⟨*full key*⟩⟨=⟨*value*⟩⟩}

此命令的定义格式是：

```
\long\def\pgfkeys@eval@key@filter@subroutine@unpack#1=#2=#3\pgfkeys@stop{%
%...
}
```

此命令的处理是：

1. 执行 \pgfkeys@spdef →P. 47\pgfkeyspred@TMP{⟨*full key*⟩}, 定义宏 \pgfkeyspred@TMP
2. 重定义 \edef\pgfkeyspred@TMP{\pgfkeyspred@TMP}
3. 执行 \pgfkeys@spdef\pgfkeyspred@TMPB{#2}, 定义宏 \pgfkeyspred@TMPB
4. 检查 \pgfkeyspred@TMPB 与 \pgfkeysnovalue@text 的定义是否相同，
   - 如果相同，就是没有 ⟨=⟨*value*⟩⟩ 这一部分，则检查命令 pgfk@\pgfkeyspred@TMP/.@def 是否已定义，
     - 如果已定义，则执行

       ```
       \pgfkeysgetvalue{\pgfkeyspred@TMP/.@def}{\pgfkeyspred@TMPB}
       ```

       获取默认值
     - 如果未定义，则什么也不做
   - 如果不同，就是有 ⟨=⟨*value*⟩⟩ 这一部分，则什么也不做
5. 检查 \pgfkeyspred@TMPB 与 \pgfkeysvaluerequired 的定义是否相同，
   - 如果相同，则执行

     ```
     \pgfkeysvalueof{/errors/value required/.@cmd}\pgfkeyspred@TMP
     ↪  \pgfkeyspred@TMPB\pgfeov
     ```

   - 如果不同，则检查命令 pgfk@\pgfkeyspred@TMP/.@cmd 是否已定义，
     - 如果已定义，则执行

       ```
       \pgfkeysgetvalue{\pgfkeyspred@TMP/.@cmd}{\pgfkeys@code}%
       \expandafter\pgfkeys@code\pgfkeyspred@TMPB\pgfeov
       ```

     - 如果未定义，则执行

       ```
       \pgfkeysvalueof{/errors/no such key filter/.@cmd}\pgfkeyspred@TMP
       ↪  \pgfkeyspred@TMPB\pgfeov
       ```

可见，在有意义的情况下，键 ⟨*full key*⟩ 应当是能处理参数的函数，并且应当写出 =⟨*value*⟩, 或者为这个键定义默认值。本命令调用函数 pgfk@⟨*full key*⟩/.@cmd 处理键值 ⟨*value*⟩。

\pgfkeys@non@outer@newif{⟨*string*⟩}

```
\def\pgfkeys@non@outer@newif@#1#2{%
    \expandafter\edef\csname #2true\endcsname{\noexpand\let\noexpand#1=
    ↪  \noexpand\iftrue}%
    \expandafter\edef\csname #2false\endcsname{\noexpand\let\noexpand#1=
    ↪  \noexpand\iffalse}%
    \csname #2false\endcsname
}%

% For latex and context, this here has the same effect as a \newif
% applied to 'if#1'. For plain tex, it has also the same effect, but
```

```
% it is not an \outer macro as the plain-tex \newif.
\def\pgfkeys@non@outer@newif#1{%
    \expandafter\pgfkeys@non@outer@newif@\csname if#1\endcsname{#1}%
}
```

本命令定义 3 个控制序列：

- \csname ⟨*string*⟩true\endcsname, 这个控制序列展开为

  ```
  \expandafter\let\csname if⟨string⟩\endcsname=\iftrue
  ```

  展开效果是使得 \csname if⟨*string*⟩\endcsname 等于 \iftrue.
  参考命令 \pgfkeysactivatefamily→P. 87{⟨*family name*⟩}.
- \csname ⟨*string*⟩false\endcsname, 这个控制序列展开为

  ```
  \expandafter\let\csname if⟨string⟩\endcsname=\iffalse
  ```

  展开效果是使得 \csname if⟨*string*⟩\endcsname 等于 \iffalse.
  参考命令 \pgfkeysactivatefamily→P. 87{⟨*family name*⟩}.
- \csname if⟨*string*⟩\endcsname, 由于本命令执行 \csname ⟨*string*⟩false\endcsname, 所以本命令将 \csname if⟨*string*⟩\endcsname 初始化为 \iffalse.

手柄 /.is family 会导致 \pgfkeys@non@outer@newif 被执行.

\pgfkeys@unknown@filtered

```
\def\pgfkeys@unknown@filtered{%
    % CASE ZERO: an unknown option.
    \def\pgfkeyscasenumber{0}%
    \pgfkeys@key@predicate%
    \ifpgfkeysfiltercontinue
%\message{PROCESSING  KEY \pgfkeyscurrentkey!}%
        % start normal 'unknown' handlers:
        \pgfkeys@orig@unknown% 等于 \pgfkeys@unknown→P. 51
    \else
%\message{FILTERED OUT  KEY \pgfkeyscurrentkey!}%
        \pgfkeys@filtered@handler%
    \fi
}
```

命令 \pgfkeys@install@filter@and@invoke→P. 85 会

```
\let\pgfkeys@unknown=\pgfkeys@unknown@filtered
```

\ifpgfkeysfiltercontinue

这个 TEX-if 通常意义是 (参考 \pgfkeys@case@one@filtered→P. 92)：

- 如果这个条件的真值是 true, 就意味着当前键值对符合筛选标准，言外之意是，下一步将执行这个键值对；
- 如果这个条件的真值是 false, 就意味着当前键值对不符合筛选标准，言外之意是，下一步将对这个键值对执行一些特别的操作，例如执行 \pgfkeys@filtered@handler→P. 84.

能影响这个 TEX-if 值的命令和键较多，例如：

- \pgfkeysisfamilyactive→P. 89
- \pgfkeys@cur@is@descendant@of@errors→P. 92
- /pgf/key filters/active families→P. 102
- /pgf/key filters/active families and known→P. 104
- /pgf/key filters/active families or descendants of→P. 105

- /pgf/key filters/is descendant of → P. 105
- /pgf/key filters/equals → P. 106
- /pgf/key filters/not → P. 107
- /pgf/key filters/and → P. 108
- /pgf/key filters/or → P. 108
- /pgf/key filters/true → P. 108
- /pgf/key filters/false → P. 108
- /pgf/key filters/defined → P. 109
- \pgfkeys@case@one@filtered → P. 92
- \pgfkeys@try@filtered → P. 93
- \pgfkeys@key@predicate → P. 83

\ifpgfkeyssuccess

这个 TEX-if 在文件《pgfkeys.code.tex》中被声明：

```
\newif\ifpgfkeyssuccess
```

在键筛选过程中，能影响这个 TEX-if 的值的命令是：

- \pgfkeysgetfamily，当命令 pgfk@⟨*family name key*⟩/family 有定义的情况下设置 \pgfkeyssuccesstrue
- \pgfkeys@try@filtered, 在以下命令有定义的情况下设置 \pgfkeyssuccesstrue
  - pgfk@\pgfkeyscurrentpath/.@cmd
  - pgfk@\pgfkeyscurrentpath
  - pgfk@/handlers/\pgfkeyscurrentname/.@cmd

其他情况下都是 \pgfkeyssuccessfalse.

\pgfkeyscasenumber

这个宏在以下情况下被定义

- 在执行命令 \pgfkeys@case@one@filtered → P. 92, \pgfkeys@try@filtered → P. 93 时，
  - 当命令 pgfk@\pgfkeyscurrentpath/.@cmd 有定义时，定义

    ```
    \def\pgfkeyscasenumber{1}
    ```

  - 当命令 pgfk@\pgfkeyscurrentpath 有定义时，定义

    ```
    \def\pgfkeyscasenumber{2}
    ```

  - 当命令 pgfk@/handlers/\pgfkeyscurrentname/.@cmd 有定义时，定义

    ```
    \def\pgfkeyscasenumber{3}
    ```

- 在执行命令 \pgfkeys@unknown@filtered → P. 96 时，定义

  ```
  \def\pgfkeyscasenumber{0}
  ```

2.8.3 手柄

2.8.3.1 针对 family 的手柄

/errors/family unknown={⟨*family name*⟩}

```
\pgfkeys{%
    /errors/family unknown/.code=\pgfkeys@error{%
```

```
        Sorry, I do not know family '#1' and can't work with any assoicated family
        ↪ handling. Perhaps you misspelled it?},
}
```

Key handler ⟨*full key for family name*⟩**/.is family**

这个手柄将 ⟨*full key for family name*⟩ 声明为一个 family name.

```
\pgfkeys{%
    /handlers/.is family/.append code={%
        %\newif is an \outer macro in plain tex, so this here is not portable:
        %\expandafter\newif\csname if\pgfkeyscurrentpath/familyactive\endcsname
        \edef\pgfkeyspred@TMP{pgfk@\pgfkeyscurrentpath/familyactive}%
        \expandafter\pgfkeys@non@outer@newif\expandafter{\pgfkeyspred@TMP}%
        \edef\pgfkeyspred@TMP{\pgfkeyscurrentpath/.belongs to family=
        ↪ \pgfkeyscurrentpath}%
        \expandafter\pgfkeysalso\expandafter{\pgfkeyspred@TMP}%
    },%
}
```

此定义的展开实际上是对命令

- `\csname pgfk@/handlers/.is family/.@cmd\endcsname`

进行重定义，即增加其定义内容，这实际上是定义了手柄 `/.is family`.

当执行 `\pgfkeys{`⟨*full key for family name*⟩`/.is family}` 时，导致

```
\pgfkeys@non@outer@newif{pgfk@⟨full key for family name⟩/familyactive}%
\pgfkeysalso{⟨full key for family name⟩/.belongs to family=⟨full key for family name⟩}%
```

又会导致：

1. 按照 `\pgfkeys@non@outer@newif` →P. 95 的定义，定义 3 个控制序列
   - `\csname pgfk@`⟨*full key for family name*⟩`/familyactivetrue\endcsname`
   - `\csname pgfk@`⟨*full key for family name*⟩`/familyactivefalse\endcsname`
   - `\csname ifpgfk@`⟨*full key for family name*⟩`/familyactive\endcsname`
2. 按照 `/.belongs to family` 的定义，又导致

   ```
   \pgfkeyssetfamily{⟨full key for family name⟩}{⟨full key for family name⟩}
   ```

执行 `\pgfkeys{`⟨*full key for family name*⟩`/.is family}` 的作用是把 ⟨*full key for family name*⟩ 声明为一个 family name, 并且还把 ⟨*full key for family name*⟩ 添加到这个 family 中。

Key handler ⟨*family name*⟩**/.activate family**

```
\pgfkeys{%
    /handlers/.activate family/.code=\pgfkeysactivatefamily{\pgfkeyscurrentpath},
}
```

此定义的展开会定义两个控制序列：

- `\csname pgfk@/handlers/.activate family/.@cmd\endcsname`, 这个命令定义手柄 `/.activate family`
- `\csname pgfk@/handlers/.activate family/.@body\endcsname`, 只是保存代码

当执行 `\pgfkeys{`⟨*family name*⟩`/.activate family}` 时，导致

```
\pgfkeysactivatefamily{⟨family name⟩}
```

Key handler ⟨*family name*⟩**/.deactivate family**

这个手柄抑制 ⟨*family name*⟩ 这个 family.

```
\pgfkeys{%
    /handlers/.deactivate family/.code=\pgfkeysdeactivatefamily{
    ↪  \pgfkeyscurrentpath},
}
```

此定义的展开会定义两个控制序列：

- \csname pgfk@/handlers/.deactivate family/.@cmd\endcsname, 这个命令定义手柄 /.deactivate family
- \csname pgfk@/handlers/.deactivate family/.@body\endcsname, 只是保存代码

当执行 \pgfkeys{⟨*family name*⟩/.deactivate family} 时，导致

```
\pgfkeysdeactivatefamily{⟨family name⟩}
```

Key handler ⟨*full key*⟩/.belongs to family=⟨*family name*⟩

这个手柄将 ⟨*full key*⟩ 添加到 ⟨*family name*⟩ 这个 family 中。

```
\pgfkeys{%
    /handlers/.belongs to family/.code={\pgfkeyssetfamily{\pgfkeyscurrentpath}{#1
    ↪  }},
}
```

此定义的展开会定义两个控制序列：

- \csname pgfk@/handlers/.belongs to family/.@cmd\endcsname, 能处理参数的命令，这个命令定义手柄 /.belongs to family
- \csname pgfk@/handlers/.belongs to family/.@body\endcsname, 只是保存代码

当执行 \pgfkeys{⟨*full key*⟩/.belongs to family={⟨*family name*⟩}} 时，导致

```
\pgfkeyssetfamily{⟨full key⟩}{⟨family name⟩}
```

2.8.3.2 引入过滤器的手柄

Key handler ⟨*full key*⟩/.install key filter={⟨*argu list*⟩}

```
\pgfkeys{%
    /handlers/.install key filter/.code={%
        \pgfkeysinstallkeyfilter{\pgfkeyscurrentpath}{#1}%
    },%
}
```

此定义的展开会定义两个控制序列：

- \csname pgfk@/handlers/.install key filter/.@cmd\endcsname, 能处理参数的命令，这个命令定义手柄 /.lastretry
- \csname pgfk@/handlers/.install key filter/.@body\endcsname, 只是保存代码

当执行 \pgfkeys{⟨*full key*⟩/.install key filter={⟨*argu list*⟩}} 时，导致

```
\pgfkeysinstallkeyfilter{⟨full key⟩}{{⟨argu list⟩}}
```

Key handler ⟨*full key*⟩/.install key filter handler={⟨*argu list*⟩}

```
\pgfkeys{%
    /handlers/.install key filter handler/.code={%
        \pgfkeysinstallkeyfilterhandler{\pgfkeyscurrentpath}{#1}%
    },%
}
```

此定义的展开会定义两个控制序列：

- \csname pgfk@/handlers/.install key filter handler/.@cmd\endcsname, 能处理参数的命令，这个命令定义手柄 /.lastretry

- \csname pgfk@/handlers/.install key filter handler/.@body\endcsname, 只是保存代码

当执行 \pgfkeys{⟨*full key*⟩/.install key filter handler={⟨*argu list*⟩}} 时，导致

```
\pgfkeysinstallkeyfilterhandler{⟨full key⟩}{{⟨argu list⟩}}
```

2.8.3.3 其他手柄

Key handler ⟨*full key*⟩/.lastretry=⟨*value*⟩

```
\pgfkeys{%
    /handlers/.lastretry/.code={%
        \ifpgfkeyssuccess\else
            \pgfkeys@try
            \ifpgfkeyssuccess\else
                \pgfkeys@split@path%
                \pgfkeys@unknown
            \fi
        \fi
    },
}
```

此定义的展开会定义两个控制序列：

- \csname pgfk@/handlers/.lastretry/.@cmd\endcsname, 能处理参数的命令，这个命令定义手柄 /.lastretry
- \csname pgfk@/handlers/.lastretry/.@body\endcsname, 只是保存代码

当执行 \pgfkeys{⟨*full key*⟩/.lastretry=⟨*value*⟩} 时，导致：

检查 \ifpgfkeyssuccess →P. 75 的真值，

- 若 \pgfkeyssuccesstrue, 则什么也不做
- 若 \pgfkeyssuccessfalse, 则
  1. 执行 \pgfkeys@try →P. 75
  2. 再次检查 \ifpgfkeyssuccess →P. 75 的真值，
     - 若 \pgfkeyssuccesstrue, 则什么也不做；
     - 若 \pgfkeyssuccessfalse, 则执行
       (a) \pgfkeys@split@path →P. 51
       (b) \pgfkeys@unknown →P. 51

2.8.4 Filter: 处理不符合筛选标准的键值对

/pgf/key filter handlers/append filtered to=⟨*\macro*⟩ (no default)

参数 ⟨*\macro*⟩ 应该是已定义的宏。如果 \pgfkeys@filtered@handler →P. 84 调用这个过滤器，那么不符合筛选标准的键值对“⟨*key*⟩={⟨*value*⟩}”将被添加到 ⟨*\macro*⟩ 中保存起来。

```
\pgfkeys{%
    /pgf/key filter handlers/append filtered to/.code={%
        % Produce
        %  <orig key> '={' <value> '}'
        % where both, the key and the value are expanded just ONCE:
        \pgfkeys@tmptoks=\expandafter\expandafter\expandafter{
        ↪  \expandafter\pgfkeyscurrentkeyRAW\expandafter=\expandafter{
        ↪  \pgfkeyscurrentvalue}}%
        \ifx#1\pgfkeys@empty
        \else
```

```
            % Produce <old list> ','  <orig key> '={' <value> '}'
            \pgfkeys@tmptoks=\expandafter\expandafter\expandafter{\expandafter#1
            ↪  \expandafter,\the\pgfkeys@tmptoks}%
        \fi
        \edef#1{\the\pgfkeys@tmptoks}%
    },%
}
```

这个定义的展开会定义两个控制序列

- \csname pgfk@/pgf/key filter handlers/append filtered to/.@cmd\endcsname, 这个函数的参数格式是 #1\pgfeov.
- \csname pgfk@/pgf/key filter handlers/append filtered to/.@body\endcsname, 这个控制序列只是保存上面的代码，

这个键对应的 /.@cmd 命令的主要作用是重定义宏 ⟨\macro⟩, 把当前正在处理的键值对“⟨key⟩={⟨value⟩}”添加到这个宏中保存起来。这个键通常配合其他手柄使用。如果直接执行

\pgfkeys{/pgf/key filter handlers/append filtered to=⟨\macro⟩}

例如：

macro:->A,/pgf/key filter handlers/append filtered to={\aaa }

```
\def\aaa{A}
\pgfkeys{/pgf/key filter handlers/append filtered to=\aaa}
\meaning\aaa
```

此时如果展开 \aaa 就会导致无限循环，出现错误。这个键一般这样用：

```
\def⟨\macro⟩{}
然后
\pgfkeys{/pgf/key filter handlers/append filtered to/.install key filter handler=
↪  ⟨\macro⟩}
```

/pgf/key filter handlers/ignore

如果 \pgfkeys@filtered@handler →P. 84 调用这个过滤器，那么不符合筛选标准的键值对将被忽略。

```
\pgfkeys{%
    /pgf/key filter handlers/ignore/.code={},
    /pgf/key filter handlers/ignore/.install key filter handler,
}
```

这个定义的展开是

```
\pgfkeysinstallkeyfilterhandler{/pgf/key filter handlers/ignore}{}
```

其主要作用把宏 \pgfkeys@filtered@handler 定义成空的。

/pgf/key filter handlers/log

如果 \pgfkeys@filtered@handler →P. 84 调用这个过滤器，那么不符合筛选标准的键值对将被记录到.log 文件中。

```
\pgfkeys{%
    /pgf/key filter handlers/log/.code={%
        \pgf@typeout{LOG: the option '\pgfkeyscurrentkey' (was originally '
        ↪  \pgfkeyscurrentkeyRAW') (case \pgfkeyscasenumber) has not been
        ↪  processed due to pgfkeysfiltered.}%
    },
}
```

2.8.5 Filters: 预定义的筛选标准

/pgf/key filters/active families

若 \pgfkeys@key@predicate→P. 83 调用这个过滤器，那么它会：

- 如果当前键是未定义的，设置 \pgfkeysfiltercontinuetrue,
- 如果当前键是已定义的，再检查当前键是否属于某个 family,
  - 如果属于某个 family, 就再检查这个 family 是否已被激活，
    - 如果已激活，设置 \pgfkeysfiltercontinuetrue,
    - 如果未激活，设置 \pgfkeysfiltercontinuefalse.
  - 如果不属于某个 family, 就设置 \pgfkeysfiltercontinuefalse.

在 \pgfkeysfiltercontinuetrue 的情况下，会正常执行当前键值对。

在 \pgfkeysfiltercontinuefalse 的情况下，会执行 \pgfkeys@filtered@handler→P. 84.

```
\pgfkeys{%
    /pgf/key filters/active families/.code={%
        \if\pgfkeyscasenumber0%
            % unknown options shall be processed with the
            % unknown-handlers.
            \pgfkeysfiltercontinuetrue
        \else
            \if\pgfkeyscasenumber3%
                \pgfkeysgetfamily\pgfkeyscurrentpath\pgfkeyspred@TMP
            \else
                \pgfkeysgetfamily\pgfkeyscurrentkey\pgfkeyspred@TMP
            \fi
            \ifpgfkeyssuccess
                \pgfkeysisfamilyactive{\pgfkeyspred@TMP}%
            \else% Ok, it does not belong to any family.
                \pgfkeysfiltercontinuefalse
            \fi
        \fi
    },%
}
```

这个定义的展开会定义 2 个控制序列：

- \csname pgfk@/pgf/key filters/active families/.@cmd\endcsname
- \csname pgfk@/pgf/key filters/active families/.@body\endcsname

(A1:a1)(A2:a2) Remaining options: '/my group/B=b,/my group/C=c'.

```
1  \pgfkeys{
2      /my group/A/.is family,
3      /my group/A1/.belongs to family=/my group/A,
4      /my group/A2/.belongs to family=/my group/A,
5      /my group/A3/.belongs to family=/my group/A,
6  }
7  \pgfkeys{/pgf/key filters/active families/.install key filter}
8  \pgfkeys{/my group/A/.activate family}
9  \pgfkeys{/pgf/key filter handlers/append filtered to/.install key filter handler=
   ↪  \remainingoptions}
10 \def\remainingoptions{}
11 \pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2, /my group/B=b, /my group/C=c}
12 Remaining options: `\remainingoptions'.
```

上面第 9 行参考 /pgf/key filter handlers/append filtered to→P. 100, 这一行实际上定义宏 \pgfkeys@filte
其作用是把不属于 /my group/A 这个 family 的键值对依次保存到 \remainingoptions 中。

/pgf/key filters/active families or no family={⟨*full key 1*⟩⟨=⟨*value 1*⟩⟩}{⟨*full key 2*⟩⟨=⟨*value 2*⟩⟩}

若 \pgfkeys@key@predicate →P. 83 调用这个过滤器，那么它会：

- 如果当前键是未知键，则执行命令 \pgfkeysevalkeyfilterwith →P. 94{⟨*full key 2*⟩⟨=⟨*value 2*⟩⟩}
- 如果当前键是已定义的，再检查当前键是否属于某个 family,
  - 如果属于某个 family, 就再检查这个 family 是否处于激活状态，
    - 如果已激活，设置 \pgfkeysfiltercontinuetrue,
    - 如果未激活，设置 \pgfkeysfiltercontinuefalse.
  - 如果不属于某个 family, 执行命令 \pgfkeysevalkeyfilterwith →P. 94{⟨*full key 1*⟩⟨=⟨*value 1*⟩⟩}.

```
/pgf/key filters/active families or no family/.code 2 args={%
    \if\pgfkeyscasenumber0%
        \pgfkeysevalkeyfilterwith{#2}%
    \else
        \if\pgfkeyscasenumber3%
            \pgfkeysgetfamily\pgfkeyscurrentpath\pgfkeyspred@TMP
        \else
            \pgfkeysgetfamily\pgfkeyscurrentkey\pgfkeyspred@TMP
        \fi
        \ifpgfkeyssuccess
            \pgfkeysisfamilyactive{\pgfkeyspred@TMP}%
        \else% Ok, it does not belong to any family.
            \pgfkeysevalkeyfilterwith{#1}%
        \fi
    \fi
},
```

按照手柄 /.code 2 args 的定义，这个定义的展开会定义 3 个控制序列：

- \csname pgfk@/pgf/key filters/active families or no family/.@cmd\endcsname, 能处理 2 个参数的命令
- \csname pgfk@/pgf/key filters/active families or no family/.@args\endcsname
- \csname pgfk@/pgf/key filters/active families or no family/.@body\endcsname

/pgf/key filters/active families or no family DEBUG={⟨*full key 1*⟩⟨=⟨*value 1*⟩⟩}{⟨*full key 2*⟩⟨=⟨*value 2*⟩⟩}

若 \pgfkeys@key@predicate →P. 83 调用这个过滤器，那么它会：

- 如果当前键是未知键，则
  1. 把当前键的信息记录到.log 文件，然后
  2. 执行 \pgfkeysevalkeyfilterwith →P. 94{⟨*full key 2*⟩⟨=⟨*value 2*⟩⟩}
- 如果当前键是已定义的，再检查当前键是否属于某个 family,
  - 如果属于某个 family, 就再检查这个 family 是否处于激活状态，
    - 如果已激活，设置 \pgfkeysfiltercontinuetrue, 再把当前键的信息记录到.log 文件，
    - 如果未激活，设置 \pgfkeysfiltercontinuefalse, 再把当前键的信息记录到.log 文件，
  - 如果不属于某个 family, 则
    1. 把当前键的信息记录到.log 文件，然后
    2. 执行 \pgfkeysevalkeyfilterwith →P. 94{⟨*full key 1*⟩⟨=⟨*value 1*⟩⟩}

```
\pgfkeys{%
    /pgf/key filters/active families or no family DEBUG/.code 2 args={%
        \if\pgfkeyscasenumber0%
            \pgf@typeout{[pgfkeyshasactivefamilyornofamily(\pgfkeyscurrentkey,
            ↪  \pgfkeyscasenumber) invoking unknown handler '#2']}%
            \pgfkeysevalkeyfilterwith{#2}%
        \else
            \if\pgfkeyscasenumber3%
                \pgfkeysgetfamily\pgfkeyscurrentpath\pgfkeyspred@TMP
            \else
                \pgfkeysgetfamily\pgfkeyscurrentkey\pgfkeyspred@TMP
            \fi
            \ifpgfkeyssuccess
                \pgfkeysisfamilyactive{\pgfkeyspred@TMP}%
                \ifpgfkeysfiltercontinue
                    \pgf@typeout{[pgfkeyshasactivefamilyornofamily(
                    ↪  \pgfkeyscurrentkey, \pgfkeyscasenumber) family is ACTIVE]}
                    ↪  %
                \else
                    \pgf@typeout{[pgfkeyshasactivefamilyornofamily(
                    ↪  \pgfkeyscurrentkey, \pgfkeyscasenumber) family is NOT
                    ↪  active.]}%
                \fi
            \else% Ok, it does not belong to any family.
                \pgf@typeout{[pgfkeyshasactivefamilyornofamily(\pgfkeyscurrentkey,
                ↪  \pgfkeyscasenumber) invoking has-no-family-handler '#1']}%
                \pgfkeysevalkeyfilterwith{#1}%
            \fi
        \fi
    },
}
```

/pgf/key filters/active families and known

```
\pgfkeys{%
    % A (faster) shortcut for
    % /pgf/key filters/active families or no family=
    %   {/pgf/keys filters/false}
    %   {/pgf/keys filters/false}
    /pgf/key filters/active families and known/.code={%
        \if\pgfkeyscasenumber0%
            \pgfkeysfiltercontinuefalse
        \else
            \if\pgfkeyscasenumber3%
                \pgfkeysgetfamily\pgfkeyscurrentpath\pgfkeyspred@TMP
            \else
                \pgfkeysgetfamily\pgfkeyscurrentkey\pgfkeyspred@TMP
            \fi
            \ifpgfkeyssuccess
                \pgfkeysisfamilyactive{\pgfkeyspred@TMP}%
            \else% Ok, it does not belong to any family.
                \pgfkeysfiltercontinuefalse
            \fi
        \fi
    },
}
```

/pgf/key filters/active families or descendants of={⟨*selected key path*⟩}

此键的参数 ⟨*selected key path*⟩ 可以是完整的键，也可以是不完整的键，但应该以斜线“/”开头；参数 ⟨*selected key path*⟩ 应该是字符串的形式，不应该是宏，因为从下面的定义看，这个键不对参数 ⟨*selected key path*⟩ 作展开。

若 \pgfkeys@key@predicate →P. 83 调用这个过滤器，那么它会：

- 如果当前键是未定义的，设置 \pgfkeysfiltercontinuefalse,
- 如果当前键是已定义的，再检查当前键是否属于某个 family,
  - 如果属于某个 family, 就再检查这个 family 是否已被激活，
    - 如果已激活，设置 \pgfkeysfiltercontinuetrue,
    - 如果未激活，设置 \pgfkeysfiltercontinuefalse.
  - 如果不属于某个 family, 再检查当前键 \pgfkeyscurrentkey 是否以 ⟨*selected key path*⟩ 开头，从纯字符的角度看，
    - 如果以 ⟨*selected key path*⟩ 开头，设置 \pgfkeysfiltercontinuetrue,
    - 如果不以 ⟨*selected key path*⟩ 开头，设置 \pgfkeysfiltercontinuefalse.

```
\pgfkeys{%
    % A (faster) shortcut for
    % /pgf/key filters/active families or no family=
    %   {/pgf/key filters/is descendant of=#1}% for keys without family
    %   {/pgf/keys filters/false}
    /pgf/key filters/active families or descendants of/.code={%
        \if\pgfkeyscasenumber0%
            \pgfkeysfiltercontinuefalse
        \else
            \if\pgfkeyscasenumber3%
                \pgfkeysgetfamily\pgfkeyscurrentpath\pgfkeyspred@TMP
            \else
                \pgfkeysgetfamily\pgfkeyscurrentkey\pgfkeyspred@TMP
            \fi
            \ifpgfkeyssuccess
                \pgfkeysisfamilyactive{\pgfkeyspred@TMP}%
            \else% Ok, it does not belong to any family.
                % the 'is descendant of' implementation has been
                % COPY PASTED here:
                %
                % string prefix comparison:
                \def\pgfkeysisdescendantof@impl##1#1##2\pgf@@eov{%
                    \def\pgfkeyspred@TMP{##1}%
                    \ifx\pgfkeyspred@TMP\pgfkeys@empty
                        \pgfkeysfiltercontinuetrue
                    \else
                        \pgfkeysfiltercontinuefalse
                    \fi
                }%
                \expandafter\pgfkeysisdescendantof@impl\pgfkeyscurrentkey#1
                ↪  \pgf@@eov
            \fi
        \fi
    },
}
```

/pgf/key filters/is descendant of={⟨*selected key path*⟩}

此键的参数 ⟨*selected key path*⟩ 可以是完整的键，也可以是不完整的键，但应该以斜线“/”开头；参

数 ⟨*selected key path*⟩ 应该是字符串的形式，不应该是宏，因为从下面的定义看，这个键不对参数 ⟨*selected key path*⟩ 作展开。

若 \pgfkeys@key@predicate →P. 83 调用这个过滤器，那么它会：

- 如果当前键 \pgfkeyscurrentkey 是未定义的，设置 \pgfkeysfiltercontinuetrue，
- 如果当前键 \pgfkeyscurrentkey 是已定义的，再检查当前键是否以 ⟨*selected key path*⟩ 开头，
  - 如果以 ⟨*selected key path*⟩ 开头，设置 \pgfkeysfiltercontinuetrue，
  - 如果不以 ⟨*selected key path*⟩ 开头，设置 \pgfkeysfiltercontinuefalse.

```
\pgfkeys{%
    % Processes only options which are children of #1.
    % Example:
    %   is descendant of/.install key filter=/foo
    % will be true for
    %   /foo/bar/x=y
    %   /foo/.cd
    %   /foo/bar/.style=...
    % but not for
    %   /bar/foo/...
    /pgf/key filters/is descendant of/.code={%
        \if\pgfkeyscasenumber0%
%\message{'\pgfkeyscurrentkey' (case \pgfkeyscasenumber) is UNKNOWN. Calling unknown handler.}%
            % unknown options shall be processed with the
            % unknown-handlers.
            \pgfkeysfiltercontinuetrue
        \else
            % string prefix comparison:
            % [ note : this has been COPY-PASTED to
            %   |active families or descendants of| ]
            \def\pgfkeysisdescendantof@impl##1#1##2\pgf@@eov{%
                \def\pgfkeyspred@TMP{##1}%
                \ifx\pgfkeyspred@TMP\pgfkeys@empty
%\message{'\pgfkeyscurrentkey' (case \pgfkeyscasenumber) is descendant of '#1': TRUE.}%
                    \pgfkeysfiltercontinuetrue
                \else
%\message{'\pgfkeyscurrentkey' (case \pgfkeyscasenumber) is descendant of '#1': FALSE.}%
                    \pgfkeysfiltercontinuefalse
                \fi
            }%
            \expandafter\pgfkeysisdescendantof@impl\pgfkeyscurrentkey#1\pgf@@eov
        \fi
    },%
}
```

/pgf/key filters/equals={⟨*key*⟩}

此键的参数 ⟨*key*⟩ 可以是完整的键，也可以是不完整的键，但应该以斜线 “/” 开头；参数 ⟨*key*⟩ 应该是字符串的形式，不应该是宏，因为从下面的定义看，这个键不对参数 ⟨*key*⟩ 作展开。

若 \pgfkeys@key@predicate →P. 83 调用这个过滤器，那么它会：

- 如果当前键 \pgfkeyscurrentkey 是未定义的，设置 \pgfkeysfiltercontinuetrue，
- 如果当前键是已定义的，再检查当前键与 ⟨*key*⟩ 是否同一个键，
  - 如果是，设置 \pgfkeysfiltercontinuetrue，
  - 如果不是，设置 \pgfkeysfiltercontinuefalse.

```
\pgfkeys{%
    % Returns true if the currently processed full key equals #2.
    /pgf/key filters/equals/.code={%
        \if\pgfkeyscasenumber0%
            % Unknown option:
            \pgfkeysfiltercontinuetrue
        \else
            \def\pgfkeyspred@TMP{#1}%
            \ifx\pgfkeyscurrentkey\pgfkeyspred@TMP
                \pgfkeysfiltercontinuetrue
            \else
                \pgfkeysfiltercontinuefalse
            \fi
        \fi
    },%
}
```

(A:a)

```
1 \pgfkeys{
2     /group 1/A/.code={(A:#1)},
3     /group 1/B/.code={(B:#1)},
4     /pgf/key filters/equals/.install key filter=/group 1/A}
5 \pgfqkeysfiltered{/group 1}{A=a,B=b}
```

上面例子中，

- 第 2, 3 行定义两个键
- 第 4 行 (参考 /.install key filter) 导致定义命令

  ```
  \def\pgfkeys@key@predicate{\pgfkeys@key@predicate@/group 1/A\pgfeov}
  ```

  其中 \pgfkeys@key@predicate@ 等于键 /pgf/key filters/equals 对应的 /.@cmd 命令。
- 在第 5 行执行命令 \pgfqkeysfiltered→P. 86 处理键值对 (默认键路径是 /group 1)，
  - 处理 A=a 时，命令 \pgfkeys@key@predicate 会设置 \pgfkeysfiltercontinuetrue, 因此导致用命令 pgfk@/group 1/A/.@cmd 来处理键值 a, 得到 “(A:a)” .
  - 处理 B=b 时，命令 \pgfkeys@key@predicate 会设置 \pgfkeysfiltercontinuefalse, 因此导致执行 \pgfkeys@filtered@handler→P. 84 (等于空的)，导致键值对 B=b 被忽略。

/pgf/key filters/not={⟨*full key*⟨=⟨*value*⟩⟩}

若 \pgfkeys@key@predicate→P. 83 调用这个过滤器，那么它会：

1. 执行 \pgfkeysevalkeyfilterwith→P. 94{⟨*full key*⟨=⟨*value*⟩⟩}，
2. 反转 \ifpgfkeysfiltercontinue→P. 96 的真值。

所以执行键值对 ⟨*full key*⟩=⟨*value*⟩ 应当能设置 \ifpgfkeysfiltercontinue→P. 96 的真值。这个过滤器的筛选标准是，与 ⟨*full key*⟩=⟨*value*⟩ 的逻辑意思相反的那些情况。

```
\pgfkeys{%
    /pgf/key filters/not/.code={%
        \pgfkeysevalkeyfilterwith{#1}%
        \ifpgfkeysfiltercontinue
            \pgfkeysfiltercontinuefalse
        \else
            \pgfkeysfiltercontinuetrue
        \fi
    },%
}
```

(C:c)

```
1 \pgfkeys{
2     /group 1/A/.code={(A:#1)},
3     /group 1/foo/bar/B/.code={(B:#1)},
4     /group 2/C/.code={(C:#1)},
5     /pgf/key filters/not/.install key filter={/pgf/key filters/is descendant of={/group 1}}}
6 \pgfkeysfiltered{/group 1/A=a,/group 1/foo/bar/B=b,/group 2/C=c}
```

上面第 5 行导致定义命令

```
\def\pgfkeys@key@predicate{\pgfkeys@key@predicate@{/pgf/key filters/is descendant
↪   of={/group 1}}\pgfeov}
```

其中 \pgfkeys@key@predicate@ 等于键 /pgf/key filters/not 对应的 /.@cmd 命令。

执行 \pgfkeys@key@predicate 时，导致

1. 执行 \pgfkeysevalkeyfilterwith{/pgf/key filters/is descendant of={/group 1}}, 导致用命令 pgfk@/pgf/key filters/is descendant of/.@cmd 处理 “/group 1”,
2. 反转 \ifpgfkeysfiltercontinue 的真值，这又导致那些 “不是 /group 1 的子键” 的键值对被执行，而 “是 /group 1 的子键” 的键值对被忽略。

/pgf/key filters/and={⟨*full key 1*⟨=⟨*value 1*⟩⟩}{⟨*full key 2*⟨=⟨*value 2*⟩⟩}

```
\pgfkeys{%
    /pgf/key filters/and/.code 2 args={%
        \pgfkeysevalkeyfilterwith{#1}%
        \ifpgfkeysfiltercontinue
            \pgfkeysevalkeyfilterwith{#2}%
        \fi
    },%
}
```

/pgf/key filters/or={⟨*full key 1*⟨=⟨*value 1*⟩⟩}{⟨*full key 2*⟨=⟨*value 2*⟩⟩}

```
\pgfkeys{%
    /pgf/key filters/or/.code 2 args={%
        \pgfkeysevalkeyfilterwith{#1}%
        \ifpgfkeysfiltercontinue
        \else
            \pgfkeysevalkeyfilterwith{#2}%
        \fi
    },%
}
```

/pgf/key filters/true

若 \pgfkeys@key@predicate→P. 83 调用这个过滤器，那么它只会设置 \pgfkeysfiltercontinuetrue.

```
\pgfkeys{%
    /pgf/key filters/true/.code={\pgfkeysfiltercontinuetrue},%
    /pgf/key filters/true/.install key filter,
}
```

注意，上面代码执行了

```
\pgfkeys{/pgf/key filters/true/.install key filter}
```

这就设置了 \pgfkeys@key@predicate→P. 83 的初始值。

/pgf/key filters/false

```
\pgfkeys{%
    /pgf/key filters/false/.code={%
        \pgfkeysfiltercontinuefalse
    },%
}
```

/pgf/key filters/defined

若 \pgfkeys@key@predicate→P. 83 调用这个过滤器，那么它：

- 若当前键是未定义的，设置 \pgfkeysfiltercontinuefalse,
- 若当前键是已定义的，设置 \pgfkeysfiltercontinuetrue.

```
\pgfkeys{%
    % Returns false if the current key is unknown, which avoids calling
    % the unknown handlers.
    /pgf/key filters/defined/.code={%
        \if\pgfkeyscasenumber0%
            \pgfkeysfiltercontinuefalse
        \else
            \pgfkeysfiltercontinuetrue
        \fi
    },
}
```

第三章　数学引擎

PGF 会自动载入数学引擎，数学引擎也可以独立于 PGF 使用。

```
\usepackage{pgfmath} % LaTeX
\input pgfmath.tex % plain TeX
\usemodule[pgfmath] % ConTeXt
```

数学引擎有 3 个层次：

- The top layer, 提供命令 \pgfmathparse, 还有一些能够设置尺寸或计数器的函数。
- The calculation layer.
- The implementation layer.

目前，数学引擎完全在 TEX 中做成，由于 TEX 是个排版程序而不是专门的数学程序，所以用 TEX 程序编制数学引擎是个很有吸引力的挑战，其中对精度和效率做了权衡。如果你觉得数学引擎的计算精度不够，那你需要去修改 implementation layer 中的算法。

文件《pgfmath.code.tex》会载入《pgfkeys.code.tex》，所以数学引擎需要 key 机制的支持。

3.1　解析表达式的命令

3.1.1　命令

数学引擎的解析命令主要是 \pgfmathparse:

\pgfmathparse{⟨*expression*⟩}

这个宏解析 ⟨*expression*⟩, 在正式解析之前会先用 \edef 将 ⟨*expression*⟩ 展开。在解析过程中，如果 ⟨*expression*⟩ 中有长度单位，就先把所有长度都转换为以 pt 为单位的长度，再计算 ⟨*expression*⟩ 的结果，然后将计算结果中的单位 pt 去掉，仅仅把计算结果中的数字（作为一个十进制数字）保存在宏 \pgfmathresult 中。如果 ⟨*expression*⟩ 中没有长度单位，那么解析的结果就跟带有长度单位 pt 的情况一样。在通常情况下，数学引擎利用 TEX 的寄存器（如尺寸寄存器，计数器）来做计算，所以计算能力受到 TEX 本身的限制。在任何时刻都应保证计算范围不超过 ±16383.99999pt，这是 TEX 的尺寸寄存器能允许的范围。

例如：

```
\pgfmathparse{2+3.5pt} \pgfmathresult \qquad
\pgfmathparse{2pt+3.5pt} \pgfmathresult
```

5.5　　5.5

关于这个宏需要注意以下几点：

- 在各个情况下，保存在宏 \pgfmathresult 中的总是十进制无单位的数值。
- 解析器能够识别并处理 TEX 的寄存器和盒子尺寸，类似 \mydimen（表示一个自定义的尺寸），0.5\mydimen（表示自定义尺寸的一半），\wd\mybox，0.5\dp\mybox，\mycount\mydimen 这样的尺寸或者计数器数值都是可以用在表达式 ⟨*expression*⟩ 中的。

- 解析器能识别并处理 ε-TEX 扩展的命令 \dimexpr，\numexpr，\glueexpr，\muexpr，只需要在这些命令前面加上 \the 就可以用这些命令的结果参与运算。
- 在表达式 ⟨*expression*⟩ 中可以使用圆括号来规定运算次序。
- 在表达式 ⟨*expression*⟩ 中可以使用多种函数，函数的参数也可以是表达式。
- 表达式 ⟨*expression*⟩ 接受科学计数法，如 1.234e+4，其中的指数符号可用小写 e 或大写 E.
- 在表达式 ⟨*expression*⟩ 中，如果一个整数以 0 开头，就默认这个整数是八进制数，会被自动转为十进制数。

```
10  \pgfmathparse{0 12}% 注意空格
    \pgfmathresult
```

- 在表达式 ⟨*expression*⟩ 中，如果一个整数以 0x 或 0X 开头，就默认这个整数是十六进制数，会被自动转为十进制数。十六进制数中的字母符号可以是大写也可以是小写。
- 在表达式 ⟨*expression*⟩ 中，如果一个整数以 0b 或 0B 开头，就默认这个整数是二进制数，会被自动转为十进制数。
- 在表达式 ⟨*expression*⟩ 中，如果有一串符号被双引号“"”括起来，就不对这一串符号执行计算。
- 表达式 ⟨*expression*⟩ 可以是一个用逗号分隔的列表，每个列表项都是一个表达式。此时会对列表项逐个做解析，将解析结果放到花括号中，这些花括号构成一个列表 (没有分隔符号)，保存到 \pgfmathresult 中。

  macro:->{1}{2.99571}{0.5}

```
\pgfmathparse{1,ln(20),sin(30)}
\meaning\pgfmathresult
```

  macro:->{}{2.99571}{0.5}

```
\pgfmathparse{,ln(20),sin(30)}
\meaning\pgfmathresult
```

这个命令在文件《pgfmathparser.code.tex》中定义，它的大致处理过程是：

```
\begingroup
  \pgfmath@catcodes
  \pgfmath@quickparsefalse
  % 检查 \ifpgfmathfloat 的真值
  % 若有真值 \pgfmathfloattrue, 则 \pgfmathfloatparsenumber
  % 若有真值 \pgfmathfloatfalse, 则 \pgfmathparse@
  % 假设 \pgfmathparse@
  ⟨ 解析表达式⟩
  % 解析完毕
  \pgfmathpostparse
  \pgfmath@smuggleone\pgfmathresult%
\endgroup
\ignorespaces
```

从定义看：

- \pgfmathparse 的所有操作都放入一个组内执行,在结尾处使用 \pgfmath@smuggleone 将 \pgfmathresult 推出组外。
- 如果有 \pgfmathfloattrue, 那么 \pgfmathparse 就导致 \pgfmathfloatparsenumber→P. 573, 这个命令只解析一个不带长度单位的数值，将此数值转为浮点数格式，并以纯文本形式保存在命令 \pgfmathresult 中
- 如果有 \pgfmathfloatfalse, 那么 \pgfmathparse 就导致 \pgfmathparse@, 这是通常情况下的 \pgfmathparse 的作用

但 fpu 库的选项 /pgf/fpu→P. 571 和命令 \pgfmathfloatparse→P. 596 会改变 \pgfmathparse 的作用。

\pgfmathqparse{⟨*expression*⟩}

这个宏与 \pgfmathparse 类似,解析 ⟨*expression*⟩ 并将结果作成无单位的数值保存在宏 \pgfmathresult 中。注意：(1) \pgfmathqparse 不解析函数、科学计数法、非十进制数，也不把双引号、问号、冒号当作具有特殊作用的符号。前面提到，表达式中双引号引起来的部分会被忽略。问号“?”和冒号“:”用于构成条件句。(2) 除了像 0.5\pgf@x 这种式子，⟨*expression*⟩ 中所有数值后面都要带单位。因为这两个限制，\pgfmathqparse 的速度大约是 \pgfmathparse 的两倍。

\pgfmathpostparse

命令 \pgfmathpostparse 出现在解析过程的结尾处（在得到 \pgfmathresult 之后，组结束之前），用它可以执行某些代码（例如对 \pgfmathresult 做某些调整），在默认下这个命令等于 \relax, 什么也不做。如果设置：

```
\def\pgfmathresultunitscale{⟨n⟩}
\let\pgfmathpostparse=\pgfmathscaleresult
```

那么保存在 \pgfmathresult 中的数值会被乘上因子 ⟨*n*⟩, 但这个做法有副作用：八进制、十六进制、二进制不会被自动转为十进制，表达式中也不能使用双引号。

\pgfmathprint{⟨*expression*⟩}

在文件《pgfmathparser.code.tex》中，此命令的定义是：

```
\def\pgfmathprint#1{\pgfmathparse{#1}\pgfmathresult}
```

下面介绍几个能给寄存器或计数器赋值或增值的命令。对于这几个命令，如果其中的 ⟨*expression*⟩ 以加号“+”开头，那么就不会解析 ⟨*expression*⟩, 而仅仅执行 TEX 的赋值或增值。这些命令的有效范围受到 TEX 分组的限制。

\pgfmathsetlength{⟨*register*⟩}{⟨*expression*⟩}

⟨*register*⟩ 代表一个 TEX 的寄存器。如果 ⟨*expression*⟩ 以加号“+”开头就只是给 ⟨*register*⟩ 赋值，举例来说：

1.0pt plus 1.0fil
```
\skipdef\myskip=0  % 定义弹性尺寸命令 \myskip
\pgfmathsetlength{\myskip}{+1pt plus 1fil} \the\myskip
```

上面例子中，先定义弹性尺寸命令 \myskip, 该命令占用寄存器 \skip0, 然后为它赋以弹性尺寸。

如果 ⟨*expression*⟩ 不以加号“+”开头，就用命令 \pgfmathparse 解析表达式，解析结果的数值部分会保存在 \pgfmathresult 中。在解析 ⟨*expression*⟩ 时，如果解析器遇到数学单位 mu，解析器就会默认 ⟨*register*⟩ 是 \muskip 类型的寄存器，会把 \pgfmathresult 中的数值带上单位 mu 并赋予⟨*register*⟩. 在解析 ⟨*expression*⟩ 时，如果解析器没有遇到数学单位 mu, 解析器就会默认 ⟨*register*⟩ 是刚性尺寸寄存器或弹性尺寸寄存器，会把 \pgfmathresult 中的数值带上单位 pt 并赋予 ⟨*register*⟩, 注意对寄存器的这个赋值不是全局地。

13.0mu
```
\muskipdef\mymuskip=0
\pgfmathsetlength{\mymuskip}{1mu+3*4mu} \the\mymuskip
```

13.0pt
```
\dimendef\mydimen=0
\pgfmathsetlength{\mydimen}{1pt+3*4pt} \the\mydimen
```

13.0pt
```
\skipdef\myskip=0
\pgfmathsetlength{\myskip}{1+3*4} \the\myskip
```

注意，下面的命令

```
\pgfmathsetlength{\myskip}{1pt plus 1fil}
```

是无效的，因为目前解析器还不支持 `fil`, 不过可以使用加号“+”来赋值：

```
\skipdef\myskip=0
\pgfmathsetlength{\myskip}{+1pt plus 1fil} \the\myskip
```

1.0pt plus 1.0fil

因为 ⟨*expression*⟩ 会被命令 `\pgfmathparse` 解析，所以在 ⟨*expression*⟩ 中可以使用纯数值表达式：

```
\newdimen\dddd
\pgfmathsetlength\dddd{10+20}
\the\dddd
```

30.0pt

`\pgfmathaddtolength{`⟨*register*⟩`}{`⟨*expression*⟩`}`

该命令类似 `\pgfmathsetlength`. 该命令将 ⟨*expression*⟩ 加到 ⟨*register*⟩ 中。

`\pgfmathsetcount{`⟨*count register*⟩`}{`⟨*expression*⟩`}`

该命令针对整数寄存器，类似 `\pgfmathsetlength`. 首先解析 ⟨*expression*⟩, 如果解析结果是小数，就直接去掉小数部分（没有“四舍五入”），将整数部分作为整数寄存器 ⟨*count register*⟩ 的值.

`\pgfmathaddtocount{`⟨*count register*⟩`}{`⟨*expression*⟩`}`

该命令针对整数寄存器，先解析 ⟨*expression*⟩, 如果解析结果是小数，就直接去掉小数部分（没有“舍入”），将整数部分加到整数寄存器 ⟨*count register*⟩ 中。

`\pgfmathsetcounter{`⟨*counter*⟩`}{`⟨*expression*⟩`}`

该命令针对 LaTeX 计数器，先解析 ⟨*expression*⟩, 如果解析结果是小数，就直接去掉小数部分（没有“四舍五入”），将整数部分作为计数器 ⟨*counter*⟩ 的值。

`\pgfmathaddtocounter{`⟨*counter*⟩`}{`⟨*expression*⟩`}`

该命令针对 LaTeX 计数器，先解析 ⟨*expression*⟩, 如果解析结果是小数，就直接去掉小数部分（没有“四舍五入”），将整数部分加到计数器 ⟨*counter*⟩ 中。

`\pgfmathsetmacro{`⟨*macro*⟩`}{`⟨*expression*⟩`}`

该命令定义宏 ⟨*macro*⟩, 并把解析 ⟨*expression*⟩ 的结果，即保存在 `\pgfmathresult` 中的数值赋予 ⟨*macro*⟩, 注意解析结果是无单位的数值。

`\pgfmathsetlengthmacro{`⟨*macro*⟩`}{`⟨*expression*⟩`}`

该命令定义宏 ⟨*macro*⟩, 并把解析 ⟨*expression*⟩ 的结果，即保存在 `\pgfmathresult` 中的数值带上单位 `pt` 赋予 ⟨*macro*⟩.

`\pgfmathtruncatemacro{`⟨*macro*⟩`}{`⟨*expression*⟩`}`

该命令定义宏 ⟨*macro*⟩, 把解析 ⟨*expression*⟩ 的结果去掉小数部分（无“舍入”），只把整数部分赋予 ⟨*macro*⟩.

3.1.2 长度单位的“显”、“隐”

`\ifpgfmathunitsdeclared`

这是个 TeX 的条件判断命令，如果在该命令之前曾经使用 `\pgfmathparse` 解析表达式，并且表达式中出现了长度单位，那么 `\ifpgfmathunitsdeclared` 的真值就是 true, 即有 `\pgfmathunitsdeclaredtrue`, 否则它的真值就是 false, 即有 `\pgfmathunitsdeclaredfalse`. 在解析表达式的过程中，通常这个条件的真值都是被全局设定的，不受分组的限制。

scalar(⟨*expression*⟩)

\pgfmathscalar{⟨*expression*⟩}

此命令的作用是：它使得 \ifpgfmathunitsdeclared 忽略 ⟨*expression*⟩ 中的长度单位，也就是说本命令不改变 ⟨*expression*⟩ 的解析结果，但会把 \ifpgfmathunitsdeclared 的真值设为 false. 注意，如果 ⟨*expression*⟩ 中的字符串（或其它的东西）里含有长度单位，那么本命令也是起作用的。

0.5 without unit

```
\pgfmathparse{scalar(1pt/2pt)} \pgfmathresult\
\ifpgfmathunitsdeclared with \else without \fi unit
```

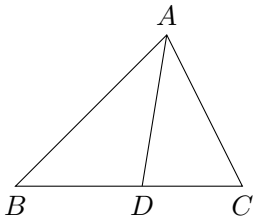
注意 1pt+scalar(1pt) 与 scalar(1pt)+1pt 是不同的表达式：

2.0 without unit
2.0 with unit

```
\pgfmathparse{1pt+scalar(1pt)} \pgfmathresult\
\ifpgfmathunitsdeclared with \else without \fi unit

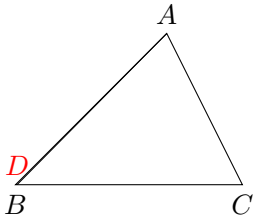
\pgfmathparse{scalar(1pt)+1pt} \pgfmathresult\
\ifpgfmathunitsdeclared with \else without \fi unit
```

一般情况下，在 \tikzmath{...} 中所做的赋值，如 \a=1+2; 不必带长度单位，保存在 \pgfmathresult 中的值是无单位的，但在很多情况下表达式的计算结果通常是带单位的，当引用带单位的计算结果时，如果只需要结果的数值部分参与运算，就可以使用 scalar 函数。比较下面两个例子：




```
\tikz{
  \coordinate["$A$"] (A) at (2,2);
  \coordinate["$B$" below] (B) at (0,0);
  \coordinate["$C$" below] (C) at (3,0);
  \draw (A) -- (B) -- (C) -- cycle;
  \path
    let \p1 =($(A)-(B)$), \p2 =($(A)-(C)$),
      \n1 = {veclen(\x1,\y1)}, \n2 = {veclen(\x2,\y2)}
    in coordinate ["$D$" below]
      (D) at ($ (B)!scalar(\n1/(\n1+\n2))!(C) $);
  \draw (A) -- (D);
}
```

在上面的例子中，表达式 \n1/(\n1+\n2) 中含有长度单位，用 scalar 将其包裹起来得到无单位的数值，代表一个“比例”；否则它代表一个“尺寸”，如下：




```
\tikz{
  \coordinate["$A$"] (A) at (2,2);
  \coordinate["$B$" below] (B) at (0,0);
  \coordinate["$C$" below] (C) at (3,0);
  \draw (A) -- (B) -- (C) -- cycle;
  \path
    let \p1 =($(A)-(B)$), \p2 =($(A)-(C)$),
      \n1 = {veclen(\x1,\y1)}, \n2 = {veclen(\x2,\y2)}
    in coordinate ["$D$"{text=red, above}]
      (D) at ($ (B)!\n1/(\n1+\n2)!(C) $);
  \draw (A) -- (D);
}
```

TEX 的数学单位“mu”（math units）会被特殊处理。

\ifpgfmathmathunitsdeclared

这个 TEX 条件判断命令与 \ifpgfmathunitsdeclared 类似，只是针对数学单位 mu, 并且函数 scalar 对该命令无影响。

3.2 数学表达式中的算子

x + y 中置算子，调用 add 函数

x - y 中置算子，调用 subtract 函数

- x 前置算子，调用 neg 函数

x * y 中置算子，调用 multiply 函数

x / y 中置算子，调用 divide 函数，如果 y 是 0 会导致错误。

x ^ y 中置算子，调用 pow 函数，计算 x^y.

x ! 后置算子，调用 factorial 函数，计算阶乘。

x r 后置算子，调用 deg 函数，这个式子假定 x 为弧度数，并将 x 转为角度。

x ? y : z 条件算子，调用 ifthenelse 函数，如果式子 x 的计算结果是非 0 值，则认为其值是 true.

x == y 中置算子，调用 equal 函数，判断 x 是否恒等于 y, 如果是，则返回 1, 否则返回 0.

x > y 中置算子，调用 greater 函数

x < y 中置算子，调用 less 函数

x != y 中置算子，调用 notequal 函数

x >= y 中置算子，调用 notless 函数

x <= y 中置算子，调用 notgreater 函数

x && y 中置算子，调用 and 函数

x || y 中置算子，调用 or 函数

!x 前置算子，调用 not 函数

(x) 组算子，圆括号有两个用处，一是用来规定运算次序，一是用来标示数学函数的参数，例如 sin(30*10) 或 mod(72,3). 注意，sin 30（留意其中的空格）等效于 sin(30), 而 sin 30*10（留意其中的空格）等效于 sin(30)*10.

{x} 组算子，在数学表达式中，花括号可以构造数组，例如，{1,2,3}. 花括号还可以套嵌起来构成多维数组，例如，{1, {2,3}, {4,5}, 6}，这是个 2 维数组。可以用 TEX 的定义命令将数组保存在一个宏中，以便于在别处引用，例如：

```
\def\myarray{{1,2,3}}
```

在数组中，除数字、运算符号、函数运算外，其它由字母、文字构成的元素要用双引号引起来：

1, two, 3.0, IV, cinq, sechs, 7.0, 文字,

```
\def\myarray{{1,"two",2+1,"IV","cinq","sechs",sin(\i*5)*14,"\kaishu 文字"}}
\foreach \i in {0,...,7}{\pgfmathparse{\myarray[\i]}\pgfmathresult, }
```

还要注意区分“逗号分隔的列表”与“数组”：逗号分隔的列表的外围没有花括号，而数组是用花括号括起来的列表。

```
\def\aaaa{1,2,3}% 这是“逗号分隔的列表”
\def\bbbb{{1,2,3}}% 这是数组
```

[x] 数组索引算子，方括号可用以索引数组，数组中各个维度的元素编号都是从 0 开始，如果索引序号过大或过小都会导致错误。索引多维数组时，可以连续使用多个方括号括起来的索引号，一个方括号确定一个“地址”，或者说，把数组看作是一个多层次的结构，第一个方括号中的数字确定数组的第一层次中的某个“子数组” G_x，第二个方括号中的数字确定 G_x 之下的某个“子数组” G_{xx}……

```
5 \pgfmathparse{{1,2,3,{4,5}}[3][1]} \pgfmathresult
```

```
\def\print#1{\pgfmathparse{#1}\pgfmathresult}
\def\identitymatrix{{{1,2,3},{"a","b","c"},{" 壹"," 贰"," 叁"}}}
\tikz[x=0.5cm,y=0.5cm]{
  \foreach \i in {0,1,2} \foreach \j in {0,1,2}
  \node at (\i,\j) [anchor=base] {\print{\identitymatrix[\i][\j]}};
  \node at (1,4) [anchor=base] {\print{\identitymatrix[2][1]}};
}
```

贰
3 c 叁
2 b 贰
1 a 壹

"x" 组算子，双引号引起来的部分表示“引用”，如果在双引号之内有宏，那么在数学引擎解析整个表达式之前，这些宏会被展开，这类似命令 \edef 的作用。如果不希望这些宏被展开，就在这些宏的前面加上 \noexpand，例如，\noexpand\Huge.

5 is Bigger than 0. 5 is smaller than 10.

```
\def\x{5}
\foreach \y in {0,10}{
  \pgfmathparse{\x > \y ? "\noexpand\Large Bigger" : "\noexpand\tiny smaller"}
  \x\ is \pgfmathresult\ than \y.
}
```

在使用 PGF 的数组时要注意以下现象：下面例子中，数组 \agroup 中似乎只有一项，但对该数组的索引结果却不是这样：

```
9 \def\agroup{{96}}
6 \pgfmathsetmacro{\onenine}{\agroup[0]}\onenine\\
  \pgfmathsetmacro{\onesix}{\agroup[1]}\onesix
```

3.3 数学表达式中的函数

每个函数都有对应的 PGF 命令版本，注意函数的 PGF 命令版本一般不用做命令 \pgfmathparse 的参数，而是单独使用，它们都把结果保存在命令 \pgfmathresult 中。例如：

```
3.0 \pgfmathadd{1}{2} \pgfmathresult
```

取整方式 对一个实数有 3 种取整方式：

Floor 向下取整，如 $\lfloor 2.7 \rfloor = 2$，$\lfloor -2.3 \rfloor = -3$.

Ceil 向上取整，如 $\lceil 2.3 \rceil = 3$，$\lceil -2.7 \rceil = -2$.

向 0 取整 即直接去掉小数部分，只保留整数部分。这个取整方式可以由前两种取整方式定义，即对正实数向下取整，对负实数向上取整。

取整函数用于截尾 可以用取整函数定义针对小数的“截尾”操作，即把某个位置之后的小数数字全部去掉，不做舍入。

如果 $x \in \mathbb{R}_+$，设 $n \in \mathbb{N}_0$，对 x 的截尾可以是

$$\operatorname{trunc}(x, n) = \frac{\lfloor 10^n \cdot x \rfloor}{10^n}.$$

如果 $x \in \mathbb{R}_-$，设 $n \in \mathbb{N}_0$，对 x 的截尾可以是

$$\operatorname{trunc}(x, n) = \frac{\lceil 10^n \cdot x \rceil}{10^n}.$$

整数除法的余数 在做整数除法时，通常用的是欧几里得辗转相除法，例如计算 $a \div b$，最终的结果表现为：

$$a = b \cdot q + r,\ q \in \mathbb{Z},\ |r| < |b|.$$

在数学上，一般规定 $r \geqslant 0$，但在程序计算上有多种选择：

1. 令 r 的符号非负。此时，$0 \leqslant r < b$，

$$\frac{a}{|b|} = \begin{cases} q + \dfrac{r}{b},\ b > 0, \\ -q - \dfrac{r}{b},\ b < 0, \end{cases} \quad \text{故} \quad \left\lfloor \frac{a}{|b|} \right\rfloor = \begin{cases} q,\ b > 0, \\ -q,\ b < 0, \end{cases}$$

所以

$$r = a - \operatorname{sign}(b) \cdot b \cdot \left\lfloor \frac{a}{|b|} \right\rfloor.$$

2. 令 r 的符号与 $a \div b$ 相同。
3. 令 r 的符号与 a 相同，这个情况叫作 truncated division，
   - 若 $a > 0$，则 $r > 0$ 且 $|r| < |b|$，此时

$$\frac{a}{|b|} = \begin{cases} q + \dfrac{r}{b},\ b > 0, \\ -q - \dfrac{r}{b},\ b < 0, \end{cases} \quad \text{故} \quad \left\lfloor \frac{a}{|b|} \right\rfloor = \begin{cases} q,\ b > 0, \\ -q,\ b < 0, \end{cases}$$

   所以

$$r = a - \operatorname{sign}(b) \cdot b \cdot \left\lfloor \frac{a}{|b|} \right\rfloor.$$

   - 若 $a < 0$，则 $r < 0$ 且 $|r| < |b|$，此时

$$\frac{a}{|b|} = \begin{cases} q + \dfrac{r}{b},\ b > 0, \\ -q - \dfrac{r}{b},\ b < 0, \end{cases} \quad \text{故} \quad \left\lceil \frac{a}{|b|} \right\rceil = \begin{cases} q,\ b > 0, \\ -q,\ b < 0, \end{cases}$$

   所以

$$r = a - \operatorname{sign}(b) \cdot b \cdot \left\lceil \frac{a}{|b|} \right\rceil.$$

4. 令 r 的符号与 b 相同，这个情况叫作 floored division，

$$\frac{a}{b} = q + \frac{r}{b},\ \text{故} \ \left\lfloor \frac{a}{b} \right\rfloor = q,$$

所以

$$r = a - b \cdot \left\lfloor \frac{a}{b} \right\rfloor.$$

举例来说，因为 $-4 = 3 \cdot (-1) - 1$，所以按 truncated division 方式计算 $-4 \div 3$ 的余数就是 -1；另一方面 $-4 = 3 \cdot (-2) + 2$，所以按 floored division 方式计算 $-4 \div 3$ 的余数就是 2.

3.3.1 基本算术函数

add(x,y)

\pgfmathadd{⟨x⟩}{⟨y⟩}

加法。

```
81.0 \pgfmathparse{add(75,6)} \pgfmathresult \par
81.0 \pgfmathadd{75}{6} \pgfmathresult
```

subtract(x,y)

\pgfmathsubtract{⟨x⟩}{⟨y⟩}

减法。

neg(x)

\pgfmathneg{⟨x⟩}

相反数。

multiply(x,y)

\pgfmathmultiply{⟨x⟩}{⟨y⟩}

乘法。

divide(x,y)

\pgfmathdivide{⟨x⟩}{⟨y⟩}

除法，计算 $x \div y$.

```
12.5 \pgfmathparse{divide(75,6)} \pgfmathresult
```

div(x,y)

\pgfmathdiv{⟨x⟩}{⟨y⟩}

除法，并将结果四舍五入，即变成最靠近的整数。

```
8 \pgfmathparse{div(75,9)} \pgfmathresult
```

factorial(x)

\pgfmathfactorial{⟨x⟩}

计算阶乘。

sqrt(x)

\pgfmathsqrt{⟨x⟩}

计算 $\sqrt{x}$.

pow(x,y)

\pgfmathpow{⟨x⟩}{⟨y⟩}

计算 x^y, 当 y 是整数时精度较好，如果 y 不是整数则近似计算 $\mathrm{e}^{y \ln x}$.

e

\pgfmathe

这是自然对数底常数，约等于 2.718281828.

exp(x)

\pgfmathexp{⟨x⟩}

计算 e^x.

ln(x)

\pgfmathln{⟨x⟩}

近似计算自然对数。

4.99997 `\pgfmathparse{ln(exp(5))} \pgfmathresult`

log10(x)

\pgfmathlogten{⟨x⟩}

近似计算以 10 为底的常用对数。

1.99997 `\pgfmathparse{log10(100)} \pgfmathresult`

log2(x)

\pgfmathlogtwo{⟨x⟩}

近似计算以 2 为底的对数。

6.99994 `\pgfmathparse{log2(128)} \pgfmathresult`

abs(x)

\pgfmathabs{⟨x⟩}

计算绝对值。

mod(x,y)

\pgfmathmod{⟨x⟩}{⟨y⟩}

使用 truncated division 方式计算除法 $\frac{x}{y}$ 的余数，但是余数的符号与 $\frac{x}{y}$ 相同。

Mod(x,y)

\pgfmathMod{⟨x⟩}{⟨y⟩}

使用 floored division 方式计算除法 $\frac{x}{y}$ 的余数，但是余数的符号总是非负。

sign(x)

\pgfmathsign{⟨x⟩}

返回 x 的符号。

3.3.2 舍入函数

round(x)

\pgfmathround{⟨x⟩}

四舍五入为整数。

floor(x)

\pgfmathfloor{⟨x⟩}

向下取整。

ceil(x)

\pgfmathceil{⟨x⟩}

向上取整。

int(x)

\pgfmathint{⟨x⟩}

返回 x 的整数部分，即向 0 取整。

frac(x)

\pgfmathfrac{⟨x⟩}

返回 x 的小数部分。

real(x)

\pgfmathreal{⟨x⟩}

声明（确保）x 是个十进制小数，带有小数点。

```
4.0  \pgfmathparse{real(4)} \pgfmathresult
```

3.3.3 几个整数运算函数

gcd(x,y)

\pgfmathgcd{⟨x⟩}

计算 x 与 y 的最大公因子。

isodd(x)

\pgfmathisodd{⟨x⟩}

如果 x 的整数部分是奇数就返回 1，否则返回 0.

iseven(x)

\pgfmathiseven{⟨x⟩}

如果 x 的整数部分是偶数就返回 1，否则返回 0.

isprime(x)

\pgfmathisprime{⟨x⟩}

如果 x 的整数部分是素数就返回 1，否则返回 0.

3.3.4 三角函数

三角函数的参数都默认为角度制的数。

pi

\pgfmathpi{⟨x⟩}

圆周率常数，约等于 3.141592654.

```
179.99962 \pgfmathparse{pi r} \pgfmathresult
```

rad(x)

\pgfmathrad{⟨x⟩}

假定 x 是角度制的数，并将它转换为弧度制的数。

deg(x)

\pgfmathdeg{⟨x⟩}

假定 x 是弧度制的数，并将它转换为角度制的数。

sin(x)

\pgfmathsin{⟨x⟩}

正弦函数，默认 x 是角度制的数。

cos(x)

\pgfmathcos{⟨x⟩}

```
1Y8.6603e-1]
11
{
  \pgfkeys{/pgf/fpu,}% 使用 fpu 库
  \pgfmathcos{30}\pgfmathresult\par
  \the\catcode`\@
}
```

上面例子表明，在激活 fpu 库函数的情况下，符号 @ 的类代码是 12 (其他字符)，所以下面代码导致错误：

```
{
  \pgfkeys{/pgf/fpu,}% 使用 fpu 计算
  \makeatletter% 把 @ 的类代码设为 11, 即字母
  \pgfmathcos@{30}\pgfmathresult
  \makeatother
}
```

tan(x)

\pgfmathtan{⟨x⟩}

sec(x)

\pgfmathsec{⟨x⟩}

cosec(x)

\pgfmathcosec{⟨x⟩}

cot(x)

\pgfmathcot{⟨x⟩}

asin(x)

\pgfmathasin{⟨x⟩}

反正弦函数，值域是 $[-90°, 90°]$.

acos(x)

\pgfmathacos{⟨x⟩}

反余弦函数，值域是 $[0°, 180°]$.

atan(x)

\pgfmathatan{⟨x⟩}

反正切函数，计算出来的函数值默认为角度制的数，属于区间 $(-90°, 90°)$. 如果提前设置了 /pgf/trig format=rad, 则把角度制的结果转为弧度制的结果。

atan2(y,x)

\pgfmathatantwo{⟨y⟩}{⟨x⟩}

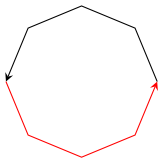
计算向量 (x, y) 所对应的方向，默认是一个角度制下的数值，属于区间 $[-180°, 180°]$. 如果 $(x, y) = (0, 0)$, 则返回 $90°$.

```
\pgfmathparse{atan2(-4,-4)} \pgfmathresult\\
\pgfmathparse{atan2(0,0)} \pgfmathresult
```

-135.0
90.0

/pgf/trig format=deg|rad (no default, initially deg)

在涉及角度的地方，例如三角函数的参数，极坐标点的坐标（如 (60:2pt)）, node 的角度位置（如 a.30）等等，都使用角度制数值，这是因为 PGF 的初始设置为 trig format=deg，使用 trig format=rad 可以将这些数值转换到弧度制。



```
\begin{tikzpicture}
\draw[-stealth] (0:1) -- (45:1) -- (90:1) -- (135:1) -- (180:1);
\draw[-stealth,trig format=rad,red]
  (pi:1) -- (5/4*pi:1) -- (6/4*pi:1) -- (7/4*pi:1) -- (2*pi:1);
\end{tikzpicture}
```

但是注意，PGF 的某些个算子、命令、程序库可能依赖初始设置 trig format=deg，因此最好局部地使用 trig format=rad. 本选项的定义是：

```
\pgfkeys{
    /pgf/trig format/.is choice,
    % attention: the luamath library will hook into these code keys.
    /pgf/trig format/deg/.code={\def\pgfmath@trig@format@choice{0}},
    /pgf/trig format/rad/.code={\def\pgfmath@trig@format@choice{1}},
    /pgf/trig format/deg,
}
```

可见本选项影响宏 \pgfmath@trig@format@choice 的值。

\pgfmathsincos{⟨x⟩}

在文件《pgfmathfunctions.trigonometric.code.tex》中定义 \pgfmathsincos 如下：

```
% \pgfmathsincos
%
% Calculate the sin and cosine of #1 (in degrees).
```

```
%
\def\pgfmathsincos#1{%
  \pgfmathparse{#1}%
  \expandafter\pgfmathsincos@\expandafter{\pgfmathresult}}
\def\pgfmathsincos@#1{%
  \edef\pgfmath@temparg{#1}%
  \pgfmathsin@{\pgfmath@temparg}\edef\pgfmathresulty{\pgfmathresult}%
  \pgfmathcos@{\pgfmath@temparg}\edef\pgfmathresultx{\pgfmathresult}%
}
```

由以上定义可见，命令 \pgfmathsincos{⟨x⟩} 的处理是：

1. 由命令 \pgfmathparse 解析 ⟨x⟩, 解析结果保存在 \pgfmathresult 中。
2. 命令 \pgfmathsincos@ 处理上一步得到的 \pgfmathresult.
   (a) 把 \pgfmathresult 中保存的结果展开后转存到 \pgfmath@temparg 中。
   (b) 计算正弦值 \pgfmathsin@{\pgfmath@temparg}, 正弦值保存在 \pgfmathresulty 中。
   (c) 计算余弦值 \pgfmathcos@{\pgfmath@temparg}, 余弦值保存在 \pgfmathresultx 中。

3.3.5 比较函数与逻辑函数

equal(x,y)

\pgfmathequal{⟨x⟩}{⟨y⟩}

如果 $x = y$ 则返回 1，否则返回 0.

```
1 \pgfmathparse{equal(20,20)} \pgfmathresult
```

greater(x,y)

\pgfmathgreater{⟨x⟩}{⟨y⟩}

如果 $x > y$ 则返回 1，否则返回 0.

less(x,y)

\pgfmathless{⟨x⟩}{⟨y⟩}

如果 $x < y$ 则返回 1，否则返回 0.

notequal(x,y)

\pgfmathnotequal{⟨x⟩}{⟨y⟩}

如果 $x \neq y$ 则返回 1，否则返回 0.

notgreater(x,y)

\pgfmathnotgreater{⟨x⟩}{⟨y⟩}

如果 $x \leqslant y$ 则返回 1，否则返回 0.

notless(x,y)

\pgfmathnotless{⟨x⟩}{⟨y⟩}

如果 $x \geqslant y$ 则返回 1，否则返回 0.

and(x,y)

\pgfmathand{⟨x⟩}{⟨y⟩}

如果 x 和 y 的解析结果都是非 0 值，则返回 1，否则返回 0.

or(x,y)

\pgfmathor{⟨x⟩}{⟨y⟩}

如果 x 和 y 的解析结果不都是 0 值，则返回 1，否则返回 0.

not(x)

\pgfmathnot{⟨x⟩}

如果 $x = 0$, 则返回 1，否则返回 0.

ifthenelse(x,y,z)

\pgfmathifthenelse{⟨x⟩}{⟨y⟩}{⟨z⟩}

如果 x 的解析结果不是 0 值，则执行 y，否则执行 z.

true

\pgfmathtrue

执行结果是 1.

yes
```
\pgfmathparse{true ? "yes" : "no"} \pgfmathresult
```

false

\pgfmathfalse

执行结果是 0.

3.3.6 伪随机函数

rnd

\pgfmathrnd

产生一个在 0 到 1 之间的服从均匀分布的伪随机数。

0.85863, 0.22765, 0.11626, 0.98111, 0.29697, 0.76578, 0.57455, 0.55875, 0.71234, 0.12143,
```
\foreach \x in {1,...,10} {\pgfmathparse{rnd}\pgfmathresult, }
```

rand

\pgfmathrand

产生一个在 -1 到 1 之间的服从均匀分布的伪随机数。

random(x,y)

\pgfmathrandom{⟨x⟩,⟨y⟩}

这个函数可以采用 3 种形式：

- random()，\pgfmathrandom{}，产生一个在 0 到 1 之间的服从均匀分布的伪随机数。
- random(x)，产生一个在 1 到 x 之间的服从均匀分布的伪随机“整数”。
- random(x,y)，产生一个在 x 到 y 之间的服从均匀分布的伪随机“整数”。

注意，如果 ⟨x⟩ 或 ⟨y⟩ 是宏，则会被展开：

9
```
\pgfmathparse{real(2)}
\pgfmathparse{random(\pgfmathresult,10)}\pgfmathresult
```

3.3.7 基本的转换函数

下面的函数将十进制数转换为二进制数、八进制数、十六进制数，转换结果不能再参与进一步的计算，因为解析器只能对十进制数做计算。

hex(x)

\pgfmathhex{⟨x⟩}

假定 x 是十进制数，并将它转换为十六进制数，其中的字母使用小写，转换结果不能再参与计算。

```
ffff  \pgfmathparse{hex(65535)} \pgfmathresult
```

Hex(x)

\pgfmathHex{⟨x⟩}

假定 x 是十进制数，并将它转换为十六进制数，其中的字母使用大写，转换结果不能再参与计算。

oct(x)

\pgfmathoct{⟨x⟩}

假定 x 是十进制数，并将它转换为八进制数，转换结果不能再参与计算。

bin(x)

\pgfmathbin{⟨x⟩}

假定 x 是十进制数，并将它转换为二进制数，转换结果不能再参与计算。

3.3.8 其它函数

min(x1,x2,...,xn)

\pgfmathmin{⟨x_1,x_2,...⟩}{⟨...,x_{n-1},x_n⟩}

返回某一组数值中的最小值，由于历史的原因，命令 \pgfmathmin 的参数要分成两组，每一组的个数任意。

```
-8.0  \pgfmathparse{min(3,-2,-8,100)} \pgfmathresult
```

观察下面的例子：

- 下面例子中，数组 \agroup 中只有一项，函数 min 的结果是：

```
96.0  \def\agroup{{96}}
      \pgfmathsetmacro{\onenum}{min(\agroup)}\onenum
```

- 下面例子中，列表 \agroup 中似乎只有一项，但 min 函数的结果却不是这样：

```
6.0  \def\asecquence{96}
     \pgfmathsetmacro{\onenum}{min(\asecquence)}\onenum
```

- 还有：

```
69.0  \pgfmathsetmacro{\onenum}{min({96},{69})}\onenum
```

- 下面的命令导致错误：

```
\pgfmathsetmacro{\onenum}{min({96,69})}
```

错误信息：！Missing number, treated as zero.

注意，函数 min, max 的“公共”版宏使用两组花括号，这是为了兼容旧版本函数句法。

16383.0
```
\pgfmathmin{}{}
\pgfmathresult
```

max(x1,x2,...,xn)

\pgfmathmax{⟨x_1,x_2,...⟩}{⟨...,x_{n-1},x_n⟩}

返回某一组数值中的最大值。

veclen(x,y)

\pgfmathveclen{⟨x⟩}{⟨y⟩}

计算 $\sqrt{x^2+y^2}$.

array(x,y)

\pgfmatharray{⟨x⟩}{⟨y⟩}

这里 x 是个数组，y 是个索引数，本函数索引数组 x 中编号为 y 的元素，元素编号从 0 开始。

17
```
\pgfmathparse{array({9,13,17,21},2)} \pgfmathresult
```

sinh(x)

\pgfmathsinh{⟨x⟩}

计算双曲正弦值。

cosh(x)

\pgfmathcosh{⟨x⟩}

计算双曲余弦值。

tanh(x)

\pgfmathtanh{⟨x⟩}

计算双曲正切值。

width("x")

\pgfmathwidth{"x"}

这里 x 代表能够放在一个 TEX 的水平盒子里的内容，本函数返回该盒子的宽度，宽度数值的默认单位为 pt，双引号防止 x 被解析。注意在整个表达式被解析之前，双引号内的宏会被展开。

82.69815
```
\pgfmathparse{width("Some Lovely Text")} \pgfmathresult
```

height("x")

\pgfmathheight{"x"}

这里 x 代表能够放在一个 TEX 的水平盒子里的内容，本函数返回该盒子的高度，高度数值的默认单位为 pt，双引号防止 x 被解析。注意在整个表达式被解析之前，双引号内的宏会被展开。

depth("x")

\pgfmathdepth{"x"}

这里 x 代表能够放在一个 TEX 的水平盒子里的内容，本函数返回该盒子的深度，深度数值的默认单位为 pt，双引号防止 x 被解析。注意在整个表达式被解析之前，双引号内的宏会被展开。

3.4 其它数学命令

3.4.1 基本算术函数

\pgfmathreciprocal{⟨x⟩}

这个命令计算 $\frac{1}{x}$，即倒数，当 x 的值很小时，该命令能有较高的精确度。

3.4.2 比较与逻辑函数

\pgfmathapproxequalto{⟨x⟩}{⟨y⟩}

如果 ⟨x⟩ 与 ⟨y⟩ 的绝对值之差满足 $|x-y| < 0.0001$，该命令会把 1.0 保存在 \pgfmathresult 中；除了这个情况外，把 0.0 保存在 \pgfmathresult 中. 该命令还会使得 TEX 的条件判断命令 \ifpgfmathcomparison 的真值被相应地设定，这个条件判断命令是全局命令。

```
\pgfmathapproxequalto{0.01}{0.002} \pgfmathresult \\
\ifpgfmathcomparison true \else false \fi
```

0.0
false

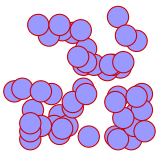
3.4.3 伪随机数

\pgfmathgeneratepseudorandomnumber

本命令生成一个属于 $[1, 2^{31}-1]$ 的随机数，保存在 \pgfmathresult 中。本命令使用 linear congruency generator.

\pgfmathrandominteger{⟨macro⟩}{⟨minimum⟩}{⟨maximum⟩}

定义宏 ⟨macro⟩, 这个宏保存一个从 ⟨minimum⟩（含）到 ⟨maximum⟩（含）的随机整数。



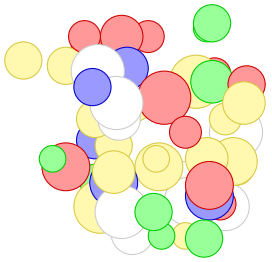
```
\begin{pgfpicture}
 \foreach \x in {1,...,50}{
  \pgfmathrandominteger{\a}{1}{50}
  \pgfmathrandominteger{\b}{1}{50}
  \pgfpathcircle{\pgfpoint{+\a pt}{+\b pt}}{+4pt}
  \color{blue!40!white}
  \pgfsetstrokecolor{red!80!black}
  \pgfusepath{stroke, fill}
 }
\end{pgfpicture}
```

\pgfmathdeclarerandomlist{⟨list name⟩}{{⟨item-1⟩}{⟨item-2⟩}...}

这个命令创建一个列表，列表的名称是 ⟨list name⟩, 注意各个列表项要用花括号括起来，而不是用逗号分隔。该命令一般与下一个命令一起使用。

\pgfmathrandomitem{⟨macro⟩}{⟨list name⟩}

其中的 ⟨list name⟩ 是用上一个命令创建的列表名称，⟨macro⟩ 是本命令定义的宏。本命令从 ⟨list name⟩ 中随机选择一个列表项，并把选出的列表项保存在宏 ⟨macro⟩ 中。



```
\begin{pgfpicture}
  \pgfmathdeclarerandomlist{color}{{red}{blue}{green}{yellow}{white}}
  \foreach \a in {1,...,50}{
    \pgfmathrandominteger{\x}{1}{85}
    \pgfmathrandominteger{\y}{1}{85}
    \pgfmathrandominteger{\r}{5}{10}
    \pgfmathrandomitem{\c}{color}
    \pgfpathcircle{\pgfpoint{+\x pt}{+\y pt}}{+\r pt}
    \color{\c!40!white}
    \pgfsetstrokecolor{\c!80!black}
    \pgfusepath{stroke, fill}
  }
\end{pgfpicture}
```

\pgfmathsetseed{⟨*integer*⟩}

显式地设置随机数生成器的种子。⟨*integer*⟩ 的默认值是 \time×\year.

3.4.4 整数的进位制转换

目前，PGF 能把 0 到 $2^{31}-1$ 之间的十进制整数转换为 p 进制数，这里 $2 \leqslant p \leqslant 36$. 如果 $p > 10$，那么数值中的字母可以使用大写也可以使用小写。

\pgfmathbasetodec{⟨*macro*⟩}{⟨*number*⟩}{⟨*base*⟩}

定义宏 ⟨*macro*⟩; 将 ⟨*number*⟩ 看作基数是 ⟨*base*⟩ 的数，把它转换为 10 进制数，保存在宏 ⟨*macro*⟩ 中。⟨*number*⟩ 中的字母可以使用大写或小写。

4223 `\pgfmathbasetodec\mynumber{107f}{16} \mynumber`

\pgfmathdectobase{⟨*macro*⟩}{⟨*number*⟩}{⟨*base*⟩}

定义宏 ⟨*macro*⟩; 将 ⟨*number*⟩ 看作基数是 10 进制数，将其转换成基数为 ⟨*base*⟩ 的数，其中的字母使用小写，并保存在宏 ⟨*macro*⟩ 中。

ffff `\pgfmathdectobase\mynumber{65535}{16} \mynumber`

\pgfmathdectoBase{⟨*macro*⟩}{⟨*number*⟩}{⟨*base*⟩}

定义宏 ⟨*macro*⟩; 将 ⟨*number*⟩ 看作基数是 10 进制数，将其转换成基数为 ⟨*base*⟩ 的数，其中的字母使用大写，并保存在宏 ⟨*macro*⟩ 中。

FFFF `\pgfmathdectoBase\mynumber{65535}{16} \mynumber`

\pgfmathbasetobase{⟨*macro*⟩}{⟨*number*⟩}{⟨*base-1*⟩}{⟨*base-2*⟩}

定义宏 ⟨*macro*⟩; 将 ⟨*number*⟩ 看作基数是 ⟨*base-1*⟩ 的数，把它转换成基数为 ⟨*base-2*⟩ 的数，其中的字母使用小写，并保存在宏 ⟨*macro*⟩ 中。

\pgfmathbasetoBase{⟨*macro*⟩}{⟨*number*⟩}{⟨*base-1*⟩}{⟨*base-2*⟩}

定义宏 ⟨*macro*⟩; 将 ⟨*number*⟩ 看作基数是 ⟨*base-1*⟩ 的数，把它转换成基数为 ⟨*base-2*⟩ 的数，其中的字母使用大写，并保存在宏 ⟨*macro*⟩ 中。

\pgfmathsetbasenumberlength{⟨*integer*⟩}

进位制转换时，结果值的位数可以用这个命令设置。如果结果值的位数小于 ⟨*integer*⟩ 就用 0 补足。

00001111
```
\pgfmathsetbasenumberlength{8}
\pgfmathdectobase\mynumber{15}{2} \mynumber
```

\pgfmathtodigitlist{⟨*macro*⟩}{⟨*number*⟩}

定义宏 ⟨*macro*⟩; 在数字 ⟨*number*⟩ 的相邻两个数字符号之间插入逗号，作成一个列表，保存在 ⟨*macro*⟩ 中。

1,1,1,1,0,0
1
```
\pgfmathdectobase{\binary}{60}{2}
\pgfmathtodigitlist{\digitlist}{\binary}
\digitlist \par
\pgfmathparse{{\digitlist}[3]} \pgfmathresult
```

从上面的例子看出，本命令得到的列表并不是“数组”，要想把它做成数组还需要在它外围加上花括号。

```
\pgfmathsetbasenumberlength{8}
\begin{tikzpicture}[x=0.25cm, y=0.25cm]
  \foreach \n [count=\y] in {0, 60, 102, 102, 126, 102, 102, 102, 0}{
    \pgfmathdectobase{\binary}{\n}{2}
    \pgfmathtodigitlist{\digitlist}{\binary}
    \foreach \digit [count=\x, evaluate={\c=\digit*50+15;}] in \digitlist
      \fill [fill=black!\c] (\x, -\y) rectangle ++(1,1);
  }
\end{tikzpicture}
```

3.4.5 角度计算

下面的两个命令必须与 PGF 的内核一起使用才有效，它们都把结果保存在命令 \pgfmathresult 中。

\pgfmathanglebetweenpoints{⟨*p*⟩}{⟨*q*⟩}

这里 ⟨*p*⟩ 与 ⟨*q*⟩ 是 PGF 命令规定的坐标点，设想一条起始点为 ⟨*p*⟩ 且方向向右的水平射线，一条起始点为 ⟨*p*⟩ 且过点 ⟨*q*⟩ 的射线，两条射线构成一个角，从水平射线到第 2 条射线的（角度制下的）角就是本命令所返回的数值。

45.0
```
\pgfmathanglebetweenpoints
  {\pgfpoint{1cm}{3cm}} {\pgfpoint{2cm}{4cm}}
\pgfmathresult
```

\pgfmathanglebetweenlines{⟨*p1*⟩}{⟨*q1*⟩}{⟨*p2*⟩}{⟨*q2*⟩}

这里 ⟨*p1*⟩, ⟨*q1*⟩, ⟨*p2*⟩, ⟨*q2*⟩ 都是 PGF 命令规定的坐标点，设想过点 ⟨*p1*⟩, ⟨*q1*⟩ 的直线 l_1 以及过点 ⟨*p2*⟩, ⟨*q2*⟩ 的直线 l_2, 从直线 l_1 到直线 l_2 的角度就是本命令所返回的数值。

270.0
```
\pgfmathanglebetweenlines
  {\pgfpoint{1cm}{3cm}}{\pgfpoint{2cm}{4cm}}
  {\pgfpoint{0cm}{1cm}}{\pgfpoint{1cm}{0cm}}
\pgfmathresult
```

3.5 用数学引擎自定义函数

可以自定义一个函数，像使用 add(x,y)，\pgfmathadd{x,y} 那样使用它。这主要用到下面的命令。

\pgfmathdeclarefunction{⟨*name*⟩}{⟨*number of arguments*⟩}{⟨*code*⟩}

\pgfmathdeclarefunction*{⟨name⟩}{⟨number of arguments⟩}{⟨code⟩}

这个命令的有效范围受到 TEX 分组的限制，此命令在文件《pgfmathfunctions.code.tex》中定义。

⟨name⟩ 是自定义函数的名称，其中可以使用字母（大小写皆可）、数字、下划线（下标符号），但是注意不能以数字开头，也不能包含空格。函数名称应当是尚未使用过的名称，如果不确定某个函数名称是否已经被使用，可以使用带星号“*”版本的命令。

若重复使用函数名称来做定义，则带星号“*”的命令会将函数的定义改写为“新版”，而不带星号“*”的命令会发出错误信息。注意最好不要改变预定义的函数，因为在某些命令、程序库中可能会用到它们。最好只用不带星号的命令以避免出错。

⟨number of arguments⟩ 声明函数的参数个数，可以是 0，正整数，或者是省略号“...”，省略号表示函数的参数个数是可变的。PGF 将常数，如 pi，e 当作是有 0 个参数的函数。如果一个函数的参数个数超过 9 个或者是可变的，就会被特殊处理 (被当作 1 个参数的样子)。

⟨code⟩ 是函数的定义内容，它应当是这样的代码：解析 ⟨code⟩ 得到一个结果，去掉结果中的长度单位（如果有的话）后保存在宏 \pgfmathresult 中。函数的定义最好不要有其它副作用，例如不要改变全局变量，所以，通常 ⟨code⟩ 是一个组，在组内计算得到结果，然后使用 \pgfmathreturn 将结果推到组外，并保存在 \pgfmathresult 中。也可以参考 \pgfmath@smuggleone→P. 40.

下面的例子定义一个函数 double, 它将其参数变成原值的 2 倍：

88.6
```
\makeatletter
\pgfmathdeclarefunction{double}{1}{
  \begingroup  % 开启一个 TeX 分组
    \pgf@x=#1pt\relax
    \multiply\pgf@x by2\relax
    \pgfmathreturn\pgf@x
  \endgroup  % 结束 TeX 分组
}
\makeatother
\pgfmathparse{double(44.3)} \pgfmathresult
```

在上面的代码中，命令\pgfmathreturn⟨tokens⟩ 之后必须跟上 \endgroup.

在文件《pgfmathutil.code.tex》中定义命令 \pgfmathreturn:

```
\def\pgfmath@returnone#1\endgroup{%
  \pgfmath@x#1%
  \edef\pgfmath@temp{\pgfmath@tonumber{\pgfmath@x}}%
  \expandafter\endgroup\expandafter\def\expandafter\pgfmathresult\expandafter{
  ↪  \pgfmath@temp}%
}

\let\pgfmathreturn=\pgfmath@returnone
```

命令 \pgfmathreturn\pgf@x\endgroup 会把展开 \pgf@x 后的结果（去掉单位）推到组外，并保存在 \pgfmathresult 中。

命令 \pgfmathdeclarefunction 还会创建以下两个宏，它们是函数的 PGF 命令版本。

- “公共版本”

  \pgfmath⟨function name⟩

  例如，对于上面定义的函数 double, 有相应的 PGF 命令 \pgfmathdouble, 因此可以写出：

  88.6
  ```
  \pgfmathdouble{44.3} \pgfmathresult
  ```

  88.6
  ```
  \pgfmathdouble{44.3pt} \pgfmathresult
  ```

  这个宏是自定义函数 ⟨function name⟩ 的“公共”版，它只是使用 ⟨function name⟩ 的一个外

在“接口”（interface），实际上并不计算函数值。当执行命令 \pgfmath⟨*function name*⟩{⟨参数⟩} 计算函数值时，该命令的⟨参数⟩会被 \pgfmathparse 处理，然后再把处理结果（不带长度单位）传递给下面的“个人版”宏，由下面的“个人版”宏计算函数值。所以“公共”版宏的参数可以带有长度单位，也可以不带长度单位。

- “个人版本”

  \pgfmath⟨*function name*⟩@

  例如，对于上面定义的函数 double，有相应的 PGF 命令 \pgfmathdouble@.
  这个宏是自定义函数 ⟨*function name*⟩ 的“个人”版，计算函数值时，为了提高速度，解析器调用“个人”版宏，而不是调用“公共”版宏。命令 \pgfmathdeclarefunction 的参数 ⟨*code*⟩ 就是“个人版本”命令 \pgfmath⟨*function name*⟩@ 的定义内容。这个宏接受“公共”版宏传递来的（无单位）参数，代入 ⟨*code*⟩ 中做计算。所以 ⟨*code*⟩ 中的变量 #1, #2 等应当代表无单位的纯数值。如果要把参数手工传递给这个宏，则参数必须是不带单位的。

88.6
```
\makeatletter
\pgfmathdouble@{44.3}
\makeatother
\pgfmathresult
```

pt 88.6
```
\makeatletter
\pgfmathdouble@{44.3pt}
\makeatother
\pgfmathresult
```

对于自定义的函数，如果其参数个数不超过 9 个，则该函数的“公共”版宏和“个人”版宏也可以使用通常的定义 TEX 宏的方式来定义，例如命令 \pgfmathsincos →P. 122 的定义。
如果自定义函数的参数个数超过 9 个，或使用省略号表示参数个数（可变个数），那么函数的“公共”版宏和“个人”版宏都定义为“好像是只有一个参数”的宏，此时的 ⟨*code*⟩ 应当能够逐个处理给出的参数。
此时使用“公共”版宏的句法是，例如：

```
\pgfmathVariableArgs{1.1,3.5,-1.5,2.6}
```

即只是用逗号分隔各个参数。使用“个人”版宏的句法是：

```
\pgfmathVariableArgs@{{1.1}{3.5}{-1.5}{2.6}}
```

即用花括号把各个参数括起来。

重定义一个函数使用下面的命令。

\pgfmathredeclarefunction{⟨*function name*⟩}{⟨*algorithm code*⟩}

重定义函数 ⟨*function name*⟩ 时，命令会用 ⟨*algorithm code*⟩ 重定义 \pgfmath⟨*function name*⟩@, 但是注意不能改变原来函数的参数个数，而且本命令的有效范围受到花括号分组的限制，因此可以局部地重定义函数。

3.6 命令 \pgfmathdeclarefunction 的处理

3.6.1 处理过程

在《pgfmathfunctions.code.tex》中定义了命令 \pgfmathdeclarefunction.
命令 \pgfmathdeclarefunction{⟨*name*⟩}{⟨*number of arguments*⟩}{⟨*code*⟩} 的处理过程主要是：

1. 读取前两个参数 ⟨*name*⟩ 和 ⟨*number of arguments*⟩
2. 检查 ⟨*name*⟩ 是否已定义的函数，如果已定义，就发出错误信息，如果未定义，则
3. 定义函数的几个属性：

```
\pgfmath@namedef{pgfmath@function@#1}{#1}%
\pgfmath@namedef{pgfmath@operation@#1@type}{function}%
\pgfmath@namedef{pgfmath@operation@#1@arity}{#2}%
\pgfmath@namedef{pgfmath@operation@#1@infix}{#1}%
\pgfmath@namedef{pgfmath@operation@#1@precedence}{1000}%
```

4. 检查参数的个数，如果参数个数 ⟨*number of arguments*⟩ 大于 9 或者是省略号，则重定义：

```
\pgfmath@namedef{pgfmath@operation@#1@arity}{1}
```

5. 执行选项

```
\pgfkeysvalueof{/pgf/declare function/execute at end function}
```

6. 读取第 3 个参数 ⟨*code*⟩
7. 执行选项

```
\pgfkeysvalueof{/pgf/declare function/execute at end function}
```

8. 如果参数个数 ⟨*number of arguments*⟩ 是 0, 则

```
\def\pgfmath⟨name⟩{\pgfmath⟨name⟩@}
\def\pgfmath⟨name⟩@{⟨code⟩}
```

9. 如果参数个数 ⟨*number of arguments*⟩ 是 1, 则

```
\def\pgfmath⟨name⟩#1{%
  \pgfmathparse{#1}\expandafter\pgfmath⟨name⟩@\expandafter{\pgfmathresult}}
\def\pgfmath⟨name⟩@#1{⟨code⟩}
```

   如果参数个数 ⟨*number of arguments*⟩ 大于 9 或者是省略号，当使用函数 \pgfmath⟨*name*⟩ 时，要把所有参数用一个花括号括起来。此时的 ⟨*code*⟩ 应当能逐个处理这些参数。

10. 如果参数个数 ⟨*number of arguments*⟩ 大于 1(不大于 9), 则利用循环命令 \pgfmathloop →P. 38 做定义。例如，当 ⟨*number of arguments*⟩ 是 3 时，定义 2 个函数：
    (a) 定义 \pgfmath⟨*name*⟩ 为

```
\def\pgfmath⟨name⟩#1#2#3{
    \pgfmathparse{#1}%
    \expandafter\let\csname pgfmath@argument@1\endcsname=\pgfmathresult%
    \pgfmathparse{#2}%
    \expandafter\let\csname pgfmath@argument@2\endcsname=\pgfmathresult%
    \pgfmathparse{#3}%
    \expandafter\let\csname pgfmath@argument@3\endcsname=\pgfmathresult%
    \pgfmath⟨name⟩@{\csname pgfmath@argument@1\endcsname}%
                   {\csname pgfmath@argument@2\endcsname}%
                   {\csname pgfmath@argument@3\endcsname}%
}
```

    (b) 定义

```
\def\pgfmath⟨name⟩@#1#2#3{⟨code⟩}
```

可见,如果 \pgfmath⟨*name*⟩ 或 \pgfmath⟨*name*⟩@ 的参数个数超过一个,最好不要直接把宏 \pgfmathresult 用作该命令的参数，如果要用的话，最好用作第一个参数。例如：

2.0
```
\def\pgfmathresult{-1}
\pgfmathadd{1}{\pgfmathresult}
\pgfmathresult
```

0.0
```
\def\pgfmathresult{-1}
\pgfmathadd{\pgfmathresult}{1}
\pgfmathresult
```

如果参数个数 ⟨*number of arguments*⟩ 大于 9 或者是省略号,要注意 ⟨*code*⟩ 的设计应该照顾 \pgfmath⟨*name*⟩ 或 \pgfmath⟨*name*⟩@ 的参数格式。例如，文件《pgfmathfunctions.misc.code.tex》中有 min 的定义：

```
\pgfmathdeclarefunction{min}{...}{%
  \begingroup%
    \pgfmath@x=16383pt\relax%
    \pgfmathmin@@#1{}}

\def\pgfmathmin@@#1{%
  \def\pgfmath@temp{#1}%
  \ifx\pgfmath@temp\pgfmath@empty%
    \let\pgfmath@next=\pgfmathmin@@@%
  \else%
    \ifdim#1pt<\pgfmath@x%
      \pgfmath@x=#1pt\relax%
    \fi%
    \let\pgfmath@next=\pgfmathmin@@%
  \fi%
  \pgfmath@next}

\def\pgfmathmin@@@{\pgfmath@returnone\pgfmath@x\endgroup}%

% For computability with old code.
\def\pgfmathmin#1#2{%
    \pgfmathparse{#1,#2}%
    \expandafter\pgfmathmin@\expandafter{\pgfmathresult}%
}
```

可见函数 min 采用的是逐项比较的办法来确定最小值的。当执行 \pgfmathparse{min(96)} 时，因为 min 函数只有一个数值参数 96, 所以预期得到的结果就是这个数值本身，但实际上结果是数值 6. 这是因为，在这个过程中，数值 96 被解析为两个字符 96, 保存到 \pgfmathresult 中；然后用“个人版”命令 \pgfmathmin@ 处理展开一次的 \pgfmathresult, 即两个字符 96, 此时字符 9 和 6 会分别成为命令 \pgfmathmin@@ 的参数。这个过程不会调用“公共版”命令 \pgfmathmin, 也不会检查函数 min 的参数的个数。而当使用“公共版”命令 \pgfmathmin 时，要为它提供两个参数：

```
\pgfmathmin{⟨arg 1⟩}{⟨arg 2⟩}
```

并且第一个参数 ⟨*arg 1*⟩ 最好不要是空的。

3.6.2 相关选项

```
\pgfkeys{%
    /pgf/declare function/.code={%
        \pgfmath@local@functions#1@=@;%
    },
    /pgf/declare function/execute at begin function/.initial={},
    /pgf/declare function/execute at end function/.initial={},
    /pgf/declare function/ignore spaces/.is choice,
```

```
    /pgf/declare function/ignore spaces/.default=true,
    /pgf/declare function/ignore spaces/true/.style={%
        /pgf/declare function/execute at begin function={%
            \begingroup
            \catcode`\^^I=9\relax
            \catcode`\ =9\relax
            \catcode`\~=10\relax
            \endlinechar=`\ \relax
        },
        /pgf/declare function/execute at end function={%
            \endgroup
        },
    },
    /pgf/declare function/ignore spaces/false/.style={%
        /pgf/declare function/execute at begin function={},
        /pgf/declare function/execute at end function={},
    },
}
```

`/pgf/declare function/execute at begin function=`⟨*tokens*⟩ (no default)

这个选项被 `\pgfmathdeclarefunction` 利用。

`/pgf/declare function/execute at end function=`⟨*tokens*⟩ (no default)

这个选项被 `\pgfmathdeclarefunction` 利用。

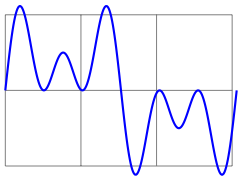
`/pgf/declare function/ignore spaces=true|false` (initially `false`, default `true`)

3.7 自定义局部函数的选项

下面的键（key）用于自定义局部函数。

`/pgf/declare function=`{⟨*function definitions*⟩} (no default)

这个键用来自定义局部有效的、不太复杂的函数，先看一个例子：



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw [blue, thick, x=0.0085cm, y=1cm,
    declare function={
      sines(\t,\a,\b)=1 + 0.5*(sin(\t)+sin(\t*\a)+sin(\t*\b));
    }]
    plot [domain=0:360, samples=144, smooth] (\x,{sines(\x,3,5)});
\end{tikzpicture}
```

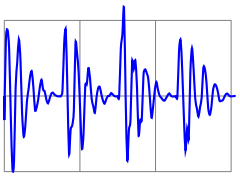
本选项的参数 ⟨*function definitions*⟩ 可以是罗列形式：

```
⟨name 1⟩(⟨arguments 1⟩)=⟨definition 1⟩;
⟨name 2⟩(⟨arguments 2⟩)=⟨definition 2⟩;
...
```

其中可以罗列多个句子，每个句子声明一个函数，每个句子中都要有等号 `=`, 并以分号结束。

对于声明函数的句子 “⟨*name*⟩(⟨*arguments*⟩)=⟨*definition*⟩;”，其中：

- ⟨*name*⟩ 是所定义的函数名，⟨*arguments*⟩ 是该函数的参数列表，⟨*definition*⟩ 是定义函数的表达式。
- ⟨*arguments*⟩ 是用逗号分隔的“宏”列表，例如：“`\x,\y`”，注意这里不能定义可变个数的参数。
- ⟨*definition*⟩ 必须是能被数学引擎解析的表达式，其中应当使用 ⟨*arguments*⟩ 中列出的宏。
- ⟨*function definitions*⟩ 中可以定义多个函数，每个函数定义都以分号结束，所用函数名在当前环境内不能重复，后定义的函数可以调用前定义的函数。



```
\begin{tikzpicture}[
  declare function={
    excitation(\t,\w) = sin(\t*\w);
    noise = rnd - 0.5;
    source(\t) = excitation(\t,20) + noise;
    filter(\t) = 1 - abs(sin(mod(\t, 90)));
    speech(\t) = 1 + source(\t)*filter(\t);
  }
]
  \draw [help lines] (0,0) grid (3,2);
  \draw [blue, thick, x=0.0085cm, y=1cm] (0,1) --
    plot [domain=0:360, samples=144, smooth] (\x,{speech(\x)});
\end{tikzpicture}
```

3.8 几个函数

下面是文件《pgfmathfunctions.basic.code.tex》定义的几个函数。

3.8.1 函数 ln 的定义

函数 `ln(`⟨*x*⟩`)` 的计算思路如下。

记 $e^t = x$, 目标是求解 t. 有 $t = \frac{\log_2 x}{\log_2 e} = \log_2 x \cdot \ln 2$. 其中 $\ln 2 \approx 0.6931471806$.

设 $2^m \le x \le 2^{m+1}$, 则 $x = 2^m \cdot u = 2^{m+1} \cdot v$, $1 \le u \le 2, 0.5 \le v \le 1$, 于是 $\log_2 x = m + \log_2 u = m + 1 + \log_2 v$. 所以

$$t \approx 0.6931471806 \times (m + \log_2 u) \approx 0.6931471806 \times (m + 1 + \log_2 v).$$

只要近似计算 $\log_2 u$ 或 $\log_2 v$ 就可以近似 t 了。

函数 `ln(`⟨*x*⟩`)` 使用多项式 `A+(B+(C+(D+(E+F*x)*x)*x)*x)*x` 来近似计算，见下文。

```
\pgfmathdeclarefunction{ln}{1}{%
    \begingroup%
        \expandafter\pgfmath@x#1pt\relax%
        \ifdim\pgfmath@x>0pt\else%
            \pgfmath@error{I cannot calculate the logarithm of `#1'}{}%
        \fi%
        \c@pgfmath@counta=0\relax%
        \ifdim\pgfmath@x>2pt\relax%
            %..... 省略若干
        \fi%
        %
        % Use A+(B+(C+(D+(E+F*x)*x)*x)*x)*x
        %
        % where:
        % A = -2.787927935
        % B =  5.047861502
        % C = -3.489886985
```

```
        % D =  1.589480044
        % E = -.4025153233
        % F =  0.04300521312
        %
        \edef\pgfmath@temp{\pgfmath@tonumber{\pgfmath@x}}%
        \pgfmath@x=0.04300521312\pgfmath@x%
        \advance\pgfmath@x by-.4025153233pt\relax%
        \pgfmath@x=\pgfmath@temp\pgfmath@x%
        \advance\pgfmath@x by1.589480044pt\relax%
        \pgfmath@x=\pgfmath@temp\pgfmath@x%
        \advance\pgfmath@x by-3.489886985pt\relax%
        \pgfmath@x=\pgfmath@temp\pgfmath@x%
        \advance\pgfmath@x by5.047861502pt\relax%
        \pgfmath@x=\pgfmath@temp\pgfmath@x%
        \advance\pgfmath@x by-2.787927935pt\relax%
        \advance\pgfmath@x by\c@pgfmath@counta pt\relax%
        \pgfmath@x=0.6931471806\pgfmath@x%
        \pgfmath@returnone\pgfmath@x%
    \endgroup%
}
```

1. 首先开启一个组，`\begingroup`
2. 执行 `\expandafter\pgfmath@x`$\langle x\rangle$`pt\relax`.
3. 执行 `\ifdim` 做检查，如果 `\pgfmath@x` 不大于 0pt, 即 $\langle x\rangle \le 0$, 则给出错误提示。
4. 设置计数器值 `\c@pgfmath@counta=0`
5. 执行 `\ifdim` 做检查，结果可用下表解释：

区间 I_i	变换因子 f	计数器 c
$(-\infty, 2^{-13}]$	2^{13}	-13
$(2^{-13}, 2^{-12}]$	2^{12}	-12
$(2^{-12}, 2^{-11}]$	2^{11}	-11
$(2^{-11}, 2^{-10}]$	2^{10}	-10
$(2^{-10}, 2^{-9}]$	2^{9}	-9
$(2^{-9}, 2^{-8}]$	2^{8}	-8
$(2^{-8}, 2^{-7}]$	2^{7}	-7
$(2^{-7}, 2^{-6}]$	2^{6}	-6
$(2^{-6}, 2^{-5}]$	2^{5}	-5
$(2^{-5}, 2^{-4}]$	2^{4}	-4
$(2^{-4}, 2^{-3}]$	2^{3}	-3
$(2^{-3}, 2^{-1}]$	2^{2}	-2
$(2^{-1}, 2^{0})$	2^{1}	-1
$[1, 2]$	1	0
$(2^{1}, 2^{2})$	2^{-1}	1
$[2^{2}, 2^{3})$	2^{-2}	2
$[2^{3}, 2^{4})$	2^{-3}	3
$[2^{4}, 2^{5})$	2^{-4}	4

区间 I_i	变换因子 f	计数器 c
$[2^5, 2^6)$	2^{-5}	5
$[2^6, 2^7)$	2^{-6}	6
$[2^7, 2^8)$	2^{-7}	7
$[2^8, 2^9)$	2^{-8}	8
$[2^9, 2^{10})$	2^{-9}	9
$[2^{10}, 2^{11})$	2^{-10}	10
$[2^{11}, 2^{12})$	2^{-11}	11
$[2^{12}, 2^{13})$	2^{-12}	12
$[2^{13}, \infty)$	2^{-13}	13

上面表格的第一列列出了 27 个区间，每个区间 I_i 对应一个因子 f 和一个计数器值 c. 如果 $\langle x\rangle \in I_i$, 则令 \pgfmath@x=$f\times$\pgfmath@x, 令 \c@pgfmath@counta=c, 所以，如果 $\langle x\rangle$ 不是过于极端的数值，那么 \pgfmath@x 的值就属于区间 $(0.5\text{pt}, 2\text{pt}]$.

6. 近似计算

7. 执行 \pgfmath@returnone\pgfmath@x, 将数值结果保存到 \pgfmathresult 中，并且结束分组。

3.8.2 函数 pow 的定义

```
\pgfmathdeclarefunction{pow}{2}{%
  \begingroup%
    \pgfmath@xa=#1pt%
    \pgfmath@xb=#2pt%
    \ifdim\pgfmath@xa=0pt\relax%
      \ifdim\pgfmath@xb=0pt\relax%
        \pgfmath@x=1pt\relax%
      \else%
        \pgfmath@x=0pt\relax%
      \fi%
    \else%
      \afterassignment\pgfmath@x%
      \expandafter\c@pgfmath@counta\the\pgfmath@xb\relax%
      \ifnum\c@pgfmath@counta<0\relax%
        \c@pgfmath@counta=-\c@pgfmath@counta%
        \pgfmathreciprocal@{#1}%
        \pgfmath@xa=\pgfmathresult pt\relax%
      \fi
      \ifdim\pgfmath@x=0pt\relax%
        \pgfmath@x=1pt\relax%
        \pgfmathloop%
          \ifnum\c@pgfmath@counta>0\relax%
            \ifodd\c@pgfmath@counta%
              \pgfmath@x=\pgfmath@tonumber{\pgfmath@x}\pgfmath@xa%
            \fi
            \ifnum\c@pgfmath@counta>1\relax%
              \pgfmath@xa=\pgfmath@tonumber{\pgfmath@xa}\pgfmath@xa%
            \fi%
            \divide\c@pgfmath@counta by2\relax%
        \repeatpgfmathloop%
      \else%
```

```
        \pgfmathln@{#1}%
        \pgfmath@x=\pgfmathresult pt\relax%
        \pgfmath@x=\pgfmath@tonumber{\pgfmath@xb}\pgfmath@x%
        \pgfmathexp@{\pgfmath@tonumber{\pgfmath@x}}%
        \pgfmath@x=\pgfmathresult pt\relax%
      \fi%
    \fi
    \pgfmath@returnone\pgf@x%
  \endgroup%
}
```

函数 pow(⟨x⟩,⟨y⟩) 的计算思路如下。

1. 首先开启一个组，\begingroup
2. 赋值

```
\pgfmath@xa=⟨x⟩pt%
\pgfmath@xb=⟨y⟩pt%
```

3. 执行一个条件分支
   - 如果 $\langle x\rangle = 0$
     - 如果 $\langle y\rangle = 0$, 则

       ```
       \pgfmath@x=1pt\relax
       ```

     - 如果 $\langle y\rangle \neq 0$, 则

       ```
       \pgfmath@x=0pt\relax
       ```

   - 如果 $\langle x\rangle \neq 0$

     (a) 执行

     ```
     \afterassignment\pgfmath@x%
     \expandafter\c@pgfmath@counta\the\pgfmath@xb\relax%
     ```

     意思是把 \pgfmath@xb (即 $\langle y\rangle$) 的整数部分保存到 \c@pgfmath@counta 中，把它的小数部分保存到 \pgfmath@x 中。

     (b) 如果 $\langle y\rangle$ 的整数部分小于 0,

     i. 做相反数赋值

     ```
     \c@pgfmath@counta=-\c@pgfmath@counta%
     ```

     ii. 求倒数

     ```
     \pgfmathreciprocal@{⟨x⟩}%
     ```

     iii. 把 $\langle x\rangle$ 的倒数保存到 \pgfmath@xa 中

     ```
     \pgfmath@xa=\pgfmathresult pt\relax%
     ```

     (c) 执行一个条件分支

     – 如果 $\langle y\rangle$ 的小数部分等于 0,

     i. 赋值

     ```
     \pgfmath@x=1pt\relax%
     ```

ii. 执行一个循环，循环条件是 \c@pgfmath@counta>0, 循环内容是：

A. 如果 \c@pgfmath@counta 是奇数，则

```
\pgfmath@x=\pgfmath@tonumber{\pgfmath@x}\pgfmath@xa%
```

B. 如果 \c@pgfmath@counta>1, 则

```
\pgfmath@xa=\pgfmath@tonumber{\pgfmath@xa}\pgfmath@xa%
```

C. 执行

```
\divide\c@pgfmath@counta by2\relax%
```

这个循环完成幂运算 $\langle X\rangle^n$(n 是正整数)。

– 如果 $\langle y\rangle$ 的小数部分不等于 0, 则计算 $\mathrm{e}^{\langle y\rangle \ln\langle x\rangle}$, 将结果保存在 \pgfmath@x 中。

4. 执行 \pgfmath@returnone\pgf@x, 将数值结果保存到 \pgfmathresult 中，并且结束分组。

3.8.3 函数 sin 的处理过程

在文件《pgfmathfunctions.trigonometric.code.tex》中定义了函数 sin.

当执行 \pgfmathparse{sin(x_0)} (计算 $\sin x_0$) 时，处理过程是：

1. 计算 $x_1 = \left|x_0 - \left[\frac{x_0}{360}\right]\right|$, 于是 $x_1 \in [-359, 359]$.
2. 计算 $x_2 = |x_1|$.
3. 如果 $x_2 \geqslant 180$, 则计算 $x_3 = 360 - x_2$; 如果 $x_2 < 180$, 则令 $x_3 = x_2$; 所以 $x_3 \in [0, 180]$.
   记 $x_3 = a.b$, 其中 $a(0 \leqslant a \leqslant 180)$ 是 x_3 的整数部分，b 是 x_3 的小数部分。
4. 近似计算 $(1-b)\cos(a) + b\cos(a+1) \approx \sin x_0$, 其中 $\cos(a)$ 和 $\cos(a+1)$ 由下面的定义规定：

```
\def\pgfmath@def#1#2#3{\expandafter\def\csname pgfmath@#1@#2\endcsname{#3}}
\pgfmath@def{cos}{0}{1.00000}    \pgfmath@def{cos}{1}{0.99985}
\pgfmath@def{cos}{2}{0.99939}    \pgfmath@def{cos}{3}{0.99863}
%..... 省略若干
\pgfmath@def{cos}{178}{-0.99939}    \pgfmath@def{cos}{179}{-0.99985}
\pgfmath@def{cos}{180}{-1.00000}     \pgfmath@def{cos}{181}{-0.99985}
```

3.8.4 关于伪随机函数

在文件《pgfmathfunctions.random.code-notes.tex》中定义了伪随机函数 rnd, rand, random, 此文件利用的算法参考《*Numerical Recipes in* C*: The Art of Scientific Computing*》*Second Edition*, William H. Press *and* Brian P. Flannery *and* Saul A. Teukolsky *and* William T. Vetterling, *Cambridge University Press*, 1992 年, 第 278 页。

设非负整数 m, a, q, r 满足以下关系：

$$m = aq + r,\ q = [m/a],\ r = m \mod a$$

再假设 $0 \leq r < q$, $0 < z < m - 1$, 则有

$$az \mod m = \begin{cases} a(z \mod q) - r[z/q], \text{ 此式大于 } 0, \text{ 否则等于} \\ a(z \mod q) - r[z/q] + m \end{cases}$$

文件中列出两种选择：

$$\text{第一种：}\begin{cases} m = 2^{31}-1 = 2147483647 \\ a = 16807 \\ q = 127773 \\ r = 2836 \end{cases} \qquad \text{第二种：}\begin{cases} m = 2^{31}-1 = 2147483647 \\ a = 69621 \\ q = 30845 \\ r = 23902 \end{cases}$$

文件使用第二种。$m = 2147483647$ 是 TEX 计数器所能保存的最大整数。z 作为“种子”使用，文件中的种子实际上都是正整数。得到伪随机数的过程是个固定的算法，此算法利用“种子”做计算。当种子不变时，计算出来的伪随机数也不变。要得到不同的伪随机数就要不断地改变种子值。所以在计算过程中种子值可能属于某个数列 $\{z_0, z_1, \cdots\}$. 种子值保存在宏 \pgfmath@rnd@z 中，其默认值为 \time×\year.

如果用户自己指定种子的值为 z_0:

```
\pgfmathsetseed{z_0}% 此命令将宏 \pgfmath@rnd@z 的值设为 z_0 的整数部分
```

那么之后可以使用命令 \pgfmathgeneratepseudorandomnumber 得到另一个种子值 $z_1 = (az_0 \mod m)$, 此命令利用 z_0 计算出 z_1, 并将宏 \pgfmath@rnd@z 的值改为 z_1, 这样得到的种子也可以看作是伪随机数。

\pgfmathgeneratepseudorandomnumber

此命令的定义是

```
\def\pgfmathgeneratepseudorandomnumber{%
    \begingroup%
        \c@pgfmath@counta=\pgfmath@rnd@z%
        \c@pgfmath@countb=\pgfmath@rnd@z%
        \c@pgfmath@countc=\pgfmath@rnd@q%
        \divide\c@pgfmath@counta by\c@pgfmath@countc%
        \multiply\c@pgfmath@counta by-\c@pgfmath@countc%
        \advance\c@pgfmath@counta by\c@pgfmath@countb
        \c@pgfmath@countc=\pgfmath@rnd@a%
        \multiply\c@pgfmath@counta by\c@pgfmath@countc%
        \c@pgfmath@countc=\pgfmath@rnd@q%
        \divide\c@pgfmath@countb by\c@pgfmath@countc%
        \c@pgfmath@countc=\pgfmath@rnd@r%
        \multiply\c@pgfmath@countb by\c@pgfmath@countc%
        \advance\c@pgfmath@counta by-\c@pgfmath@countb%
        \ifnum\c@pgfmath@counta<0\relax%
            \c@pgfmath@countb=\pgfmath@rnd@m%
            \advance\c@pgfmath@counta by\c@pgfmath@countb%
        \fi%
        \xdef\pgfmath@rnd@z{\the\c@pgfmath@counta}%
    \endgroup%
    \edef\pgfmathresult{\pgfmath@rnd@z}%
}
```

此命令全局地重定义 \pgfmath@rnd@z, 然后把 \pgfmath@rnd@z 的值保存到 \pgfmathresult 中。设 z 是当前的 \pgfmath@rnd@z 的值，另外给出 3 个数值 a, r, q, 计算

$$s = \left[\left[\frac{z}{q}\right](-q) + z\right] a - \left[\frac{z}{q}\right] r$$

如果 $s \geq 0$, 则 \pgfmath@rnd@z 的值就是 s; 否则\pgfmath@rnd@z 的值就是 s+\pgfmath@rnd@m.

函数 rnd, rand, random 都会调用命令 \pgfmathgeneratepseudorandomnumber, 例如函数 rnd 的定义是：

```
\pgfmathdeclarefunction{rnd}{0}{%
    \begingroup%
        \pgfmathgeneratepseudorandomnumber%
        \c@pgfmath@counta\pgfmathresult%
        \c@pgfmath@countb\c@pgfmath@counta%
        \divide\c@pgfmath@countb100001\relax% To get one.
        \multiply\c@pgfmath@countb-100001\relax%
        \advance\c@pgfmath@countb\c@pgfmath@counta%
        \advance\c@pgfmath@countb1000000\relax%
        \expandafter\pgfmathrnd@@\the\c@pgfmath@countb\pgfmath@%
        \pgfmath@returnone\pgfmath@x%
    \endgroup%
}%

\def\pgfmathrnd@@#1#2#3\pgfmath@{%
    \edef\pgfmath@temp{#2.#3}%
    \pgfmath@x=\pgfmath@temp pt\relax%
}%
```

3.8.5 计算倒数的函数

文件《pgfmathfunctions.basic.code.tex》中定义了 `\pgfmathreciprocal`.

下面看一下 `\pgfmathreciprocal{`$\langle x\rangle$`}` 的处理。

假设参数 $\langle x\rangle$ 非负。参数 $\langle x\rangle$ 可能是可展开的，把 $\langle x\rangle$ 的展开值记为 $\langle exp\ x\rangle$，它应该是形式为 $[x].p_1p_2p_3p_4p_5\cdots$ 这样的定点数，其中 $[x]\leqslant 16383$ 表示整数部分。

1. 用 `\begingroup` 开启一个组
2. 把 $\langle exp\ x\rangle$ 保存到尺寸寄存器 `\pgfmath@x` 中，此时 `\pgfmath@x` 的值应该是 $[x].p_1p_2p_3p_4p_5$pt 这样的、至多有 5 位小数、至少有一位小数的定点数尺寸。
3. 如果 `\pgfmath@x` 的值是 0pt, 则报错：

   ```
   \pgfmath@error{You asked me to calculate `1/#1', but I cannot divide any number by
   ↪  zero}{}%
   ```

4. 如果 `\pgfmath@x` 的值不是 0pt, 则

   `\pgfmathreciprocal@@`$[x].p_1p_2p_3p_4p_5$`0000000\pgfmath@`

   命令 `\pgfmathreciprocal@@...\pgfmath@` 的处理是：检查 $[x].p_1p_2p_3p_4p_5$ 的小数部分是不是 0，

   - 如果 $[x].p_1p_2p_3p_4p_5$ 的小数部分是 0，则

     ```
     \pgfmath@x1pt\relax%
     \divide\pgfmath@x#1\relax%
     ```

   - 如果 $[x].p_1p_2p_3p_4p_5$ 的小数部分不是 0，则
     - 如果 $[x]>100$, 记 $[x]=z_5z_4z_3z_2z_1$, $a_0=z_5z_4z_3z_2z_1p_1p_2p_3$ (一个至多 8 位的整数)，$b_0=10^9$, 注意此时 $a_0>10^5$, a_0 的最小可能值是 101000, a_0 的最大可能值是 16383999.
       (a) $b_1=\left[\frac{b_0}{a_0}\right]$, 此时 $b_1<10^4$
       (b) $a_1=\left[\frac{b_1}{10^4}\right]$, 此时 $a_1=0$
       (c) 赋值 `\pgfmath@x=`$a_1$pt, 也就是 `\pgfmath@x=`0pt
       (d) $a_2=a_1\cdot(-10^4)=0$
       (e) $b_2=b_1+a_2=b_1$
       (f) 赋值 `\pgfmath@y=`$b_2$pt
       (g) 赋值 `\pgfmath@y=`$\frac{\texttt{\pgfmath@y}}{10^6}$

(h) 赋值 \pgfmath@x=\pgfmath@x + \pgfmath@y, 也就是 \pgfmath@x=\pgfmath@y

– 如果 $[x] \leqslant 100$, 记 $[x] = 00z_3z_2z_1$ (为了便于比较，写法以 0 开头), $a_0 = z_3z_2z_1p_1p_2p_3p_4p_5$ (一个至多 8 位的整数)，$b_0 = 10^9$, 注意此时 a_0 的最小可能值是 1, a_0 的最大可能值是 10099999.

(a) $b_1 = \left[\frac{b_0}{a_0}\right] = \beta_9\beta_8\beta_7\beta_6\beta_5\beta_4\beta_3\beta_2\beta_1$

(b) $a_1 = \left[\frac{b_1}{10^4}\right] = \beta_9 \cdots \beta_5$

(c) 赋值 \pgfmath@x=$a_1$pt

(d) $a_2 = [a_1 \cdot (-10^4)] = -\beta_9 \cdots \beta_5 0000$

(e) $b_2 = b_1 + a_2 = \beta_4\beta_3\beta_2\beta_1$

(f) 赋值 \pgfmath@y=$b_2$pt

(g) 赋值 \pgfmath@y=0.1 · \pgfmath@y

(h) 赋值 \pgfmath@y=0.1 · \pgfmath@y

(i) 赋值 \pgfmath@y=0.1 · \pgfmath@y

(j) 赋值 \pgfmath@y=0.1 · \pgfmath@y

(k) 赋值 \pgfmath@x=\pgfmath@x + \pgfmath@y = $\beta_9\beta_8\beta_7\beta_6\beta_5.\beta_4\beta_3\beta_2\beta_1$

5. 执行

```
\pgfmath@returnone\pgfmath@x%
\endgroup%
```

用 \endgroup 结束组，并把 \pgfmath@x 的数值部分保存到 \pgfmathresult 中。

3.8.6 除法函数

在除法函数 divide 的处理中用到一个宏

```
\def\pgfmath@small@number{0.00002}
```

宏 \pgfmath@small@number 作为“阈值”使用，记之为 $\langle s \rangle$.

\pgfmathdivide{$\langle x \rangle$}{$\langle y \rangle$} 的处理：

1. 用 \begingroup 开启一个组
2. 用寄存器保存 $\langle x \rangle$, $\langle y \rangle$

```
\pgfmath@x=⟨x⟩pt\relax%
\pgfmath@y=⟨y⟩pt\relax%
```

3. \let\pgfmath@sign=\pgfmath@empty
4. 如果 \pgfmath@y 的值是 0pt, 则报错；否则继续。
5. 执行

```
\afterassignment\pgfmath@xa%
\c@pgfmath@counta\the\pgfmath@y\relax%
```

将 \pgfmath@y 的整数部分保存到计数器 \c@pgfmath@counta, 小数部分保存到 \pgfmath@xa.

6. 如果 \pgfmath@xa 的值是 0pt, 则

```
\divide\pgfmath@x by\c@pgfmath@counta%
\pgfmath@returnone\pgfmath@x%
\endgroup%
```

用 \endgroup 结束组，并把尺寸寄存器 \pgfmath@x 的值的数值部分保存到 \pgfmathresult 中，结束计算。

如果 \pgfmath@xa 的值不是 0pt, 则把 $\frac{\langle x \rangle}{\langle y \rangle}$ 的符号保存到 \pgfmath@sign, 并且，如果 \pgfmath@x

中的尺寸是负的，则变成正的；如果 \pgfmath@y 中的尺寸是负的，则变成正的。继续。

7. ▶ 如果 \pgfmath@y 的值小于 1pt, 则

```
\pgfmathreciprocal@{\pgfmath@tonumber{\pgfmath@y}}%
\pgfmath@x=\pgfmath@sign\pgfmathresult\pgfmath@x%
\pgfmath@returnone\pgfmath@x%
\endgroup%
```

得到结果 \pgfmathresult, 结束计算。

▶ 如果 \pgfmath@y 的值不小于 1pt, 则

(a) 定义 \def\pgfmathresult{0}

(b) 设置真值 \pgfmath@divide@periodtrue, 即需要在 \pgfmathresult 中添加小数点。

(c) \c@pgfmath@counta=0\relax

(d) 执行 \pgfmathdivide@@, 此命令：

假设 ⟨*s*⟩ 是某个预先给定的很小的正数值 (宏 \pgfmath@small@number),检查是否 ⟨*s*⟩pt<\pgfmath@x 并且 ⟨*s*⟩pt<\pgfmath@y:

如果不是，则什么也不做；

如果是，则检查是否 \pgfmath@y>\pgfmath@x:

- 如果 \pgfmath@y>\pgfmath@x,

  i. 检查是否需要在 \pgfmathresult 中插入小数点：

  ```
  \ifpgfmath@divide@period%
      \expandafter\def\expandafter\pgfmathresult\expandafter{
      ↪  \pgfmathresult.}%
      \pgfmath@divide@periodfalse%
  \fi%
  ```

  此时保存在 \pgfmathresult 中的是“0.⋯”这种形式的一串符号，并且有真值 \pgfmath@divide@periodfalse.

  ii. 执行 \pgfmathdivide@dimenbyten\pgfmath@y, 此命令：

  假设 \pgfmath@y 的值是 $z_5z_4z_3z_2z_1.p_1p_2p_3p_4p_5$pt, 计算 $\left[\frac{[y]}{10}\right]\cdot(-10)+[y]=z_1$, 重新赋值 \pgfmath@y= $z_5z_4z_3z_2.z_1p_1p_2p_3p_4p_5$pt, 大约让 \pgfmath@y 的值缩小 10 倍。

  iii. 再次检查是否 \pgfmath@y>\pgfmath@x,

  – 如果 \pgfmath@y>\pgfmath@x, 则定义

  ```
  \expandafter\def\expandafter\pgfmathresult\expandafter{
  ↪  \pgfmathresult0}
  ```

  此时保存在 \pgfmathresult 中的是“0.0⋯”这种形式的一串符号。然后再从头重复执行 \pgfmathdivide@@, 直到 \pgfmath@y⩽\pgfmath@x.

  – 如果并非 \pgfmath@y>\pgfmath@x, 则再从头执行 \pgfmathdivide@@.

- 如果并非 \pgfmath@y>\pgfmath@x, 则

  i. 转换

  ```
  \c@pgfmath@counta=\pgfmath@x%
  \c@pgfmath@countb=\pgfmath@y%
  ```

  有转换关系 1pt = 65536sp, 所以保存在计数器 \c@pgfmath@counta, \c@pgfmath@countb 中的整数可能较大。例如，下面代码得到一个 10 位整数：

```
1073741823
```

```
\newdimen\aaaaa%
\newcount\bbbbb%
\aaaaa=16383.99999pt%
\bbbbb=\aaaaa%
\the\bbbbb
```

ii. 赋值 \c@pgfmath@counta= $\left[\frac{\text{\c@pgfmath@counta}}{\text{\c@pgfmath@countb}}\right]$, 这是 $\left[\frac{\text{\pgfmath@x}}{\text{\pgfmath@y}}\right]$.

iii. 赋值 \pgfmath@x= \pgfmath@x − \c@pgfmath@counta · \pgfmath@y, 这样就有 \pgfmath@x<\pgfmath@y.

iv. 定义

```
\def\pgfmath@next{%
    \toks0=\expandafter{\pgfmathresult}%
    \edef\pgfmathresult{\the\toks0 \the\c@pgfmath@counta}%
}%
```

这个 \pgfmath@next 可能会在后面被执行。

v. 检查真值 \ifpgfmath@divide@period,

– 如果其值是 true, 则什么也不做。

– 如果其值是 false, 则检查 \c@pgfmath@counta 的值是否 > 9.

有可能导致 \c@pgfmath@counta> 9 的情况，例如：

```
yes
10
```

```
\newcount\aaaaaa \newcount\bbbbbb
\newdimen\cccccc \newdimen\dddddd
\cccccc=9.00000pt \dddddd=9.00001pt
\ifdim \cccccc<\dddddd yes \else no \fi \par
\aaaaaa\cccccc
\dddddd=0.900001pt
\bbbbbb\dddddd
\divide\aaaaaa\bbbbbb
\the\aaaaaa
```

∗ 如果大于 9, 则

⟨1⟩ 执行

```
\expandafter\pgfmathdivide@advance@last@digit\pgfmathresult
↪  CCCCC\@@
```

此命令先赋值 \pgfmath@ya=\pgfmathresult pt, 再检查 \pgfmathresult 所保存的数值形式：

如果是 $[r]$. 这种形式，则定义 \pgfmath@xa=1pt

如果是 $[r].r_1$ 这种形式，则定义 \pgfmath@xa=0.1pt

如果是 $[r].r_1r_2$ 这种形式，则定义 \pgfmath@xa=0.01pt

如果是 $[r].r_1r_2r_3$ 这种形式，则定义 \pgfmath@xa=0.001pt

如果是 $[r].r_1r_2r_3r_4$ 这种形式，则定义 \pgfmath@xa=0.0001pt

如果是其他形式，则定义 \pgfmath@xa=0.00001pt

再把 \pgfmath@ya + \pgfmath@xa 的数值部分保存到 \pgfmathresult 中。

⟨2⟩ 赋值 \c@pgfmath@counta= \c@pgfmath@counta − 10

⟨3⟩ 检查 \c@pgfmath@counta 的值，

如果 \c@pgfmath@counta 的值是 0, 则重定义

```
\let\pgfmath@next=\relax
```

否则，什么也不做。

* 如果不大于 9, 则什么也不做。

vi. 执行前面定义的 \pgfmath@next.

vii. 再从头执行 \pgfmathdivide@@.

命令 \pgfmathdivide@@ 是个循环操作,其终止条件是 ⟨s⟩pt≥\pgfmath@x 或者 ⟨s⟩pt≥\pgfmath@y.

(e) 赋值 \pgfmath@x=\pgfmath@sign\pgfmathresult pt\relax

(f) 执行

```
\pgfmath@returnone\pgfmath@x%
\endgroup%
```

用 \endgroup 结束组，并把尺寸寄存器 \pgfmath@x 的值的数值部分保存到 \pgfmathresult 中，结束计算。

3.8.7 (fpu 版) 加法函数

普通的 PGF 加法函数 add 在《pgfmathfunctions.basic.code.tex》中定义：

```
\pgfmathdeclarefunction{add}{2}{%
    \begingroup%
    \pgfmath@x=#1pt\relax%
    \pgfmath@y=#2pt\relax%
    \advance\pgfmath@x by\pgfmath@y%
    \pgfmath@returnone\pgfmath@x%
  \endgroup%
}
```

可见这是利用尺寸寄存器 \pgfmath@x, \pgfmath@y 和 TEX 命令 \advance 定义的。

在 fpu 库中定义的加法函数 \pgfmathfloatadd@{⟨x⟩}{⟨y⟩} 如下处理：

1. 用 \begingroup 开启一个组
2. 执行

```
\pgfmathfloattoextentedprecision@a{⟨x⟩}%
\let\pgfmathfloat@arga=\pgfmathresult
```

参考 \pgfmathfloattoextentedprecision@a→P. 585, 这是对参数 ⟨x⟩ 的尾数和指数做调整，调整后的尾数保存在 \pgfmathresult 和 \pgfmathfloat@arga 中。

3. 执行

```
\pgfmathfloattoextentedprecision@b{⟨y⟩}%
\let\pgfmathfloat@argb=\pgfmathresult
```

注意，如果此时参数 ⟨y⟩ 是 \pgfmathresult, 那么会导致错误，因为此时的 \pgfmathresult 保存的是参数 ⟨x⟩ 的调整后的尾数，是个定点数。

4. 设置真值 \pgfmathfloatcomparisontrue.
5. 检查标记 \pgfmathfloat@a@S 的值：
   - 若其值是 0，则

```
\edef\pgfmathresult{⟨y⟩}%
\pgfmathfloatcomparisonfalse
```

   - 若其值是 1，则什么也不做。
   - 若其值是 2，则保存尾数的相反数 (一个负数)：

```
\edef\pgfmathfloat@arga{-\pgfmathfloat@arga}%
```

   - 若其值是 3, 4, 5, 则

```
\pgfmathfloatcomparisonfalse
\pgfmathfloatcreate{\the\pgfmathfloat@a@S}{0.0}{0}%
```

6. 检查标记 \pgfmathfloat@b@S 的值，做法类似以上。
7. 检查 \ifpgfmathfloatcomparison 的真值，
   - 若其值是 true, 则
     (a) 比较此时的指数 \pgfmathfloat@a@E 和 \pgfmathfloat@b@E,
     – 若 \pgfmathfloat@a@E<\pgfmathfloat@b@E, 则执行

```
\pgfmathfloatadd@shift{\pgfmathfloat@arga}{\pgfmathfloat@a@E}{
↪   \pgfmathfloat@b@E}%
```

此命令重定义 \pgfmathfloat@arga 和 \pgfmathfloat@a@E, 也就是调整这个尾数和指数，使得指数 \pgfmathfloat@a@E = \pgfmathfloat@b@E (将小指数调整为大指数)，其调整步骤是：

i. 赋值 \pgf@xa=\pgfmathfloat@arga pt

ii. 分情况：

* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E = 0, 则什么也不做。
* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E = 1, 则

```
\divide\pgf@xa by10\relax
```

* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E = 2, 则

```
\divide\pgf@xa by100\relax
```

* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E = 3, 则

```
\divide\pgf@xa by1000\relax
```

* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E = 4, 则

```
\divide\pgf@xa by10000\relax
```

* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E = 5, 则

```
\divide\pgf@xa by10000\relax
\divide\pgf@xa by10\relax
```

* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E = 6, 则

```
\divide\pgf@xa by10000\relax
\divide\pgf@xa by100\relax
```

* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E = 7, 则

```
\divide\pgf@xa by10000\relax
\divide\pgf@xa by1000\relax
```

* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E = 8, 则

```
\divide\pgf@xa by10000\relax
\divide\pgf@xa by10000\relax
```

* 若 \pgfmathfloat@b@E − \pgfmathfloat@a@E > 8, 则

```
\pgf@xa=0pt
```

iii. 赋值 \pgfmathfloat@a@E=\pgfmathfloat@b@E.

iv. 重定义 \edef\pgfmathfloat@arga{\pgf@sys@tonumber\pgf@xa}.

– 若并非 \pgfmathfloat@a@E<\pgfmathfloat@b@E, 则

```
\pgfmathfloatadd@shift{\pgfmathfloat@argb}{\pgfmathfloat@b@E}{
↪  \pgfmathfloat@a@E}%
```

做类似的操作。

(b) 执行

```
\pgfmath@basic@add@{\pgfmathfloat@arga}{\pgfmathfloat@argb}%
```

这是用 PGF 的通常的加法函数做加法。

(c) 定义

```
\edef\pgfmathresult{\pgfmathresult e\the\pgfmathfloat@a@E}
```

这是类似科学记数法格式的数。

(d) 执行

```
\expandafter\pgfmathfloatqparsenumber\expandafter{\pgfmathresult}%
```

获得浮点数格式的数值，保存在 \pgfmathresult 中。

- 若其值是 false, 则什么也不做。

8. 执行

```
\pgfmath@smuggleone\pgfmathresult
\endgroup
```

结束之前开启的组，并把计算结果 \pgfmathresult 放到组外。

可见，除了开始的两个调整尾数和指数的操作外，命令 \pgfmathfloatadd@ 都是用 TEX 的寄存器做运算的。

下面例子表明，对于非常小的数值，命令 \pgfmathfloatadd@ 与通常的 PGF 加法函数 add 相比，能更好地利用有效数字：

```
\pgfmathadd{0.000005555555}{0.000001234123}\pgfmathresult\par
{
    \pgfkeys{/pgf/fpu}
    \pgfmathadd{0.000005555555}{0.000001234123}\pgfmathresult
}
```

```
0.0
1Y6.7896779e-6]
```

下面例子表明，对于较大的数值，命令 \pgfmathfloatadd@ 与通常的 PGF 加法函数 add 相比，可能会丢失有效数字：

```
\pgfmathadd{16000.00011}{0.00022}\pgfmathresult\par
{
    \pgfkeys{/pgf/fpu}
    \pgfmathadd{16000.00011}{0.00022}\pgfmathresult\par
    \pgfmathfloatsetextprecision{3}
    \pgfmathadd{16000.00011}{0.00022}\pgfmathresult
}
```

```
16000.00032
1Y1.6e4]
1Y1.60000003e4]
```

当然，多数情况下，丢掉的部分几乎没有实际的影响。

一个例子。

Fibonacci 数列的前几项是 (从第 0 项开始)：

```
% Fibonacci 数列的第 0 项到第 44 项
1,
1, 2, 3, 5, 8,
13, 21, 34, 55, 89,
144, 233, 377, 610, 987,
1597, 2584, 4181, 6765, 10946,
```

```
17711, 28657, 46368, 75025, 121393,
196418, 317811, 514229, 832040, 1346269,
2178309, 3524578, 5702887, 9227465, 14930352,
24157817, 39088169, 63245986, 102334155, 165580141,
267914296, 433494437, 701408733, 1134903170
```

下面代码得到 Fibonacci 数列的第 19, 20 项：

6765, 10946

```
\def\FibonacciFirst{1}
\def\FibonacciSecond{1}
\foreach \i in {2,...,20}
{
  \pgfmathsetmacro{\FibonacciTemp}{int(\FibonacciFirst + \FibonacciSecond)}
  \xdef\FibonacciFirst{\FibonacciSecond}
  \xdef\FibonacciSecond{\FibonacciTemp}
}
\FibonacciFirst, \FibonacciSecond
```

上面代码至多得到 Fibonacci 数列的第 20 项，因为 $6765 + 10946 = 17711 > 16383.99999$, 已经超出 TEX 关于单位 pt 的运算范围。

下面：

第 35 项：14,930,352
第 36 项：24,157,816

```
\def\FibonacciFirst{5702887}
\def\FibonacciSecond{9227465}
\pgfmathfloatparsenumber{\FibonacciFirst}
\xdef\FibonacciFirstTemp{\pgfmathresult}
\pgfmathfloatparsenumber{\FibonacciSecond}
\xdef\FibonacciSecondTemp{\pgfmathresult}
\foreach \i in {35,36}
{
    \pgfmathfloatadd{\FibonacciFirstTemp}{\FibonacciSecondTemp}%
    \xdef\FibonacciFirstTemp{\FibonacciSecondTemp}%
    \xdef\FibonacciSecondTemp{\pgfmathresult}%
%    \pgfmathprintnumber[fixed,precision=100]{\FibonacciFirstTemp}\\
    \noindent 第 \i 项：\pgfmathprintnumber[fixed,precision=100]{
    ↪  \FibonacciSecondTemp}\\
}%
```

其中的 14930352 是 \pgfmathfloatadd 对两个 7 位整数的运算结果；24157816 是 \pgfmathfloatadd 对一个 7 位整数与一个 8 位整数的运算结果，可见在计算结果的最后一位上出现偏差。

使用 \pgfmathfloatsetextprecision{3} 可以纠正这个偏差：

第 35 项：14,930,351.99
第 36 项：24,157,817

```
\pgfmathfloatsetextprecision{3}
\def\FibonacciFirst{5702887}
\def\FibonacciSecond{9227465}
\pgfmathfloatparsenumber{\FibonacciFirst}
\xdef\FibonacciFirstTemp{\pgfmathresult}
\pgfmathfloatparsenumber{\FibonacciSecond}
\xdef\FibonacciSecondTemp{\pgfmathresult}
\foreach \i in {35,36}
{
    \pgfmathfloatadd{\FibonacciFirstTemp}{\FibonacciSecondTemp}%
    \xdef\FibonacciFirstTemp{\FibonacciSecondTemp}%
    \xdef\FibonacciSecondTemp{\pgfmathresult}%
%    \pgfmathprintnumber[fixed,precision=100]{\FibonacciFirstTemp}\\
    \noindent 第 \i 项：\pgfmathprintnumber[fixed,precision=100]{
    ↪  \FibonacciSecondTemp}\\
}%
```

但是又导致第一个计算结果出现 0.01 的偏差，这个偏差来自 TEX 的尺寸寄存器运算：

14930.35199pt
```
\newdimen\aaaa%
\newdimen\bbbb%
\aaaa=5702.887pt%
\bbbb=9227.465pt%
\advance \aaaa\bbbb%
\the\aaaa%
```

2415.78171pt
```
\newdimen\aaaa%
\newdimen\bbbb%
\aaaa=1493.035199pt%
\bbbb=922.7465pt%
\advance \aaaa\bbbb%
\the\aaaa%
```

2415.7817pt
```
\newdimen\aaaa%
\newdimen\bbbb%
\aaaa=1493.035199pt%
\bbbb=9227.465pt%
\divide\bbbb by10
\advance \aaaa\bbbb%
\the\aaaa%
```

所以当对超过 7 位的大整数做加减法时应该多加注意。

另外，宏 \pgfmathresult 可以用作 \pgfmathfloatadd 的第 1 个参数，但最好不要用作第 2 个参数。

3.8.8 (fpu 版) 减法函数

PGF 的普通的减法函数是：

```
\pgfmathdeclarefunction{subtract}{2}{%
    \begingroup%
    \pgfmath@x=#1pt\relax%
    \pgfmath@y=#2pt\relax%
    \advance\pgfmath@x by-\pgfmath@y%
    \pgfmath@returnone\pgfmath@x%
  \endgroup%
}
```

Fpu 版的减法函数是：

```
\def\pgfmathfloatsubtract@#1#2{%
    \begingroup
    \edef\pgfmathresult{#2}%
    \expandafter\pgfmathfloat@decompose@tok\pgfmathresult\relax\pgfmathfloat@b@S
    ↪  \pgfmathfloat@a@Mtok\pgfmathfloat@b@E
    \ifcase\pgfmathfloat@b@S
        \edef\pgfmathresult{#1}%
    \or
        \pgfmathfloatcreate{2}{\the\pgfmathfloat@a@Mtok}{\the\pgfmathfloat@b@E}%
        \let\pgfmathfloatsub@arg=\pgfmathresult
        \pgfmathfloatadd@{#1}{\pgfmathfloatsub@arg}%
    \or
        \pgfmathfloatcreate{1}{\the\pgfmathfloat@a@Mtok}{\the\pgfmathfloat@b@E}%
        \let\pgfmathfloatsub@arg=\pgfmathresult
        \pgfmathfloatadd@{#1}{\pgfmathfloatsub@arg}%
    \else
        \pgfmathfloatcreate{\the\pgfmathfloat@b@S}{0.0}{0}%
    \fi
```

```
    \pgfmath@smuggleone\pgfmathresult
    \endgroup
}%

\let\pgfmathfloatsubtract=\pgfmathfloatsubtract@
```

可见命令 \pgfmathfloatsubtract@ 会多次重定义 \pgfmathresult, 所以尽量不要把宏 \pgfmathresult 用作这个命令的参数；如果要用，那么最好用作这个命令的第二个参数。

3.8.9 (fpu 版) 乘法函数

普通的 PGF 的乘法函数 multiply 在《pgfmathfunctions.basic.code.tex》中定义：

```
\pgfmathdeclarefunction{multiply}{2}{%
  \begingroup%
    \pgfmath@x=#1pt\relax%
    \pgfmath@y=#2pt\relax%
    \pgfmath@x=\pgfmath@tonumber{\pgfmath@y}\pgfmath@x%
    \pgfmath@returnone\pgfmath@x%
  \endgroup%
}
```

其中的命令 \pgfmath@tonumber 等效于 \pgf@sys@tonumber→P. 202.

在 fpu 库中定义的乘法函数 \pgfmathfloatmultiply@{⟨x⟩}{⟨y⟩} 的处理是：

1. 用 \begingroup 开启一个组。
2. 执行 \pgfmathfloatsetextprecision→P. 585{1} 来规定对尾数的调整。
3. 执行

   ```
   \pgfmathfloattoextentedprecision@a{⟨x⟩}%
   \let\pgfmathfloat@arga=\pgfmathresult
   ```

   参考 \pgfmathfloattoextentedprecision@a→P. 585, 这是对参数 ⟨x⟩ 的尾数和指数做调整，调整后的尾数保存在 \pgfmathresult 和 \pgfmathfloat@arga 中。
4. 执行

   ```
   \pgfmathfloattoextentedprecision@b{⟨y⟩}%
   \let\pgfmathfloat@argb=\pgfmathresult
   ```

   注意，如果此时参数 ⟨y⟩ 是 \pgfmathresult, 那么会导致错误，因为此时的 \pgfmathresult 保存的是参数 ⟨x⟩ 的调整后的尾数，是个定点数。
5. 设置真值 \pgfmathfloatcomparisontrue.
6. 检查标记 \pgfmathfloat@a@S 的值：
   - 若其值是 0, 则

     ```
     \pgfmathfloatcreate{0}{0.0}{0}%
     \pgfmathfloatcomparisonfalse
     ```

   - 若其值是 1, 则检查标记 \pgfmathfloat@b@S 的值：
     - 若其值是 0, 则

       ```
       \pgfmathfloatcreate{0}{0.0}{0}%
       \pgfmathfloatcomparisonfalse
       ```

     - 若其值是 1, 则 \def\pgfmathresult@S{1}
     - 若其值是 2, 则 \def\pgfmathresult@S{2}
     - 若其值是 3, 4, 5, 则

```
\expandafter\pgfmathfloatcreate\the\pgfmathfloat@b@S{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

- 若其值是 2, 则检查标记 \pgfmathfloat@b@S 的值:
  - 若其值是 0, 则

```
\pgfmathfloatcreate{0}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 1, 则 \def\pgfmathresult@S{2}
  - 若其值是 2, 则 \def\pgfmathresult@S{1}
  - 若其值是 3, 则

```
\pgfmathfloatcreate{3}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 4, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 5, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

- 若其值是 3, 则

```
\pgfmathfloatcreate{3}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

- 若其值是 4, 则检查标记 \pgfmathfloat@b@S 的值:
  - 若其值是 0, 则

```
\pgfmathfloatcreate{0}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 1, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 2, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 3, 则

```
\pgfmathfloatcreate{3}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 4, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 5, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

- 若其值是 5, 则检查标记 \pgfmathfloat@b@S 的值:
  - 若其值是 0, 则

```
\pgfmathfloatcreate{0}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

– 若其值是 1, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

– 若其值是 2, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

– 若其值是 3, 则

```
\pgfmathfloatcreate{3}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

– 若其值是 4, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

– 若其值是 5, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

7. 检查 \ifpgfmathfloatcomparison 的真值：
   - 若其值是 true, 则
     (a) 计算

```
\pgfmath@basic@multiply@{\pgfmathfloat@arga}{\pgfmathfloat@argb}
```

这是用普通的 PGF 函数做乘法。

(b) 计算指数 \advance\pgfmathfloat@a@E by\pgfmathfloat@b@E

(c) 定义

```
\edef\pgfmathresult{\pgfmathresult e\the\pgfmathfloat@a@E}
```

这是类似科学记数法格式的数。

(d) 执行

```
\expandafter\pgfmathfloatqparsenumber\expandafter{\pgfmathresult}
```

将浮点数保存在 \pgfmathresult 中。

(e) 执行

```
\expandafter\pgfmathfloat@decompose@tok\pgfmathresult\relax
↪  \pgfmathfloat@a@S\pgfmathfloat@a@Mtok\pgfmathfloat@a@E
```

获取标记、尾数、指数。

(f) 执行

```
\pgfmathfloatcreate{\pgfmathresult@S}{\the\pgfmathfloat@a@Mtok}{\the
↪  \pgfmathfloat@a@E}
```

将乘积——是个浮点数——保存在 \pgfmathresult 中，其标记是 \pgfmathresult@S.

   - 若其值是 false, 则什么也不做。
8. 执行

```
\pgfmath@smuggleone\pgfmathresult
\endgroup
```

可见命令 \pgfmathfloatmultiply@ 先将其参数的尾数的小数点右移一位，指数减去 1, 此时尾数的整数部分至多有 2 位数；然后再用 TEX 的尺寸寄存器对尾数做乘法，乘积的整数部分至多有 4 位数，这样就不会出现“dimension is too large”的错误。

若参数的尾数的小数部分的位数超过 6 位，就可能丢失有效数字。

如果把宏 \pgfmathresult 用作这个命令的参数，那么最好用作这个命令的第一个参数。

3.8.10 fpu 版的除法函数

在 fpu 库中定义的除法函数 \pgfmathfloatdivide@{⟨x⟩}{⟨y⟩} 的处理是：

1. 用 \begingroup 开启一个组
2. 用 \pgfmathfloatsetextprecision{1} 规定对尾数的调整。
3. 定义 \edef\pgfmathfloat@arga{⟨x⟩}
4. 执行

```
\pgfmathfloattoextentedprecision@a{\pgfmathfloat@arga}%
\let\pgfmathfloat@arga=\pgfmathresult
```

调整第一个参数的尾数和指数，尾数的小数点右移 1 位，指数减去 1, 尾数保存在 \pgfmathresult 和 \pgfmathfloat@arga 中。

5. 定义 \edef\pgfmathfloat@argb{⟨y⟩}
6. 执行

```
\pgfmathfloattoextentedprecision@b{\pgfmathfloat@argb}%
\let\pgfmathfloat@argb=\pgfmathresult
```

7. 设置 \pgfmathfloatcomparisontrue
8. 检查标记 \pgfmathfloat@a@S 的值：
   - 若其值是 0, 则

     ```
     \pgfmathfloatcreate{0}{0.0}{0}%
     \pgfmathfloatcomparisonfalse
     ```

   - 若其值是 1, 则检查标记 \pgfmathfloat@b@S 的值：
     - 若其值是 0, 则

       ```
       \pgfmathfloatcreate{4}{0.0}{0}%
       \pgfmathfloatcomparisonfalse
       ```

     - 若其值是 1, 则 \def\pgfmathresult@S{1}
     - 若其值是 2, 则 \def\pgfmathresult@S{2}
     - 若其值是 3, 则

       ```
       \pgfmathfloatcreate{3}{0.0}{0}%
       \pgfmathfloatcomparisonfalse
       ```

     - 若其值是 4, 5, 则

       ```
       \pgfmathfloatcreate{0}{0.0}{0}%
       \pgfmathfloatcomparisonfalse
       ```

   - 若其值是 2, 则检查标记 \pgfmathfloat@b@S 的值：
     - 若其值是 0, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 1, 则 `\def\pgfmathresult@S{2}`
  - 若其值是 2, 则 `\def\pgfmathresult@S{1}`
  - 若其值是 3, 则

```
\pgfmathfloatcreate{3}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 4, 5, 则

```
\pgfmathfloatcreate{0}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

- 若其值是 3, 则

```
\pgfmathfloatcreate{3}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

- 若其值是 4, 则检查标记 `\pgfmathfloat@b@S` 的值：
  - 若其值是 0, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 1, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 2, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 3, 则

```
\pgfmathfloatcreate{3}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 4, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 5, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

- 若其值是 5, 则检查标记 `\pgfmathfloat@b@S` 的值：
  - 若其值是 0, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 1, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

  - 若其值是 2, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

– 若其值是 3, 则

```
\pgfmathfloatcreate{3}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

– 若其值是 4, 则

```
\pgfmathfloatcreate{5}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

– 若其值是 5, 则

```
\pgfmathfloatcreate{4}{0.0}{0}%
\pgfmathfloatcomparisonfalse
```

9. 检查 \ifpgfmathfloatcomparison 的真值：
   - 若其值是 true, 则
     (a) 计算

```
\pgfmath@basic@divide@{\pgfmathfloat@arga}{\pgfmathfloat@argb}%
```

这是用普通的 PGF 函数做除法。

(b) 计算指数 \advance\pgfmathfloat@a@E by-\pgfmathfloat@b@E

(c) 定义

```
\edef\pgfmathresult{\pgfmathresult e\the\pgfmathfloat@a@E}
```

这是类似科学记数法格式的数。

(d) 执行

```
\expandafter\pgfmathfloatqparsenumber\expandafter{\pgfmathresult}
```

将浮点数保存在 \pgfmathresult 中。

(e) 执行

```
\expandafter\pgfmathfloat@decompose@tok\pgfmathresult\relax
↪  \pgfmathfloat@a@S\pgfmathfloat@a@Mtok\pgfmathfloat@a@E
```

获取标记、尾数、指数。

(f) 执行

```
\pgfmathfloatcreate{\pgfmathresult@S}{\the\pgfmathfloat@a@Mtok}{\the
↪  \pgfmathfloat@a@E}
```

将结果——是个浮点数——保存在 \pgfmathresult 中，其标记是 \pgfmathresult@S.

   - 若其值是 false, 则什么也不做。

10. 执行

```
\pgfmath@smuggleone\pgfmathresult
\endgroup
```

除法函数的处理过程类似乘法函数。

宏 \pgfmathresult 可以用作 \pgfmathfloatdivide 的第 1 个参数，但最好不要用作第 2 个参数。

总结，宏 \pgfmathresult 可以用作：

- \pgfmathfloatadd 的第 1 个参数
- \pgfmathfloatsubtract 的第 2 个参数

- \pgfmathfloatmultiply 的第 1 个参数
- \pgfmathfloatdivide 的第 1 个参数

3.8.11 fpu 版的倒数函数

命令 \pgfmathfloatreciprocal@ 是用 \pgfmathfloatdivide@ 定义的：

```
\def\pgfmathfloatreciprocal@#1{%
    \begingroup
    % FIXME optimize
    \edef\pgfmathfloat@loc@TMPa{#1}%
    \pgfmathfloatcreate{1}{1.0}{0}%
    \pgfmathfloatdivide@{\pgfmathresult}{\pgfmathfloat@loc@TMPa}%
    \pgfmath@smuggleone\pgfmathresult
    \endgroup
}%
```

3.9 输出数值的格式

本节介绍文件《pgfmathfloat.code.tex》提供的方法，作为数学引擎的一部分，这个文件总是会被 PGF 自动载入。注意 fpu 库的文件《pgflibraryfpu.code.tex》不会被自动载入。

输出的数值处在数学模式中，所以使用某些自定义输出格式的选项时，例如，frac TeX=⟨*macro*⟩, dec sep=⟨*text*⟩, mantissa sep=⟨*text*⟩ 等，要记住是在数学模式中使用代码。

3.9.1 基本的命令与选项

\pgfmathprintnumber[⟨*options*⟩]{⟨*x*⟩}

此命令在文件《pgfmathfloat.code.tex》中定义，它要比 \pgfmathprint →P. 112 复杂。本命令是数值输出命令，它使用命令 \pgfmathfloatparsenumber 解析实数 ⟨*x*⟩, 并输出到显示器上。⟨*x*⟩ 可以是定点数，浮点数，或科学计数法格式的数值，可以是很大的数值 (参考 fpu 程序库)。本命令可以与 fpu 程序库的命令一起使用，这与数学引擎的函数计算命令不同。

\pgfmathprintnumberto{⟨*x*⟩}{⟨*macro*⟩}

解析实数 ⟨*x*⟩, 并将它保存在宏 ⟨*macro*⟩ 中，而不是输出到显示器上。

/pgf/number format/fixed (no value)

这个 key 针对 \pgfmathprintnumber, 使得该命令输出的数值的小数部分具有固定位数（位数由选项 precision= 规定，该选项的初始设置是 2 位），多余位数的小数会被四舍五入，即使用定点小数。

4.57 0 0.1 24,415.98 123,456.12

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

上面的例子中使用 \pgfkeys 为之后的输出命令设置选项。也可以使用带选项的输出命令：

4.57 `\pgfmathprintnumber [fixed, precision=2] {4.568}`

/pgf/number format/fixed zerofill={⟨*boolean*⟩} (default true)

该布尔选项决定：当输出数值为定点数且小数部分的位数小于选项 precision= 的规定时，是否用符号 0 来填充。本选项对 /pgf/number format/fixed 以及 /pgf/number format/std → P. 157 都有影响。

4.5600 `\pgfmathprintnumber [fixed, fixed zerofill, precision=4] {4.56}`

/pgf/number format/sci (no value)

这个选项使得输出的数值为科学计数法格式，该格式包括：符号、尾数（mantissa）、幂（exponent，以 10 为底）三部分。注意尾数的整数部分有且只有一位，即个位，个位上的数应当非 0 （如果可能的话）。尾数会参照选项 precision= 或 sci precision= 的规定做舍入。

$4.57 \cdot 10^{0}$ $5 \cdot 10^{-4}$ $1 \cdot 10^{-1}$ $2.44 \cdot 10^{4}$ $1.23 \cdot 10^{5}$

```
\pgfkeys{/pgf/number format/.cd,sci,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

/pgf/number format/sci zerofill={⟨*boolean*⟩} (default true)

该布尔选项决定：当输出数值为科学计数法格式且小数部分的位数小于选项 precision= 的规定时，是否用 0 来填充。

$4.5600 \cdot 10^{0}$ `\pgfmathprintnumber [sci, sci zerofill, precision=4] {4.56}`

/pgf/number format/zerofill={⟨*boolean*⟩} (style, default true)

同时设置 fixed zerofill 和 sci zerofill.

/pgf/number format/std (no value)

/pgf/number format/std=⟨*lower e*⟩ (no default)

/pgf/number format/std=⟨*lower e*⟩:⟨*upper e*⟩ (no default)

这些选项的工作方式是：假设输入的数值是 n, 在科学计数法之下 $n = s \cdot m \cdot 10^{e}$（这里用 s 代表符号，m 代表尾数，e 代表幂指数），那么，当 ⟨*lower e*⟩ $\leqslant e \leqslant$ ⟨*upper e*⟩ 时，输出的数值为 fixed 格式；否则输出的数值为 sci 格式。

- 对于选项 std 来说，⟨*lower e*⟩ 等于 $-\frac{p}{2}$，这里 p 是由选项 precision=p 规定的精度；⟨*upper e*⟩ 等于 4. 即当 $-\frac{p}{2} \leqslant e \leqslant 4$ 时，输出的数值为 fixed 格式；否则输出的数值为 sci 格式。
  注意，如果输出命令不使用选项 fixed 或 sci 来指定输出数值的格式，则默认使用该选项。
- 对于选项 std=⟨*lower e*⟩ 来说，⟨*upper e*⟩ 等于 4.

$5 \cdot 10^{-4}$ `\pgfmathprintnumber [std, precision=2] {5e-4}`

0.0005 `\pgfmathprintnumber [std, precision=8] {5e-4}`

0.0005 `\pgfmathprintnumber [std=-4, precision=4] {5e-4}`

0.001 `\pgfmathprintnumber [std=-4, precision=3] {5e-4}`

$5.2 \cdot 10^5$ `\pgfmathprintnumber[std=-4:4, precision=1]{51.5e4}`

注意上面后两个例子，根据指定的精度，输出结果有所舍入。

/pgf/number format/relative*=⟨*exponent base 10*⟩ (no default)

注意其中的星号“*”. 这个选项设置输出命令。

这里的 ⟨*exponent base 10*⟩ 是个整数。假设 ⟨*exponent base 10*⟩ $= r$，待输出的数值是 n, 将 n 写成 $n = s \cdot M \cdot 10^r$, 注意其中的幂指数是 r; s 代表符号。此时按照选项 `precision=` 规定的精度对 M 做舍入，得到 $\overline{M}$，于是 n 变成 $\overline{n} = s \cdot \overline{M} \cdot 10^r$, 然后参照输出格式选项，例如 `fixed`, `sci`, `std`, 输出 $\overline{n}$.

$1.23457 \cdot 10^8$ $1.23457 \cdot 10^8$

```
\pgfkeys{/pgf/number format/.cd,relative*={3},precision=0}
\pgfmathprintnumber{123456999}\hspace{1em}
\pgfmathprintnumber{123456999.12}
```

在上面的例子中，指定的数量级是 10^3, 记 $123456999 = 123456.999 \cdot 10^3$, 精度为 0, 即将小数部分向个位做舍入，得到 123457, 然后按选项 `std` 的规定输出。

/pgf/number format/every relative (style, no value)

这是个样式，该样式的初始设置是：

```
\pgfkeys{/pgf/number format/every relative/.style=std}
```

/pgf/number format/relative style={⟨*options*⟩} (no default)

等效于 `every relative/.append style=`{⟨*options*⟩}.

/pgf/number format/fixed relative

本选项设置输出命令，其作用是：假设待输出的数值是 n, 所设置的输出数值的精度是 p；从左向右考察 n 的各个位置上的数字，首先找出第一个非零数字，记为 w_1，记 w_1 右侧的数字为 w_2，记 w_2 右侧的数字为 w_3，……直到数字 w_p，然后按“四舍五入”的原则，将 w_p 右侧的数字舍去，得到需要输出的数值 $\overline{n}$.

0.0101 `\pgfmathprintnumber[fixed relative,precision=3]{0.010073452}`

在上面例子中，输出精度是 3，待输出数值 0.010073452 的第 1 个非零数字是 1，因此需要保留的是 0.0100，而将 73452 “四舍五入”，于是得到 0.0101.

本选项会忽略 `/pgf/number format/fixed zerofill` →P. 157.

/pgf/number format/int detect (no value)

本选项设置输出命令，其作用是：检查待输出的数值是否是整数，如果是整数就将它输出为不带小数点的整数，否则输出为科学计数法格式。

15 20 $2.04 \cdot 10^1$ $1 \cdot 10^{-2}$ 0

```
\pgfkeys{/pgf/number format/.cd,int detect,precision=2}
\pgfmathprintnumber{15}\hspace{1em}
\pgfmathprintnumber{20}\hspace{1em}
\pgfmathprintnumber{20.4}\hspace{1em}
\pgfmathprintnumber{0.01}\hspace{1em}
\pgfmathprintnumber{0}
```

\pgfmathifisint{⟨*number constant*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

这个命令检查待输出数值 ⟨*number constant*⟩ 是否为整数，如果是整数就执行 ⟨*true code*⟩, 否则执行 ⟨*false code*⟩.

本命令调用 \pgfmathfloatparsenumber 来解析 ⟨*number constant*⟩, 解析结果会保存在宏 \pgfretval 中。

15 is an int: 15. 15.5 is no int

```
15 \pgfmathifisint{15}{is an int: \pgfretval.}{is no int}\hspace{2em}
15.5 \pgfmathifisint{15.5}{is an int: \pgfretval.}{is no int}
```

/pgf/number format/int trunc (no value)

将待输出的数值的小数部分去掉，无舍入，只保留整数部分，输出之。

4 0 0 24,415 123,456

```
\pgfkeys{/pgf/number format/.cd,int trunc}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

/pgf/number format/frac (no value)

将数值输出为真分数或带分数。真分数、带分数的分数部分的样式，由下面的选项 frac TeX=\⟨*macro*⟩ 来规定。

$\frac{1}{3}$ $\frac{1}{2}$ $\frac{16}{75}$ $\frac{3}{25}$ $\frac{2}{75}$ $-\frac{1}{75}$ $\frac{18}{25}$ $\frac{1}{15}$ $\frac{2}{15}$ $-\frac{1}{75}$ $3\frac{1}{3}$ $1\frac{6787}{28935}$ 1 -6

```
\pgfkeys{/pgf/number format/frac}
\pgfmathprintnumber{0.333333333333333}\hspace{1em}
\pgfmathprintnumber{0.5}\hspace{1em}
\pgfmathprintnumber{2.133333333333325e-01}\hspace{1em}
\pgfmathprintnumber{0.12}\hspace{1em}
\pgfmathprintnumber{2.666666666666646e-02}\hspace{1em}
\pgfmathprintnumber{-1.333333333333334e-02}\hspace{1em}
\pgfmathprintnumber{7.200000000000000e-01}\hspace{1em}
\pgfmathprintnumber{6.666666666666667e-02}\hspace{1em}
\pgfmathprintnumber{1.333333333333333e-01}\hspace{1em}
\pgfmathprintnumber{-1.333333333333333e-02}\hspace{1em}
\pgfmathprintnumber{3.3333333}\hspace{1em}
\pgfmathprintnumber{1.23456}\hspace{1em}% 这一行需要 fp 宏包的支持
\pgfmathprintnumber{1}\hspace{1em}
\pgfmathprintnumber{-6}
```

本选项利用命令 \pgfmathprintnumber@frac 做计算，此命令需要 fpu 库的支持, 某些情况下也需要 fp 宏包的支持。

/pgf/number format/frac TeX=\⟨*macro*⟩ (no default, initially \frac)

这个选项与上面的选项 frac 相配合，如果要使用这个选项就必须同时使用选项 frac. 本选项将输出数值的格式规定为 TEX 宏 \⟨*macro*⟩ 所表现的形式，初始为 \frac.

这里 \⟨*macro*⟩ 应当是带有两个参数的宏,待输出的数值在选项 frac 的作用下表现为 \⟨*macro*⟩{⟨*m*⟩}{⟨*n*⟩} 的实现形式。如果 \⟨*macro*⟩ 是只有一个参数的宏，那么待输出的数值在选项 frac 的作用下的表现可能有点奇怪。

$\frac{3}{5}$, $\frac{3}{5}$, $\sqrt{35}$,
$1^{3}5$, $1_{3}5$, $\ln_{3}5$,

```
\pgfmathprintnumber[frac]{.6},
\pgfmathprintnumber[frac,frac TeX={\dfrac}]{.6},
\pgfmathprintnumber[frac,frac TeX={\sqrt}]{.6},\\[15pt]
\pgfmathprintnumber[frac,frac TeX={^}]{1.6},
\pgfmathprintnumber[frac,frac TeX={_}]{1.6},
\pgfmathprintnumber[frac,frac TeX={\ln_}]{.6},
```

/pgf/number format/frac denom=⟨*int*⟩ (no default, initially empty)

这个选项与上面的选项 frac 相配合，如果要使用这个选项就必须同时使用选项 frac. 本选项将输出数值的分数部分的分母设为整数 ⟨*int*⟩.

$\frac{1}{10}$ $\frac{5}{10}$ $1\frac{2}{10}$ $-\frac{6}{10}$ $-1\frac{4}{10}$

```
\pgfkeys{/pgf/number format/.cd,frac, frac denom=10}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{0.5}\hspace{1em}
\pgfmathprintnumber{1.2}\hspace{1em}
\pgfmathprintnumber{-0.6}\hspace{1em}
\pgfmathprintnumber{-1.4}\hspace{1em}
```

注意本选项设置的 ⟨*int*⟩ 最好是 10 的整数倍。

$\frac{3}{5}$, $\frac{1}{2}$, $\frac{7}{12}$, $\frac{-12}{-20}$,

```
\pgfmathprintnumber[frac]{.6},
\pgfmathprintnumber[frac,frac denom=2]{.6},\\[10pt]
\pgfmathprintnumber[frac,frac denom=12]{.6},
\pgfmathprintnumber[frac,frac denom=-20,frac TeX={\dfrac}]{.6},
```

/pgf/number format/frac whole=true|false (no default, initially true)

这个选项与上面的选项 frac 相配合，如果要使用这个选项就必须同时使用选项 frac.

当本选项的值为 true 时，其作用是：如果待输出数值无整数部分，就将它输出为真分数；如果待输出数值有整数部分，就将它输出为带分数。

当本选项的值为 false 时，其作用是：如果待输出数值无整数部分，就将它输出为真分数；如果待输出数值有整数部分，就将它输出为假分数。

注意本选项的初始值是 true.

$\frac{201}{10}$ $\frac{11}{2}$ $\frac{6}{5}$ $-\frac{28}{5}$ $-\frac{7}{5}$

```
\pgfkeys{/pgf/number format/.cd,frac, frac whole=false}
\pgfmathprintnumber{20.1}\hspace{1em}
\pgfmathprintnumber{5.5}\hspace{1em}
\pgfmathprintnumber{1.2}\hspace{1em}
\pgfmathprintnumber{-5.6}\hspace{1em}
\pgfmathprintnumber{-1.4}\hspace{1em}
```

/pgf/number format/frac shift={⟨*integer*⟩} (no default, initially 4)

参考命令 \pgfmathfloatgetfrac → P. 580.

/pgf/number format/frac warning

这个选项的定义是：

```
\pgfkeys{%
    /pgf/number format/.is family,
    %...
    frac warning/.is if=pgfmathprintnumber@frac@warn,
    frac warning=true,
    %...
}
```

这个选项设置 TEX-if 命令 \ifpgfmathprintnumber@frac@warn 的真值，其初始值是 true, 参考命令 \pgfmathfloatgetfrac → P. 580.

/pgf/number format/precision={⟨*number*⟩} (initially 2)

本选项设置输出数值的精度，即保留原数值的小数点后的 ⟨*number*⟩ 位小数，将其余小数做四舍五入。如果原数值没有小数点，或小数部分的位数不足 ⟨*number*⟩ 位，本选项不会为输出数值添加小数点或

用 0 补足位数。如果原数值有小数点，但没有小数数字或没有非 0 小数数字，输出时本选项会把小数部分去掉。

本选项对 fixed 格式和 sci 格式 (针对尾数 (mantissa)) 都有效。

10 10 10 10.01 $5 \cdot 10^6$ $5.01 \cdot 10^6$

```
\pgfkeys{/pgf/number format/precision=2}
\pgfmathprintnumber{10} \quad
\pgfmathprintnumber{10.} \quad
\pgfmathprintnumber{10.0} \quad
\pgfmathprintnumber{10.005} \quad
\pgfmathprintnumber{5.00e6} \quad
\pgfmathprintnumber{5.005e6}
```

本选项的初始值是：

```
\def\pgfmathfloat@round@precision{2}
```

本选项只是针对输出结果的“舍入”标准，不是计算标准。在具体的计算过程中，计算的精度由计算命令自己的算法决定。

/pgf/number format/sci precision=⟨*number or empty*⟩ (no default, initially empty)

本选项针对 sci 格式的尾数（mantissa），设置其精度，即保留尾数的小数点后的 ⟨*number*⟩ 位小数，将其余小数做四舍五入。对于 sci 格式的输出来说，本选项要比选项 precision= 优先。

/pgf/number format/read comma as period=true|false (no default, initially false)

这个选项影响数值解析器的行为。若设置本选项的值是 true，数值解析器在读取待输出数值（即输入的数值）时，会把其中的逗号当作是小数点。不过如果没有其它相关设置，在输出数值时，仍然使用点号作为小数点。

1,234.56

```
\pgfkeys{/pgf/number format/read comma as period}
\pgfmathprintnumber{1234,56}
```

3.9.2 输出数值的样式以及标点符号

/pgf/number format/set decimal separator={⟨*text*⟩} (no default)

将 ⟨*text*⟩ 作为输出数值中的小数点符号，默认是点号。

1.5, ...

```
\pgfkeysgetvalue{/pgf/number format/set decimal separator}{\aspoint}
1\aspoint 5,\quad \aspoint \aspoint \aspoint
```

/pgf/number format/dec sep={⟨*text*⟩} (style, no default)

等效于 set decimal separator={⟨*text*⟩}.

/pgf/number format/set thousands separator={⟨*text*⟩} (no default)

为了便于读数，对于输出数值的整数部分，可以每隔 3 个数字放置一个分隔符，叫作“千位分隔符”。本选项将 ⟨*text*⟩ 作为千位分隔符，默认使用逗号。

a,b

```
\pgfkeysgetvalue{/pgf/number format/set thousands separator}\asthsep
a\asthsep b
```

如果要取消千位分隔符，就把 ⟨*text*⟩ 留空。如果 ⟨*text*⟩ 是一个逗号“,”，则输出时，逗号“,”后面会有一个“逗号空白”。如果 ⟨*text*⟩ 是两重花括号括起来的逗号“{{,}}”，则输出时，逗号“,”后面没有“逗号空白”。比较下面的输出：

```
\pgfmathprintnumber{1234.56} \par
\pgfmathprintnumber[set thousands separator={}]{1234.56} \par
\pgfmathprintnumber[set thousands separator={,}]{1234.56} \par
\pgfmathprintnumber[set thousands separator={{{,}}}]{1234.56}
```

1,234.56
1234.56
1, 234.56
1,234.56

/pgf/number format/1000 sep={⟨*text*⟩} (style, no default)

等效于 `set thousands separator={⟨text⟩}`.

/pgf/number format/1000 sep in fractionals={⟨*boolean*⟩} (no default, initially `false`)

如果这个选项的值是 true，则在输出数值的整数部分和小数部分中都使用“千位分隔符”；如果这个选项的值是 false，则只在输出数值的整数部分中都使用“千位分隔符”。

注意本选项的默认值是 true, 初始值是 false.

1.234&123&456&7 · 10^3

```
\pgfkeys{/pgf/number format/.cd, std=-2:2,
  precision=10, set thousands separator={\&},
  1000 sep in fractionals }
\pgfmathprintnumber{1234.1234567}
```

/pgf/number format/min exponent for 1000 sep={⟨*number*⟩} (no default, initially 01)

这个选项的作用是：假设待输出的数值是 n, 将 n 写成科学计数法形式 $n = s\cdot m\cdot 10^e$, 当 $e \geqslant \langle number\rangle$ 时，才会在输出 n 时使用千位分隔符。

如果 ⟨*number*⟩ 是 0, 则取消该选项；如果 ⟨*number*⟩ 是负数，则忽略之。

1000
10000
100 000

```
\pgfkeys{/pgf/number format/.cd, int detect,
1000 sep={\ }, min exponent for 1000 sep=5}
\pgfmathprintnumber{1000} \par
\pgfmathprintnumber{10000} \par
\pgfmathprintnumber{100000}
```

/pgf/number format/use period (no value)

这个选项是默认的，即默认小数点用点号“.”，千位分隔符用逗号“,”。

/pgf/number format/use comma (no value)

这个选项决定：小数点用逗号“,”，千位分隔符用点号“.”，恰好与上一个选项相反。

1.234,56

```
\pgfmathprintnumber[use comma]{1234.56}
```

/pgf/number format/skip 0.={⟨*boolean*⟩} (no default, initially `false`)

如果这个选项的值是 true，则 0.5 会被输出为 .5.

.56

```
\pgfmathprintnumber[skip 0.]{0.56}
```

/pgf/number format/showpos={⟨*boolean*⟩} (no default, initially `false`)

如果这个选项的值是 true，则输出非负数时会在数值前面添上正号“+”.

+12.3
+0
−12.3

```
\pgfkeys{/pgf/number format/showpos}
\pgfmathprintnumber{12.3} \par
\pgfmathprintnumber{0} \par
\pgfmathprintnumber{-12.3}
```

/pgf/number format/print sign={⟨*boolean*⟩} (style, no default)

与上一个选项 `showpos` 等效。

/pgf/number format/sci 10e (no value)

这个选项的作用是：输出数值时，按照默认设置或者某些手工设置的选项，如果需要将数值输出为科学计数法格式，那么该数值采用的科学计数法格式是 $s \cdot m \cdot 10^e$, 这是默认的科学计数法格式。

/pgf/number format/sci 10^e (no value)

等效于上一个选项 `sci 10e`.

/pgf/number format/sci e (no value)

这个选项的作用是：输出数值时，按照默认设置或者某些手工设置的选项，如果需要将数值输出为科学计数法格式，那么该数值采用的科学计数法格式是 $s\ m\ \mathrm{e} \pm r$. 如 `1.5e-4` 等于 $1.5 \cdot 10^{-4}$；`-1.5e+4` 等于 $-1.5 \cdot 10^4$.

$-1.23e{+}1$

```
\pgfkeys{/pgf/number format/.cd,sci,sci e}
\pgfmathprintnumber{-12.345}
```

/pgf/number format/sci E (no value)

与上一个选项类似，只是这里使用大写“E”.

$1.23E{+}1$

```
\pgfkeys{/pgf/number format/.cd,sci,sci E}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci subscript (no value)

这个选项的作用是：输出数值时，按照默认设置或者某些手工设置的选项，如果需要将数值输出为科学计数法格式，那么该数值采用的科学计数法格式是 $s\ m_r$，即把幂指数 r 作为尾数的下标。

1.23_1

```
\pgfkeys{/pgf/number format/.cd,sci,sci subscript}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci superscript (no value)

这个选项的作用是：输出数值时，按照默认设置或者某些手工设置的选项，如果需要将数值输出为科学计数法格式，那么该数值采用的科学计数法格式是 $s\ m^r$，即把幂指数 r 作为尾数的上标。

1.23^1

```
\pgfkeys{/pgf/number format/.cd,sci,sci superscript}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci generic={⟨*keys*⟩} (no default)

这个选项用于自定义一种科学计数法格式，其中 ⟨*keys*⟩ 可以使用以下选项。

/pgf/number format/sci generic/mantissa sep={⟨*text*⟩} (no default, initially `empty`)

将 ⟨*text*⟩ 作为尾数与幂之间的分隔符号，默认的科学计数法格式中使用乘积点 `\cdot`.

/pgf/number format/sci generic/exponent={⟨*text*⟩} (no default, initially `empty`)

定义幂的格式，在参数 ⟨*text*⟩ 中可以使用一个变量符号“`#1`”来表示幂指数。注意这里默认 ⟨*text*⟩ 处于数学模式中。

$1.23 \times 拾^{1}$;　$1.23 \times 拾^{-4}$

```
\pgfkeys{/pgf/number format/.cd, sci,
  sci generic={mantissa sep=\times,exponent={\text{拾}^{#1}}}}
\pgfmathprintnumber{12.345}; \quad \pgfmathprintnumber{0.00012345}
```

实际上，在 `sci generic={`⟨*keys*⟩`}` 的 ⟨*keys*⟩ 中可以出现 3 个变量：`#1` 代表幂指数，`#2` 代表数值的符号，`#3` 代表尾数。

如果数值是正数，则 #2 的值应是 1；如果数值是负数，则 #2 的值应是 2.
#3 代表的尾数是未经格式化处理的，例如其中没有千位分隔符。

1.23 玩一下$_1^{1.23}$拾^1； −1.23 玩一下$_2^{1.23}$拾$^{-4}$

```
\pgfkeys{/pgf/number format/.cd, sci,
  sci generic={mantissa sep={\text{\quad 玩一下}_#2^{#3}},exponent={\text{拾}^{#1}}}}
\pgfmathprintnumber{12.345}; \quad \pgfmathprintnumber{-0.00012345}
```

/pgf/number format/retain unit mantissa=true|false (no default, initially true)

本选项针对科学记数法格式的输出数值（包括选项 std 规定的格式）。如果本选项的值是 false, 那么，当按照规定的精度对尾数做舍入后，若尾数等于 1, 则忽略尾数。

$1.05\cdot10^1$; 10^1; $1.01\cdot10^3$; -10^3;

```
\pgfkeys{/pgf/number format/.cd,sci, retain unit mantissa=false}
\pgfmathprintnumber{10.5};
\pgfmathprintnumber{10};
\pgfmathprintnumber{1010};
\pgfmathprintnumber[precision=1]{-1010};
```

/pgf/number format/@dec sep mark={⟨*text*⟩} (no default)

将 ⟨*text*⟩ 放在输出数值的小数点位置的左侧，即使数值是没有小数点的整数也会在末尾添加这个 ⟨*text*⟩, 通常用作占位符。

12&.35
12&

```
\makeatletter
\pgfkeys{/pgf/number format/@dec sep mark={\&}}
\pgfmathprintnumber{12.345} \par
\pgfmathprintnumber{12}
\makeatother
```

/pgf/number format/@sci exponent mark={⟨*text*⟩} (no default)

这个选项针对科学计数法格式的输出数值，将 ⟨*text*⟩ 放在尾数与幂之间的分隔符的左侧，通常用作占位符。

1.23& · 10^1
1& · 10^0

```
\makeatletter
\pgfkeys{/pgf/number format/@sci exponent mark={\&}}
\pgfmathprintnumber[sci]{12.345} \par
\pgfmathprintnumber[sci]{1}
\makeatother
```

/pgf/number format/assume math mode={⟨*boolean*⟩} (default true)

在输出数值之前，会检查当前模式是否是数学模式。如果不是，就用命令 \pgfutilensuremath{\pgfmathresult} 把数值放入数学模式中输出。如果设置本选项的值是 true, 就总是假设当前模式是数学模式，直接用 \pgfmathresult 输出数值（当然此时未必处于数学模式中）。这里所谓的“假设”的意思是设置 \ifpgfmathprintnumber@assumemathmode 的真值为 true.

/pgf/number format/verbatim

用“抄录”形式输出数值，而不是用数学模式输出数值。这个选项会重置 1000 sep, dec sep, print sign, skip 0. 等选项的设置，但保留 precision, fixed zerofill, sci zerofill, fixed, sci, int detect 等选项的效果。

1.23e1; 1.23e-4; 3.27e6

```
\pgfkeys{
  /pgf/fpu,
  /pgf/number format/.cd, sci, verbatim}
\pgfmathprintnumber{12.345};
\pgfmathprintnumber{0.00012345};
```

```
\pgfmathparse{exp(15)}
\pgfmathprintnumber{\pgfmathresult}
```

第四章　重复操作：foreach 句法

实现 foreach 句法的是宏包 pgffor，这个宏包会被 Ti*k*Z 自动加载，但 PGF 并不自动加载这个宏包，这个宏包可以独立于 PGF 使用。所以要想在 PGF 下使用 foreach 句法，应当先调用这个宏包。

```
\usepackage{pgffor} % LaTeX
\input pgffor.tex % plain TeX
\usemodule[pgffor] % ConTeXt
```

这个宏包主要定义了命令 \foreach 和 \breakforeach. 宏包文件《pgffor.code.tex》会自动载入《pgfmath.code.tex》，进而载入其他文件，所以宏包 pgffor 需要 PGF 数学引擎以及 key 机制的支持。

4.1　句法

\foreach[⟨*options*⟩]⟨*variables*⟩[⟨*options*⟩] in ⟨*list*⟩ ⟨*commands*⟩

参数 ⟨*options*⟩ 是选项。

⟨*variables*⟩ 是循环变量，是以反斜线开头的 TEX 命令形式，例如 \x，\point, 如果你想做一些有趣的事情，那么 ⟨*variables*⟩ 也可以是活动符 (active characters, 类代码为 13)。

⟨*list*⟩ 是用逗号分隔的循环变量的取值列表。

⟨*commands*⟩ 是循环体。

注意，\foreach 语句各个组成部分之间最好不要有空行。

对于 ⟨*list*⟩ 中的每个变量值，\foreach 会设置一个组，在这个组内将变量值赋予变量 ⟨*variables*⟩, 再执行循环体 ⟨*commands*⟩. 由于单次循环放在一个组内，可以尽量避免对下次循环的影响。如果希望某个命令不受组的限制，可以使用全局定义。

使用单个变量：

[1][2][3][0]

```
\def\mylist{1,2,3,0}
\foreach \x in \mylist {[\x]}
```

在 ⟨*list*⟩ 中可以使用省略号 ...，意思是构造等差数列。

如果 ⟨*commands*⟩ 中不出现所设定的变量 ⟨*variables*⟩, 那就仅仅把 ⟨*commands*⟩ 执行若干次，所执行的次数是 ⟨*list*⟩ 中列表项的个数。

0.19365, 0.91702, 0.5358, 0.17427, 0.23373,

```
\foreach \x in {1,...,5} {\pgfmathparse{rnd}\pgfmathresult, }
```

4.1.1　列表 ⟨*list*⟩ 中的省略号

在 ⟨*list*⟩ 中将某个列举条目使用省略号代替会导致一种“递推”构造，有以下几种类别：

1. 构造公差为 1 或 −1 的等差数列：

```
\foreach \x in {1,...,6} {\x, } 得到 1, 2, 3, 4, 5, 6,
\foreach \x in {9,...,3.5} {\x, } 得到 9, 8, 7, 6, 5, 4,
```

2. 用两项规定公差，构造等差数列：

```
\foreach \x in {1,2,...,6} {\x, } 得到 1, 2, 3, 4, 5, 6,
\foreach \x in {1,2,3,...,6} {\x, } 得到 1, 2, 3, 4, 5, 6,
\foreach \x in {1,3,...,11} {\x, } 得到 1, 3, 5, 7, 9, 11,
\foreach \x in {1,3,...,10} {\x, } 得到 1, 3, 5, 7, 9, 注意不包括 10
\foreach \x in {0,0.1,...,0.5} {\x, } 得到 0, 0.1, 0.20001, 0.30002, 0.40002
```

3. 字母递推：

```
\foreach \x in {a,...,m} {\x, } 得到 a, b, c, d, e, f, g, h, i, j, k, l, m,
\foreach \x in {Z,X,...,M} {\x, } 得到 Z, X, V, T, R, P, N,
```

4. 参数递推：

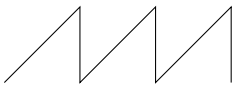
```
\foreach \x in {2^1,2^...,2^7} {$\x$, }
\foreach \x in {0\pi,0.5\pi,...\pi,2\pi} {$\x$, }
\foreach \x in {A_1,..._1,H_1} {$\x$, }
```

5. 多种混合

```
\foreach \x in {a,b,9,8,...,1,2,2.125,...,2.5} {\x, }
得到 a, b, 9, 8, 7, 6, 5, 4, 3, 2, 1, 2, 2.125, 2.25, 2.375, 2.5,
```

4.1.2 在路径中使用 foreach 语句

TikZ 允许在路径中使用 `foreach` 或 `\foreach` (二者等效)。在路径中使用 foreach 语句时，foreach 所辖的 ⟨*commands*⟩ 必须是用于构造路径的代码。

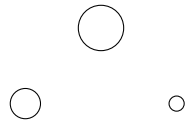


```
\tikz \draw (0,0)
  foreach \x in {1,...,3}
  { -- (\x,1) -- (\x,0) };
```

node 和 pic 操作也支持 foreach 语句。

4.1.3 多个相互关联的变量

如果在 `\foreach` 后面设置多个变量，需要在 ⟨*list*⟩ 中为每个变量设置值：



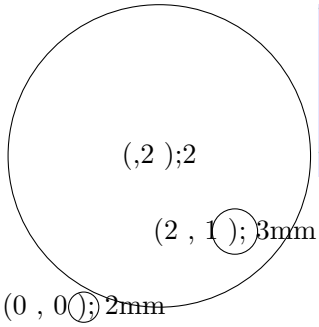
```
\begin{tikzpicture}
  \foreach \x / \y / \r in {0 / 0 / 2mm, 1 / 1 / 3mm, 2 / 0 / 1mm}
    \draw (\x,\y) circle (\r);
\end{tikzpicture}
```

如上设置的变量 `\x/\y/\r` 不是相互独立的，它们的取值实际上是 3 个向量：

(0,0,2mm) 和 (1,1,3mm) 和 (2,0,1mm)

注意以下几点：

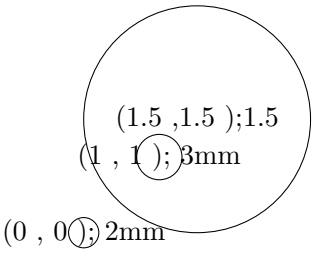
- 在 ⟨*variables*⟩ 中斜线“/”的前后可以有空格；在 ⟨*list*⟩ 中，斜线“/”的前后如果有空格，则空格会被当作变量值的一部分。
- 对于一组变量值，有的斜线“/”的前面或后面缺少变量值，那么相应的变量值通常当作“空值”处理，例如，




```
\begin{tikzpicture}
  \foreach \x / \diameter / \y  in  {0 / 2mm / 0 , 2 / 3mm / 1 , /2 }
  {
    \draw (\x,\y) circle (\diameter);
    \node at (\x,\y) {(\x,\y);\diameter};}
\end{tikzpicture}
```

上面例子中，第 3 组变量值缺少第 1 个变量的值，此时 \x 是空的。但是当 TikZ 解析坐标 (,2) 时，得到的是 (⟨x 轴的单位坐标⟩,⟨y 轴的单位坐标的 2 倍⟩).

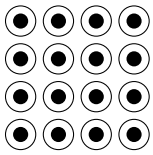
- 对于一组变量值，如果只有第 1 个变量的值，没有斜线，自然就缺少其他变量的值，此时，把第 1 个变量的值用作其他变量的值，




```
\begin{tikzpicture}
  \foreach \x / \diameter / \y  in
  {0 / 2mm / 0 , 1 / 3mm / 1 , 1.5 }
    {\draw (\x,\y) circle (\diameter);
     \node at (\x,\y) {(\x,\y);\diameter};}
\end{tikzpicture}
```

4.1.4 套嵌使用 foreach 句子

当两个 \foreach 语句直接套嵌时，内层的 \foreach 语句作为外层 \foreach 的循环代码。



```
\begin{tikzpicture}[x=0.5cm,y=0.5cm]
\foreach \x in {0,1,2,3}
  \foreach \y in {0,1,2,3}
    {
      \draw (\x,\y) circle (0.2cm);
      \fill (\x,\y) circle (0.1cm);
    }
\end{tikzpicture}
```

当两个 \foreach 语句套嵌时，两个语句中的变量是相互独立的。所以在上面的例子中，(\x,\y) 代表 16 个点。

对于下面的例子：

```
\foreach \i in {1,...,5}
{
  \foreach \j in {2,4}
  {
    ⟨commands⟩
  }
}
```

这个套嵌结构的执行次序是：

1. 对于 \i=1, 执行
   (a) \j=2
   (b) \j=4
2. 对于 \i=2, 执行
   (a) \j=2
   (b) \j=4

3. .....

4.2 \foreach 的大致处理过程

\foreach 的大致处理过程是：

1. \pgffor@atbeginforeach
2. 定义几个初始值

```
\let\pgffor@assign@before@code=\pgfutil@empty%
\let\pgffor@assign@after@code=\pgfutil@empty%
\let\pgffor@assign@once@code=\pgfutil@empty%
\let\pgffor@remember@code=\pgfutil@empty%
\let\pgffor@remember@once@code=\pgfutil@empty%
\pgffor@alphabeticsequencefalse%
\pgffor@contextfalse%
\let\pgffor@var=\pgfutil@empty
```

3. \pgffor@vars, 将 foreach 句子中的各个部分分别保存，各个选项也会被立即执行。
4. 定义几个初始值

```
\let\pgffor@last=\pgfutil@empty%
\let\pgffor@prevlast=\pgfutil@empty%
\let\pgffor@dotsend=\pgfutil@empty%
\let\pgffor@dots@pre=\pgfutil@empty%
\let\pgffor@dots@post=\pgfutil@empty%
```

5. 执行循环。解析变量值列表，对每一个变量值，或者一组变量值，执行一次循环体。此时有两种情况：一是变量值中含有省略号，二是变量值中不含有省略号。
6. 在将变量值列表处理完毕后，执行 \pgffor@after

\pgffor@atbeginforeach

初始之下，就是 \begingroup.

```
\def\pgffor@atbeginforeach{%
    \begingroup%
}
```

\pgffor@after

```
\def\pgffor@after{%
  \global\pgffor@continuetrue%
  \pgffor@atendforeach%
  \pgffor@afterhook}
```

\pgffor@atendforeach

```
\def\pgffor@atendforeach{%
        \global\edef\pgffor@remember@expanded{\pgffor@remember@code}%
        \ifx\pgffor@remember@expanded\pgfutil@empty%
        \else%
          \pgffor@remember@expanded%
          \global\let\pgffor@remember@expanded=\pgfutil@empty%
```

```
        \fi%
    \endgroup%
}
```

4.2.1 解析选项、循环变量、变量值

\foreach 句法中的选项 [⟨*options*⟩], 变量 ⟨*variables*⟩, 变量值列表 ⟨*list*⟩ 被 \pgffor@vars 解析。

选项 当 \pgffor@vars 遇到方括号时，会立即执行其中的选项：

```
\pgfkeys{/pgf/foreach/.cd,⟨options⟩}
```

循环变量 可以只用一个变量，如

```
\foreach \x[⟨options for \x⟩] in {1,2,3} ...
```

可以使用多个变量，如

```
\foreach \x[⟨options for \x⟩]/\y[⟨options for \y⟩]/\z in {1/2/3,a/b/c} ...
```

也可以把上面循环变量之间的斜线 / 去掉：

```
\x[⟨options for \x⟩]\y[⟨options for \y⟩]\z
```

被解析的变量依次保存在 \pgffor@var 中：

```
\def\pgffor@var{\x/\y/\z}% 用斜线分隔变量
```

循环变量的值列表 ⟨*list*⟩ 是循环变量所取的值。

- 如果 ⟨*list*⟩ 是以开花括号开头的通常的列表 {⟨*value list*⟩}, 则用 \pgffor@normal@list 处理它：

  ```
  \pgffor@normal@list{⟨value list⟩}
  ```

- 如果 ⟨*list*⟩ 不以开花括号开头，那么它一般就是一个宏 \⟨*macro*⟩, 此时先将它展开一次，再用 \pgffor@normal@list 处理展开结果：

  ```
  \expandafter\pgffor@normal@list\expandafter{\⟨macro⟩}
  ```

\pgffor@normal@list 的处理是：

1. 定义 \pgffor@values
   - 如果选项 /pgf/foreach/expand list→P. 180=true, 那么做彻底展开定义

     ```
     \edef\pgffor@values{⟨value list⟩ 或 展开一次的\⟨macro⟩}%
     ```

   - 如果选项 /pgf/foreach/expand list=false, 那么直接定义

     ```
     \def\pgffor@values{⟨value list⟩ 或 展开一次的\⟨macro⟩}%
     ```

   在初始之下有 /pgf/foreach/expand list=false, 所以最好直接定义

   ```
   \def\⟨macro⟩{⟨value list⟩}
   ```

   此后，
   - 如果 \pgffor@values 是空的，则重定义

     ```
     \def\pgffor@values{\pgffor@stop,}%
     ```

   - 如果 \pgffor@values 不是空的，则重定义

     ```
     \expandafter\def\expandafter\pgffor@values\expandafter{\pgffor@values,
     ↪  \pgffor@stop,}%
     ```

2. 执行

```
\global\pgffor@continuetrue%
\pgffor@collectbody
```

命令 \pgffor@collectbody 解析 ⟨*commands*⟩ 这一部分。

4.2.2 保存循环体

命令 \pgffor@collectbody 识别、保存循环体 ⟨*commands*⟩. 本命令是一个循环处理，它将作为循环体的代码保存到宏 \pgffor@body 中。

本命令主要识别 \foreach, 开花括号 {, 分号 ; 这 3 种记号：

1. 若遇到 \foreach⟨*var*⟩in⟨*list*⟩, 即 ⟨*commands*⟩ 以 \foreach 句子开头 (套嵌使用 \foreach 句子)，就把这一部分添加到循环体中，然后继续识别这 3 种记号；
2. 若遇到开花括号 {,
   - 如果 ⟨*commands*⟩ 以 \foreach 句子开头,则把花括号以及花括号内的代码添加到 \pgffor@body 中；
   - 如果 ⟨*commands*⟩ 不以 \foreach 句子开头，则把花括号内的代码添加到 \pgffor@body 中；

   然后调用 \pgffor@iterate.
3. 如果没有遇到 \foreach 或开花括号 {, 就向后查找分号 ;, 将遇到的第一个分号以及这个分号之前的代码添加到循环体中，然后调用 \pgffor@iterate.

4.2.3 执行循环

循环有 2 种情况：一是顺次执行的循环，二是套嵌的循环，这两种情况也可能混在一起。当套嵌使用 \foreach 命令时会出现套嵌的循环，由于 \foreach 命令会把所有操作放入一个组中，所以内层循环会被放入组中执行。下文说到“单次的循环”，指的是以循环体为核心的一次操作，可能指的是外层的循环，也可能指的是内层的循环，也可能指的是顺次执行的一个循环。一般情况下，变量值的个数，或者说变量值的组数 (对于有多个循环变量的情况)，决定了循环的次数。变量值保存在列表 \pgffor@values 中。单次的循环有 2 个步骤：第 1 步，解析一个或一组变量值，并定义变量宏来保存变量值；第 2 步，执行循环体。

4.2.4 从列表 \pgffor@values 中逐个 (逐组) 读取变量值

\pgffor@scan#1,

本命令逐一解析变量值列表中的列表项：

```
\expandafter\pgffor@scan\pgffor@values
```

本命令的定义是：

```
\def\pgffor@scan{\pgfutil@ifnextchar({\pgffor@scanround}{\pgffor@scanone}}
\def\pgffor@scanround(#1)#2,{\def\pgffor@value{(#1)#2}\pgffor@scanned}
\def\pgffor@scanone#1,{\def\pgffor@value{#1}\pgffor@scanned}
```

本命令以逗号作为相邻两个列表项的分隔符号，也就是说，参数 #1 代表一个列表项。

- 如果 #1 以开圆括号 (开头，则将 #1 保存到 \pgffor@value 中，这里还要求 #1 中的圆括号是平衡的，否则导致错误；
- 如果 #1 不以开圆括号 (开头，则将 #1 保存到 \pgffor@value 中；

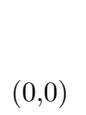
也就是说，命令 \pgffor@scan 会对以开圆括号 (开头的列表项做简单的识别、处理。

然后本命令调用 \pgffor@scanned.

按 \pgffor@scan 的定义，下面的代码导致错误：

```
\foreach \x / \y in {(0,0)/(1,1)}
{
  \tikz{ \node at\x {\x}; \node at\y {\y}; }
}
```

因为 (0,0)/(1 被当作一组变量值，可以使用花括号避免这种错误：




```
\foreach \x / \y in {(0,0)/{(1,1)}}
{
  \tikz{ \node at\x {\x}; \node at\y {\y}; }
}
```

\pgffor@scanned

本命令执行条件分支：

- 如果 \pgffor@value 等于 \pgffor@stop, 即在前一个循环中已将变量值列表处理完毕，则执行 \pgffor@after
- 如果 \pgffor@value 不等于 \pgffor@stop,
  - 如果 \pgffor@continuefalse, 则执行 \pgffor@scan
  - 如果 \pgffor@continuetrue
    - 如果 \pgffor@value 中包含省略号 ..., 则执行 \pgffor@handledots,
    - 如果 \pgffor@value 中不包含省略号 ..., 则执行 \pgffor@handlevalue,

本命令的处理受到 \ifpgffor@continue 的真值的影响。

\ifpgffor@continue

在每个单次的循环中都检查这个 TEX-if 的真值。如果这个 TEX-if 的真值是 false, 那么就只是解析、读取变量值，但不会执行循环体。如果这个 TEX-if 的真值是 true, 那么在解析、读取变量值后，就执行循环体。

4.2.4.1 单次的循环：当变量值或一组变量值中不含有省略号时

\pgffor@handlevalue

在解析某个或某一组变量值时,如果其中不包含省略号 ..., 则执行本命令。本命令调用 \pgffor@invokebody 执行一次赋值、循环操作，然后调用 \pgffor@scan 继续处理变量值。

```
\def\pgffor@handlevalue{%
  \let\pgffor@prevlast=\pgffor@last%
  \let\pgffor@last=\pgffor@value%
  \pgffor@invokebody%
  \pgffor@scan%
}
```

```
\let\pgffor@begingroup=\pgffor@default@begingroup
\let\pgffor@endgroup=\pgffor@default@endgroup
```

\pgffor@invokebody

在初始之下,本命令在一个组内做一次循环 (对应选项 /pgf/foreach/scope iterations→P. 175=true):

1. \pgffor@begingroup, 初始之下就是 \begingroup.
2. 为各个循环变量赋值，此时所有的变量都保存在 \pgffor@var 中，如果其中有多个变量，那么相邻 2 个变量之间用斜线 / 分隔。这一步会检查在 \pgffor@var 中是否含有斜线 /:
   - 如果不含有斜线 /, 那么就是只有一个循环变量，此时直接定义

```
\expandafter\expandafter\expandafter\def\expandafter\pgffor@var\expandafter
↪  {\pgffor@value}%
相当于
\def\⟨var⟩{展开一次的 \pgffor@value，即变量值}
```

- 如果含有斜线 /，那么就是有多个循环变量，此时执行命令 \pgffor@multiassign，这个命令以斜线 / 为参数定界标志，将各个变量与相应的变量值对应起来，分别定义

```
\def\⟨var i⟩{ 变量值 }
```

对于变量值缺失的情况，参考前文。

3. 如果 \pgffor@assign@once@code 非空，则执行它，参考选项 evaluate，assign，evaluate once，assign once，
4. 如果 \pgffor@assign@before@code 非空，则执行它，
5. 执行循环体

```
\expandafter\expandafter\expandafter\pgffor@reset@hooks\expandafter
↪  \pgffor@beginhook\expandafter\pgffor@body\pgffor@endhook%
```

其中 \pgffor@body 是循环体，它的前后都有 hook.

6. 如果 \pgffor@assign@after@code 非空，则执行它，
7. \pgffor@endgroup，在初始定义下，此命令 (即 \pgffor@default@endgroup) 如下处理：
   (a) 如果 \pgffor@remember@once@code 非空，则全局定义：

```
\xdef\pgffor@remember@once@expanded{\pgffor@remember@once@code}
```

   (b) 如果 \pgffor@remember@code 非空，则全局定义：

```
\xdef\pgffor@remember@expanded{\pgffor@remember@code}
```

   (c) \endgroup，结束前面 \pgffor@begingroup 开启的组。
   (d) 如果 \pgffor@remember@once@expanded 非空，则：

```
\pgffor@remember@once@expanded%
\global\let\pgffor@remember@once@expanded=\pgfutil@empty
```

   (e) 如果 \pgffor@assign@once@code 非空，则：

```
\global\let\pgffor@assign@once@code=\pgfutil@empty
```

   (f) 如果 \pgffor@remember@expanded 非空，则：

```
\pgffor@remember@expanded%
\global\let\pgffor@remember@expanded=\pgfutil@empty
```

若选项 /pgf/foreach/scope iterations→P. 175=false，那么本命令不会创建组，而是进行压栈和出栈操作。

4.2.5 循环中的 hook

在单次循环中，循环体的前后都有 hook:

```
\expandafter\expandafter\expandafter\pgffor@reset@hooks\expandafter\pgffor@beginhook
↪  \expandafter\pgffor@body\pgffor@endhook%
```

\pgffor@reset@hooks

在文件的代码中，这个 hook 设置其他 hook 的初始值：

```
\def\pgffor@reset@hooks{%
  \let\pgffor@beginhook=\relax%
  \let\pgffor@endhook=\relax%
  \let\pgffor@afterhook=\relax%
}
\pgffor@reset@hooks
```

\pgffor@afterhook 在 \pgffor@after 那里被用到。

4.3 针对变量的选项

\pgffor@ifcsregister\⟨*command*⟩{⟨*first code*⟩}{⟨*second code*⟩}

这个命令判断 \⟨*command*⟩ 是否计数器：

- 如果是宏，或者是未定义的，则设置真值\pgffor@registeriscountfalse，然后执行 ⟨*second code*⟩.
- 如果是计数器，则设置真值\pgffor@registeriscounttrue，然后执行 ⟨*first code*⟩.
- 如果非以上情况，则设置真值\pgffor@registeriscountfalse，然后执行 ⟨*first code*⟩.

判断过程利用了 \meaning\⟨*command*⟩ 的返回值。

选项的定义用到以下条件：

```
\newif\ifpgffor@assign@evaluate
\newif\ifpgffor@assign@once
\newif\ifpgffor@assign@parse
```

下面的数个选项会定义数个宏，在单次循环中，它们的执行次序是 (如果它们非空的话，参考 \pgffor@invokebody)：

1. \begingroup
2. 为循环变量赋值
3. \pgffor@assign@once@code
4. \pgffor@assign@before@code
5. 循环体
6. \pgffor@assign@after@code
7. \endgroup
8. \pgffor@remember@once@code
9. 若当前循环是 (顺次执行的数个循环中的) 第 1 次循环，则全局地清空 \pgffor@assign@once@code
10. \pgffor@remember@code

在 \foreach 的开头设置这些宏的初始值为空。

在 (顺次执行的数个循环中的) 第 1 次循环的结尾处会全局地清空 \pgffor@assign@once@code，所以，在第 2 次循环中，\pgffor@assign@once@code 很有可能是空的。所以，\pgffor@assign@once@code 中的非全局定义将被限制在包裹第 1 次循环的组中。

在所有循环结束后，在 \pgffor@atendforeach → P. 169 那里，还会再次执行 \pgffor@remember@code (如果非空的话)。

/pgf/foreach/var=⟨*variable*⟩ (no default)

这个选项用于声明变量名称，举例来说，以下叙述等效：

```
\foreach \x/\y in {0/1,a/b}
\foreach [var=\x,var=\y] in {0/1,a/b}
\foreach [var=\x] \y in {0/1,a/b}
\foreach \x [var=\y]  in {0/1,a/b}
```

注意如果使用这个选项，那么该选项应该放在其它变量选项之前，因为其它选项需要变量名称。

当 \pgffor@vars 遇到变量 \x 时，会调用 \pgffor@var@add 将 \x 添加到宏 \pgffor@var 中。本选项直接调用 \pgffor@var@add 将 \x 添加到宏 \pgffor@var 中。

/pgf/foreach/scope iterations=true|false (no default)

本选项决定是否将单次的循环放入一个组内执行。

- 如果本选项的值是 true, 则

```
\let\pgffor@begingroup=\pgffor@default@begingroup%
\let\pgffor@endgroup=\pgffor@default@endgroup%
```

这两个命令会创建组，把单次的 (或内层的) 循环放入这个组内执行，这就是文件《pgffor.code.tex》的初始设置。

- 如果本选项的值是 false, 则

```
\let\pgffor@begingroup=\pgffor@stack@begingroup%
\let\pgffor@endgroup=\pgffor@stack@endgroup%
```

这两个命令不创建组，而是做压栈和出栈操作。第一个命令将当前的某些变量值保存，然后执行一个循环 (单次的循环或者内层的循环)，然后再将保存的变量拿出来。(不过 \foreach 会把所有操作放入一个组内。)

比较：

„,e^5

```
\def\aaaa{}
\foreach \i in {1,2,...,5}
{
  \edef\aaaa{\aaaa,,,$e^\i$}
  \ifnum \i=5 \xdef\aaaa{\aaaa} \fi
}
\aaaa
```

„,e^1„,e^2„,e^3„,e^4„,e^5

```
\def\aaaa{}
\foreach [scope iterations=false] \i in {1,2,...,5}
{
  \edef\aaaa{\aaaa,,,$e^\i$}
  \ifnum \i=5 \xdef\aaaa{\aaaa} \fi
}
\aaaa
```

„,e^1„,e^a„,e^b„,e^2„,e^a„,e^b„,e^3„,e^a„,e^b„,e^4„,e^a„,e^b„,e^5„,e^a„,e^b

```
\def\aaaa{}
\foreach [scope iterations=false] \i in {1,2,...,5}
{
  \edef\aaaa{\aaaa,,,$e^\i$}
  \foreach [scope iterations=false] \j in {a,b}
  {
    \edef\aaaa{\aaaa,,,$e^\j$}
    \if b\j \xdef\aaaa{\aaaa} \fi
  }
  \ifnum \i=5 \xdef\aaaa{\aaaa} \fi
}
\aaaa
```

/pgf/foreach/evaluate={⟨*evaluate statement*⟩} (no default)

这个选项的主要作用是，将某些数学表达式保存，以待需要时利用。它的参数可以是以下形式：

- 类似 math 库那样，罗列多个句子：

```
\aaaa=⟨expression 1⟩;
\bbbb=⟨expression 2⟩;
...
```

其中每个句子以分号结束 (最后一个句子可以不用分号结束)，并且使用等号。句子的意思是，例如，第一个句子的 ⟨*expression 1*⟩ 应当是包含循环变量 (宏) 的数学表达式，这会把代码

```
\pgfmathparse{⟨expression 1⟩}\let\aaaa=\pgfmathresult
```

添加到宏 \pgffor@assign@before@code 中 (右侧)。

注意在 ⟨*expression 1*⟩ 中使用的变量 (宏) 不需要特别的限制，也不需要是已定义的，因为本选项只是保存代码，并不解析表达式，所以，只要在执行 \pgffor@assign@before@code 时，⟨*expression 1*⟩ 中的变量 (宏) 有定义就不会导致错误。在单次循环中，执行 \pgffor@assign@before@code 的时机通常是，为循环变量赋值之后，执行循环体之前，参考 \pgffor@invokebody.

- 罗列循环变量 (宏)

```
\x 或
\x;\y 或
\x;\y;
```

这个罗列形式等效于

```
\x=\x 或
\x=\x;\y=\y 或
\x=\x;\y=\y;
```

- 句法

⟨*variable*⟩ as ⟨*macro*⟩ using ⟨*formula*⟩

其中 ⟨*variable*⟩ 是循环变量，⟨*formula*⟩ 应当是包含循环变量 (宏) 的数学表达式，这个句子等效于

```
⟨macro⟩=⟨formula⟩;
```

对比：

2^0, 2^1, 2^2, 2^3, 2^4,

```
\foreach \x in {2^0,2^...,2^4}{$\x$, }% 输出 TeX 的排版结果
```

1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0, 256.0,

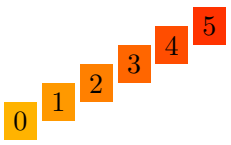
```
\foreach \x [evaluate=\x] in {2^0,2^...,2^8}{$\x$, }% 输出数学计算的结果
```

$2^0 = 1.0$, $2^1 = 2.0$, $2^2 = 4.0$, $2^3 = 8.0$, $2^4 = 16.0$, $2^5 = 32.0$, $2^6 = 64.0$, $2^7 = 128.0$, $2^8 = 256.0$,

```
\foreach \x [evaluate=\x as \xeval] in {2^0,2^...,2^8} {$\x=\xeval$, }
```

0 1 2 3 4 5 6 7 8 9 10

```
\tikz\foreach [evaluate=\x as \shade using \x*10] \x in {0,1,...,10}
\node [fill=red!\shade!yellow, minimum size=0.65cm] at (\x,0) {\x};
```




```
\tikz\foreach \x [count=\a,evaluate={\i=0.5*\x;\j=10*\x+30;}] in {0,1,...,5}
\node [fill=red!\j!yellow] at (0.5*\a,0.5*\i) {\x};
```

通常情况下 foreach 语句输出的数值结果默认是带小数点的，如果需要修改输出数值的格式可以参考 \pgfmathprintnumber → P. 156, 例如：

1, 2, 4, 8, 16, 32, 64, 128, 256,

```
\foreach \x [evaluate=\x as \xeval using int(\x)] in {2^0,2^...,2^8}{\xeval, }
```

1, 2, 4, 8, 16, 32, 64, 128, 256,

```
\foreach \x [evaluate=\x as \xeval] in {2^0,2^...,2^8}
  {\pgfkeys{/pgf/number format/int trunc} \pgfmathprintnumber{\xeval}, }
```

/pgf/foreach/assign={⟨*assign statement*⟩} (no default)

这个选项的主要作用是，将某些定义宏的代码保存起来，以待需要时利用。它的参数可以是以下罗列形式：

```
\⟨macro 1⟩={⟨replace text 1⟩};% 如果换行，要有注释符号
\⟨macro 2⟩={⟨replace text 2⟩};
...
```

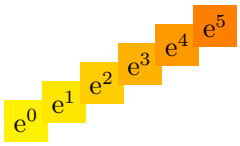
其中每个句子以分号结束 (最后一个句子可以不用分号结束)，并且使用等号。例如，第 1 个句子，这会导致把

```
\def\⟨macro 1⟩{⟨replace text 1⟩}
```

添加到宏 \pgffor@assign@before@code 中。在单次循环中，执行 \pgffor@assign@before@code 的时机通常是，为循环变量赋值之后，执行循环体之前，参考 \pgffor@invokebody.

注意，选项 evaluate，assign 都可以向宏 \pgffor@assign@before@code 中添加代码 (添加到右侧)。

下面例子先执行选项 assign, 再执行选项 evaluate:




```
\tikz\foreach [
    assign={{\a}={2};% 如果中间换行，那么这里一定要有注释符号
            {\tttt}={$\mathrm e^\x$}},
    evaluate={\i=0.5*\x+\a;\j=10*\x;}] \x in {0,1,...,5}
  \node [fill=red!\j!yellow] at (\i,0.5*\i) {\tttt};
```

/pgf/foreach/evaluate once={⟨*evaluate statement*⟩} (no default)

这个选项类似选项 evaluate, 不过本选项将代码

```
\pgfmathparse{⟨expression⟩}\let\⟨macro⟩=\pgfmathresult
```

添加到宏 \pgffor@assign@once@code 中 (右侧)，还把代码

```
\noexpand\def\noexpand\⟨macro⟩{⟨expression⟩}
```

添加到宏 \pgffor@remember@once@code 中 (右侧)。

注意在第 1 次循环的结尾处会全局地清空 \pgffor@assign@once@code.

„,6.0^a„,(1 + 2 + 3)^b„,(1 + 2 + 3)^c

```
\def\aaaa{}
\def\bbbb{x}
\foreach [evaluate once={\bbbb=(1+2+3);}] \i in {a,...,c}
{ \xdef\aaaa{\aaaa,,,$\bbbb^\i$} }
\aaaa
```

/pgf/foreach/assign once={⟨*assign statement*⟩} (no default)

本选项类似 assign, 不过本选项把

```
\def\⟨macro⟩{⟨replace text⟩}
```

添加到宏 \pgffor@assign@once@code 中，还把

```
\noexpand\def\noexpand\⟨macro⟩{⟨replace text⟩}
```

添加到宏 \pgffor@remember@once@code 中。

注意在第 1 次循环的结尾处会全局地清空 \pgffor@assign@once@code.

/pgf/foreach/remember={⟨remember statement⟩} (no default)

这个选项的主要作用是，将某些代码保存，以待需要时利用。它的参数可以是以下形式：

- 罗列多个句子：

```
\⟨remember 1⟩=\⟨cmd 1⟩;
\⟨remember 2⟩=\⟨cmd 2⟩;
...
```

其中每个句子以分号结束 (最后一个句子可以不用分号结束)，并且使用等号。句子的意思是，例如，对于第一个句子：

– 如果 \⟨cmd 1⟩ 是计数器，就把

```
\expandafter\def\expandafter\⟨remember 1⟩\expandafter{\the\⟨cmd 1⟩}
```

添加到宏 \pgffor@assign@after@code 中 (右侧)，还把

```
\noexpand\def\noexpand\⟨remember 1⟩{\⟨remember 1⟩}
```

添加到宏 \pgffor@remember@code 中 (右侧)。

– 如果 \⟨cmd 1⟩ 不是计数器，就把

```
\expandafter\def\expandafter\⟨remember 1⟩\expandafter{\⟨cmd 1⟩}
```

添加到宏 \pgffor@assign@after@code 中 (右侧)，还把

```
\noexpand\def\noexpand\⟨remember 1⟩{\⟨remember 1⟩}
```

添加到宏 \pgffor@remember@code 中 (右侧)。

- 罗列多个命令：

```
\⟨cmd 1⟩;% 如果换行，要有注释符号
\⟨cmd 2⟩;
...
```

其中每个句子以分号结束 (最后一个句子可以不用分号结束)，这等效于

```
\⟨cmd 1⟩=\⟨cmd 1⟩;% 如果换行，要有注释符号
\⟨cmd 2⟩=\⟨cmd 2⟩;
...
```

- 句子

⟨variable⟩ as ⟨macro⟩ (initially ⟨value⟩)

其中 (initially ⟨value⟩) 是可选的，⟨value⟩ 也可以是空的。这会导致：

– 把

```
\edef⟨macro⟩{⟨variable⟩}
```

添加到宏 \pgffor@assign@after@code 中 (右侧)。

– 把

```
\noexpand\def\noexpand⟨macro⟩{⟨macro⟩}
```

添加到宏 \pgffor@remember@code 中 (右侧)。

– 把

```
\edef⟨macro⟩{⟨value⟩}
```

添加到宏 \pgffor@assign@once@code 中 (右侧)。如果不写出 (initially ⟨value⟩) 或者

⟨*value*⟩ 是空的，那么此处定义的 ⟨*macro*⟩ 就是空的。

假设变量 \x 的值域是 $\{a_1, a_2, \cdots\}$, 而 ⟨*commands*⟩ 的作用相当于一个二元操作 $f(a_{n-1}, a_n)$, 其中 a_n 是变量 \x 的当前值，此时可以使用本选项。

这个选项定义宏 ⟨*macro*⟩, 并将 \x 的当前值的前一个值赋予宏 ⟨*macro*⟩. 其中 (initially ⟨*value*⟩) 是可选的，规定这个宏的初值为 ⟨*value*⟩. 可以在 ⟨*commands*⟩ 中使用宏 ⟨*macro*⟩.

```
\foreach \x [remember=\x as \lastx (initially A)] in {B,...,F}
等价于
\foreach \x / \lastx in {B/A,...,F/E}
```

$\overrightarrow{AB}$, $\overrightarrow{BC}$, $\overrightarrow{CD}$,
$\overrightarrow{DE}$, $\overrightarrow{EF}$,
$\overrightarrow{\text{初值}C}$, $\overrightarrow{CD}$, $\overrightarrow{DE}$, $\overrightarrow{EF}$,

```
\foreach \x [remember=\x as \lastx (initially A)]
  in {B,...,F}{$\overrightarrow{\lastx\x}$, }

\foreach \x [remember=\x as \lastx
    (initially {\text{初值}})]
  in {C,...,F}{$\overrightarrow{\lastx\x}$, }
```

/pgf/foreach/count=⟨*count statement*⟩ (no default)

这个选项保存某些代码，并执行一个计算。

本选项的参数形式可以是：

- ⟨*macro*⟩ from ⟨*value*⟩, 对于这个形式，本选项会把

  ```
  \noexpand\def\noexpand⟨macro⟩{⟨macro⟩}
  ```

  添加到宏 \pgffor@remember@code 中 (右侧)，把

  ```
  \pgfmathparse{int(⟨macro⟩+1)}\let⟨macro⟩=\pgfmathresult
  ```

  添加到宏 \pgffor@assign@before@code 中 (右侧)，还计算

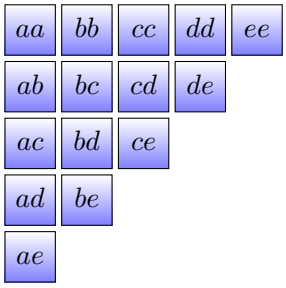
  ```
  \pgfmathparse{int(⟨value⟩-1)}\let⟨macro⟩=\pgfmathresult
  ```

  这就把 ⟨*macro*⟩ 的初始值设置为 int(<⟨*value*⟩>-1).

- ⟨*macro*⟩, 这个形式等效于 ⟨*macro*⟩ from 1.

这个选项定义宏 ⟨*macro*⟩, 并把变量的当前值在 ⟨*list*⟩ 中的序号赋予 ⟨*macro*⟩, 可以在 ⟨*commands*⟩ 中使用宏 ⟨*macro*⟩. 其中的 from ⟨*value*⟩ 是可选的，当使用这个词组后，宏 ⟨*macro*⟩ 的初始值就是 ⟨*value*⟩, 即第 1 次执行 ⟨*commands*⟩ 时宏 ⟨*macro*⟩ 的值是 ⟨*value*⟩, 第 2 次执行 ⟨*commands*⟩ 时宏 ⟨*macro*⟩ 的值是 ⟨*value*⟩+1，……

```
\foreach \x [count=\xi from -2] in {a,...,e}
等价于
\foreach \x / \xi in {a/-2,...,e/3}
```




```
\tikz[x=0.75cm,y=0.75cm]
  \foreach \x [count=\xi] in {a,...,e}
    \foreach \y [count=\yi] in {\x,...,e}
      \node [draw, top color=white, bottom color=blue!50, minimum
      ↪  size=0.666cm]
        at (\xi,-\yi) {$\mathstrut\x\y$};
```

0:1:2, 1:2:3, 2:3:4, 3:4:5,

```
\foreach \i[count=\j from 1,count=\k from 0] in {2,...,5}{\k:\j:\i, }
```

/pgf/foreach/parse=true|false (default false)

在用省略号构造变量值的过程中，如果本选项的值设为 `true`, 那么用命令 `\pgfmathparse` 来计算 (计算时会抑制 fpu 库)，所以列表项可以是很复杂的表达式。

注意 TEX 寄存器运算有误差，可能会引起问题。

1 2 3 4 5 6 7 8 9

```
\foreach \x [parse=true] in {1,...,1.0e+1 - 1}{ \x }
```

`/pgf/foreach/expand list=`{⟨*boolean*⟩} (default `false`)

如果变量值列表 ⟨*list*⟩ 是一个宏，或者列表 ⟨*list*⟩ 中含有宏，那么本选项有意思。比较：

1!!!2!!!3!!!

```
\def\aaaa{1,2,3}
\foreach \i in \aaaa{\i!!!}%\aaaa 被展开一次，恰当
```

1,2,3!!!

```
\def\aaaa{1,2,3}
\def\bbbb{\aaaa}
\foreach \i in \bbbb{\i!!!}%\bbbb 被展开一次，不恰当
```

1!!!2!!!3!!!

```
\def\aaaa{1,2,3}
\def\bbbb{\aaaa}
\def\cccc{\bbbb}
\foreach [expand list=true] \i in \cccc{\i!!!}%\cccc 被彻底展开，恰当
```

如果本选项的值是 true, 那么 ⟨*list*⟩ 会先被 `\edef` 展开，再解析、读取。如果 ⟨*list*⟩ 中含有复杂定义的宏 (用这个宏构造变量值列表)，可以使用本选项。

1,2,3,4,5,

```
\def\Iota#1#2{%
  \ifnum\numexpr#1\relax<\numexpr#2\relax
    \the\numexpr#1\relax,%
    \expandafter\Iota\expandafter{\the\numexpr(#1)+1\relax}{#2}%
  \else
    \the\numexpr#2\relax
  \fi}
\foreach [expand list=true] \x in {\Iota{1}{5}} {\x,}
```

`\breakforeach`

这个命令用在 `\foreach` 语句的 ⟨*commands*⟩ 中。`\foreach` 语句会执行 ⟨*commands*⟩ 数次，在每次执行 ⟨*commands*⟩ 的过程中，如果遇到了 `\breakforeach`, 就会终断本次执行，进入下一次执行。这是对 ⟨*commands*⟩ 的处理过程作出的限制。

```
\def\breakforeach{\global\pgffor@continuefalse}
```

```
\begin{tikzpicture}
  \foreach \x in {1,...,4}
    \foreach \y in {1,...,4}
     {
     \fill[red!50] (\x,\y) ellipse (3pt and 6pt);
     \ifnum \x<\y
       \breakforeach
     \fi
     }
\end{tikzpicture}
```

4.4 ungrouped foreach 命令

`\pgfplotsforeachungrouped`

在宏包 PgfplotsTable 中定义了命令 \pgfplotsforeachungrouped, 此命令类似 \foreach 那样执行重复操作，采用的句法也是类似的，不过如其名称所示，此命令不会把各次操作限制在 TEX 组中。观察下面的例子：

```
\def\jishu{0}
\foreach \i in {1,...,5}{\pgfmathsetmacro{\jishu}{int(\jishu+1)}}\jishu\par
\pgfplotsforeachungrouped \i in {1,...,5} {\pgfmathsetmacro{\jishu}{int(\jishu+1)}}\jishu
```

0
5

上面例子用宏 \jishu 来统计列表 1,...,5 中列表项的个数。可见命令 \pgfplotsforeachungrouped 是有便利之处的。

不过，命令 \pgfplotsforeachungrouped 没有与之配合的 \break 命令。另外，下面代码：

```
\def\aaaa{1,...,5}
\pgfplotsforeachungrouped \i in \aaaa {\pgfmathsetmacro{\jishu}{int(\jishu+1)}}
```

中的 \aaaa 也会导致错误。前面可用于 \foreach 的选项 /pgf/foreach/count →P. 179 等也不能用于此命令。

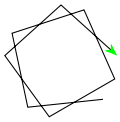
观察下面的例子：

```
--([turn]-84:1) coordinate (A1)--([turn]-84:1) coordinate (A2)--([turn]-84:1)
coordinate (A3)--([turn]-84:1) coordinate (A4)
```

```
\def\SubstitutionMark{}
\pgfplotsforeachungrouped \i in {1,...,4}{%
  \edef\SubstitutionMark{\SubstitutionMark--([turn]-84:1) coordinate (A\i)}%
}
\texttt{\SubstitutionMark}
```

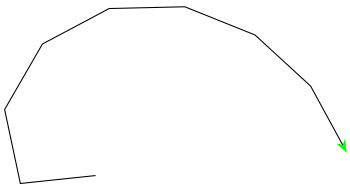
这个例子用一个迭代构造 \SubstitutionMark.

再一个例子。规定：“右转 84°, 前进一个固定长度并且在前进时画线”是一个步骤，现在要重复这个步骤若干次。观察下面的图形：



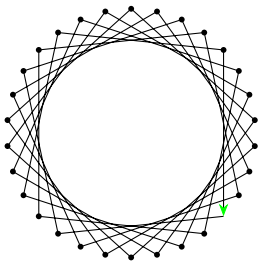
```
\def\SubstitutionMark{}
\pgfplotsforeachungrouped \i in {1,...,8}{%
  \edef\SubstitutionMark{\SubstitutionMark--([turn]-84:1) coordinate (A
  ↪  \i)}%
}
\tikz{
  \draw [-{Stealth[green]}](0,0) coordinate (A0)\SubstitutionMark;
}
```

使用 \foreach 就是这样的：



```
\tikz{
  \draw [-{Stealth[green]}](0,0) coordinate (A0)
    foreach \i in {1,...,8}{--([turn]-84:1)coordinate (A\i)};
}
```

用 \pgfplotsforeachungrouped 是这样：



```
\def\SubstitutionMark{}
\pgfplotsforeachungrouped \i in {1,...,30}{%
  \edef\SubstitutionMark{\SubstitutionMark--([turn]-84:1) coordinate (A
  ↪  \i)}%
}
\tikz[scale=2.2]{
  \draw [-{Stealth[green]}](0,0) coordinate (A0)\SubstitutionMark;
  \foreach \i in {1,...,29} \fill (A\i) circle (0.5pt);
}
```

下面的例子得到 Fibonacci 数列的前 7 项：

1, 1, 2, 3, 5, 8, 13

```
\expandafter\def\csname A0\endcsname{1}
\expandafter\def\csname A1\endcsname{1}
\pgfplotsforeachungrouped \i in {2,...,6}{%
  \pgfmathsetmacro{\j}{int(\i-1)}
  \pgfmathsetmacro{\k}{int(\i-2)}
  \expandafter\pgfmathsetmacro\expandafter{\csname A\i\endcsname}
    {%
      int(\csname A\j\endcsname + \csname A\k\endcsname)
    }
}
\foreach \i in {0,...,6}{%
  \ifnum \i=6
    \csname A\i\endcsname
  \else
    \csname A\i\endcsname,\quad
  \fi}
```

第五章　日历

```
\usepackage{pgfcalendar} % LaTeX
\input pgfcalendar.tex % plain TeX
\usemodule[pgfcalendar] % ConTeXt
```

这个宏包可以独立于 PGF 来使用。本宏包提供的命令可以在“标准 ISO 格式的日期 (如 1975-12-26)”与“儒略日 (Julian day)”之间做转换。“儒略日”是一个整数，即“天数”，其起算点 (即儒略日的第 0 日) 是公元前 4713 年 1 月 1 日中午 12 点 (世界时，星期一)。本宏包也提供了制作日历的命令。

这个宏包需要 PGF 的数学引擎的文件《pgfint.code.tex》的支持。

这个宏包可以与 translator 包配合，来翻译星期、月份的名称。

5.1　处理日期

5.1.1　日期转换

\pgfcalendardatetojulian{⟨*data*⟩}\⟨*counter*⟩

本命令将 ISO 格式的日期 ⟨*date*⟩ 转换为儒略日，保存在计数器 \⟨*counter*⟩ 中。

参数 ⟨*date*⟩ 可以是：

- ⟨年⟩-⟨月⟩-⟨日⟩ 这种格式，如 2021-1-1 或 2021-01-01
- ⟨年⟩-⟨月⟩-⟨日⟩+⟨*a positive or negative number*⟩ 这种格式，如 2021-1-1+2 或 2021-01-01+-2
- 可以用 \year, \month, \day 分别代表当前的年、月、日
- ⟨日⟩ 可以是单词 last, 代表当前月的最后一天
- ⟨*date*⟩ 可以是保存以上格式的宏，命令 \pgfcalendardatetojulian 会先把 ⟨*date*⟩ 彻底展开，然后再调用 \pgfcalendar@datetojulian 解析展开结果

参数 \⟨*counter*⟩ 是事先声明的计数器。

```
\newcount\aaaa
\pgfcalendardatetojulian{2022-2-2}{\aaaa}
\the\aaaa
```

2459613

\pgfcalendarjuliantodate{⟨*Julian day*⟩}\⟨*year macro*⟩\⟨*month macro*⟩\⟨*day macro*⟩

本命令将儒略日数 ⟨*Julian day*⟩ 转换为 ISO 格式的日期，将年、月、日分别保存到宏 \⟨*year macro*⟩, \⟨*month macro*⟩, \⟨*day macro*⟩ 中。

保存在 \⟨*year macro*⟩ 中的年份没有先导“0”，但注意，这个宏可能保存数值 0, 尽管没有第 0 年。

保存在 \⟨*month macro*⟩ 中的月份是 2 个数字符号，第一个数字符号可能是先导“0”，例如“01”就是一月份。

保存在 \⟨*day macro*⟩ 中的日子是 2 个数字符号，第一个数字符号可能是先导“0”。

参数 ⟨*Julian day*⟩ 必须是一个能适合 \count1=⟨*Julian day*⟩ 的表达式。

2022-02-02
```
\def\aaaa{2459613}
\pgfcalendarjuliantodate\aaaa{\myyear}{\mymonth}{\myday}
\myyear-\mymonth-\myday
```

\pgfcalendarjuliantoweekday{⟨*Julian day*⟩}\⟨*week day counter*⟩

本命令将儒略日数 ⟨*Julian day*⟩ 转换为星期数，保存在计数器 \⟨*week day counter*⟩ 中。

注意这个命令计算模 7 的余数，使得星期一对应数值 0, 星期二对应数值 1, 等等。

星期 3
```
\newcount\aaaa
\pgfcalendarjuliantoweekday{2459613}\aaaa
\advance\aaaa by1\relax 星期 \the\aaaa% 2022-2-2
```

\pgfcalendareastersunday{⟨*year*⟩}\⟨*counter*⟩

本命令将年份 ⟨*year*⟩ 中的复活节 (Easter Sunday, 每年春分月圆之后第一个星期日) 的日期转换为儒略日数，保存在计数器 \⟨*counter*⟩ 中。

复活节 2022-04-17
```
\newcount\aaaa
\pgfcalendareastersunday{2022}\aaaa
\pgfcalendarjuliantodate\aaaa\myyear\mymonth\myday
复活节 \myyear-\mymonth-\myday
```

5.1.2 检查日期

\pgfcalendarifdate{⟨*date*⟩}{⟨*tests*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

参数 ⟨*date*⟩ 是 ISO 格式的日期，会被命令 \pgfcalendardatetojulian → P. 183 处理，所以 ⟨*date*⟩ 的格式如前述。

参数 ⟨*tests*⟩ 是一个用逗号分隔的列表，每个列表项都是以 /pgf/calendar 为前缀路径的“键”(键值对)，每个键值对代表某种“性质”。

本命令检查日期 ⟨*date*⟩ 是否具有 ⟨*tests*⟩ 中列出的诸性质中的某一个性质；如果是，就执行 ⟨*true code*⟩，否则执行 ⟨*false code*⟩.

本命令的处理是：

1. 处理 ⟨*date*⟩

   ```
   \pgfcalendardatetojulian{⟨date⟩}{\pgfutil@tempcnta}
   ```

   将 ⟨*date*⟩ 转换为儒略日数保存到寄存器 \pgfutil@tempcnta

2. 获取 ⟨*date*⟩ 的年、月、日

   ```
   \pgfcalendarjuliantodate{\pgfutil@tempcnta}
       {\pgfcalendarifdateyear}{\pgfcalendarifdatemonth}{\pgfcalendarifdateday}%
   ```

   至此，宏 \pgfcalendarifdateyear, \pgfcalendarifdatemonth, \pgfcalendarifdateday 都可用了。

3. 保存儒略日数到宏 \pgfcalendarifdatejulian

   ```
   \edef\pgfcalendarifdatejulian{\the\pgfutil@tempcnta}
   ```

4. 计算星期数并保存到宏 \pgfcalendarifdateweekday

   ```
   \pgfcalendarjuliantoweekday{\pgfutil@tempcnta}{\pgfutil@tempcntb}%
   \edef\pgfcalendarifdateweekday{\the\pgfutil@tempcntb}%
   ```

5. 执行 \pgfcalendar@launch@ifdate{⟨*tests*⟩}{⟨*true code*⟩}{⟨*false code*⟩}.

   \pgfcalendar@launch@ifdate{⟨*tests*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

使用本命令时，以下宏应当有正确的定义：

- \pgfcalendarifdatejulian
- \pgfcalendarifdateyear
- \pgfcalendarifdatemonth
- \pgfcalendarifdateday
- \pgfcalendarifdateweekday

本命令的定义是：

```
\long\def\pgfcalendar@launch@ifdate#1#2#3{%
  % 本命令依赖前面定义的那些名称为 pgfcalendarifdatexxxx 的宏
  \pgfcalendarmatchesfalse%
  \pgfqkeys{/pgf/calendar}{#1}%
  \ifpgfcalendarmatches%
    #2%
  \else%
    #3%
  \fi%
}

\newif\ifpgfcalendarmatches
```

可见，⟨*tests*⟩ 中列出的“性质”应当都是以 /pgf/calendar 为路径的键值对。

如果日期 ⟨*date*⟩ 具有所列出的诸性质中的某一个性质，那么，这个性质对应的键应当把 \ifpgfcalendarmatches 的真值设置为 true; 如果日期 ⟨*date*⟩ 不具有所列出的任一性质，那么这些键应当不修改 \ifpgfcalendarmatches 的真值 (保持为 false)。

这个命令按照 \ifpgfcalendarmatches 的真值来执行 ⟨*true code*⟩ 或 ⟨*false code*⟩.

文件《pgfcalendar.code.tex》定义的以 /pgf/calendar 为路径的键 (可以列在 ⟨*tests*⟩ 中) 有以下。

注意，在执行以下键时，宏

- \pgfcalendarifdatejulian, 保存 ⟨*date*⟩ 的儒略日数
- \pgfcalendarifdateweekday, 保存 ⟨*date*⟩ 对应的星期数
- \pgfcalendarifdateyear, 保存 ⟨*date*⟩ 的 ⟨年⟩ 份
- \pgfcalendarifdatemonth, 保存 ⟨*date*⟩ 的 ⟨月⟩ 份
- \pgfcalendarifdateday, 保存 ⟨*date*⟩ 的 ⟨日⟩ 数

都应当是可用的。

/pgf/calendar/all

这个键的定义是：

```
\pgfkeys{/pgf/calendar/all/.code=\pgfcalendarmatchestrue}
```

/pgf/calendar/Monday

这个键的定义是：

```
\pgfkeys{/pgf/calendar/Monday/.code={\ifnum\pgfcalendarifdateweekday=0
↪  \relax\pgfcalendarmatchestrue\fi}}
```

注意 Monday 对应的宏 \pgfcalendarifdateweekday 的值是 0.

/pgf/calendar/Tuesday

这个键的定义是：

```
\pgfkeys{/pgf/calendar/Tuesday/.code={\ifnum\pgfcalendarifdateweekday=1
↪  \relax\pgfcalendarmatchestrue\fi}}
```

/pgf/calendar/Wednesday

/pgf/calendar/Thursday

/pgf/calendar/Friday

/pgf/calendar/Saturday

/pgf/calendar/Sunday

这个键的定义是：

```
\pgfkeys{/pgf/calendar/Sunday/.code={\ifnum\pgfcalendarifdateweekday=6
↪  \relax\pgfcalendarmatchestrue\fi}}
```

注意 Sunday 对应的宏 \pgfcalendarifdateweekday 的值是 6.

/pgf/calendar/workday

这个键的定义是：

```
\pgfkeys{/pgf/calendar/workday/.code={\ifnum\pgfcalendarifdateweekday<5
↪  \relax\pgfcalendarmatchestrue\fi}}
```

/pgf/calendar/weekend

这个键的定义是：

```
\pgfkeys{/pgf/calendar/weekend/.code={\ifnum\pgfcalendarifdateweekday>4
↪  \relax\pgfcalendarmatchestrue\fi}}
```

/pgf/calendar/equals=⟨*reference*⟩ (no default, 必须给出键值)

这个键的定义是：

```
\pgfkeys{/pgf/calendar/equals/.cd,.value required,.code={%
  \pgfcalendar@special@datetojulian{#1}%
  \ifnum\pgfcalendarifdatejulian=\pgfutil@tempcnta\relax%
    \pgfcalendarmatchestrue%
  \fi}%
}
```

这个键的参数 ⟨*reference*⟩ 可以是：

- 完整的 ISO 格式的日期，如 2025-2-5, 这导致

  ```
  \pgfcalendardatetojulian→P. 183{⟨ 年⟩-⟨ 月⟩-⟨ 日⟩}{\pgfutil@tempcnta}%
  ```

- 缺少年份的 ISO 格式的日期，如 2-5, 这导致

  ```
  \pgfcalendardatetojulian→P. 183{\pgfcalendarifdateyear-⟨ 月⟩-⟨ 日⟩}{
  ↪  \pgfutil@tempcnta}%
  ```

- 保存以上 2 种格式日期的宏。

这个键会比较宏 \pgfcalendarifdatejulian (即 ⟨*date*⟩ 的儒略日数) 与计数器 \pgfutil@tempcnta (即 ⟨*reference*⟩ 的儒略日数)，如果二者相等就设置 \ifpgfcalendarmatches 的真值为 true.

/pgf/calendar/at least=⟨*reference*⟩ (no default, 必须给出键值)

这个键的参数 ⟨*reference*⟩ 的格式参考 equals.

如果 ⟨*date*⟩ 不早于 ⟨*reference*⟩ (前者的儒略日数不小于后者),那么这个键就设置 \ifpgfcalendarmatches 的真值为 true.

/pgf/calendar/at most=⟨*reference*⟩ (no default, 必须给出键值)

这个键的参数 ⟨*reference*⟩ 的格式参考 equals.
如果 ⟨*date*⟩ 不晚于 ⟨*reference*⟩ (前者的儒略日数不大于后者),那么这个键就设置 \ifpgfcalendarmatches 的真值为 true.

/pgf/calendar/between=⟨*start reference*⟩ and ⟨*end reference*⟩ (no default, 必须给出键值)

这个键的参数 ⟨*reference*⟩ 的格式参考 equals.
如果 ⟨*date*⟩ 不晚于 ⟨*end reference*⟩ (前者的儒略日数不大于后者)，也不早于 ⟨*start reference*⟩ (前者的儒略日数不小于后者)，那么这个键就设置 \ifpgfcalendarmatches 的真值为 true.

/pgf/calendar/day of month=⟨*number*⟩ (no default, 必须给出键值)

这个键的参数 ⟨*number*⟩ 是一个整数，或者保存整数的宏、计数器。
注意 ⟨*date*⟩ 的组成是 ⟨年⟩-⟨月⟩-⟨日⟩，这个键比较 ⟨*number*⟩ 与 \pgfcalendarifdateday (即 ⟨*date*⟩ 的 ⟨日⟩)，如果二者相等，就设置 \ifpgfcalendarmatches 的真值为 true.

/pgf/calendar/end of month=⟨*number*⟩ (default 1)

这个键的参数 ⟨*number*⟩ 是一个整数，或者保存整数的宏、计数器，它的默认值是 1.
注意 ⟨*date*⟩ 的组成是 ⟨年⟩-⟨月⟩-⟨日⟩，这个键会判断 ⟨*date*⟩ 是否 ⟨月⟩ 的倒数第 ⟨*number*⟩ 天，如果是，就设置 \ifpgfcalendarmatches 的真值为 true.
假设 ⟨*date*⟩ 的儒略日数加上 ⟨*number*⟩ (即 \pgfcalendarifdatejulian+⟨*number*⟩) 对应的 ISO 格式日期是 ⟨*year'*⟩-⟨*month'*⟩-⟨*day'*⟩, 如果 ⟨*day'*⟩ 等于 1, 那么这个键就设置 \ifpgfcalendarmatches 的真值为 true.

/pgf/calendar/Easter=⟨*number*⟩ (default 0)

这个键的参数 ⟨*number*⟩ 是一个整数，或者保存整数的宏、计数器，它的默认值是 0.
注意 ⟨*date*⟩ 的组成是 ⟨年⟩-⟨月⟩-⟨日⟩，如果 ⟨*date*⟩ 与 ⟨年⟩ 中的复活节日期相差 ⟨*number*⟩ 天，那么这个键就设置 \ifpgfcalendarmatches 的真值为 true.

用户可以仿照以上键的定义，自定义别的键。

5.1.3 星期、月份的名称

在格式定义文件《pgfutil-latex.def》中有：

```
\ifx\translate\@undefined % check if \translate is available
  \def\pgfutil@translate#1{#1}
\else
  \def\pgfutil@translate#1{\translate{#1}}
\fi
```

这就是 LaTeX 中命令 \pgfutil@translate 的定义。

\pgfcalendarweekdayname{⟨*week day number*⟩}

参数 ⟨*week day number*⟩ 是一个整数，或者保存整数的宏、计数器。本命令返回 ⟨*week day number*⟩ 对应的星期名称 (全名)，⟨*week day number*⟩ 可以是 0, 1, ..., 6.

Monday `\pgfcalendarweekdayname{0}`

本命令的定义是：

```
\def\pgfcalendarweekdayname#1{%
  \pgfutil@translate{\ifcase#1Monday\or Tuesday\or Wednesday\or Thursday\or Friday
  ↪  \or Saturday\or Sunday\fi}%
}
```

\pgfcalendarweekdayshortname{⟨*week day number*⟩}

参数 ⟨*week day number*⟩ 是一个整数，或者保存整数的宏、计数器。本命令返回 ⟨*week day number*⟩ 对应的星期名称 (简写)，⟨*week day number*⟩ 可以是 0, 1, ..., 6.

Mon `\pgfcalendarweekdayshortname{0}`

\pgfcalendarmonthname{⟨*month number*⟩}

参数 ⟨*month number*⟩ 是一个整数，或者保存整数的宏、计数器。本命令返回 ⟨*month number*⟩ 对应的月份名称 (全名)，⟨*month number*⟩ 可以是 1, ..., 12.

January `\pgfcalendarmonthname{1}`

\pgfcalendarmonthshortname{⟨*month number*⟩}

参数 ⟨*month number*⟩ 是一个整数，或者保存整数的宏、计数器。本命令返回 ⟨*month number*⟩ 对应的月份名称 (简写)，⟨*month number*⟩ 可以是 1, ..., 12.

Jan `\pgfcalendarmonthshortname{1}`

5.2 排版日历

\pgfcalendar{⟨*prefix*⟩}{⟨*start date*⟩}{⟨*end date*⟩}{⟨*rendering code*⟩}

本命令将从 ⟨*start date*⟩ 到 ⟨*end date*⟩ 的每一个日期用 ⟨*rendering code*⟩ 排版出来。

关于本命令：

- 本命令的定义有前缀 \long
- 本命令的所有操作都限制在一个 \begingroup 与 \endgroup 的组合中
- 本命令先定义 \ifdate

  \ifdate{⟨*tests*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

  本命令的定义是：

  ```
  \let\ifdate=\pgfcalendar@local@ifdate
  ```

  \pgfcalendar@local@ifdate{⟨*tests*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

  本命令的定义是：

  ```
  \def\pgfcalendar@local@ifdate{%
    \let\pgfcalendarifdatejulian=\pgfcalendarcurrentjulian%
    \let\pgfcalendarifdateyear=\pgfcalendarcurrentyear%
    \let\pgfcalendarifdatemonth=\pgfcalendarcurrentmonth%
    \let\pgfcalendarifdateday=\pgfcalendarcurrentday%
    \let\pgfcalendarifdateweekday=\pgfcalendarcurrentweekday%
    \pgfcalendar@launch@ifdate%
  }
  ```

  可见本命令调用 \pgfcalendar@launch@ifdate →P. 184.

- 本命令 (无展开地) 保存 ⟨*prefix*⟩ 到宏 \pgfcalendarprefix

```
\def\pgfcalendarprefix{⟨prefix⟩}
```

⟨*prefix*⟩ 主要是配合 \pgfcalendarsuggestedname →P. 190 来使用。

- 参数 ⟨*start date*⟩ 与 ⟨*end date*⟩
  - ⟨*start date*⟩ 与 ⟨*end date*⟩ 都是完整 ISO 格式的日期，或者保存这种日期的宏
  - ⟨*start date*⟩ 与 ⟨*end date*⟩ 会被 \pgfcalendardatetojulian →P. 183 处理
  - ⟨*start date*⟩ 的儒略日数保存在宏 \pgfcalendarbeginjulian
  - ⟨*end date*⟩ 的儒略日数保存在宏 \pgfcalendarendjulian
  - ⟨*start date*⟩ 的彻底展开保存在宏 \pgfcalendarbeginiso
  - ⟨*end date*⟩ 的彻底展开保存在宏 \pgfcalendarendiso
- 本命令执行一个 \loop ... \repeat 循环来处理各个日期，在循环中
  1. 当前日期的儒略日数保存在计数器 \pgfcalendarcurrentjulian
  2. 循环的条件是：当前日期的儒略日数小于“⟨*end date*⟩ 的儒略日数加 1”

     ```
     \ifnum\pgfcalendarcurrentjulian<\pgfcalendarendjulianplus\relax
     ```

  3. 首先计算当前日期的相关信息
     - 宏 \pgfcalendarcurrentyear 保存年份 ⟨年⟩
     - 宏 \pgfcalendarcurrentmonth 保存月份 ⟨月⟩，是 2 位数
     - 宏 \pgfcalendarcurrentday 保存日数 ⟨日⟩，是 2 位数
     - 宏 \pgfcalendarcurrentweekday 保存星期数 ⟨日⟩
  4. 执行 ⟨*rendering code*⟩
     - 在 ⟨*rendering code*⟩ 中可以使用上述各个宏、计数器
     - 在 ⟨*rendering code*⟩ 中可以使用任何 TEX 代码，例如 {tikzpicture} 环境。
     - 如果在 {tikzpicture} 环境中使用 \pgfcalendar, 那么在 ⟨*rendering code*⟩ 中可以使用 TikZ/PGF 的命令。
     - 在 ⟨*rendering code*⟩ 中可以使用 \ifdate →P. 188

\pgfcalendarshorthand{⟨*kind spec*⟩}{⟨*format spec*⟩}

本命令按照 ⟨*format spec*⟩ 指定的输出格式来输出 ⟨*kind spec*⟩.

使用本命令时，需要以下宏有适当的定义：

- \pgfcalendarcurrentyear
- \pgfcalendarcurrentmonth
- \pgfcalendarcurrentday
- \pgfcalendarcurrentweekday

本命令可以用在 \pgfcalendar →P. 188 的 ⟨*rendering code*⟩ 中。

参数 ⟨*kind spec*⟩ 可以是

- d, 代表当前日期的日数 ⟨日⟩
- m, 代表当前日期的月份 ⟨月⟩
- y, 代表当前日期的年份 ⟨年⟩
- w, 代表当前日期的星期数 ⟨星期⟩

参数 ⟨*format spec*⟩ 指定输出格式，可以是

- -, 输出的日数、月份中没有先导 0
- =, 输出的日数、月份中用空格代替先导 0
- 0, 输出的日数、月份可能有先导 0

- t, 输出的是文本 (全名)
- ., 输出的是文本 (简写)

ISO form: 2007-01-20, long form: Saturday, January 20, 2007

```
%\usepackage {pgfcalendar}
\let\%=\pgfcalendarshorthand
\pgfcalendar{cal}{2007-01-20}{2007-01-20}
{ ISO form: \%y0-\%m0-\%d0, long form: \%wt, \%mt \%d-, \%y0}
```

\pgfcalendarsuggestedname

本命令用在 \pgfcalendar→P. 188 的 ⟨*rendering code*⟩ 中。

这个命令配合 \pgfcalendar→P. 188 的 ⟨*prefix*⟩ 使用。在排版日期时，如果需要把日期做成 node 并为它命名时，可以用这个命令生成一个名称。

本命令：

- 如果 ⟨*prefix*⟩ 是空的，则返回空的内容。
- 如果 ⟨*prefix*⟩ 不是空的，则返回

```
\pgfcalendarprefix-\pgfcalendarcurrentyear-\pgfcalendarcurrentmonth-
↪  \pgfcalendarcurrentday
```

即“⟨前缀⟩-⟨年⟩-⟨月⟩-⟨日⟩”这样一串符号。

可见，如果要使用本命令，那么 ⟨*prefix*⟩ 最好不是空的。

5.3 与 translator 包的配合

利用 translator 包可以将星期、月份的名称翻译为非英语名称。

先载入 translator 包，再载入 pgfcalendar 包 (或 Ti*k*Z 的 calendar 库)。

如果载入了 translator 包，那么 pgfcalendar 包会载入词典 translator-months-dictionary

```
\usedictionary{translator-months-dictionary}
```

命令 \usedictionary 属于 translator 包。

参考前文的 \pgfcalendarweekdayname→P. 187, \pgfutil@translate.

第二部分

系统层

第六章　格式与文件

6.1　Ti*k*Z 支持的格式

Ti*k*Z 支持的格式：LaTeX, Plain TeX, ConTeXt.

- 在 LaTeX 中使用 PGF 和 Ti*k*Z 都是很简单的，只需要使用命令

```
\usepackage{pgf}
或者
\usepackage{tikz}
```

- 使用 Plain TeX 格式时需要使用命令

```
\input{pgf.tex}
或者
\input{tikz.tex}
```

来载入 PGF 和 Ti*k*Z, 而环境名称则使用

```
\pgfpicture
⟨content⟩
\endpgfpicture
```

当使用 `pdftex`，或者 `tex+dvips` 编译时，PGF 能自己判断输出格式，其他情况下你需要定义宏 `\pgfsysdriver` 来指定驱动文件。

- 在 ConTeXt 格式下，使用命令

```
\usemodule[pgf]
或者
\usemodule[tikz]
```

来载入 PGF 和 Ti*k*Z, 而环境名称则使用

```
\startpgfpicture
⟨content⟩
\stoppgfpicture
```

或者

```
\starttikzpicture
⟨content⟩
\stoptikzpicture
```

6.2　支持的输出格式

TeX 产生输出的步骤有 2 步：第一，处理排版的文本文件，得到 `.dvi` 文件 (device-independent file)；第二，将 `.dvi` 文件转换为，例如 PostScript 文件。

编译方式例如：

- .tex 文件 $\xrightarrow{\texttt{latex}}$.dvi 文件 $\xrightarrow{\texttt{dvips}}$.ps 文件 $\xrightarrow{\texttt{ps2pdf}}$.pdf 文件
- .tex 文件 $\xrightarrow{\texttt{tex}}$.dvi 文件 $\xrightarrow{\texttt{dvipdfm}}$.pdf 文件
- .tex 文件 $\xrightarrow{\texttt{latex}}$.dvi 文件 $\xrightarrow{\texttt{tex4ht}}$.html 文件
- .tex 文件 $\xrightarrow{\texttt{pdftex}}$.pdf 文件
- .tex 文件 $\xrightarrow{\texttt{pdflatex}}$.pdf 文件

6.3 LaTeX 格式下的主要文件类型

在 LaTeX 格式下，Ti*k*Z 使用的文件主要有 3 种：

- .sty 文件，命令 \usepackage, \RequirePackage 读取的文件。
- .code.tex 文件，这类文件实现 Ti*k*Z/PGF 的各种命令。.sty 文件会调用这些文件。
- .def 文件，这是驱动文件，或者是与 LaTeX 格式相关的定义文件。

6.4 LaTeX 格式下 Ti*k*Z 宏包载入文件的次序

当使用 LaTeX 格式时，使用命令

```
\uaepackage{tikz}
```

会导致如下的文件载入次序：

```
*tikz.sty
␣␣**pgf.sty
␣␣␣␣***pgfrcs.sty
␣␣␣␣␣␣****pgfutil-common.tex
␣␣␣␣␣␣****pgfutil-latex.def%␣LaTeX␣ 格式相关的定义
␣␣␣␣␣␣****pgfrcs.code.tex
␣␣␣␣***pgfcore.sty
␣␣␣␣␣␣␣␣*****graphicx.sty%␣ 宏包 ␣graphicx
␣␣␣␣␣␣␣␣*****pgfsys.sty
␣␣␣␣␣␣␣␣␣␣******pgfrcs.sty
␣␣␣␣␣␣␣␣␣␣******pgfsys.code.tex
␣␣␣␣␣␣␣␣␣␣␣␣*******pgfkeys.code.tex
␣␣␣␣␣␣␣␣␣␣␣␣******* 在 ␣pgfsys.code.tex␣ 末尾载入配置文件 ␣pgf.cfg
␣␣␣␣␣␣␣␣␣␣␣␣******* 载入配置文件 ␣pgf.cfg␣ 后，载入驱动文件 ␣pgfsys-xxxx.def
␣␣␣␣␣␣␣␣␣␣******pgfsyssoftpath.code.tex
␣␣␣␣␣␣␣␣␣␣******pgfsysprotocol.code.tex
␣␣␣␣␣␣␣␣*****keyval.sty%␣ 宏包 ␣keyval
␣␣␣␣␣␣␣␣*****xcolor.sty%␣ 宏包 ␣xcolor
␣␣␣␣␣␣␣␣*****pgfcore.code.tex
␣␣␣␣␣␣␣␣␣␣******pgfmath.code.tex
␣␣␣␣␣␣␣␣␣␣******pgfint.code.tex
␣␣␣␣␣␣␣␣␣␣******pgfcorepoints.code.tex
␣␣␣␣␣␣␣␣␣␣******pgfcorepathconstruct.code.tex␣ 等等
␣␣␣␣***pgfmoduleshapes.code.tex
␣␣␣␣***pgfmoduleplot.code.tex
␣␣**pgffor.sty
␣␣␣␣***pgfrcs.sty
␣␣␣␣***pgfkeys.sty
␣␣␣␣␣␣****pgfkeys.code.tex
␣␣␣␣***pgfmath.sty
␣␣␣␣␣␣****pgfrcs.sty
␣␣␣␣␣␣****pgfkeys.sty
```

```
␣␣␣␣␣␣****pgfmath.code.tex
␣␣␣␣␣␣␣␣*****pgfkeys.code.tex
␣␣␣␣␣␣␣␣*****pgfmathcalc.code.tex
␣␣␣␣␣␣␣␣*****pgfmathfloat.code.tex
␣␣␣␣***pgffor.code.tex
␣␣␣␣␣␣****pgfmath.code.tex
␣␣**tikz.code.tex
␣␣␣␣***pgflibraryplothandlers.code.tex
␣␣␣␣***pgfmodulematrix.code.tex
␣␣␣␣***tikzlibrarytopaths.code.tex
```

《pgfutil-latex.def》是使用 LaTeX 格式时才会载入的文件，其中利用 LaTeX 的命令做了一些定义。

在文件《pgf.sty》中用命令 `\DeclareOption` 声明了宏包选项 `draft`, `version=0.65`, `version=0.96`, `version=1.18`, `version=latest`.

在文件《pgfsys.sty》中声明了宏包选项 `dvisvgm`, 这个选项选择驱动文件：

```
\def\pgfsysdriver{pgfsys-dvisvgm.def}
```

6.5 选择后台驱动

使用 LaTeX 的情况下 PGF 借鉴 graphics 宏包的机制来自己识别编译驱动，具体过程是：在 LaTeX 格式相关文件《pgfutil-latex.def》中有：

```
\def\pgfutil@guessdriver{
  \ifx\HCode\@undefined%
    \edef\pgfsysdriver{pgfsys-\Gin@driver}% should be right
  \else%
    \def\pgfsysdriver{pgfsys-tex4ht.def}% should be right
  \fi%
}
```

其中 `\Gin@driver` 是 graphics, xcolor 宏包都用到的宏，它一般保存 `dvips.def`, `dvipdfmx.def`, `dvisvgm.def`, `xetex.def`, `pdftex.def`, `luatex.def` 等字符串。

在文件《pgfsys.code.tex》中执行命令 `\pgfutil@guessdriver`, 从而定义 `\pgfsysdriver`, 然后执行 `\pgfutil@InputIfFileExists` 载入相应的驱动文件。驱动文件名称也会被写入 `.log` 文件，例如：“Driver file for pgf: pgfsys-xetex.def”。

如果 TikZ 不能自己决定驱动文件，那么你需要自己指定一个驱动文件，也就是说，在载入 `pgf` 之前，需要先为宏 `\pgfsysdriver` 定义一个合适的值，例如，如果你想使用 `dvips`, 那么你需要定义

```
\def\pgfsysdriver{pgfsys-dvips.def}
```

如果你想使用 `pdftex` 或 `pdflatex`, 那么你需要定义

```
\def\pgfsysdriver{pgfsys-pdftex.def}
```

6.5.1 输出 PDF 格式的文件

文件 pgfsys-pdftex.def

这是针对 pdfTeX 的驱动文件，此文件会调用文件 `pgfsys-common-pdf.def`.

文件 pgfsys-dvipdfm.def

这是针对 `(la)tex`*to*`dvipdfm` 的驱动文件，此文件会调用文件 `pgfsys-common-pdf.def`. 这个驱动支持 PGF 的多数特性，但有以下限制：

1. 在 LaTeX 下，它使用 graphics 的插图机制，不支持 masking.

2. 在 plainTEX 下，不支持插图机制。

文件 pgfsys-xetex.def
这是针对 xe(la)tex→xdvipdfmx 的驱动文件。

文件 pgfsys-vtex.def
这是针对 vtex 的驱动文件。此文件会调用文件 pgfsys-common-postscript.def. 使用命令行参数可以让 vtex 输出 PDF 或 Postscript 格式的文件。
这个驱动支持 PGF 的多数特性，但有以下限制：

1. 在 LATEX 下，它使用 graphics 的插图机制，不支持 masking.
2. 在 plainTEX 下，不支持插图机制。
3. 支持颜色渐变 (Shading), 但输出质量与 dvips 相同。
4. 不支持透明度。
5. 不能记住图形在页面上的位置。

6.5.2 输出 PostScript 格式的文件

文件 pgfsys-dvips.def
这是针对 (la)tex→dvips 的驱动文件，此文件会调用文件 pgfsys-common-postscript.def. 这个驱动支持 PGF 的多数特性，但有以下限制：

1. 在 LATEX 下，它使用 graphics 的插图机制，不支持 masking.
2. 在 plainTEX 下，不支持插图机制。
3. 支持颜色渐变 (Shading), 但输出质量不如与 pdf 文件。
4. 支持透明度，但需要新版 Ghostscript 的支持。
5. 能记住图形在页面上的位置，但需要新版 pdftex 的支持。

文件 pgfsys-textures.def
这是针对 textures 的驱动文件。此文件会调用文件 pgfsys-common-postscript.def.

6.5.3 输出 SVG 格式的文件

文件 pgfsys-dvisvgm.def
这个驱动将 DVI 文件转为 SVG 文件。由于 graphics 宏包尚不支持这个驱动 (?)，所以注意：在 LATEX 下，如果 tikz 宏包带有 dvisvgm 选项，则选定此驱动 (?)；如果调用 PGF 的宏包，例如 beamer 文类的情况下，要使用文类选项 dvisvgm.
例如

```
\documentclass[dvisvgm]{minimal}
\usepackage{tikz}
\begin{document}
Hello \tikz [baseline] \fill [fill=blue!80!black] (0,.75ex) circle[radius=.75ex];
\end{document}
```

执行：

```
latex example
dvisvgm example
```

或

```
lualatex --output-format=dvi example
dvisvgm example
```

文件 pgfsys-tex4ht.def

这个驱动文件针对 tex4ht. 此驱动调用 pgfsys-common-svg.def.

tex4ht 把.dvi 文件转为.html 文件，由于 HTML 格式不能用于绘制图形，所以这个驱动会要求 PGF 产生 SVG 图形。

使用这个驱动时注意：

1. 在 LaTeX 下，它使用 graphics 的插图机制。
2. 在 plainTeX 下，不支持插图机制。
3. 不能记住图形在页面上的位置。
4. 对 pgfpicture 环境内的文字的支持不是很好。因为目前 SVG 规范对文字的支持不是很好。
5. Unlike for other output formats, the bounding box of a picture “really crops” the picture.
6. 不支持矩阵。
7. 不支持函数颜色渐变 (Shading), 但输出质量不如与 pdf 文件。

这个驱动的工作方式是：在 pgfpicture 环境开始时，使用 \special 命令将 tex4ht 的输出定向到一个新的、名称为\jobname-xxx.svg 的文件中 (其中 xxx 是数字)，在 pgfpicture 环境的末尾，每个系统层的图形命令会在输出文件中创建 SVG 词语。PostScript/PDF 的 imaging model and the processing model 与 SVG 不同，但结果相近。

由于目前 SVG 对文字的支持不是很好，你可能需要下面的选项来调整处理文字的方式：

/pgf/tex4ht node/escape=⟨*boolean*⟩ (default false)

本选项决定文字渲染方法。本选项的值 false 的作用是，将文字转为 SVG 文字，但只能处理：简单符号 (字母，数字，标点符号，某些数学符号……), subscripts and superscripts (but not subsubscripts), 其他文字符号都会被忽略，或变成无效的 HTML 代码。也就是说，下面两种文字能被较好地渲染：

- 没有数学模式、特殊符号的文字，
- 含有上下标的简单的数学文本。

如果你使用含有特殊符号的文字，例如 α, 会破坏图形。

当使用 node[/pgf/tex4ht node/escape=true] {⟨*text*⟩} 时，PGF 会退回到 HTML 来渲染 ⟨*text*⟩, 这会得到较好的文字效果。

/pgf/tex4ht node/css=⟨*filename*⟩ (default \jobname)

这个选项告诉浏览器使用哪个 CSS 文件来决定 node 的展示样式，本选项只在 tex4ht node/escape=true 下有效。

/pgf/tex4ht node/class=⟨*class name*⟩ (default foreignobject)

这个选项允许你给 node 设置一个类名称，本选项只在 tex4ht node/escape=true 下有效。

/pgf/tex4ht node/id=⟨*id name*⟩ (default \jobname picturenumber-nodenumber)

这个选项允许你给 node 设置一个唯一的 id, 本选项只在 tex4ht node/escape=true 下有效。

6.5.4 输出 DVI 格式的文件

文件 pgfsys-dvi.def

这个驱动文件可以适应任何输出驱动，除了 texr4ht.

6.5.5 驱动文件《pgfsys-pdftex.def》

驱动文件《pgfsys-pdftex.def》做 2 件事：

1. 载入文件《pgfsys-common-pdf.def》，文件《pgfsys-common-pdf.def》的主要内容是，对系统层的路径命令、颜色模式、图形参数等涉及图形外观的命令做进行重定义 (之前《pgfsys.code.tex》对这些命令有不完善的定义)，使之对应 PDF 语言的要素并调用 `\pgfsysprotocol@literal`, 这个命令 (在通常情况下) 调用 `\pgfsys@invoke`, 再调用 pdftex 引擎的命令 `\pdfliteral`.

2. 做其他定义，这些定义调用 pdftex 引擎的命令处理 PDF 语言的要素。

第七章　概略

系统层由一些命令组成，这些命令的直接 (主要) 目的衔接编译引擎的命令，主要涉及：

- 格式相关的文件，如《pgfutil-latex.def》
- 文件《pgfsys.code.tex》，它包含
  - 配置文件《pgf.cfg》(目前没有什么用处)
  - 驱动文件，如《pgfsys-pdftex.def》《pgfsys-xetex.def》
  - 驱动文件包含的文件，如
    - 《pgfsys-pdftex.def》包含《pgfsys-common-pdf.def》
    - 《pgfsys-xetex.def》包含《pgfsys-dvipdfmx.def》进而包含《pgfsys-common-pdf.def》
- 软路径文件《pgfsyssoftpath.code.tex》
- 文件《pgfsysprotocol.code.tex》

系统层的命令基本都以 `\pgfsys@` 开头。在利用 Ti*k*Z/PGF 的命令绘图时，最好不要直接使用 `\special` 向.dvi 文件中写入内容，也不要直接使用编译引擎命令，例如 `\pdfliteral` 向.pdf 文件中写入内容，否则可能破坏输出文件的一致性和完整性。系统层的命令会自己决定向输出文件中写入什么内容。

用下面的命令加载系统层文件宏包：

```
\usepackage{pgfsys} % LaTeX
\input pgfsys.tex % plain TeX
\usemodule[pgfsys] % ConTeXt
```

文件《pgfsys.code.tex》会先定义一些未完善的命令，这种命令会导致警告信息，提示它们是“未实现的”(not implemented)，实际上，系统命令是由驱动文件 (driver files) 来完善的 (implement, 实际上是把命令与输出语言，例如 PDF 语言的要素联系起来)。

文件《pgfsys.code.tex》的结尾会加载配置文件 pgf.cfg (目前这个文件没有实质作用)，然后再加载驱动文件。例如对于 pdftex 编译引擎来说，相应的驱动文件是《pgfsys-pdftex.def》。驱动文件会加载《pgfsys-common-pdf.def》或者《pgfsys-common-postscript.def》之类的文件，来完善系统层的命令。

`\pgfsysdriver`

一般情况下在格式相关的.def 文件中定义了 `\pgfutil@guessdriver`. 在《pgfsys.code.tex》的结尾处，如果没有发现用户自己定义了 `\pgfsysdriver`, 那么执行 `\pgfutil@guessdriver`, 从而定义 `\pgfsysdriver`.

`\pgfsysdriver` 保存驱动文件的名称，它的默认值是 `pgfsys-\Gin@driver`, 对于 LaTeX 格式来说这个默认值比较合适。对于 plain TeX, 当使用 `pdftex` 时，它被设置为 `pgfsys-pdftex.def`, 否则被设为 `pgfsys-dvips.def`.

然后利用 `\pgfsysdriver` 载入驱动文件，驱动文件会完善系统层的命令。

第八章　文件《pgfsysprotocol.code.tex》

这个文件提供的命令的作用是将系统层的某些命令 (主要是构建路径的命令，影响图形外观的命令) 转换为输出语言 (例如 PDF 语言) 的要素 (纯文本)，并做两种处理：

- 如果 \ifpgfsysprotocol@buffered 的真值是 true, 则将得到的输出语言缓存到 (添加到) 宏 \pgfsysprotocol@c 中，简称为“缓存 (cache)”。
- 如果 \ifpgfsysprotocol@buffered 的真值是 false, 则用 \pgfsys@invoke 处理缓存的输出语言，简称为“invoke”。

命令 \pgfsys@invoke 会调用编译引擎的输出命令 (例如 pdftex 的 \pdfliteral, xetex 的 \special)。

以上操作称为“protocolling”，主要利用 \pgfsysprotocol@literal 完成。

下面的例子显示 protocolling 结果：

```
q 0 G 0 g 0.3985 w q q 0.8 1 0.8 rg 0.0 0.0 m 28.3468 28.3468 l 56.69362 0.0 l B Q Q q Q n Q
```

```
\makeatletter%
\pgfsysprotocol@getcurrentprotocol\xxxx%
\pgfsysprotocol@setcurrentprotocol\pgfutil@empty%
\pgfsysprotocol@bufferedtrue%
\begin{tikzpicture}[overlay]%
  \filldraw[fill=green!20,save path=\aaa](0,0)--(1,1)--(2,0);%
\end{tikzpicture}%
{\ttfamily\pgfsysprotocol@currentprotocol}% 展示保存的 PDF 语言要素
\pgfsysprotocol@bufferedfalse%
\pgfsysprotocol@setcurrentprotocol\xxxx%
\makeatother%
```

\pgfsysprotocol@currentprotocol

系统层的某些命令被转换为输出语言 (例如 PDF 语言) 的要素 (纯文本) 后，一般会被保存到这个宏中。这个宏的初始值被 let 为 \pgfutil@empty.

\ifpgfsysprotocol@buffered

这个 TEX-if 决定是否继续向宏 \pgfsysprotocol@currentprotocol 中添加缓存。如果不再缓存 (真值 false)，就调用 \pgfsys@invoke → P. 203 将其释放。

\pgfsysprotocol@literal#1

本命令缓存 {#1\space}, 然后根据 \ifpgfsysprotocol@buffered 的真值，决定保留缓存或者释放缓存。

```
\def\pgfsysprotocol@literal#1{%
  \pgfsysprotocol@literalbuffered{#1}%
  \ifpgfsysprotocol@buffered%
  \else%
    \pgfsysprotocol@flushcurrentprotocol%
  \fi%
}
```

\pgfsysprotocol@literalbuffered#1

本命令将 {#1\space} 全局地添加到 \pgfsysprotocol@currentprotocol 中。

其定义是：

```
\def\pgfsysprotocol@literalbuffered#1{%
  \edef\pgfsysprotocol@temp{{#1\space}}%
  \expandafter\pgfutil@g@addto@macro\expandafter\pgfsysprotocol@currentprotocol
  ↪  \pgfsysprotocol@temp%
}
```

\pgfsysprotocol@flushcurrentprotocol

本命令释放宏 \pgfsysprotocol@currentprotocol 的缓存，再清空它。

```
\def\pgfsysprotocol@flushcurrentprotocol{%
  \pgfsysprotocol@invokecurrentprotocol%
  \pgfsysprotocol@setcurrentprotocol\pgfutil@empty%
}
```

\pgfsysprotocol@invokecurrentprotocol

本命令释放缓存。

```
\def\pgfsysprotocol@invokecurrentprotocol{%
  \ifx\pgfsysprotocol@currentprotocol\pgfutil@empty%
  \else%
    \expandafter\pgfsys@invoke\expandafter{\pgfsysprotocol@currentprotocol}
    ↪  %
  \fi%
}
```

参考 \pgfsys@invoke → P. 203.

\pgfsysprotocol@getcurrentprotocol\⟨*macro*⟩

将 \⟨*macro*⟩ let 为当前的缓存内容。

```
\def\pgfsysprotocol@getcurrentprotocol#1{%
  \let#1=\pgfsysprotocol@currentprotocol%
}
```

\pgfsysprotocol@setcurrentprotocol\⟨*macro*⟩

将 \⟨*macro*⟩ 保存的内容作为当前的缓存。

```
\def\pgfsysprotocol@setcurrentprotocol#1{%
  \global\let\pgfsysprotocol@currentprotocol=#1%
}
```

在《pgfsys-common-pdf.def》中有：

```
\def\pgfsys@moveto#1#2{\pgf@sys@bp{#1}\pgf@sys@bp{#2}\pgfsysprotocol@literal{m}}
```

按 \pgf@sys@bp → P. 202 的定义，如果使用 pdftex 引擎，即载入《pgfsys-common-pdf.def》，那么

```
\pgfsys@moveto{1pt}{0pt}
等效于
\pgfsysprotocol@literal{0.99627 0 m }
```

第九章　文件《pgfsys.code.tex》

这个文件会载入《pgfkeys.code.tex》，即需要 key 机制的支持。

这个文件说的 graphic scope 指的是 \pgfsys@beginscope 和 \pgfsys@endscope 的组合；说的 id scope 指的是 \pgfsys@begin@idscope 和 \pgfsys@end@idscope 的组合;说的 TEX scope 指的是 \begingroup 和 \endgroup 的组合。

本文件使用一些寄存器：

```
\newdimen\pgf@x
\newdimen\pgf@y
\newdimen\pgf@xa
\newdimen\pgf@ya
\newdimen\pgf@xb
\newdimen\pgf@yb
\newdimen\pgf@xc
\newdimen\pgf@yc
\newdimen\pgf@xd
\newdimen\pgf@yd

\newwrite\w@pgf@writea
\newread\r@pgf@reada
\let\pgfutil@inputcheck=\r@pgf@reada

% internal counters that are always present when pgfsys is loaded
\newcount\c@pgf@counta
\newcount\c@pgf@countb
\newcount\c@pgf@countc
\newcount\c@pgf@countd

\newtoks\t@pgf@toka
\newtoks\t@pgf@tokb
\newtoks\t@pgf@tokc

% Ensure that math registers are the same (math is broken in case it
% is loaded first)
\let\pgfmath@x\pgf@x
\let\pgfmath@xa\pgf@xa
\let\pgfmath@xb\pgf@xb
\let\pgfmath@xc\pgf@xc

\let\pgfmath@y\pgf@y
\let\pgfmath@ya\pgf@ya
\let\pgfmath@yb\pgf@yb
\let\pgfmath@yc\pgf@yc

\let\c@pgfmath@counta\c@pgf@counta
```

```
\let\c@pgfmath@countb\c@pgf@countb
\let\c@pgfmath@countc\c@pgf@countc
\let\c@pgfmath@countd\c@pgf@countd
```

9.1 基本命令

\pgfset

这是 \pgfkeys 的变化版本。

```
\def\pgfset{\pgfqkeys{/pgf}}
```

\pgf@sys@tonumber\⟨*dimension register*⟩

参数 \⟨*dimen register*⟩ 是尺寸寄存器，本命令将这个尺寸的单位去掉，仅将数值插入到当前位置。

```
\def\pgf@sys@tonumber#1{\expandafter\Pgf@geT\the#1}
```

\Pgf@geT⟨*dimension string*⟩

参数 ⟨*dimension string*⟩ 是类似 1.2pt 这样的尺寸字符。本命令返回尺寸的数值部分。

\pgf@sys@bp⟨*dimension expression*⟩

参数 ⟨*dimension expression*⟩ 应当是能为尺寸寄存器 \pgf@x 赋值的表达式，赋值结果是以 pt 为单位的尺寸。本命令将 ⟨*dimension expression*⟩ 转换为以 bp 为单位的尺寸字符串 (1pt=0.99627bp)，然后将这个尺寸的数值部分和一个空格 "⟨*number*⟩␣" 全局地添加到宏 \pgfsysprotocol@currentprotocol 中。

\pgf@sys@pt⟨*dimension expression*⟩

参数 ⟨*dimension expression*⟩ 应当是能为尺寸寄存器 \pgf@x 赋值的表达式，赋值结果是以 pt 为单位的尺寸。本命令将这个尺寸的数值部分和一个空格"⟨*number*⟩␣"全局地添加到宏 \pgfsysprotocol@currentprotoco 中。

\pgf@sys@fail⟨*string*⟩

本命令检查控制序列

\csname pgf@sys@fail@⟨*string*⟩\endcsname

是否等于 \pgfutil@empty (空的)，如果是，则什么也不做；否则，将这个控制序列 let 为 \pgfutil@empty，然后发出警告，警告内容大略是：当前的驱动文件不支持 ⟨*string*⟩ 这种特性。

这个命令的目的大概是，假设想要让输出的图形具有某种 ⟨特性⟩，于是定义

```
\def\⟨macro⟩⟨arguments⟩{\pgf@sys@fail{⟨ 特性⟩}}
```

然后在驱动文件中重定义 \⟨*macro*⟩

```
\def\⟨macro⟩⟨arguments⟩{用编译引擎的命令处理 ⟨ 特性⟩}
```

这样 \⟨*macro*⟩ 就不会再调用 \pgf@sys@fail. 但如果编译引擎不能处理 ⟨特性⟩，那就无法重定义 \⟨*macro*⟩，于是 \⟨*macro*⟩ 就会调用 \pgf@sys@fail.

例如，\pgfsys@moveto 的最初定义是：

```
\def\pgfsys@moveto#1#2{\pgf@sys@fail{path constructions}}
```

而文件《pgfsys-common-pdf.def》将 \pgfsys@moveto 重定义为：

```
\def\pgfsys@moveto#1#2{\pgf@sys@bp{#1}\pgf@sys@bp{#2}\pgfsysprotocol@literal{m}}
```

\pgfsys@invoke{⟨*output language*⟩}

最初定义：

```
\def\pgfsys@invoke{\pgf@sys@fail{invoking specials}}
```

在文件《pgfsys-dvipdfmx.def》中被重定义为：

```
\def\pgfsys@invoke#1{\special{pdf:code #1}}% 命令 \special 是不可展开的
```

在文件《pgfsys-pdftex.def》中被重定义为：

```
\def\pgfsys@invoke#1{\pdfliteral{#1}}
```

\pgfsys@strcmp{⟨*tok 1*⟩}{⟨*tok 2*⟩}

本命令将 ⟨*tok 1*⟩ 和 ⟨*tok 2*⟩ 作为字符串做比较。

本命令的定义是：

```
\ifdefined\pdfstrcmp
  \let\pgfsys@strcmp\pdfstrcmp
\else\ifdefined\strcmp
  \let\pgfsys@strcmp\strcmp
\else\ifdefined\directlua
  \directlua{
  local lft = lua.get_functions_table()
  lft[\string#lft+1] = function()
      local lhs = token.scan_string()
      local rhs = token.scan_string()
      if lhs < rhs then
          tex.sprint(-2, "-1")
      elseif lhs == rhs then
          tex.sprint(-2, "0")
      else
          tex.sprint(-2, "1")
      end
  end
  token.set_lua("pgfsys@strcmp", \string#lft, "global")
  }
\else
  \def\pgfsys@strcmp#1#2{\pgf@sys@fail{string comparison}}%
\fi\fi\fi
```

在《pdftex-a.pdf》中对 \pdfstrcmp⟨*tok 1*⟩⟨*tok 2*⟩ 的解释是：如果两个字符串相等，返回 0; 如果前者先于后者，返回 −1; 其他情况返回 1.

9.2 系统命令流的开启与结束

\pgfsys@beginpicture

此命令用在 {pgfpicture} 环境的开端处，用来设置某些东西。它的最初定义是：

```
\def\pgfsys@beginpicture{}
```

如果使用 xetex 编译引擎，那么《pgfsys-dvipdfmx.def》对它的重定义是：

```
\def\pgfsys@beginpicture{\special{pdf:bcontent}}
```

\pgfsys@endpicture

此命令用在 {pgfpicture} 环境的结束处，用来设置某些东西。它的最初定义是：

```
\def\pgfsys@endpicture{}
```

如果使用 xetex 编译引擎，那么《pgfsys-dvipdfmx.def》对它的重定义是：

```
\def\pgfsys@endpicture{\special{pdf:econtent}}
```

\pgfsys@typesetpicturebox⟨*box*⟩

参数 ⟨*box*⟩ 是 TEX 的原始盒子。在图形 (picture) 编制完成后，描述图形的底层命令 (系统层命令的最后展开形式，多数是 \special 命令) 会被放入盒子 ⟨*box*⟩ 中。本命令计算盒子 ⟨*box*⟩ 高度、宽度、左侧边界、基线位置、右侧边界，并将它放到 TEX 的处理流程中。

编译下面的命令：

```
\showboxdepth=\maxdimen
\showboxbreadth=\maxdimen
A%
\makeatletter%
\newbox\aaaa%
\setbox\aaaa=\hbox{%
  \pgfsys@beginpicture%
  \pgfsys@moveto{0pt}{0pt}%
  \pgfsys@lineto{10pt}{10pt}%
  \pgfsys@lineto{20pt}{0pt}%
  \pgfsys@stroke%
  \pgfsys@endpicture%
}%
\makeatother%
\showbox\aaaa%
\box\aaaa%
BCD
```

会在.log 文件中看到：

```
\aaaa=\box62
> \box62=
\hbox(0.0+0.0)x0.0
.\special{pdf:bcontent}%beginpicture
.\special{pdf:code 0.0 0.0 m }%moveto
.\special{pdf:code 9.96277 9.96277 l }%lineto
.\special{pdf:code 19.92554 0.0 l }%lineto
.\special{pdf:code S }%stroke
.\special{pdf:econtent}%endpicture
```

\pgfsys@beginpurepicture

当绘制的图形中不包含 escaped boxes 时，可以使用这个环境。所谓“escaped boxes”指的是，中断 PGF 的绘图流程，插入的 TEX 盒子，其中可以包含各种内容。

本命令的最初定义等于 \pgfsys@beginpicture:

```
\def\pgfsys@beginpurepicture{\pgfsys@beginpicture}
```

有的驱动文件可能重定义这个命令，例如《pgfsys-common-pdf-via-dvi.def》中有：

```
\def\pgfsys@beginpurepicture{\special{pdf: content q}}
```

\pgfsys@endpurepicture

本命令的最初定义等于 \pgfsys@endpicture:

```
\def\pgfsys@endpurepicture{\pgfsys@endpicture}
```

\pgfsys@hbox{⟨*box number*⟩}

⟨*box number*⟩ 是 TEX 盒子 (水平或垂直)。

本命令中断 PGF 的绘图流程，创建“escaped box”，即插入 TEX 盒子 ⟨*box number*⟩ 的内容 (放入水平盒子里)，其中可以包含各种内容。

本命令的最初定义是：

```
\def\pgfsys@hbox#1{%
  \pgfsys@begin@idscope%
  \pgfsys@beginscope%
    \setbox#1=\hbox{\box#1}%
    \wd#1=0pt%
    \ht#1=0pt%
    \dp#1=0pt%
    \box#1%
  \pgfsys@endscope%
  \pgfsys@end@idscope%
}
```

在《pgfsys-dvipdfmx.def》中有重定义：

```
\def\pgfsys@hbox#1{%
  \pgfsys@begin@idscope%
  \pgfsys@beginscope%
    \setbox#1=\hbox{\box#1}%
    \wd#1=0pt%
    \ht#1=0pt%
    \dp#1=0pt%
    \pgfsys@dvipdfmx@suspendcontent%
    \pgfsys@invoke{0 J [] 0 d}% reset line cap and dash
    \pgfsys@dvipdfmx@start@force@reset@color%
    \box#1%
    \pgfsys@dvipdfmx@stop@force@reset@color%
    \pgfsys@dvipdfmx@unsuspendcontent%
  \pgfsys@endscope%
  \pgfsys@end@idscope%
}
```

\pgfsys@hboxsynced{⟨*box number*⟩}

类似 \pgfsys@hbox, 不过本命令会对插入的盒子应用画布变换矩阵，相当于先 \pgflowlevelsynccm, 再 \pgfsys@hbox.

本命令的最初定义是：

```
\def\pgfsys@hboxsynced#1{%
  \pgfsys@beginscope\pgflowlevelsynccm\pgfsys@hbox#1\pgfsys@endscope%
}%
```

在《pgfsys-dvipdfmx.def》中有重定义：

```
\def\pgfsys@hboxsynced#1{%
  \pgfsys@begin@idscope%
  \pgfsys@beginscope%
    \setbox#1=\hbox{\box#1}%
    \wd#1=0pt%
    \ht#1=0pt%
    \dp#1=0pt%
    \pgfsys@dvipdfmx@suspendcontent%
    \pgfsys@invoke{0 J [] 0 d}% reset line cap and dash
```

```
    \pgfsys@dvipdfmx@start@force@reset@color%
    \pgf@sys@bp@correct\pgf@pt@x%
    \pgf@sys@bp@correct\pgf@pt@y%
    \special{pdf:btrans matrix \pgf@pt@aa\space \pgf@pt@ab\space \pgf@pt@ba\space
    ↪  \pgf@pt@bb\space
      \pgf@sys@tonumber{\pgf@pt@x} \pgf@sys@tonumber{\pgf@pt@y}}%
    \box#1%
    \special{pdf:etrans}%
    \pgfsys@dvipdfmx@stop@force@reset@color%
    \pgfsys@dvipdfmx@unsuspendcontent%
  \pgfsys@endscope%
  \pgfsys@end@idscope%
}
```

\pgfsys@pictureboxsynced{⟨*box number*⟩}

本命令的最初定义是：

```
\def\pgfsys@pictureboxsynced#1{%
  {%
    \setbox0=\hbox{\pgfsys@beginpicture\box#1\pgfsys@endpicture}%
    \pgfsys@hboxsynced0%
  }%
}
```

9.3 scope 命令

\pgfsys@beginscope

本命令的最初定义是：

```
\def\pgfsys@beginscope{\pgf@sys@fail{scoping}}
```

在《pgfsys-common-pdf.def》中有重定义：

```
\def\pgfsys@beginscope{\pgfsysprotocol@literal{q}}
```

\pgfsys@endscope

本命令的最初定义是：

```
\def\pgfsys@endscope{\pgf@sys@fail{scoping}}
```

在《pgfsys-common-pdf.def》中有重定义：

```
\def\pgfsys@endscope{\pgfsysprotocol@literal{Q}}
```

9.4 构建路径的系统层命令

例如 \pgfsys@moveto, \pgfsys@lineto, \pgfsys@curveto, \pgfsys@rect, \pgfsys@closepath 之类。

9.5 设置画布变换的系统层命令

\pgfsys@transformcm{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}{⟨*d*⟩}{⟨*e*⟩}{⟨*f*⟩}

这个命令设置一个画布变换矩阵，它的参数是画布变换矩阵的元素，其中 ⟨*a*⟩, ⟨*b*⟩, ⟨*c*⟩, ⟨*d*⟩ 是不带长度单位的数值，⟨*e*⟩, ⟨*f*⟩ 是带长度单位的 TEX 尺寸。

参考《PDF Reference》(6 edition, v 1.7) 的 §4.2.3. 假设 x, y 是尺寸，对点 $(x, y, 1)$ 做变换得到点 $(x', y', 1)$，则变换为：

$$(x', y', 1) = (x, y, 1) \cdot \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

例如平移变换：

```
\pgfsys@transformcm{1}{0}{0}{1}{1cm}{1cm}
```

文件《pgfsys-common-pdf.def》对这个命令的定义是：

```
\def\pgfsys@transformcm#1#2#3#4#5#6{%
  \pgfsysprotocol@literalbuffered{#1 #2 #3 #4}\pgf@sys@bp{#5}\pgf@sys@bp{#6}
  ↪  \pgfsysprotocol@literal{cm}}
```

定义中的 "cm" 代表变换矩阵。《PDF Reference》中提到，变换的对象是坐标系统，而不是图形对象。

\pgfsys@transformshift{⟨*x displacement*⟩}{⟨*y displacement*⟩}

这是平移变换，它的定义是：

```
\def\pgfsys@transformshift#1#2{\pgfsys@transformcm{1}{0}{0}{1}{#1}{#2}}
```

参数 ⟨*x displacement*⟩ 是 x 轴方向的平移分量，⟨*y displacement*⟩ 是 y 轴方向的平移分量。

\pgfsys@transformxyscale{⟨*x scale*⟩}{⟨*y scale*⟩}

这是放缩变换，它的定义是：

```
\def\pgfsys@transformxyscale#1#2{\pgfsys@transformcm{#1}{0}{0}{#2}{0bp}{0bp}}
```

参数 ⟨*x scale*⟩ 是 x 轴方向的放缩因子，⟨*y scale*⟩ 是 y 轴方向的放缩因子。

\pgfsys@viewboxmeet{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩}{⟨x_1'⟩}{⟨y_1'⟩}{⟨x_2'⟩}{⟨y_2'⟩}

本命令的作用是：以 (⟨x_1⟩,⟨y_1⟩) 左下角点，以 (⟨x_2⟩,⟨y_2⟩) 为右上角点确定矩形 R; 以 (⟨x_1'⟩,⟨y_1'⟩) 左下角点，以 (⟨x_2'⟩,⟨y_2'⟩) 为右上角点确定矩形 R'; 对 R' 施加画布变换 (放缩、平移，保持长宽比)，使得 R' 的中心点与 R 重合，$R' \subset R$, 并且 R' 的边界恰好接触 R 的边界 (透过 R 观察 R').

本命令有默认的完善形式，主要用于动画中的 view box. 本命令的定义是

```
\def\pgfsys@viewboxmeet#1#2#3#4#5#6#7#8{\pgfsys@beginscope
↪  \pgf@sys@default@viewbox@impl{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{<}}
```

可见本命令开启一个 scope, 在本命令之后必须用 \pgfsys@endviewbox 结束。

\pgfsys@endviewbox

结束 \pgfsys@viewboxmeet 或 \pgfsys@viewboxslice 开启的 scope.

```
\def\pgfsys@endviewbox{\pgfsys@endscope}
```

\pgfsys@viewboxslice{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩}{⟨x_1'⟩}{⟨y_1'⟩}{⟨x_2'⟩}{⟨y_2'⟩}

类似 \pgfsys@viewboxmeet, 不过本命令对 R' 施加画布变换 (放缩、平移，保持长宽比)，使得恰好 $R \subset R'$.

本命令的定义是

```
\def\pgfsys@viewboxslice#1#2#3#4#5#6#7#8{\pgfsys@beginscope
↪  \pgf@sys@default@viewbox@impl{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}{>}}
```

可见本命令开启一个 scope, 在本命令之后必须用 \pgfsys@endviewbox 结束。

9.6 画、填充、剪切路径的系统命令

例如 \pgfsys@stroke, \pgfsys@fill, \pgfsys@fillstroke, \pgfsys@clipnext, \pgfsys@discardpath, \pgfsys@closestroke 之类。

9.7 图形状态

\pgfsys@setlinewidth{⟨*width*⟩}

本命令设置路径线条的线宽为 ⟨*width*⟩, 它必须是 TEX 尺寸。

《PDF Reference》中提到，线宽为 0 的线条是设备能够画出的最细的线条，其实际视觉效果是依赖设备的，因此不推荐使用 0 线宽。PGF 默认线宽值是 0.4pt.

\pgfsys@buttcap

本命令设置线条的“冠”为 butt cap.

\pgfsys@roundcap

本命令将线冠设置为 round cap.

\pgfsys@rectcap

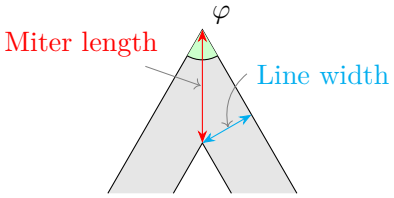
本命令将线冠设置为 rectangular cap.

\pgfsys@miterjoin

本命令将设置线结合为 miter join.

\pgfsys@setmiterlimit{⟨*factor*⟩}

本命令将设置线结合的极限为 ⟨*factor*⟩, 如果 $\langle factor\rangle < \frac{\text{Miter length}}{\text{Line width}} = \frac{1}{\sin(\frac{\varphi}{2})}$，那么线结合就自动变成 bevel join 形式。参考《PDF Reference》(6 edition, v 1.7) 的 §4.3.2.



\pgfsys@roundjoin

本命令将设置线结合为 round join.

\pgfsys@beveljoin

本命令将设置线结合为 bevel join.

\pgfsys@setdash{⟨*pattern*⟩}{⟨*phase*⟩}

这个命令确定线型，⟨*pattern*⟩ 是一个用逗号分隔的 TEX 尺寸列表，⟨*phase*⟩ 是个 TEX 尺寸。例如

```
\pgfsys@setdash{4pt,3pt}{0pt}
```

其中的 4pt,3pt 表示 4pt on,3pt off,4pt on,3pt off,...，也就是说，线型以 4pt on,3pt off 为一个周期。⟨*pattern*⟩ 中的第一个尺寸总是对应“on”。如果 ⟨*pattern*⟩ 中的尺寸个数是奇数个，例如

```
\pgfsys@setdash{4pt,3pt,2pt}{0pt}
```

那么最后一个尺寸 2pt 会被利用两次，也就是说，线型的周期是 4pt on,3pt off,2pt on,2pt off. 如果 ⟨*pattern*⟩ 是空的，那么就代表“on”。

⟨*phase*⟩ 是“相位”。例如

```
\pgfsys@setdash{4pt,3pt,2pt}{2pt}
```

就是 2pt on,3pt off,2pt on,2pt off,4pt on,3pt off,2pt on,2pt off.

\ifpgfsys@eorule

这个 TEX-if 提示是否使用奇偶规则来做填充、剪切。

9.8 颜色

9.9 图样

\pgfsys@declarepattern{⟨*name*⟩}{⟨*x1*⟩}{⟨*y1*⟩}{⟨*x2*⟩}{⟨*y2*⟩}{⟨*x step*⟩}{⟨*y step*⟩}
{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}{⟨*d*⟩} {⟨*e*⟩}{⟨*f*⟩}{⟨*code*⟩}{⟨*flag*⟩}

本命令声明一个新的图样。

如果 ⟨*flag*⟩ 的值是 0 则声明一个不可变色的图样。如果 ⟨*flag*⟩ 的值是 1 则声明一个可变色的图样。

⟨*name*⟩ 是图样的名称。

⟨*x1*⟩, ⟨*y1*⟩, ⟨*x2*⟩, ⟨*y2*⟩, ⟨*x step*⟩, ⟨*y step*⟩ 都是尺寸。

(⟨*x1*⟩，⟨*y1*⟩) 是“边界盒子”的 bottom left 点；(⟨*x2*⟩，⟨*y2*⟩) 是“边界盒子”的 top right 点。

⟨*code*⟩ 是绘制图样的命令。

⟨*x step*⟩ 和 ⟨*y step*⟩ 指定 tiling step, 即“图样砖盒子”的右上角点，“图样砖盒子”的左下角点默认为“图样砖坐标系”的原点。

⟨*a*⟩, ⟨*b*⟩, ⟨*c*⟩, ⟨*d*⟩, ⟨*e*⟩, ⟨*f*⟩ 指定针对图样的变换矩阵，其中 ⟨*a*⟩, ⟨*b*⟩, ⟨*c*⟩, ⟨*d*⟩ 是不带单位的数值，而 ⟨*e*⟩, ⟨*f*⟩ 是带单位的尺寸。

\pgfsys@patternmatrix

本命令的默认定义是 (见文件《pgfcorepatterns.code.tex》)：

```
\def\pgfsys@patternmatrix{{1.0}{0.0}{0.0}{1.0}{0.0pt}{0.0pt}}
```

本命令为 \pgfsys@declarepattern 提供参数 ⟨*a*⟩, ⟨*b*⟩, ⟨*c*⟩, ⟨*d*⟩, ⟨*e*⟩, ⟨*f*⟩, 即指定针对图样的初始的变换矩阵。

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

其中 a, b, c, d 是不带单位的数值，而 e, f, x, y, x', y' 是带单位的尺寸。

\pgfsys@setpatternuncolored{⟨*name*⟩}{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

引用名称为 ⟨*name*⟩ 的图样并为它设置颜色。在本命令之前应当使用 \pgfsys@declarepattern 声明图样 ⟨*name*⟩, 并且图样 ⟨*name*⟩ 的 ⟨*flag*⟩ 应当是 0. 参数 ⟨*red*⟩, ⟨*green*⟩, ⟨*blue*⟩ 是 rgb 模式的颜色参数。

本命令的设置会一直有效，直到再次使用本命令。

此命令的最初定义是：

```
\def\pgfsys@setpatternuncolored#1#2#3#4{\pgf@sys@fail{patterns}}
```

文件《pgfsys-pdftex.def》《pgfsys-dvipdfmx.def》对它的重定义是：

```
\def\pgfsys@setpatternuncolored#1#2#3#4{%
  \pgfsysprotocol@literal{/pgfprgb cs #2 #3 #4 /pgfpat#1\space scn}%
}
```

\pgfsys@setpatterncolored{⟨name⟩}

引用名称为 ⟨name⟩ 的图样并设置颜色。在本命令之前应当使用 \pgfsys@declarepattern 声明图样 ⟨name⟩, 并且图样 ⟨name⟩ 的 ⟨flag⟩ 应当是 1.

此命令的最初定义是：

```
\def\pgfsys@setpatterncolored#1{\pgf@sys@fail{patterns}}
```

文件《pgfsys-pdftex.def》《pgfsys-dvipdfmx.def》对它的重定义是：

```
\def\pgfsys@setpatterncolored#1{%
  \pgfsysprotocol@literal{/Pattern cs /pgfpat#1\space scn}%
}
```

9.10 图像

9.11 渐变

9.12 透明度

9.13 动画

9.14 对象的 id

可以为图形中的多种对象添加 id 标签，当某个对象具有 id 后，可以把它的 id 用于制作超链接 (hyperlinking, 用于生成动画)。

下面的命令创建的对象可以具有 id:

1. Graphic scopes, 当调用 \pgfsys@begin@idscope →P. 214 时。
2. view boxes, 当调用 \pgfsys@viewboxmeet →P. 207 或 \pgfsys@viewboxslice →P. 207 时。
3. paths, 当调用 \pgfsys@fill, \pgfsys@stroke 等命令时。
4. text boxes, 当调用 \pgfsys@hbox →P. 205, \pgfsys@hboxsynced →P. 205 等命令时。
5. animations, 当调用 \pgfsys@animate 时。

为对象创建 id 的步骤有两步。首先用命令 \pgfsys@new@id 创建一个 id 名称 (标签)，把这个名称保存在某个宏中。然后，在创建某个对象 (上文列出的命令) 之前使用命令 \pgfsys@use@id 将这个 id 赋予该对象。命令 \pgfsys@use@id 只对它后面的一个对象有效。对象不仅可以具有 id 标签, 还可以具有 type 标签。当一个对象具有数个组成部分时，对象本身可以有一个 id, 而它的每个部分都可以具有一个 type 标签。例如，一个 node 由多个部分组成 (如 background path, text 等)，各个组成部分都可以获得一个 type 标签。当引用对象的某个组成部分时，就可以使用 ⟨id⟩.⟨type⟩ 这种格式。

\pgf@sys@id@count

这是个计数器，它的值用作对象的 id.

\pgfsys@new@id{⟨macro⟩}

本命令创建一个新的 id 并保存在 ⟨macro⟩ 中，以备后用。此命令的定义是：

```
\newcount\pgf@sys@id@count

\def\pgfsys@new@id#1{%
  \edef#1{pgf\the\pgf@sys@id@count}%
  \global\advance\pgf@sys@id@count by1\relax%
}
```

执行 \pgfsys@new@id{⟨*macro*⟩} 的效果是：

1. 定义 ⟨*macro*⟩ 的值是 pgf⟨*\the\pgf@sys@id@count* 值⟩.
2. 将计数器 pgf@sys@id@count 的值 (全局地) 加 1.

\pgfsys@use@id{⟨*id*⟩}

本命令使用之前 \pgfsys@new@id 创建的 ⟨*id*⟩ 标签，将此 ⟨*id*⟩ 赋予本命令之后的一个 (仅一个) 图形对象。当对象具有 ⟨*id*⟩ 后，格式 ⟨*id*⟩.⟨*type*⟩ 就是可用的。此命令的定义是：

```
\def\pgfsys@use@id#1{%
  \edef\pgf@sys@id@current@id{#1}%
  \let\pgfsys@current@type\pgfutil@empty%
}
\let\pgf@sys@id@current@id\pgfutil@empty
```

假设 ⟨*macro*⟩ 是命令 \pgfsys@new@id 定义的宏，执行 \pgfsys@use@id{⟨*macro*⟩} 的效果是：

1. 将 ⟨*macro*⟩ 的展开值保存到 \pgf@sys@id@current@id 中。
2. 令 \pgfsys@current@type 等于 \pgfutil@empty.

\pgf@sys@id@current@id

这个宏保存一个 id, 这个 id 可以指派给下一个对象。这个宏的初始状态等于 \pgfutil@empty.

\pgfsys@clear@id

清空 \pgf@sys@id@current@id. 当一个 id 指派给某个对象后，就应当清空它。

```
\def\pgfsys@clear@id{%
  \let\pgf@sys@id@current@id\pgfutil@empty%
}
```

\pgfsys@register@type{⟨*type*⟩}

⟨*type*⟩ 是文字或者展开为文字的宏，这些文字用作 type 名称。
本命令检查 \csname pgf@sys@reg@type@⟨*type*⟩\endcsname 是否已定义，如果未定义，则设置一个花括号组，在这个组内执行：

1. 把宏 \pgf@sys@type@count 的值赋予寄存器 \c@pgf@counta.
2. 全局地把寄存器 \c@pgf@counta 的值加 1.
3. 把寄存器 \c@pgf@counta 的值保存在宏 \pgf@sys@type@count 中。
4. 全局地定义命令 \csname pgf@sys@reg@type@⟨*type*⟩\endcsname 的值为

   y⟨ 值 *\the\c@pgf@counta*⟩

   这就是 ⟨*type*⟩ 对应的控制序列，它的内部值有一个编号。

以上操作是在组内执行的，所以 type 对应的编号总是从 0 开始。

\pgf@sys@type@count

这个宏保存一个整数值，它的初始值是 0.

命令 \pgfsys@register@type{⟨type⟩} 的作用是“注册”⟨type⟩, 也就是确保命令

\csname pgf@sys@reg@type@⟨type⟩\endcsname

有定义。文件列出了预定义的 type, 即 background, path, text, background.path, 它们的定义是:

```
\def\pgf@sys@reg@type@{}
\def\pgf@sys@reg@type@background{b}
\def\pgf@sys@reg@type@path{p}
\def\pgf@sys@reg@type@text{t}
\expandafter\def\csname pgf@sys@reg@type@background.path\endcsname{bp}
```

\pgfsys@use@type{⟨type⟩}

此命令的定义是:

```
\def\pgfsys@use@type#1{%
  \edef\pgfsys@current@type{#1}%
  \pgfsys@register@type\pgfsys@current@type%
}
\let\pgfsys@current@type\pgfutil@empty
```

执行 \pgfsys@use@type{⟨type⟩} 导致:

1. 执行 \edef\pgf@sys@current@type{⟨type⟩}.
2. 执行 \pgfsys@register@type\pgfsys@current@type, 注册 ⟨type⟩.

\pgfsys@append@type{⟨text⟩}

本命令检查 \pgfsys@current@type 的值是否为空:

- 如果 \pgfsys@current@type 的值为空，则执行 \pgfsys@use@type{⟨text⟩}, 将 ⟨type⟩ 注册为一个 type.
- 否则,执行 \pgfsys@use@type{\pgfsys@current@type.⟨text⟩}, 这导致对 \pgfsys@current@type 的重定义，给它原来的值加后缀“.⟨text⟩”, 注册一个新的 type.

\pgfsys@push@type

将当前 type 的定义放入一个 stack 中，保存这个 stack 的是宏 \pgf@sys@typestack, 这个宏不是全局定义的，因此最好不要用 TEX 组限制它。

本命令对宏 \pgf@sys@typestack 的定义是:

```
\def\pgf@sys@typestack{%
  \def\pgfsys@current@type{展开一次的 \pgfsys@current@type}%
  \def\pgf@sys@typestack{展开一次的 \pgf@sys@typestack}%
}
```

可见，如果多次使用本命令，那么宏 \pgf@sys@typestack 保存一个多层套嵌的 \def 结构。

在 3 次使用命令 \pgfsys@push@type 后得到:

```
\def\pgf@sys@typestack{
  \def\pgfsys@current@type{⟨type 3⟩}%
  \def\pgf@sys@typestack{
    \def\pgfsys@current@type{⟨type 2⟩}%
    \def\pgf@sys@typestack{
      \def\pgfsys@current@type{⟨type 1⟩}%
      \def\pgf@sys@typestack{}
    }
  }
}%
```

\pgfsys@pop@type

调出 stack of types 中顶端的 type 的定义：

```
\def\pgfsys@pop@type{\pgf@sys@typestack}
% Pops the last id from the stack.
```

\pgfsys@id@ref{⟨*id number*⟩}{⟨*type name*⟩}

此命令的定义是：

```
\def\pgfsys@id@ref#1#2{#1\csname pgf@sys@reg@type@#2\endcsname}
```

按代码注释，⟨*id number*⟩ 是 id 编号，⟨*type name*⟩ 是用 \pgfsys@id@register@type 注册过的 type 名称。

\pgfsys@id@ref{⟨*id number*⟩}{⟨*type name*⟩} 得到一个 id-type-pair.

\pgfsys@id@refcurrent

此命令的定义是：

```
\def\pgfsys@id@refcurrent{\pgfsys@id@ref{\pgf@sys@id@current@id}{
↪  \pgfsys@current@type}}
% Expands to a text that can be inserted as a reference to the current
% id-type pair in use.
```

命令 \pgf@sys@id@current@id 是 \pgfsys@use@id 的子命令。

命令 \pgfsys@current@type 是 \pgfsys@use@type 的子命令。

本命令得到当前可用的 (可以指派给下一个对象的) id-type-pair.

\pgfsys@call@save

此命令的定义是：

```
\def\pgfsys@call@save{%
  \pgfsys@beginscope%
  \pgfsys@beg@save%
  \expandafter\global\expandafter\let\csname pgf@sys@att@beg@\pgfsys@id@refcurrent
  ↪  \endcsname\relax%
  \expandafter\global\expandafter\let\csname pgf@sys@att@end@\pgfsys@id@refcurrent
  ↪  \endcsname\relax%
}
```

此命令：

1. 用 \pgfsys@beginscope 开启一个 graphics scope.
2. 执行 \pgfsys@beg@save, 此命令由 \pgfsys@begin@idscope →P. 214 的展开过程定义，就是控制序列 \csname pgf@sys@att@beg@⟨*id*⟩⟨*type*⟩\endcsname.
3. 全局地规定 \csname pgf@sys@att@beg@\pgfsys@id@refcurrent\endcsname 等于 \relax.
4. 全局地规定 \csname pgf@sys@att@end@\pgfsys@id@refcurrent\endcsname 等于 \relax.

\pgfsys@call@end

此命令的定义是：

```
\def\pgfsys@call@end{%
  \pgfsys@end@save%
  \pgfsys@endscope%
}
```

\pgfsys@end@save 由 \pgfsys@begin@idscope →P. 214 的展开过程定义,就是控制序列 \csname pgf@sys@att@end

\pgfsys@invalidate@currentid

此命令的定义是：

```
\def\pgfsys@invalidate@currentid{%
  \expandafter\global\expandafter\let\csname pgf@sys@id@keylist@
  ↪  \pgfsys@id@refcurrent\endcsname\pgfutil@empty%
}
% Mark the current id-type pair as used.
```

\pgfsys@begin@idscope

开启一个"id scope", 本命令开启的 id scope 可能是 (也可能不是) 一个 graphics scope; 如果是一个 graphics scope, 则这个 scope 中的图形会获得 id-type-pair, 以备稍后引用它。如果本命令之前没有使用 id 或者 id-type-pair 已经被用过，那么就未必开启一个 graphics scope(这依赖驱动), 但总会开启一个 TEX scope.

注意 \pgfsys@beginscope 并不会利用当前的 id-type-pair.

本命令：

1. \begingroup
2. 获取当前可用的 id-type-pair.
3. 将 \pgfsys@beg@save let 为 \csname pgf@sys@att@beg@⟨*id*⟩⟨*type*⟩\endcsname, 这个控制序列由 \pgfsys@attach@to@id 定义。
4. 将 \pgfsys@end@save let 为 \csname pgf@sys@att@end@⟨*id*⟩⟨*type*⟩\endcsname, 这个控制序列由 \pgfsys@attach@to@id 定义。
5. 如果 \pgfsys@beg@save 与 \pgfsys@end@save 两者之一有定义 (不等于 \relax),就调用 \pgfsys@call@save
6. \pgfsys@invalidate@currentid
7. \begingroup

\pgfsys@end@idscope

本命令结束由 \pgfsys@begin@idscope 开启的 graphic scopes 和 TEX scopes.

```
\def\pgfsys@end@idscope{
      \endgroup%
    \ifx\pgfsys@beg@save\relax%
      \ifx\pgfsys@end@save\relax%
      \else%
        \pgfsys@call@end%
      \fi%
    \else%
      \pgfsys@call@end%
    \fi%
  \endgroup
}
% Ends an id scope.
```

\pgfsys@attach@to@id{⟨*id*⟩}{⟨*type*⟩}{⟨*begin code*⟩}{⟨*end code*⟩}{⟨*setup code*⟩}

在使用 ⟨*id*⟩-⟨*type*⟩-pair 之前 (赋予某个对象之前)，可以使用本命令将 ⟨*id*⟩-⟨*type*⟩-pair 跟这些 code 联系起来。其中的 ⟨*id*⟩ 必须是由 \pgfsys@new@id 创建的。

本命令：

1. \pgfsys@register@type{⟨*type*⟩}, 检查或注册 ⟨*type*⟩.
2. 全局定义 \csname pgf@sys@att@beg@⟨*id*⟩⟨*type*⟩\endcsname, 假设这个控制序列原来保存的内容是 ⟨*origin begin code*⟩, 那么重定义它：

```
\gdef\csname pgf@sys@att@beg@⟨id⟩⟨type⟩\endcsname{⟨origin begin code⟩⟨begin code⟩}
```

3. 全局定义 \csname pgf@sys@att@end@⟨*id*⟩⟨*type*⟩\endcsname, 假设这个控制序列原来保存的内容是 ⟨*origin end code*⟩, 那么重定义它：

```
\gdef\csname pgf@sys@att@end@⟨id⟩⟨type⟩\endcsname{⟨end code⟩⟨origin end code⟩}
```

本命令的定义格式只有 4 个变量，⟨*setup code*⟩ 不属于本命令的参数。

\pgfsys@begin@idscope 与 \pgfsys@end@idscope 的组合是：

```
\begingroup
%...
% 假设 \csname pgf@sys@att@beg@idtype\endcsname 或
% \csname pgf@sys@att@end@idtype\endcsname 有定义
  \pgfsys@beginscope
% 执行 \csname pgf@sys@att@beg@idtype\endcsname
% 清除 \csname pgf@sys@att@beg@idtype\endcsname 和
% \csname pgf@sys@att@end@idtype\endcsname
% \pgfsys@invalidate@currentid
    \begingroup
%... 组合内容
    \endgroup
  \pgfsys@endscope
\endgroup
```

9.15 RDF

9.16 重复利用对象

参考 \pgfsys@invoke → P. 203.

\pgfsys@defobject{⟨*name*⟩}{⟨*lower left*⟩}{⟨*upper right*⟩}{⟨*code*⟩}

本命令的定义是：

```
\def\pgfsys@defobject#1#2#3#4{%
  \pgfsysprotocol@getcurrentprotocol\pgfsys@temp%
  {%
    \pgfpicturetrue%
    \pgfsysprotocol@setcurrentprotocol\pgfutil@empty%
    \pgfsysprotocol@bufferedtrue%
    \pgfsys@beginscope%
    #4%
    \pgfsys@endscope%
    \pgfsysprotocol@getcurrentprotocol\pgfsys@@temp%
    \expandafter\global\expandafter\let\csname #1\endcsname=\pgfsys@@temp%
  }%
  \pgfsysprotocol@setcurrentprotocol\pgfsys@temp%
}
```

可见本命令的第 2 第 3 个参数无用。

本命令在 system scope 内执行 ⟨*code*⟩ (绘制对象的代码),这些代码会被 protocoling. 在真值 \pgfsysprotocol@buf 之下，所得的 protocoling 代码被缓存，然后再全局地保存到控制序列 \csname ⟨*name*⟩\endcsname 中。

\pgfsys@useobject{⟨*name*⟩}{⟨*extra code*⟩}

本命令的定义是：

```
\def\pgfsys@useobject#1#2{%
  \pgfsysprotocol@getcurrentprotocol\pgfsys@temp%
  {%
    \pgfsysprotocol@setcurrentprotocol\pgfutil@empty%
    \pgfsysprotocol@bufferedfalse%
    #2%
    \expandafter\pgfsysprotocol@setcurrentprotocol\csname #1\endcsname%
    \pgfsysprotocol@invokecurrentprotocol%
  }%
  \pgfsysprotocol@setcurrentprotocol\pgfsys@temp%
}
```

本命令先执行 ⟨*extra code*⟩, 再引入由 ⟨*name*⟩ 缓存的代码。

\pgfsys@marker@declare{⟨*macro*⟩}{⟨*code*⟩}

类似 \pgfsys@defobject, 不过本命令保存的 mark 可以作为箭头用于动画中。本命令全局定义宏 ⟨*macro*⟩, 令它保存一个对象 id 编号。执行 ⟨*code*⟩ 得到的缓存会被全局地保存在控制序列 \csname pgf@sys@marker@prot@⟨*macro*⟩\endcsname 中。

\pgfsys@marker@use{⟨*macro*⟩}

参数 ⟨*macro*⟩ 是上一命令定义的宏。本命令 invoke 这个宏保存的代码。

```
\def\pgfsys@marker@use#1{%
  \pgfsysprotocol@literal{\csname pgf@sys@marker@prot@#1\endcsname}%
}
```

9.17 使得对象不可见的命令

\pgfsys@begininvisible

处于 \pgfsys@begininvisible 和 \pgfsys@endinvisible 之间的输出将被抑制，从而不可见。本命令的定义是：

```
\def\pgfsys@begininvisible{\pgfsys@transformcm{1}{0}{0}{1}{2000bp}{2000bp}}
```

可见本命令做一个画布平移，平移向量具有较大的尺寸，也就是将对象平移到了较远的地方，超出一般的页面幅度。

\pgfsys@endinvisible

本命令配合 \pgfsys@begininvisible, 其定义是：

```
\def\pgfsys@endinvisible{\pgfsys@transformcm{1}{0}{0}{1}{-2000bp}{-2000bp}}
```

9.18 页面尺寸

9.19 跟踪页面上的位置

跟踪页面上某个位置的机制分为两个步骤，至少需要编译.tex 文件两次：首先在页面的某个位置处“做标记”(具体细节依赖后台驱动)，第一次编译，将这个“标记”的页面坐标输出到.aux 文件中；然后第二次编译，从.aux 文件中读取这个页面坐标。

\pgfsys@markposition{⟨*name*⟩}

标记本命令所处的页面位置，并将这个位置命名为 ⟨*name*⟩, 可以在其他地方使用 ⟨*name*⟩ 引用这个页面位置。应当在正常排版的流程中使用本命令，例如：

```
The value of $x$ is \pgfsys@markposition{here}important.
```

本命令的最初定义是：

```
\def\pgfsys@markposition#1{\pgf@sys@fail{marking the current position}}
```

文件《pgfsys-pdftex.def》对它的重定义是：

```
\def\pgfsys@markposition#1{%
  \pdfsavepos%
  \edef\pgf@temp{#1}%
  \expandafter\pgfutil@writetoaux\expandafter{%
    \expandafter\noexpand\expandafter\pgfsyspdfmark\expandafter{\pgf@temp}{
    ↪  \the\pdflastxpos}{\the\pdflastypos}}%
}

\def\pgfsyspdfmark#1#2#3{%
  \expandafter\gdef\csname pgf@sys@pdf@mark@pos@#1\endcsname{\pgfqpoint{#2sp}{#3sp
  ↪  }}%
  \pgfutil@check@rerun{#1}{{#2}{#3}}%
}
```

其中 \pdfsavepos 是 pdftex 的命令，它获取当前位置的绝对坐标，参考点是页面的左下角，可以用 \pdflastxpos 和 \pdflastypos 得到这个位置的 x, y 坐标尺寸 (\the)。在页面排版输出后 (\shipout)，这个位置的坐标被写入.aux 文件。

在使用 LaTeX 格式的情况下，文件《pgfutil-latex.def》对 \pgfutil@check@rerun 的重定义是：

```
\def\pgfutil@check@rerun#1#2{\@newl@bel{pgf@lab}{#1}{#2}}
```

综合以上，\pgfsys@markposition 会向.aux 文件中 (用\pgfutil@writetoaux) 写入命令：

```
\pgfsyspdfmark{⟨name⟩}{⟨x coord⟩}{⟨y coord⟩}
```

当第 2 次编译.tex 文件，载入.aux 文件时，这个 \pgfsyspdfmark 会全局地定义 2 个控制序列：

- \csname pgf@sys@pdf@mark@pos@⟨*name*⟩\endcsname, 保存位置点 \pgfqpoint{⟨*x dimension*⟩}{⟨*y dimension*⟩}.
- \csname pgf@lab@⟨*name*⟩\endcsname, 保存 {⟨*x dimension*⟩}{⟨*y dimension*⟩}.

\pgfsys@getposition{⟨*name*⟩}{⟨*macro*⟩}

本命令将名称为 ⟨*name*⟩ 的页面上的位置点保存到宏 ⟨*macro*⟩ 中。

如果位置 ⟨*name*⟩ 没有被标记，或者在第一次编译.tex 文件时 (此时位置点尚不可用)，⟨*macro*⟩ 会等于 \relax. 如果成功引用 ⟨*name*⟩ 位置,那么宏 ⟨*macro*⟩ 会被定义为 \pgfqpoint...，例如 \pgfqpoint{2cm}{3cm} 代表的位置是：从页面的原点向右 2cm，向上 3cm 所到达的页面位置。如果“引用者”与“被引用者”处于同一页面，那么它们使用同一页面的坐标系，原点在页面左下角；如果“引用者”与“被引用者”不处于同一页面，那么就不能保证二者所参考的原点一致。

页面的左下角位置的名称被预定义为 pgfpageorigin, 可以用命令 \pgfsys@getposition{pgfpageorigin} 引用这个位置。

本命令的最初定义是：

```
\def\pgfsys@getposition#1#2{\let#2=\relax}
```

文件《pgfsys-pdftex.def》对它的重定义是：

```
\def\pgfsys@getposition#1#2{%
  \edef\pgf@marshal{\let\noexpand#2=\expandafter\noexpand\csname
  ↪ pgf@sys@pdf@mark@pos@#1\endcsname}%
  \pgf@marshal%
}
```

可见宏 ⟨*macro*⟩ 保存的是 `\pgfqpoint{`⟨*x dimension*⟩`}{`⟨*y dimension*⟩`}`.

下面的例子：

```
The value of $x$ is \pgfsys@markposition{here}important.

Lots of text.

\hbox{\pgfsys@markposition{myorigin}%
\begin{pgfpicture}
  % Switch of size protocol
  \pgfpathmoveto{\pgfpointorigin}
  \pgfusepath{use as bounding box}
  \pgfsys@getposition{here}{\hereposition}
  \pgfsys@getposition{myorigin}{\thispictureposition}
  \pgftransformshift{\pgfpointscale{-1}{\thispictureposition}}% 回到页面的原点
  \pgftransformshift{\hereposition}% 平移到 here 位置
  \pgfpathcircle{\pgfpointorigin}{1cm}
  \pgfusepath{draw}
\end{pgfpicture}}
```

第十章　软路径

以 \pgfsys@ 开头的路径构建命令，例如 \pgfsys@moveto, \pgfsys@lineto, \pgfsys@curveto 是构建“硬路径”的命令，它们的展开会导致 protocoling 操作。

以 \pgfsyssoftpath@ 开头的路径构建命令创建“软路径”，软路径命令展开后变成硬路径命令。所谓软路径指的是由 \pgfsyssoftpath@movetotoken, \pgfsyssoftpath@linetotoken 之类的软路径 token 构成的记号序列。

Ti*k*Z 对命令 \draw (0bp,0bp) -- (10bp,0bp); 的处理是个复杂的过程：

1. 首先这个命令转换成基本层的命令

```
\pgfpathmoveto{\pgfpoint{0bp}{0bp}}
\pgfpathlineto{\pgfpoint{10bp}{0bp}}
\pgfusepath{stroke}
```

2. 基本层的命令 \pgfpathxxxx 调用软路径的命令，例如
   - \pgfpathmoveto 调用 \pgfsyssoftpath@moveto, 向当前的软路径中添加 \pgfsyssoftpath@movetotoken,
   - \pgfpathlineto 调用 \pgfsyssoftpath@lineto, 向当前的软路径中添加 \pgfsyssoftpath@linetotoken,

   当前软路径会被保存在一个内部宏中，每使用一个软路径命令，当前软路径就会被延伸，保存软路径的内部宏也会被更新。在创建软路径的过程中，软路径是有缓存的 (有 2 个级别的缓存)，最终的软路径保存在 \pgfsyssoftpath@thepath 中。
3. 当遇到命令 \pgfusepath 时，保存在内部宏中的软路径会被调用，软路径中的各种记号，如
   - \pgfsyssoftpath@movetotoken 会调用 \pgfsys@moveto,
   - \pgfsyssoftpath@linetotoken 会调用 \pgfsys@lineto,

   创建硬路径。

也就是说，先创建软路径，然后用软路径创建硬路径。采用这种处理的原因是：

(i) 在软路径变成硬路径之前，可以修改软路径。

(ii) PDF 要求在路径内部不能有“不用于创建路径”的命令，例如下面的代码

```
\pgfsys@moveto{0pt}{0pt}%
\pgfsys@setlinewidth{1pt}%
\pgfsys@lineto{10pt}{10pt}%
\pgfsys@stroke%
```

被 invoke 后得到的 PDF 语言要素是：

```
0.0 0.0 m 0.99628 w 9.96277 9.96277 l S
```

其中的 \pgfsys@setlinewidth{1pt} (生成 0.99628 w) 会导致 PDF 文件被“损坏”，有的 PDF 阅读器不能容忍这种“损坏”。由于只有软路径命令向 \pgfsyssoftpath@thepath 中添加软路径记号，所以最终得到的软路径中不会包含线宽、颜色等要素，从而避免这种“损坏”问题。

10.1 保存、调用一个软路径

当使用命令 \pgfsyssoftpath@xxxx 时就开启软路径创建过程。在创建软路径的过程中，可以把当前软路径保存在某个宏中，之后可以通过这个宏调用所保存的软路径。

\pgfsyssoftpath@getcurrentpath{⟨*macro name*⟩}

这个命令把当前软路径保存在 ⟨*macro name*⟩ 中，本命令定义 ⟨*macro name*⟩.

\pgfsyssoftpath@setcurrentpath{⟨*macro name*⟩}

这个命令把保存在宏 ⟨*macro name*⟩ 中的软路径调出，用作当前软路径，调出这个软路径后，可以对这个软路径做修改。

\pgfsyssoftpath@invokecurrentpath

此命令的定义是：

```
\def\pgfsyssoftpath@invokecurrentpath{%
  \pgfsyssoftpath@thepath%
  \pgfsyssoftpath@bigbuffer%
  \pgfsyssoftpath@smallbuffer%
}
```

软路径的各部分可能 (按构造次序) 分别保存在宏

- \pgfsyssoftpath@thepath → P. 222
- \pgfsyssoftpath@bigbuffer → P. 222
- \pgfsyssoftpath@smallbuffer → P. 222

中，所以此命令会把软路径插入到当前位置。

\pgfsyssoftpath@flushcurrentpath

此命令的定义是：

```
\def\pgfsyssoftpath@flushcurrentpath{%
  \pgfsyssoftpath@invokecurrentpath%
  \pgfsyssoftpath@setcurrentpath\pgfutil@empty%
}
```

此命令将软路径插入到当前位置，然后清空软路径。

10.2 创建软路径的命令

\pgfsyssoftpath@moveto{⟨*x*⟩}{⟨*y*⟩}

这个命令给当前软路径添加一个 moveto 操作。⟨*x*⟩, ⟨*y*⟩ 都是 TEX 尺寸。

```
\pgfsyssoftpath@moveto{0pt}{0pt}
\pgfsyssoftpath@lineto{10pt}{10pt}
\pgfsyssoftpath@flushcurrentpath
\pgfsys@stroke
```

\pgfsyssoftpath@lineto{⟨*x*⟩}{⟨*y*⟩}

这个命令给当前软路径添加一个 lineto 操作。⟨*x*⟩, ⟨*y*⟩ 都是 TEX 尺寸。

\pgfsyssoftpath@curveto{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}{⟨*d*⟩}{⟨*x*⟩}{⟨*y*⟩}

这个命令给当前软路径添加一个 curveto 操作 (控制曲线)，以当前点为起点，以 (x, y) 为终点，以 (a, b)，(c, d) 为控制点。

\pgfsyssoftpath@rect{⟨lower left x⟩}{⟨lower left y⟩}{⟨width⟩}{⟨height⟩}

这个命令给当前软路径添加一个矩形。

\pgfsyssoftpath@closepath

这个命令给当前软路径添加一个 close 操作，使之闭合。

10.3 软路径数据结构

软路径按照某个标准化的结构来保存，以利于对其做修改。通常，一个软路径是由很多个三元组构成的序列。一个 token 和两个尺寸组成一个三元组，例如：

```
\pgfsyssoftpath@movetotoken{0bp}{1bp}
```

是个三元组，它的三个组成部分是：\pgfsyssoftpath@movetotoken, {0bp}, {1bp}. 再如：

```
\pgfsyssoftpath@movetotoken{0bp}{0bp}\pgfsyssoftpath@linetotoken{10bp}{0bp}
```

是两个三元组。

一个矩形命令会被转为 2 个有序 三元组，例如

```
\pgfsyssoftpath@rect{0bp}{0bp}{1cm}{2cm}
```

会被保存为：

```
\pgfsyssoftpath@rectcornertoken{0bp}{0bp}
\pgfsyssoftpath@rectsizetoken{1cm}{2cm}
```

一个 curve-to 操作会被转为 3 个有序 三元组，例如

```
\pgfsyssoftpath@curveto{1bp}{2bp}{3bp}{4bp}{5bp}{6bp}
```

会被保存为：

```
\pgfsyssoftpath@curvetosupportatoken{1bp}{2bp}
\pgfsyssoftpath@curvetosupportbtoken{3bp}{4bp}
\pgfsyssoftpath@curvetotoken{5bp}{6bp}
```

构成三元组的 token 有如下几个：

- \pgfsyssoftpath@movetotoken, 对应 moveto 操作。
- \pgfsyssoftpath@linetotoken, 对应 lineto 操作。
- \pgfsyssoftpath@curvetosupportatoken, 对应 curveto 操作的第一个控制点。
- \pgfsyssoftpath@curvetosupportbtoken, 对应 curveto 操作的第二个控制点。
- \pgfsyssoftpath@curvetotoken, 对应 curveto 操作的终点。
- \pgfsyssoftpath@rectcornertoken, 对应矩形命令的左下角点。
- \pgfsyssoftpath@rectsizetoken 矩形命令的宽度和高度。
- \pgfsyssoftpath@closepath, 对应 close 操作。

10.4 文件《pgfsyssoftpath.code.tex》

```
\let\pgfsyssoftpath@thepath=\pgfutil@empty
\let\pgfsyssoftpath@bigbuffer=\pgfutil@empty
\let\pgfsyssoftpath@smallbuffer=\pgfutil@empty
\newcount\pgfsyssoftpath@smallbuffer@items
\newcount\pgfsyssoftpath@bigbuffer@items
```

\pgfsyssoftpath@thepath

这个宏用于保存当前的软路径，它一般由 \pgfsyssoftpath@flushbuffers 定义，其初始值被 let 为 \pgfutil@empty.

\pgfsyssoftpath@bigbuffer

这个宏是用于临时保存某些软路径构造命令的“大容器”，其初始值被 let 为 \pgfutil@empty, 它会被 \pgfsyssoftpath@addtocurrentpath, \pgfsyssoftpath@smalloverflow 重定义。

\pgfsyssoftpath@smallbuffer

这个宏是用于临时保存某些软路径构造命令的“小容器”，其初始值被 let 为 \pgfutil@empty, 它会被 \pgfsyssoftpath@addtocurrentpath, \pgfsyssoftpath@smalloverflow 重定义。

“小容器”被装满后，其内容会被转移到“大容器”。

“大容器”被装满后，其内容会被转移到 \pgfsyssoftpath@thepath.

\pgfsyssoftpath@flushbuffers

这个命令的定义是：

```
\def\pgfsyssoftpath@flushbuffers{%
  \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
  \expandafter\gdef%
  \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
  \expandafter\pgfsyssoftpath@thepath%
  \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
  \expandafter{%
    \expandafter\expandafter\expandafter\pgfsyssoftpath@thepath%
    \expandafter\pgfsyssoftpath@bigbuffer\pgfsyssoftpath@smallbuffer}%
  \global\let\pgfsyssoftpath@smallbuffer=\pgfutil@empty
  \global\let\pgfsyssoftpath@bigbuffer=\pgfutil@empty
  \global\pgfsyssoftpath@bigbuffer@items0\relax%
  \global\pgfsyssoftpath@smallbuffer@items0\relax%
}
```

本命令全局地重定义宏 \pgfsyssoftpath@thepath, 然后全局地重置两个宏、计数器的值。在定义代码中用了数个 \expandafter, 其作用是先展开 \pgfsyssoftpath@smallbuffer, 得到这个宏的定义内容，再展开 \pgfsyssoftpath@bigbuffer, 再展开 \pgfsyssoftpath@thepath, 即：

```
\gdef\pgfsyssoftpath@thepath{%
            第三展开的 \pgfsyssoftpath@thepath%
        第二展开的 \pgfsyssoftpath@bigbuffer
    第一展开的 \pgfsyssoftpath@smallbuffer
}%
```

也就是把保存在 \pgfsyssoftpath@bigbuffer, \pgfsyssoftpath@smallbuffer 中的内容，附加到 \pgfsyssoftpath@thepath 中。从字面上看，本命令是“刷新” \pgfsyssoftpath@thepath 的命令。

\pgfsyssoftpath@getcurrentpath⟨*a macro*⟩

这个命令的定义是：

```
\def\pgfsyssoftpath@getcurrentpath#1{%
  \pgfsyssoftpath@flushbuffers%
  \let#1=\pgfsyssoftpath@thepath%
}
```

本命令使得 ⟨*a macro*⟩ 等于当前的 \pgfsyssoftpath@thepath.

\pgfsyssoftpath@setcurrentpath⟨*a macro*⟩

此命令的定义是：

```
\def\pgfsyssoftpath@setcurrentpath#1{%
  \global\let\pgfsyssoftpath@thepath=#1%
  \global\let\pgfsyssoftpath@smallbuffer=\pgfutil@empty
  \global\let\pgfsyssoftpath@bigbuffer=\pgfutil@empty
  \global\pgfsyssoftpath@bigbuffer@items0\relax%
  \global\pgfsyssoftpath@smallbuffer@items0\relax%
}
```

本命令全局地重定义 \pgfsyssoftpath@thepath，使之等于 ⟨*a macro*⟩，然后全局地重置两个宏、计数器的值。

\pgfsyssoftpath@addtocurrentpath⟨软路径构造命令⟩

此命令的定义是：

```
\def\pgfsyssoftpath@addtocurrentpath#1{%
  \global\advance\pgfsyssoftpath@smallbuffer@items by1\relax%
  \ifnum\pgfsyssoftpath@smallbuffer@items<40\relax% good number?
    \expandafter\gdef\expandafter\pgfsyssoftpath@smallbuffer\expandafter{
    ↪  \pgfsyssoftpath@smallbuffer#1}%
  \else%
    \pgfsyssoftpath@smalloverflow{#1}%
  \fi%
}
```

本命令首先全局地给计数器 \pgfsyssoftpath@smallbuffer@items 的值加 1，然后检查这个计数器的值，

- 如果这个计数器的值 < 40，则全局地重定义 \pgfsyssoftpath@smallbuffer，向其中添加 ⟨软路径构造命令⟩。
- 如果这个计数器的值 $\geqslant 40$，则执行 \pgfsyssoftpath@smalloverflow.

\pgfsyssoftpath@smalloverflow⟨软路径构造命令⟩

此命令的定义是：

```
\def\pgfsyssoftpath@smalloverflow#1{%
  \global\advance\pgfsyssoftpath@bigbuffer@items by1\relax%
  \ifnum\pgfsyssoftpath@bigbuffer@items<30\relax% good number?
    \expandafter\expandafter\expandafter\gdef%
    \expandafter\expandafter\expandafter\pgfsyssoftpath@bigbuffer%
    \expandafter\expandafter\expandafter{\expandafter\pgfsyssoftpath@bigbuffer
    ↪  \pgfsyssoftpath@smallbuffer#1}%
    \global\let\pgfsyssoftpath@smallbuffer=\pgfutil@empty%
    \global\pgfsyssoftpath@smallbuffer@items0\relax%
  \else%
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
    \expandafter\gdef%
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
    \expandafter\pgfsyssoftpath@thepath%
    \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
    \expandafter{%
      \expandafter\expandafter\expandafter\pgfsyssoftpath@thepath
      \expandafter\pgfsyssoftpath@bigbuffer\pgfsyssoftpath@smallbuffer#1}%
    \global\let\pgfsyssoftpath@smallbuffer=\pgfutil@empty
    \global\let\pgfsyssoftpath@bigbuffer=\pgfutil@empty
    \global\pgfsyssoftpath@bigbuffer@items0\relax%
    \global\pgfsyssoftpath@smallbuffer@items0\relax%
```

```
  \fi%
}
```

本命令被 \pgfsyssoftpath@addtocurrentpath 调用 (\pgfsyssoftpath@smallbuffer@items$\geqslant$ 40). 本命令首先全局地给计数器 \pgfsyssoftpath@bigbuffer@items 的值加 1，然后检查这个计数器的值，

- 如果这个计数器的值 <30, 则全局地重定义 \pgfsyssoftpath@bigbuffer:

```
\gdef\pgfsyssoftpath@bigbuffer{
            第二展开的 \pgfsyssoftpath@bigbuffer
        第一展开的 \pgfsyssoftpath@smallbuffer
    ⟨ 软路径构造命令⟩
}
```

  然后再全局地清空 \pgfsyssoftpath@smallbuffer.

  然后再全局地重置 \pgfsyssoftpath@smallbuffer@items 的值为 0.

- 如果这个计数器的值 $\geqslant 30$, 则全局地重定义 \pgfsyssoftpath@thepath:

```
\gdef\pgfsyssoftpath@thepath{
                第三展开的 \pgfsyssoftpath@thepath
            第二展开的 \pgfsyssoftpath@bigbuffer
        第一展开的 \pgfsyssoftpath@smallbuffer
    ⟨ 软路径构造命令⟩
}
```

  然后全局地重置两个宏、计数器的值。

\pgfsyssoftpath@moveto{⟨*dimend x*⟩}{⟨*dimend y*⟩}

参数 ⟨*dimend x*⟩, ⟨*dimend y*⟩ 应当是 10pt 之类的尺寸，或者保存尺寸的宏，或者是 \the⟨尺寸寄存器⟩ 等形式。此命令的定义是：

```
\def\pgfsyssoftpath@moveto#1#2{%
  \edef\pgfsyssoftpath@coord{{#1}{#2}}%
  \expandafter\pgfsyssoftpath@addtocurrentpath\expandafter{
  ↪  \expandafter\pgfsyssoftpath@movetotoken\pgfsyssoftpath@coord}%
  \ifpgfsyssoftpathmovetorelevant%
    \global\let\pgfsyssoftpath@lastmoveto\pgfsyssoftpath@coord%
  \fi%
}
```

此命令调用 \pgfsyssoftpath@addtocurrentpath 将

```
\pgfsyssoftpath@movetotoken{⟨dimend x⟩}{⟨dimend y⟩}
```

添加到 \pgfsyssoftpath@smallbuffer, 或 \pgfsyssoftpath@bigbuffer, 或 \pgfsyssoftpath@thepath 中。如果 \ifpgfsyssoftpathmovetorelevant 的真值是 true, 还会全局地保存“{⟨*dimend x*⟩}{⟨*dimend y*⟩}” 到 \pgfsyssoftpath@lastmoveto.

\pgfsyssoftpath@lineto{⟨*dimend x*⟩}{⟨*dimend y*⟩}

参数 ⟨*dimend x*⟩, ⟨*dimend y*⟩ 应当是 10pt 之类的尺寸，或者保存尺寸的宏，或者是 \the⟨尺寸寄存器⟩ 等形式。此命令的定义是：

```
\def\pgfsyssoftpath@lineto#1#2{%
  \edef\pgfsyssoftpath@coord{{#1}{#2}}%
  \expandafter\pgfsyssoftpath@addtocurrentpath\expandafter{
  ↪  \expandafter\pgfsyssoftpath@linetotoken\pgfsyssoftpath@coord}%
}
```

此命令调用 \pgfsyssoftpath@addtocurrentpath 将

```
\pgfsyssoftpath@linetotoken{⟨dimend x⟩}{⟨dimend y⟩}
```

添加到 \pgfsyssoftpath@smallbuffer，或 \pgfsyssoftpath@bigbuffer，或 \pgfsyssoftpath@thepath 中。

\pgfsyssoftpath@curveto{⟨dimend 1⟩}{⟨dimend 2⟩}{⟨dimend 3⟩}{⟨dimend 4⟩}{⟨dimend 5⟩}{⟨dimend 6⟩}

此命令的定义是：

```
\def\pgfsyssoftpath@curveto#1#2#3#4#5#6{%
  \edef\pgfsyssoftpath@temp{{%
      \noexpand\pgfsyssoftpath@curvetosupportatoken{#1}{#2}%
      \noexpand\pgfsyssoftpath@curvetosupportbtoken{#3}{#4}%
      \noexpand\pgfsyssoftpath@curvetotoken{#5}{#6}%
    }}%
  \expandafter\pgfsyssoftpath@addtocurrentpath\pgfsyssoftpath@temp%
}
```

此命令调用 \pgfsyssoftpath@addtocurrentpath 将

```
\pgfsyssoftpath@curvetosupportatoken{⟨dimend 1⟩}{⟨dimend 2⟩}
\pgfsyssoftpath@curvetosupportbtoken{⟨dimend 3⟩}{⟨dimend 4⟩}
\pgfsyssoftpath@curvetotoken{⟨dimend 5⟩}{⟨dimend 6⟩}
```

添加到 \pgfsyssoftpath@smallbuffer，或 \pgfsyssoftpath@bigbuffer，或 \pgfsyssoftpath@thepath 中。

\pgfsyssoftpath@rect{⟨dimend 1⟩}{⟨dimend 2⟩}{⟨dimend 3⟩}{⟨dimend 4⟩}

此命令的定义是：

```
\def\pgfsyssoftpath@rect#1#2#3#4{%
  \edef\pgfsyssoftpath@temp{{%
    \noexpand\pgfsyssoftpath@rectcornertoken{#1}{#2}%
    \noexpand\pgfsyssoftpath@rectsizetoken{#3}{#4}%
  }}%
  \expandafter\pgfsyssoftpath@addtocurrentpath\pgfsyssoftpath@temp%
}
```

此命令调用 \pgfsyssoftpath@addtocurrentpath 将

```
\pgfsyssoftpath@rectcornertoken{⟨dimend 1⟩}{⟨dimend 2⟩}
\pgfsyssoftpath@rectsizetoken{⟨dimend 3⟩}{⟨dimend 4⟩}
```

添加到 \pgfsyssoftpath@smallbuffer，或 \pgfsyssoftpath@bigbuffer，或 \pgfsyssoftpath@thepath 中。按定义，\pgfsyssoftpath@rectsizetoken 是 \pgfsyssoftpath@rectcornertoken 的一个参数。

\pgfsyssoftpath@closepath

此命令的定义是：

```
\def\pgfsyssoftpath@closepath{%
  \expandafter\pgfsyssoftpath@addtocurrentpath\expandafter{
  ↪  \expandafter\pgfsyssoftpath@closepathtoken\pgfsyssoftpath@lastmoveto}%
}
```

此命令调用 \pgfsyssoftpath@addtocurrentpath 将

```
\expandafter\pgfsyssoftpath@closepathtoken\pgfsyssoftpath@lastmoveto
```

添加到 \pgfsyssoftpath@smallbuffer，或 \pgfsyssoftpath@bigbuffer，或 \pgfsyssoftpath@thepath

中。

\pgfsyssoftpath@specialround{⟨*dimend 1*⟩}{⟨*dimend 2*⟩}

此命令的定义是：

```
\def\pgfsyssoftpath@specialround#1#2{%
  \edef\pgfsyssoftpath@temp{{#1}{#2}}%
  \expandafter\pgfsyssoftpath@addtocurrentpath\expandafter{
  ↪  \expandafter\pgfsyssoftpath@specialroundtoken\pgfsyssoftpath@temp}%
}
```

此命令调用 \pgfsyssoftpath@addtocurrentpath 将

```
\pgfsyssoftpath@specialroundtoken{⟨dimend 1⟩}{⟨dimend 2⟩}
```

添加到 \pgfsyssoftpath@smallbuffer, 或 \pgfsyssoftpath@bigbuffer, 或 \pgfsyssoftpath@thepath 中。

\pgfsyssoftpath@specialroundtoken{⟨*dimend 1*⟩}{⟨*dimend 2*⟩}

文件《pgfsyssoftpath.code.tex》中有：

```
\let\pgfsyssoftpath@specialroundtoken=\pgfutil@gobbletwo
```

也就是直接把参数 ⟨*dimend 1*⟩, ⟨*dimend 2*⟩ 吃掉。

关于圆角的处理，参考 \pgfprocessround→ P. 230, \pgfsetcornersarced→ P. 291.

\pgfsyssoftpath@movetotoken 等会调用创建硬路径的命令：

```
\def\pgfsyssoftpath@movetotoken#1#2{\pgfsys@moveto{#1}{#2}}
\def\pgfsyssoftpath@linetotoken#1#2{\pgfsys@lineto{#1}{#2}}
\def\pgfsyssoftpath@rectcornertoken#1#2#3#4#5{\pgfsys@rect{#1}{#2}{#4}{#5}}
↪  % #3 = \pgfsyssoftpath@rectsizetoken
\def\pgfsyssoftpath@curvetosupportatoken#1#2#3#4#5#6#7#8{\pgfsys@curveto{#1}{#2}{#4
↪  }{#5}{#7}{#8}}
\def\pgfsyssoftpath@closepathtoken#1#2{\pgfsys@closepath}
\let\pgfsyssoftpath@specialroundtoken=\pgfutil@gobbletwo
\def\pgfsyssoftpath@curvetosupportbtoken#1#2{curvetotokenb}
↪  % make sure this \ifx-equal only to itself
\def\pgfsyssoftpath@curvetotoken#1#2{curvetotoken}
↪  % make sure this \ifx-equal only to itself
```

10.5 文件《pgfcorepathprocessing.code.tex》

本文件的命令针对软路径做处理。

10.5.1 拆分软路径的命令

一个路径可能包含数个构造路径的命令，其中的 moveto 命令将路径划分为数个 subpath, 即“子路径”。例如

```
\draw (0,0)--(1,0) (0,0)--(0,1);
```

其中有两个 subpath.

\pgfprocesssplitpath⟨*soft path macro*⟩

参数 ⟨*soft path macro*⟩ 是保存软路径的宏，此命令的定义是：

```
\def\pgfprocesssplitpath#1{%
  \let\pgfprocessresultpathprefix\pgfutil@empty%
  \let\pgfprocessresultpathsuffix\pgfutil@empty%
  \let\pgf@next\pgf@process@split%
  \expandafter\pgf@process@split#1\pgfsyssoftpath@movetotoken{}{}\pgf@stop%
}
```

本命令将 ⟨*soft path macro*⟩ 的最后一个 subpath 保存到 `\pgfprocessresultpathsuffix`; 将 ⟨*soft path macro*⟩ 的最后一个 subpath 之前的部分保存到 `\pgfprocessresultpathprefix`.

macro:->\pgfsyssoftpath@movetotoken {0.0pt}{0.0pt}\pgfsyssoftpath@linetotoken {28.45274pt}{0.0pt}\pgfsyssoftpath@movetotoken {0.0pt}{0.0pt}\pgfsyssoftpath@linetotoken {0.0pt}{28.45274pt}
macro:->\pgfsyssoftpath@movetotoken {0.0pt}{0.0pt}\pgfsyssoftpath@linetotoken {28.45274pt}{0.0pt}
macro:->\pgfsyssoftpath@movetotoken {0.0pt}{0.0pt}\pgfsyssoftpath@linetotoken {0.0pt}{28.45274pt}

```
\tikz \draw [save path=\aaaa] (0,0)--(1,0) (0,0)--(0,1);\par
\pgfprocesssplitpath\aaaa
\meaning\aaaa\par
\meaning\pgfprocessresultpathprefix\par
\meaning\pgfprocessresultpathsuffix
```

`\pgfprocessresultpathsuffix`

如上。

`\pgfprocessresultpathprefix`

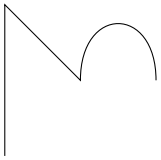
如上。

`\pgfprocesssplitsubpath`⟨*subpath macro*⟩

参数 ⟨*subpath macro*⟩ 是保存一个 subpath 的宏，此命令的定义是：

```
\def\pgfprocesssplitsubpath#1{%
  % First, we need to find the end:
  \let\pgf@tempa\pgfutil@emtpy%
  \let\pgf@tempb\pgfutil@emtpy%
  \let\pgf@tempc\pgfutil@emtpy%
  \let\pgf@tempd\pgfutil@emtpy%
  \let\pgfprocessresultsubpathprefix\pgfutil@empty%
  \let\pgfprocessresultsubpathsuffix\pgfutil@empty%
  \let\pgf@next\pgf@split@subpath%
  \expandafter\pgf@split@subpath#1\pgf@stop%
}
```

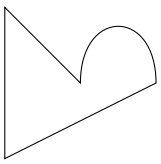
构建路径的操作指的是 moveto, lineto, curveto, closepath 这 4 类。如果 ⟨*subpath macro*⟩ 的最后一个操作是 curveto, 那么将与这个 curveto 相关的 4 个构建软路径的命令保存到 `\pgfprocessresultsubpathsuffix`; 将其余部分保存到 `\pgfprocessresultsubpathprefix`, 例如



```
macro:->\pgfsyssoftpath@movetotoken {-28.45274pt}{-28.45274pt}\pgfsyssoftpath@linetotoken
{-28.45274pt}{28.45274pt}
macro:->\pgfsyssoftpath@linetotoken {0.0pt}{0.0pt}\pgfsyssoftpath@curvetosupportatoken
{0.0pt}{28.45274pt}\pgfsyssoftpath@curvetosupportbtoken
{28.45274pt}{28.45274pt}\pgfsyssoftpath@curvetotoken {28.45274pt}{0.0pt}
```

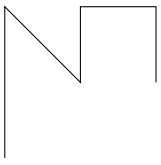
```
\tikz \draw [save path=\aaaa] (-1,-1)--(-1,1)--(0,0)..controls(0,1)and(1,1)..(1,0);\par
\pgfprocesssplitsubpath\aaaa
{\ttfamily
\meaning\pgfprocessresultsubpathprefix\par
\meaning\pgfprocessresultsubpathsuffix
}
```

如果 ⟨*subpath macro*⟩ 的最后一个操作不是 curveto, 那么将 ⟨*soft path macro*⟩ 的最后两个构建软路径的命令保存到 `\pgfprocessresultsubpathsuffix`; 将其余部分保存到 `\pgfprocessresultsubpathprefix`, 例如



```
macro:->\pgfsyssoftpath@movetotoken {-28.45274pt}{-28.45274pt}\pgfsyssoftpath@linetotoken
{-28.45274pt}{28.45274pt}\pgfsyssoftpath@linetotoken
{0.0pt}{0.0pt}\pgfsyssoftpath@curvetosupportatoken
{0.0pt}{28.45274pt}\pgfsyssoftpath@curvetosupportbtoken {28.45274pt}{28.45274pt}
macro:->\pgfsyssoftpath@curvetotoken {28.45274pt}{0.0pt}\pgfsyssoftpath@closepathtoken
{-28.45274pt}{-28.45274pt}
```

```
\tikz \draw [save path=\aaaa] (-1,-1)--(-1,1)--(0,0)..controls(0,1)and(1,1)..(1,0)--cycle;\par
\pgfprocesssplitsubpath\aaaa
{\ttfamily
\meaning\pgfprocessresultsubpathprefix\par
\meaning\pgfprocessresultsubpathsuffix
}
```



```
macro:->\pgfsyssoftpath@movetotoken {-28.45274pt}{-28.45274pt}\pgfsyssoftpath@linetotoken
{-28.45274pt}{28.45274pt}\pgfsyssoftpath@linetotoken
{0.0pt}{0.0pt}\pgfsyssoftpath@linetotoken {0.0pt}{28.45274pt}
macro:->\pgfsyssoftpath@linetotoken {28.45274pt}{28.45274pt}\pgfsyssoftpath@linetotoken
{28.45274pt}{0.0pt}
```

```
\tikz \draw [save path=\aaaa] (-1,-1)--(-1,1)--(0,0)--(0,1)--(1,1)--(1,0);\par
\pgfprocesssplitsubpath\aaaa
{\ttfamily
\meaning\pgfprocessresultsubpathprefix\par
\meaning\pgfprocessresultsubpathsuffix
```

```
}
```

当需要对 ⟨*subpath macro*⟩ 的末尾做修改时，可能要用到 \pgfprocessresultsubpathsuffix.

\pgfprocessresultsubpathprefix

如上。

\pgfprocessresultsubpathsuffix

如上。

\pgfprocesspathextractpoints⟨*soft path macro*⟩

参数 ⟨*soft path macro*⟩ 是保存软路径的宏，本命令针对的是 ⟨*soft path macro*⟩ 的最后一个 subpath. 本命令将最后一个 subpath 的：

- 第一个坐标保存到 \pgfpointfirstonpath
- 第二个坐标保存到 \pgfpointsecondonpath
- 倒数第一个坐标保存到 \pgfpointsecondlastonpath
- 倒数第二个坐标保存到 \pgfpointlastonpath

保存的是 \pgfqpoint{⟨*x*⟩}{⟨*y*⟩} 这种形式。如果最后一个 subpath 中只有一个坐标点，则这 4 个宏保存同一个点；如果 ⟨*soft path macro*⟩ 等于 \pgfutil@empty(空的)，则这 4 个宏保存 \pgfpointorigin. 此命令的定义是：

```
\def\pgfprocesspathextractpoints#1{%
  \ifx#1\pgfutil@empty%
    \let\pgfpointfirstonpath=\pgfpointorigin%
    \let\pgfpointsecondonpath=\pgfpointorigin%
    \let\pgfpointsecondlastonpath=\pgfpointorigin%
    \let\pgfpointlastonpath=\pgfpointorigin%
  \else%
    \expandafter\pgf@extractprocessorfirst#1\pgf@stop%
  \fi%
}
```

\pgfpointfirstonpath

如上。

\pgfpointsecondonpath

如上。

\pgfpointsecondlastonpath

如上。

\pgfpointlastonpath

如上。

\pgfprocesscheckclosed⟨*soft path*⟩⟨*code or macro*⟩

参数 ⟨*soft path*⟩ 是软路径或保存软路径的宏；⟨*code or macro*⟩ 是一些代码，或执行某些代码的命令。此命令的定义是：

```
\def\pgfprocesscheckclosed#1#2{%
  {%
    \global\let\pgf@proc@todo=\relax%
    \let\pgfsyssoftpath@movetotoken=\pgfutil@gobbletwo%
    \let\pgfsyssoftpath@linetotoken=\pgfutil@gobbletwo%
    \let\pgfsyssoftpath@curvetosupportatoken=\pgfutil@gobbletwo%
    \let\pgfsyssoftpath@curvetosupportbtoken=\pgfutil@gobbletwo%
    \let\pgfsyssoftpath@curvetotoken=\pgfutil@gobbletwo%
```

```
    \def\pgfsyssoftpath@rectcornertoken{\gdef\pgf@proc@todo{#2}\pgfutil@gobbletwo}
    ↪ %
    \def\pgfsyssoftpath@rectsizetoken{\gdef\pgf@proc@todo{#2}\pgfutil@gobbletwo}%
    \def\pgfsyssoftpath@closepathtoken{\gdef\pgf@proc@todo{#2}\pgfutil@gobbletwo}%
    #1%
  }%
  \pgf@proc@todo%
}
```

可见本命令能检查 ⟨*soft path*⟩ 中是否出现了

- \pgfsyssoftpath@rectcornertoken 或者
- \pgfsyssoftpath@rectsizetoken 或者
- \pgfsyssoftpath@closepathtoken

如果出现了，那么就执行 ⟨*code or macro*⟩.

10.5.2 处理圆角

构建路径的基本命令，例如 \pgfpathlineto:

```
\def\pgfpathlineto#1{%
  \pgfpointtransformed{#1}%
  \pgf@roundcornerifneeded%
  \pgf@nlt@lineto{\pgf@x}{\pgf@y}%
  \global\pgf@path@lastx=\pgf@x%
  \global\pgf@path@lasty=\pgf@y%
}
```

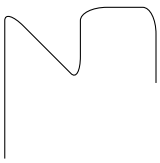
会在路径的“转角”的前面执行 \pgf@roundcornerifneeded, 如果 \ifpgf@arccorners 的真值是 true, 这会把命令

```
\expandafter\pgfsyssoftpath@specialround\pgf@corner@arc
```

放到“转角”的前面。命令 \pgfsyssoftpath@specialround → P. 226 会在当前的软路径中添加记号

```
\pgfsyssoftpath@specialroundtoken{⟨x⟩}{⟨y⟩}
```

例如：



```
macro:->\pgfsyssoftpath@movetotoken {-28.45274pt}{-28.45274pt}\pgfsyssoftpath@specialroundtoken
{5.0pt}{10.0pt}\pgfsyssoftpath@linetotoken {-28.45274pt}{28.45274pt}\pgfsyssoftpath@specialroundto
{5.0pt}{10.0pt}\pgfsyssoftpath@linetotoken {0.0pt}{0.0pt}\pgfsyssoftpath@specialroundtoken
{5.0pt}{10.0pt}\pgfsyssoftpath@linetotoken {0.0pt}{28.45274pt}\pgfsyssoftpath@specialroundtoken
{5.0pt}{10.0pt}\pgfsyssoftpath@linetotoken {28.45274pt}{28.45274pt}\pgfsyssoftpath@specialroundtok
{5.0pt}{10.0pt}\pgfsyssoftpath@linetotoken {28.45274pt}{0.0pt}
```

```
\begin{tikzpicture}
    \draw [save path=\aaaa]
        \pgfextra{\pgfsetcornersarced{\pgfpoint{5pt}{10pt}}}
        (-1,-1)--(-1,1)--(0,0)--(0,1)--(1,1)--(1,0);
\end{tikzpicture}\par
{\ttfamily \meaning\aaaa}
```

\pgfprocessround⟨*macro 1*⟩⟨*macro 2*⟩

⟨*macro 1*⟩ 是保存软路径的宏。本命令检查在 ⟨*macro 1*⟩ 中是否包含记号 `\pgfsyssoftpath@specialroundtoken`，如果不包含，就直接令 ⟨*macro 2*⟩ 等于 ⟨*macro 1*⟩; 如果包含，就执行一个复杂的循环处理，将具有“尖角”的软路径变成具有“圆角”的软路径，并保存到 ⟨*macro 2*⟩ 中。

处理圆角的流程大体上是：

1. 命令 `\pgfsetcornersarced`→P. 291 会设置 `\ifpgf@arccorners` 的真值以及圆角的尺寸；
2. 如果 `\ifpgf@arccorners` 的真值是 true, 那么构建路径的基本命令 (例如 `\pgfpathlineto`) 会调用 `\pgf@roundcornerifneeded`→P. 292 向软路径中添加记号 `\pgfsyssoftpath@specialroundtoken`;
3. 之后命令 `\pgfusepath`→P. 295 调用 `\pgfprocessround` 处理软路径中的尖角。

由于 TEX 组能限制 `\ifpgf@arccorners` 的真值，所以在一个路径中，可以用组来规定这个路径的哪一部分使用圆角，哪一部分使用尖角，也就是说，尖角和圆角可以共存于一个路径中。

10.5.3 改变首尾坐标

`\pgfprocesspathreplacestartandend`{⟨*soft path macro*⟩}{⟨*new start point*⟩}{⟨*new end point*⟩}

参数 ⟨*soft path macro*⟩ 应当是保存软路径的宏。本命令使用 ⟨*new start point*⟩ 替换 ⟨*soft path macro*⟩ 的第一个坐标，使用 ⟨*new end point*⟩ 替换 ⟨*soft path macro*⟩ 的最后一个坐标，得到新的软路径，仍然保存到 ⟨*soft path macro*⟩.

```
macro:->\pgfsyssoftpath@movetotoken {-56.9055pt}{-56.9055pt}\pgfsyssoftpath@linetotoken
{0.0pt}{28.45274pt}\pgfsyssoftpath@linetotoken
{28.45274pt}{28.45274pt}\pgfsyssoftpath@linetotoken {56.9055pt}{56.9055pt}
```

```
\begin{tikzpicture}
    \draw [save path=\aaaa]
        (0,0)--(0,1)--(1,1)--(1,0);
\end{tikzpicture}\par
\pgfprocesspathreplacestartandend\aaaa{\pgfpoint{-2cm}{-2cm}}{\pgfpoint{2cm}{2cm}}
{\ttfamily \meaning\aaaa}
```

第三部分

基本层

第十一章　基本层简介

基本层建立在系统层之上，主要包括与文件《pgfcore.code.tex》相关的文件以及模块。基本层的命令名称有以下规律：

1. 命令、环境都以 pgf 开头。
2. 指定坐标点的命令以 \pgfpoint 开头。
3. 开启或者延伸一个路径的命令以 \pgfpath 开头。
4. 设置或者修改图形参数的命令以 \pgfset 开头。
5. 针对刚刚创建的对象（如一个路径）进行操作的命令（使用路径的命令）以 \pgfuse 开头。
6. 坐标变换命令以 \pgftransform 开头。
7. 与箭头有关的命令以 \pgfarrows 开头。
8. “快速”延展路径或“快速”画出路径的命令以 \pgfpathq 或 \pgfusepathq 开头。
9. 与矩阵有关的命令以 \pgfmatrix 开头。

```
\usepackage{pgfcore}  % LaTeX
\input pgfcore.tex  % plain TeX
\usemodule[pgfcore]  % ConTeXt
```

这个命令会载入 PGF 的基本层的内核，但不载入任何模块，也不载入任何前端层（如 Ti*k*Z），但会把系统层一并载入。使用命令 \usepgfmodule 载入模块。

```
\usepackage{pgf}  % LaTeX
\input pgf.tex  % plain TeX
\usemodule[pgf]  % ConTeXt
```

这个命令会载入宏包 pgfcore，模块 shapes 和模块 plot. 在 LaTeX 中，这个命令有两个选项：

```
\usepackage[draft]{pgf}
```

带上这个选项后，所有图形都被方框代替，加快编译速度。

```
\usepackage[version=⟨version⟩]{pgf}
```

这个选项指示版本信息，如果 ⟨*version*⟩ 是 0.65，那么会载入很多“兼容命令”。如果 ⟨*version*⟩ 是 0.96，这些“兼容命令”不会被载入。如果不给出版本信息，那么所有版本的命令都会被载入。

\usepgfmodule{⟨*module names*⟩}

\usepgfmodule[⟨*module names*⟩]

载入内核后，可以用这个命令进一步载入模块。在 ⟨*module names*⟩ 中可以列出多个模块名称，之间用逗号分隔。包裹 ⟨*module names*⟩ 的可以是花括号或者方括号。多次载入同一模块不会有特别的作用。例如

```
\usepgfmodule{matrix,shapes}
```

这个命令实际上会载入文件 pgfmodule⟨*module*⟩.code.tex，例如文件 pgfmodulematrix.code.tex，你可以自己编辑一个这种文件，放在 TeX 能找到的地方。

\usepgflibrary{⟨*list of libraries*⟩}

\usepgflibrary[⟨*list of libraries*⟩]

⟨*list of libraries*⟩ 中列出程序库名称，之间用逗号分隔。例如

```
\usepgflibrary{arrows}
```

这个命令实际上会载入文件 pgflibrary⟨*library*⟩.code.tex, 例如 pgflibraryarrows.code.tex.

Library 与 module 的区别在于，库提供基本层的“附加”内容，而模块则提供新的功能。例如，库 decoration 提供多种装饰路径，而模块 decoration 则提供装饰功能来操作装饰路径。

一个图形 (graphic) 具有层级结构，即套嵌的 scope, 图形中主要使用 2 种 scope, 即“环境”和“TEX 组”，规则如下：

1. PGF 的最外层的 scope 是 {pgfpicture} 环境,即图形要在这个环境中画出。一般来说,在 {pgfpicture} 环境之外,不能把某个绘图参数设为“全局参数”,例如,在文档开头使用命令 \pgfsetlinewidth{1pt} 并不能将所有 {pgfpicture} 环境中的线宽设为 1pt，你只能在每个环境中分别设置线宽。

2. 可以在 {pgfpicture} 环境中使用一个或数个 {pgfscope} 环境来限制图形状态参数 (影响图形外观的某些参数)。

   注意 TEX 组也是一种 scope, 可以把 {pgfscope} 环境放入一个 TEX 组内，也可以把一个 TEX 组放入 {pgfscope} 环境内。“图形状态” (graphic state) 参数都接受 {pgfscope} 环境的限制，而不接受 TEX 组的限制。有的参数 (如箭头)、命令 (如坐标变换)，接受 TEX 组的限制。

3. 注意 {pgfscope} 环境会自动创建一个 TEX 组，因此接受 TEX 组的限制的参数、命令也接受 {pgfscope} 环境的限制。而且，不能将 {pgfscope} 环境与 TEX 组交错使用。

4. 一个路径不能被拆散到不同的 {pgfscope} 环境中。如果要对一个路径中的不同部分做不同的坐标变换，应当将各个部分分别放入一个 TEX 组内，分别做变换。

5. 命令 \pgftext 会创建一个 scope, 在这个 scope 内是通常的 TEX 状态，因此各种 TEX 的命令、模式、环境都可以用作命令 \pgftext 的参数，例如，可以把 {pgfpicture} 环境或者表格环境用作命令 \pgftext 的参数。

遵循以下原则会避免很多令人费解的问题：

- 绘图命令要放入 {pgfpicture} 环境中。
- 使用 {pgfscope} 环境来明确图形的层次。
- 在绘图环境中不要使用 TEX 分组，除非要限制坐标变换。

11.1 文件《pgfcorescopes.code.tex》

11.1.1 {pgfpicture} 环境

\begin{pgfpicture}
  ⟨*environment content*⟩
\end{pgfpicture}

这个环境没有可选项，但可以带有 4 个表达式参数：

\begin{pgfpicture}{⟨expression 1⟩}{⟨expression 2⟩}{⟨expression 3⟩}{⟨expression 4⟩}
⟨environment content⟩
\end{pgfpicture}

{pgfpicture} 后面的开花括号 { 会导致执行 \pgf@oldpicture，它会使得参数 ⟨expression 1⟩, ⟨expression 2⟩, ⟨expression 3⟩, ⟨expression 4⟩ 被 \pgfmathparse 解析，解析结果用于指定 {pgfpicture} 环境的上下左右界限坐标 (尺寸)。

```
\def\pgf@oldpicture#1#2#3#4{%
  \pgfmathsetlength\pgf@picminx{#1}%
  \pgfmathsetlength\pgf@picminy{#2}%
  \pgfmathsetlength\pgf@picmaxx{#3}%
  \pgfmathsetlength\pgf@picmaxy{#4}%
  \pgf@relevantforpicturesizefalse%
  \pgf@picture}
```

命令 \begin{pgfpicture} 导致 \pgfpicture. 命令 \end{pgfpicture} 导致 \endpgfpicture. {pgfpicture} 环境的结构大致是 (假设不执行 \pgf@oldpicture):

```
\begingroup%.....1
  \pgfpicturetrue%
  \global\advance\pgf@picture@serial@count by1\relax%
  \edef\pgfpictureid{pgfid\the\pgf@picture@serial@count}%
  \let\pgf@nodecallback=\pgfutil@gobble%
  \pgf@picmaxx=-16000pt\relax%
  \pgf@picminx=16000pt\relax%
  \pgf@picmaxy=-16000pt\relax%
  \pgf@picminy=16000pt\relax%
  \pgf@relevantforpicturesizetrue%
  \pgf@resetpathsizes%
  % \pgfutil@ifnextchar\bgroup\pgf@oldpicture\pgf@picture
  \setbox\pgfpic\hbox to0pt\bgroup%
    \begingroup%.....2
      \pgfsys@beginpicture%
      \pgfsys@beginscope%
        \begingroup%.....3
          \pgfsetcolor{.}%
          \pgfsetlinewidth{0.4pt}%
          \pgftransformreset%
          \pgfsyssoftpath@setcurrentpath\pgfutil@empty%
          \begingroup%.....4
            \let\pgf@setlengthorig=\setlength%
            \let\pgf@addtolengthorig=\addtolength%
            \let\pgf@selectfontorig=\selectfont%
            \let\setlength=\pgf@setlength%
            \let\addtolength=\pgf@addtolength%
            \let\selectfont=\pgf@selectfont%
            \nullfont\spaceskip0pt\xspaceskip0pt%
            \setbox\pgf@layerbox@main\hbox to0pt\bgroup%
              \begingroup%.....5
% 以上 \pgfpicture
                %%%%%%%%%%%%%%%
                ⟨ 环境内容⟩
                %%%%%%%%%%%%%%%
% 以下 \endpgfpicture
                \ifpgfrememberpicturepositiononpage%
```

```
                \hbox to0pt{\pgfsys@markposition{\pgfpictureid}}%
              \fi%
              % ok, now let's position the box
              \ifdim\pgf@picmaxx=-16000pt\relax%
                % empty picture. make size 0.
                \global\pgf@picmaxx=0pt\relax%
                \global\pgf@picminx=0pt\relax%
                \global\pgf@picmaxy=0pt\relax%
                \global\pgf@picminy=0pt\relax%
              \fi%
              % Shift baseline outside:
              \pgf@relevantforpicturesizefalse%
              \pgf@process{\pgf@baseline}%
              \xdef\pgf@shift@baseline{\the\pgf@y}%
              %
              \pgf@process{\pgf@trimleft}%
              \global\advance\pgf@x by-\pgf@picminx
              % prepare \hskip\pgf@trimleft@final.
              % note that \pgf@trimleft@final is also queried
              % by the pgf image externalization.
              \xdef\pgf@trimleft@final{-\the\pgf@x}%
              %
              \pgf@process{\pgf@trimright}%
              \global\advance\pgf@x by-\pgf@picmaxx
              % prepare \hskip\pgf@trimright@final.
              % note that \pgf@trimright@final is also queried
              % by the pgf image externalization.
              \xdef\pgf@trimright@final{\the\pgf@x}%
  %
  \pgf@remember@layerlist@globally
            \endgroup%.........5
            \hss%
          \egroup%
  \pgf@restore@layerlist@from@global
           \pgf@insertlayers%
        \endgroup%.........4
        \pgfsys@discardpath%
      \endgroup%.........3
     \pgfsys@endscope%
     \pgfsys@endpicture%
   \endgroup%.........2
   \hss
  \egroup%
  \pgfsys@typesetpicturebox\pgfpic%
\endgroup%.........1
```

这个环境使用 5 个组。注释：

4. `\pgfpictureid` 保存图形的 id.

5. `\pgf@nodecallback` 以 node 名称为参数。

6-9. 设置尺寸寄存器 `\pgf@picmaxx`, `\pgf@picminx`, `\pgf@picmaxy`, `\pgf@picminy` 的初始值，

```
\pgf@picmaxx=-16000pt\relax%
\pgf@picminx=16000pt\relax%
\pgf@picmaxy=-16000pt\relax%
\pgf@picminy=16000pt\relax%
```

这 4 个寄存器用于记录图形的上下左右边界坐标。

10. 条件命令 `\ifpgf@relevantforpicturesize` 见文件《pgfcorepoints.code.tex》，其字面意思是：是否将之后的某个 (某些) 尺寸、坐标点与 picture size 相关联，即是否计入当前 picture 的边界盒子内。例如在文件《pgfcorepathconstruct.code.tex》中 `\pgf@protocolsizes` 的定义是：

```
\def\pgf@protocolsizes#1#2{%
  \ifpgf@relevantforpicturesize%
    \ifdim#1<\pgf@picminx\global\pgf@picminx#1\fi%
    \ifdim#1>\pgf@picmaxx\global\pgf@picmaxx#1\fi%
    \ifdim#2<\pgf@picminy\global\pgf@picminy#2\fi%
    \ifdim#2>\pgf@picmaxy\global\pgf@picmaxy#2\fi%
  \fi%
  \ifpgf@size@hooked%
    \let\pgf@size@hook@x#1\let\pgf@size@hook@y#2\pgf@path@size@hook%
  \fi%
  \ifdim#1<\pgf@pathminx\global\pgf@pathminx#1\fi%
  \ifdim#1>\pgf@pathmaxx\global\pgf@pathmaxx#1\fi%
  \ifdim#2<\pgf@pathminy\global\pgf@pathminy#2\fi%
  \ifdim#2>\pgf@pathmaxy\global\pgf@pathmaxy#2\fi%
}
\newif\ifpgf@size@hooked
\let\pgf@path@size@hook=\pgfutil@empty%
```

`\pgf@protocolsizes` 的两个参数是路径当前点的 x, y 坐标尺寸，这个命令在条件 true 时，将当前点纳入图形的边界盒子内；另外，将当前点纳入当前路径的边界盒子内。

11. 命令 `\pgf@resetpathsizes` 见文件《pgfcorepathconstruct.code.tex》，它全局地重置尺寸寄存器 `\pgf@pathmaxx`, `\pgf@pathminx`, `\pgf@pathmaxy`, `\pgf@pathminy` 的值，这 4 个寄存器用于记录路径的边界坐标。

```
\def\pgf@resetpathsizes{%
  \global\pgf@pathmaxx=-16000pt\relax%
  \global\pgf@pathminx=16000pt\relax%
  \global\pgf@pathmaxy=-16000pt\relax%
  \global\pgf@pathminy=16000pt\relax%
}
```

13. 设置一个名称为 `\pgfpic` 的 0 宽度盒子，这个盒子将在 `\endpgfpicture` 那里用 `\egroup` 结束。

14. 盒子内容放入一个组中。

15. 在不同的驱动文件中 `\gfsys@beginpicture` 有不同定义

16. 使用系统层的 scope: `\pgfsys@beginscope`

17. 套嵌一个组 `\begingroup`, 限制之后的内容。

18. 命令 `\pgfsetcolor{.}` 设置颜色“.”，这个点号代表当前色 (似乎来自 xcolor 宏包)。

19. 命令 `\pgfsetlinewidth{0.4pt}` 设置线宽

20. 命令 `\pgftransformreset` 将变换矩阵设为单位矩阵。

21. 命令 `\pgfsyssoftpath@setcurrentpath\pgfutil@empty` 将当前的软路径设为空的。

22. 套嵌一个组 `\begingroup`, 限制之后的内容。

28. 关于字体有命令：

```
\def\pgf@selectfont{\pgf@selectfontorig\nullfont}
```

29. 命令 `\nullfont\spaceskip0pt\xspaceskip0pt` 清空字体命令和不必要的间距。

30. 再设置一个名称为 `\pgf@layerbox@main` 的 0 宽度盒子，盒子内容放入一个组中。这个盒子将在 `\endpgfpicture` 那里用 `\egroup` 结束。

34. 注意 ⟨环境内容⟩ ⊂ 盒子 `\pgf@layerbox@main` ⊂ 盒子 `\pgfpic` 中。

37. 条件命令 \ifpgfrememberpicturepositiononpage 见文件《pgfcorescopes.code.tex》，字面意思是：是否要记住图形在页面上的位置。

49. 不再计算图形的边界盒子。

51. 宏 \pgf@shift@baseline 全局地保存图形的基线坐标。

58. 宏 \pgf@trimleft@final 全局地保存图形的左侧边界坐标。

65. 宏 \pgf@trimright@final 全局地保存图形的右侧边界坐标。

67. 命令 \pgf@remember@layerlist@globally 的定义是：

```
\def\pgf@remember@layerlist@globally{%
  \global\let\pgf@layerlist@=\pgf@layerlist
}%
```

参考文件《pgfcorelayers.code.tex》的命令 \pgf@layerlist, \pgfsetlayers → P. 358.

69. 用 \hss 生成一段水平弹性空白。

70. 用 \egroup 结束盒子 \pgf@layerbox@main.

71. 命令 \pgf@restore@layerlist@from@global 的定义是：

```
\def\pgf@restore@layerlist@from@global{%
  \let\pgf@layerlist=\pgf@layerlist@
}%
```

72. 命令 \pgf@insertlayers 的定义是：

```
\def\pgf@insertlayers{%
  \box\pgf@layerbox@main%
}
```

此命令将盒子 \pgf@layerbox@main 中的内容取出，放在当前位置，并清空这个盒子。注意取出的内容还包含在盒子 \pgfpic 中。

文件《pgfcorelayers.code.tex》会重定义这个命令，参考 \pgf@insertlayers → P. 362.

74. 执行 \pgfsys@discardpath 丢弃当前的路径。

79. 执行 \hss 生成一段水平弹性空白。

81. \pgfsys@typesetpicturebox 计算盒子 \pgfpic 的基线和尺寸，按计算的结果将盒子 \pgfpic 插入到文档中。

11.1.2 {pgfscope} 环境

\begin{pgfscope}
  ⟨*environment content*⟩
\end{pgfscope}

该环境内的“图形状态 (graphic state)”参数只在该环境内起作用。“图形状态”参数包括：

- 线宽 line width.
- 线条颜色，填充色。
- 实线或虚线等线条样式 dash pattern.
- 线结合、线冠 line join and cap.
- 线条转角处的尖锐程度 miter limit.
- 画布变换矩阵。
- 剪切路径。

其它类型的参数也会影响图形的外观，但是不属于“图形状态”参数。例如，箭头命令不属于图形状态参数，箭头命令的有效范围受到 TEX 组的限制。{pgfscope} 环境创建一个 TEX 组。

在开启 {pgfscope} 环境时，当前路径必须是空的，也就是说，不能在构建路径的过程中开启 {pgfscope} 环境。

这个环境的定义是：

```
\def\pgfscope{%
  \pgfsyssoftpath@setcurrentpath\pgfutil@empty%
  \pgfsys@beginscope%
    \pgf@resetpathsizes%
    \edef\pgfscope@linewidth{\the\pgflinewidth}%
    \let\pgfscope@stroke@color=\pgf@strokecolor@global%
    \let\pgfscope@fill@color=\pgf@fillcolor@global%
    \begingroup}
\def\endpgfscope{%
    \endgroup%
    \global\pgflinewidth=\pgfscope@linewidth%
    \global\let\pgf@strokecolor@global=\pgfscope@stroke@color%
    \global\let\pgf@fillcolor@global=\pgfscope@fill@color%
  \pgfsys@endscope}
```

限制图形状态参数的是 \pgfsys@beginscope 和 \pgfsys@endscope.

```
\pgfscope
  ⟨environment contents⟩
\endpgfscope
```

这是 Plain TeX 的用法。

```
\startpgfscope
  ⟨environment contents⟩
\stoppgfscope
```

这是 ConTeXt 的用法。

11.1.3 选定 \nullfont 字体

\pgf@selectfont

此命令选定 \nullfont 字体，即取消字体。

```
\def\pgf@selectfont{\pgf@selectfontorig\nullfont}
```

11.1.4 设置环境的基线、边界

\pgfsetbaseline{⟨*expression*⟩ 或 default}

本命令将其参数保存为一个 PGF 点的 y 坐标，这个 y 坐标将用作环境的基线。注意本命令并不展开其参数，只是将参数保存起来，所以本命令的参数可以含有未定义的内容。在 \endpgfpicture 那里会展开这个参数 (一个 PGF 点)，点的 y 坐标值用作环境的基线。

本命令的定义是：

```
\def\pgfsetbaseline#1{%
  \def\pgf@temp{#1}%
  \ifx\pgf@temp\pgf@default@text
    \pgfsetbaseline{\pgf@picminy}%
  \else
    \pgfsetbaselinepointlater{\pgfpoint{0pt}{#1}}%
  \fi
}
\pgfsetbaseline{\pgf@picminy}
```

本命令的参数是：

- 单词 default, 这导致

```
\pgfsetbaselinepointlater{\pgfpoint{0pt}{\pgf@picminy}}%
```

- 能被 \pgfmathsetlength 处理的 ⟨*expression*⟩, 这导致

```
\pgfsetbaselinepointlater{\pgfpoint{0pt}{⟨expression⟩}}%
```

\pgfsetbaselinepointlater{⟨*PGF point*⟩}

本命令不展开其参数，只是将其参数保存到宏 \pgf@baseline 中。在 \endpgfpicture 那里会用这个宏决定环境的基线。

```
\def\pgfsetbaselinepointlater#1{\def\pgf@baseline{#1}}
```

\pgfsetbaselinepointnow{⟨*PGF point*⟩}

本命令展开其参数 ⟨*PGF point*⟩, 将得到坐标尺寸保存到宏 \pgf@baseline 中，其中的 y 坐标值 (一个尺寸) 将用作环境的基线 (在 \endpgfpicture 那里)。由于本命令会展开其参数，所以，本命令之后，环境的基线坐标值就不会再改变。

```
\def\pgfsetbaselinepointnow#1{%
  \pgf@process{#1}%
  \edef\pgf@setter@baseline{\noexpand\pgfpoint{\the\pgf@x}{\the\pgf@y}}%
  \pgfsetbaselinepointlater{\pgf@setter@baseline}%
}
```

\pgfsettrimleftpointlater{⟨*PGF point*⟩}

本命令不展开其参数，只是将其参数保存到宏 \pgf@trimleft 中。在 \endpgfpicture 那里会用这个宏决定环境的左侧边界坐标值。

```
\def\pgfsettrimleftpointlater#1{\def\pgf@trimleft{#1}}
```

\pgfsettrimleftpointnow{⟨*PGF point*⟩}

本命令展开其参数 ⟨*PGF point*⟩, 将得到坐标尺寸保存到宏 \pgf@trimleft 中，其中的 x 坐标值 (一个尺寸) 将用作环境的左侧边界坐标值 (在 \endpgfpicture 那里)。由于本命令会展开其参数，所以，本命令之后，环境的左侧边界坐标值就不会再改变。

```
\def\pgfsettrimleftpointnow#1{%
  \pgf@process{#1}%
  \edef\pgf@setter@baseline{\noexpand\pgfpoint{\the\pgf@x}{\the\pgf@y}}%
  \pgfsettrimleftpointlater{\pgf@setter@baseline}%
}
```

\pgfsettrimleft{⟨*expression*⟩ 或 default}

本命令将其参数保存为一个 PGF 点的 x 坐标，这个 x 坐标将用作环境的左侧边界坐标值。注意本命令并不展开其参数，只是将参数保存起来，所以本命令的参数可以含有未定义的内容。在 \endpgfpicture 那里会展开这个参数 (一个 PGF 点)，点的 x 坐标值用作环境的左侧边界坐标值。

```
\def\pgfsettrimleft#1{%
  \def\pgf@temp{#1}%
  \ifx\pgf@temp\pgf@default@text
    \pgfsettrimleft{\pgf@picminx}
  \else
    \pgfsettrimleftpointlater{\pgfpoint{#1}{0pt}}%
  \fi
}
```

```
\pgfsettrimleft{\pgf@picminx}
```

本命令的参数是：

- 单词 default, 这导致

  ```
  \pgfsettrimleftpointlater{\pgfpoint{\pgf@picminx}{0pt}}%
  ```

- 能被 \pgfmathsetlength 处理的 ⟨*expression*⟩, 这导致

  ```
  \pgfsettrimleftpointlater{\pgfpoint{⟨expression⟩}{0pt}}%
  ```

\pgfsettrimrightpointlater{⟨*PGF point*⟩}

本命令不展开其参数，只是将其参数保存到宏 \pgf@trimright 中。在 \endpgfpicture 那里会用这个宏决定环境的右侧边界坐标值。

```
\def\pgfsettrimrightpointlater#1{\def\pgf@trimright{#1}}
```

\pgfsettrimrightpointnow{⟨*PGF point*⟩}

本命令展开其参数 ⟨*PGF point*⟩, 将得到坐标尺寸保存到宏 \pgf@trimright 中，其中的 x 坐标值 (一个尺寸) 将用作环境的右侧边界坐标值 (在 \endpgfpicture 那里)。由于本命令会展开其参数，所以，本命令之后，环境的右侧边界坐标值就不会再改变。

```
\def\pgfsettrimrightpointnow#1{%
  \pgf@process{#1}%
  \edef\pgf@setter@baseline{\noexpand\pgfpoint{\the\pgf@x}{\the\pgf@y}}%
  \pgfsettrimrightpointlater{\pgf@setter@baseline}%
}
```

\pgfsettrimright{⟨*expression*⟩ 或 default}

本命令将其参数保存为一个 PGF 点的 x 坐标，这个 x 坐标将用作环境的右侧边界坐标值。注意本命令并不展开其参数，只是将参数保存起来，所以本命令的参数可以含有未定义的内容。在 \endpgfpicture 那里会展开这个参数 (一个 PGF 点)，点的 x 坐标值用作环境的右侧边界坐标值。

```
\def\pgfsettrimright#1{%
  \def\pgf@temp{#1}%
  \ifx\pgf@temp\pgf@default@text
    \pgfsettrimright{\pgf@picmaxx}%
  \else
    \pgfsettrimrightpointlater{\pgfpoint{#1}{0pt}}%
  \fi
}
\pgfsettrimright{\pgf@picmaxx}
```

本命令的参数是：

- 单词 default, 这导致

  ```
  \pgfsettrimrightpointlater{\pgfpoint{\pgf@picmaxx}{0pt}}%
  ```

- 能被 \pgfmathsetlength 处理的 ⟨*expression*⟩, 这导致

  ```
  \pgfsettrimrightpointlater{\pgfpoint{⟨expression⟩}{0pt}}%
  ```

11.1.5 各种 interrupt 环境

```
\begin{pgfinterruptpath}
  ⟨environment content⟩
\end{pgfinterruptpath}
```

在构建路径的过程中插入这个环境，可以中断当前路径的构建过程，把当前路径的构建状态暂时封存起来，本环境结束时，会调出封存的路径状态，继续构建它。在本环境内你可以构建新的路径，并对此路径做某些操作。

本环境的内容被放入一个组中。本环境不会开启 {pgfscope} 环境，也不会调用任何 \pgfsys@ 命令 (这在路径中间很危险)。

利用本环境的一个例子是给路径添加箭头。在路径中，使用本环境；在本环境中进行箭头缓存 (caching)。

本环境的定义是：

```
\def\pgfinterruptpath
{%
  \begingroup%
  % save all sorts of things...
  \edef\pgf@interrupt@savex{\the\pgf@path@lastx}%
  \edef\pgf@interrupt@savey{\the\pgf@path@lasty}%
  \pgf@getpathsizes\pgf@interrupt@pathsizes%
  \pgfsyssoftpath@getcurrentpath\pgf@interrupt@path%
  \pgfsyssoftpath@setcurrentpath\pgfutil@empty%
  \let\pgf@interrupt@lastmoveto=\pgfsyssoftpath@lastmoveto%
  \begingroup%
}
\def\endpgfinterruptpath
{%
  \endgroup%
  \pgfsyssoftpath@setcurrentpath\pgf@interrupt@path%
  \pgf@setpathsizes\pgf@interrupt@pathsizes%
  \global\pgf@path@lastx\pgf@interrupt@savex%
  \global\pgf@path@lasty\pgf@interrupt@savey%
  \global\let\pgfsyssoftpath@lastmoveto\pgf@interrupt@lastmoveto%
  \endgroup%
}
```

```
\pgfinterruptpath
  ⟨environment contents⟩
\endpgfinterruptpath
```

这是 Plain TeX 的用法。

```
\startpgfinterruptpath
  ⟨environment contents⟩
\stoppgfinterruptpath
```

这是 ConTeXt 的用法。

\begin{pgfinterruptpicture}

⟨environment content⟩

\end{pgfinterruptpicture}

这个环境用在 {pgfpicture} 环境中，它会暂时中断当前环境。可以在本环境环境中插入一个新的 {pgfpicture} 环境。设计这个环境的目的是，在一个盒子的开头或结尾使用本环境，然后将该盒子作为命令 \pgfqbox 的参数，引入 {pgfpicture} 环境中。本环境不能直接放到 {pgfpicture} 环境中。

本环境的定义是：

```
\def\pgfinterruptpicture
{%
  \begingroup%
    \csname tikz@inside@picturefalse\endcsname%
    \pgfinterruptboundingbox%
```

```
      \pgftransformreset%
      \pgfinterruptpath%
        \ifx\pgf@selectfontorig\@undefined%
        \else%
          \let\setlength\pgf@setlengthorig%
          \let\addtolength\pgf@addtolengthorig%
          \let\selectfont\pgf@selectfontorig%
        \fi%
        \pgfutil@selectfont%
        \pgfpicturefalse%
        \let\pgf@positionnodelater@macro\relax%
        \pgf@savelayers%
}
\def\endpgfinterruptpicture
{%
        \pgf@restorelayers%
      \endpgfinterruptpath%
    \endpgfinterruptboundingbox%
  \endgroup%
}

\let\pgf@savelayers=\relax
\let\pgf@restorelayers=\relax
```

文件《pgfcorelayers.code.tex》会重定义 \pgf@savelayers, \pgf@restorelayers.

Sub-picture.

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{0cm}}
  \newbox\mybox  % 定义一个盒子
  \setbox\mybox=\hbox{  % 设置盒子为一个水平盒子,
    \begin{pgfinterruptpicture}  % 将这个打断环境放入盒子中
      Sub-\begin{pgfpicture}  % 插入一个新的绘图环境
        \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
        \pgfpathlineto{\pgfpoint{1cm}{1cm}}
        \pgfusepath{stroke}
      \end{pgfpicture}-picture.  % 结束插入新环境
    \end{pgfinterruptpicture}  % 结束打断环境
  }
  \pgfqbox{\mybox}  % 引入新定义的盒子
  \pgfpathlineto{\pgfpoint{0cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}\hskip3.9cm
```

```
\pgfinterruptpicture
  ⟨environment contents⟩
\endpgfinterruptpicture
```

这是 Plain TeX 的用法。

```
\startpgfinterruptpicture
  ⟨environment contents⟩
\stoppgfinterruptpicture
```

这是 ConTeXt 的用法。

\begin{pgfinterruptboundingbox}

⟨environment content⟩

\end{pgfinterruptboundingbox}

这个环境打断对于图形的边界盒子的计算，将边界盒子的计算状态暂时封存，然后创建一个针对本环

境的 ⟨*environment contents*⟩ 的边界盒子并计算它。待本环境结束后，再调出封存的边界盒子的计算状态，继续计算原来的（外层环境的）边界盒子。这样做的效果是，{pgfinterruptboundingbox} 环境所绘的图形会被外层环境的边界盒子忽略。

本环境的定义是：

```
\def\pgfinterruptboundingbox
{%
  \begingroup%
    \edef\pgf@interrupt@savemaxx{\the\pgf@picmaxx}%
    \edef\pgf@interrupt@saveminx{\the\pgf@picminx}%
    \edef\pgf@interrupt@savemaxy{\the\pgf@picmaxy}%
    \edef\pgf@interrupt@saveminy{\the\pgf@picminy}%
    \pgf@picmaxx=-16000pt\relax%
    \pgf@picminx=16000pt\relax%
    \pgf@picmaxy=-16000pt\relax%
    \pgf@picminy=16000pt\relax%
    \pgf@size@hookedfalse%
    \let\pgf@path@size@hook=\pgfutil@empty%
}
\def\endpgfinterruptboundingbox
{%
    \global\pgf@picmaxx=\pgf@interrupt@savemaxx%
    \global\pgf@picmaxy=\pgf@interrupt@savemaxy%
    \global\pgf@picminx=\pgf@interrupt@saveminx%
    \global\pgf@picminy=\pgf@interrupt@saveminy%
  \endgroup%
}
```

```
\pgfinterruptboundingbox
  ⟨environment contents⟩
\endpgfinterruptboundingbox
```

这是 Plain TeX 的用法。

```
\startpgfinterruptboundingbox
  ⟨environment contents⟩
\stoppgfinterruptboundingbox
```

这是 ConTeXt 的用法。

11.1.6 插入文字

\pgftext[⟨*options*⟩]{⟨*text*⟩}

这个命令创建一个水平盒子，在这个盒子内部是通常的 TEX 状态。本命令把 ⟨*text*⟩ 放入盒子中，⟨*text*⟩ 可以是 TEX 状态下的文字、环境、命令等。在 ⟨*text*⟩ 中可以使用抄录命令。本命令将这个盒子内容放入一个 graphic scope 中，并插入到当前位置。

坐标变换命令对此命令插入的盒子有影响。

在默认下，这个盒子的中心位于坐标系的原点位置，可以通过选项来改变盒子的位置。盒子本身有上（top）、下（bottom）、左（left）、右（right）等“部位”，这些“部位”就是盒子边界上的点。如果把盒子中的文字看作是一种注释，那么注释应该有指向的目标点，类似 node 的“锚定点”（但它不是 node）。

本命令的处理过程大致是：

```
1 \setbox\pgf@hbox=\hbox\bgroup% 定义水平盒子 \pgf@hbox
2   \pgfinterruptpicture%
```

```
    \bgroup%
%       \aftergroup\pgf@collectresetcolor%
      \let\next={⟨text⟩} % 开花括号 { 被吃掉，闭花括号 } 配合前面的 \bgroup
%       \pgf@collectresetcolor
  \endpgfinterruptpicture%
\egroup% 结束水平盒子定义
%\pgf@after@text
{% 开启一个 TeX 组
  ⟨ 处理本命令的选项，决定平移矩阵⟩

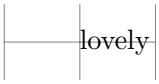
  ↪ ⟨ 用 \pgfapproximatenonlineartransformation 修改线性变换矩阵来近似非线性变换，并清除非线性变
  ⟨ 执行 \pgf@protocolsizes，用盒子更新图形、路径的边界盒子⟩
  ⟨ 执行 \pgfsys@hboxsynced，插入盒子内容⟩
}% 结束 TeX 组
```

可见，参数 {⟨*text*⟩} 中，包裹 ⟨*text*⟩ 的花括号应当看作是一个能限制定义有效范围的 TEX 组 (scope)。

/pgf/text/left (no value)

将盒子的 left 点放在锚定点上。

下面例子中，默认文字盒子的锚定点是原点，盒子的 left 点放在原点上：




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[left] {lovely}}
```

/pgf/text/right (no value)

将盒子的 right 点放在锚定点上。

/pgf/text/top (no value)

将盒子的 top 点放在锚定点上。

/pgf/text/bottom (no value)

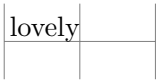
将盒子的 bottom 点放在锚定点上。

/pgf/text/base (no value)

将盒子中文字的基线的中点放在锚定点上。


```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[base] {lovely}}
```

以上选项可以配合使用，例如：

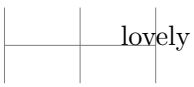



```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[bottom,right] {lovely}}
```


```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[base,right] {lovely}}
```

/pgf/text/at=⟨*point*⟩ (no default)

指定盒子的锚定点为 ⟨*point*⟩.



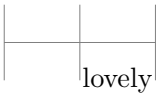

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[base,at={\pgfpoint{1cm}{0cm}}] {lovely}}
```

/pgf/text/x=⟨*dimension*⟩ (no default)

规定盒子的锚定点沿着 x 轴方向平移 ⟨*dimension*⟩, 可以是负值尺寸。

/pgf/text/y=⟨*dimension*⟩ (no default)

规定盒子的锚定点沿着 y 轴方向平移 ⟨*dimension*⟩, 可以是负值尺寸。

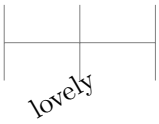



```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[at={\pgfpoint{1cm}{0cm}},x=-0.5cm,y=-0.5cm] {lovely}}
```

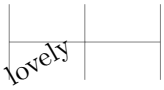
/pgf/text/rotate=⟨*degree*⟩ (no default)

将盒子围绕锚定点旋转，旋转角度由 ⟨*degree*⟩ 指定。

注意以上选项 `left`, `top`, `x=`, `y=` 等都是平移选项，选项 `rotate` 是旋转选项，如果它们的先后次序不同会导致不同的结果。比较下面的例子：




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[rotate=30,left,x=-1cm,y=-0.5cm,] {lovely}}
```




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[left,x=-1cm,y=-0.5cm,rotate=30,] {lovely}}
```

11.1.7 对象的 id

图形对象 (graphical objects) 可以有一个 identifier, 利用这个 identifier 可以（在稍后）引用这个图形对象。例如，可以利用这个 identifier 给图形对象创建一个超链接，或者将图形对象加入到动画 (animation) 中。

给图形对象添加 identifier 只需要两步：

1. 使用命令 `\pgfuseid{`⟨*id*⟩`}` 选择一个 identifier 名称，作为名称的 ⟨*id*⟩ 就是一串普通的符号。
2. 使用 `\pgfidscope` 或 `\pgftext` 等命令创建一个对象，这个对象自动具有 ⟨*id*⟩.

下面的系统层命令可以创建具有 id 的对象：

1. `\pgfsys@begin@idscope`, 创建一个 graphic scope.
2. `\pgfsys@viewboxmeet` 或 `\pgfsys@viewboxslice`, 创建一个 view box.
3. `\pgfsys@fill`, `\pgfsys@stroke` 以及其它“使用”路径的命令。
4. `\pgfsys@hbox` 或 `\pgfsys@hboxsynced`, 创建 text boxes.
5. `\pgfsys@animate` ... 等命令，创建动画。

以上系统层命令被以下基本层命令调用：

- `\pgfidscope`, 创建一个 id scope.
- `\pgfviewboxscope`, 创建一个 view box.
- `\pgfusepath`, 创建路径。

- \pgftext, \pgfnode, \pgfmultipartnode, 创建 text boxes, node.
- \pgfanimateattribute, 创建动画。

\begin{pgfidscope}
⟨*environment content*⟩
\end{pgfidscope}

本环境创建一个 graphic scope, 这个环境可以具有 id, 其 id 用命令 \pgfuseid 添加。

《pgfcorescopes.code.tex》中有：

```
\def\pgfidscope{\pgfsys@begin@idscope}
\def\endpgfidscope{\pgfsys@end@idscope}
```

参考 \pgfsys@begin@idscope→P. 214.

```
\pgfidscope
  ⟨environment contents⟩
\endpgfidscope
```

PlainTeX 中的环境。

```
\startpgfidscope
  ⟨environment contents⟩
\stoppgfidscope
```

ConTeXt 中的环境。

\pgfuseid{⟨*name*⟩}

⟨*name*⟩ 是一串符号，或者保存一串符号的宏，用作对象的 id. 在本命令之后的第一个 graphic object 会以 ⟨*name*⟩ 作为其 id (还要求这个 graphic object 与此命令处于同一个 TEX 组中)。在本命令之后的第二个、第三个等其它 graphic object 不会带有 id.

两个不同的 graphic object 可以具有相同的 id, 只需分别在它们前面使用命令 \pgfuseid 来指定相同的 id 名称。

本命令会将 ⟨*name*⟩ 与控制序列 \csname pgf@id@names@⟨*name*⟩\endcsname 对应起来。本命令先检查 ⟨*name*⟩ 这个 id 是否被使用过，也就是检查这个控制序列是否已定义 (不等于 \relax)，

- 如果未定义，即首次使用 ⟨*name*⟩ 这个 id, 则
  - 定义宏 \pgf@prev@id, 它等于 \pgfutil@empty.
  - 用 \pgfsys@new@id 声明宏 \pgf@next@id 为一个新的、当前可用的 id, 然后用 \pgfsys@use@id 处理这个宏。这个宏保存当前正在使用 (可以指派给下一个图形对象) 的 id. 注意在系统层中，id 的形式为 pgf⟨编号 N⟩, 其中并不包含这里给出的 ⟨*name*⟩.
  - 全局定义控制序列 \csname pgf@id@names@⟨*name*⟩\endcsname, 它保存“{pgf⟨编号 N⟩}{⟨“一个空组”⟩}”
- 如果已定义，即重复使用 ⟨*name*⟩ 这个 id, 则
  - 定义宏 \pgf@prev@id, 它保存之前被 \pgfsys@use@id 处理过的 id, 其形式为 pgf⟨编号 N⟩.
  - 用 \pgfsys@new@id 声明宏 \pgf@next@id 为一个新的、当前可用的 id, 然后用 \pgfsys@use@id 处理这个宏。这个宏保存当前正在使用 (可以指派给下一个图形对象) 的 id, 其形式为 pgf⟨编号 $N+1$⟩.
  - 全局定义控制序列 \csname pgf@id@names@⟨*name*⟩\endcsname, 它保存“{pgf⟨编号 $N+1$⟩}{⟨一个空组⟩}”.

另外，identifier 具有“类型”这一特征，即 identifier type, 当引用图形对象的 id 时，是综合 identifier 与 type 来引用的。Id 的 type 用下面的命令设置：

\pgfusetype{⟨*type*⟩}

本命令设置图形对象的类型。如果 ⟨*type*⟩ 以点号“.”开头，则表示它是当前类型的附加 (type 之下的 type)。

一个 graphic object 可以由多个部分组成，在它的任何一个部分中都可以使用命令 \pgfusetype 来为该部分规定一个 type. 在创建 graphic object 后，就可以用该对象的 id 名称 ⟨*name*⟩ 和 ⟨*type*⟩ 来引用 graphic object 的各个部分。

目前可用的 type 如下：

- 在命令 \pgfviewboxscope 中可以使用类型 .view, 它针对 view object.
- 在命令 \pgfmultipartnode 中，类型 .behind background 针对 node 的 behind background path; 类似地，也有类型 .before background, .behind foreground, .before foreground.
- 在 node 中，类型 .background 针对 background path; 类型 .foreground 针对 foreground path. 假如一个 node 的 id 是 mynode, 那么就可以用 mynode.background 来引用其 background path.
- 在 node 中，各个文字部分的名称同时也是类型，每个 node 都有一个默认的文字盒子，其名称是 text, 所以 .text 是针对该文字盒子的类型。例如 shapes.multipart 库提供的 circle split 有两个文字部分，其名称分别是 text 和 lower, 所以 .text 和 .lower 是 circle split 的两个 type.

在 Ti*k*Z 中：

- 当一个路径使用了 name 选项时，该路径有 .path 类型。
- 针对 the scope of the optional path picture 的类型 .path picture.
- 类型 .path fill, 针对被 (颜色、图样) 填充路径。
- 类型 .path shade, 针对用于制作颜色渐变效果的路径。

如果想临时修改当前的类型，可以使用下面两个命令：

\pgfpushtype

将当前的类型放到一个内部的、全局定义的栈中。就是 \pgfsys@push@type.

\pgfpoptype

将保存在一个内部的、全局定义的栈的顶层的类型调出。就是 \pgfsys@pop@type.

\pgfclearid

将 local scope 中的当前的 id 以及 type 都清除掉。就是 \pgfsys@clear@id.

\pgfidrefnextuse{⟨*macro*⟩}{⟨*name*⟩}

本命令的处理是：

- 如果 ⟨*name*⟩ 这个 id 之前使用过，那么控制序列 \csname pgf@id@names@⟨*name*⟩\endcsname 保存“{pgf⟨编号 N⟩}{⟨“一个空组”⟩}”
  1. 定义宏 \pgf@prev@id, 它保存 pgf⟨编号 N⟩.
  2. 用 \pgfsys@new@id 声明宏 \pgf@next@id 为一个新的、当前可用的 id, 其形式为 pgf⟨编号 X⟩.
  3. 全局定义控制序列 \csname pgf@id@names@⟨*name*⟩\endcsname, 它保存“{pgf⟨编号 N⟩}{pgf⟨编号 X⟩}”.
  4. 令 ⟨*macro*⟩ 等于 \pgf@next@id.
- 如果 ⟨*name*⟩ 这个 id 之前未被使用过，

1. 令宏 \pgf@prev@id 等于 \pgfutil@empty.
2. 用 \pgfsys@new@id 声明宏 \pgf@next@id 为一个新的、当前可用的 id, 其形式为 pgf⟨编号 X⟩.
3. 全局定义控制序列 \csname pgf@id@names@⟨*name*⟩\endcsname, 它保存“{一个空组}{pgf⟨编号 X⟩}”.
4. 令 ⟨*macro*⟩ 等于 \pgf@next@id.

本命令将宏 ⟨*macro*⟩ 与 ⟨*name*⟩ 联系起来，使之成为一个 id, 之后可以用命令 \pgfuseid{⟨*macro*⟩} 把这个 id 用作某个 graphic object 的 id, 而且这个 graphic object 可以与此命令位于不同的页面上。选项 /pgf/animation/whom 会调用此命令。

\pgfidrefprevuse{⟨*macro*⟩}{⟨*name*⟩}

本命令类似 \, 只是本命令最后令 ⟨*macro*⟩ 等于 \pgf@prev@id.

\pgfaliasid{⟨*alias*⟩}{⟨*name*⟩}

在当前的 TEX scope 中，为名称 ⟨*name*⟩ 创建一个别名 ⟨*alias*⟩, 把两个名称“捆绑”起来。但是如果再次使用命令 \pgfuseid{⟨*name*⟩}, 那么 ⟨*name*⟩ 与 ⟨*alias*⟩ 之间的“捆绑”关系就没有了。

\pgfgaliasid{⟨*alias*⟩}{⟨*name*⟩}

本命令的作用类似 \pgfaliasid, 不过本命令所建立的捆绑关系是全局的。

\pgfifidreferenced{⟨*name*⟩}{⟨*then code*⟩}{⟨*else code*⟩}

本命令检查 ⟨*name*⟩ 是否已经被声明的 id, 也就是检查控制序列 \csname pgf@id@names@⟨*name*⟩\endcsname 是否已定义 (不等于 \relax)。如果已定义，就执行 ⟨*then code*⟩, 否则执行 ⟨*else code*⟩.

定义这个控制序列的是 \pgfuseid, \pgfidrefnextuse, \pgfidrefprevuse, \pgfaliasid, \pgfgaliasid.

第十二章　点

文件《pgfcorepoints.code.tex》。

PGF 的点实际上都是 2 维点，点的坐标保存在尺寸寄存器 \pgf@x 和 \pgf@y 中。对点的运算基本上都是利用寄存器完成的。当说“一个 PGF 点”时，指的是类似 \pgfpoint 这样的命令，或者是为尺寸寄存器 \pgf@x 和 \pgf@y 赋值，或者能修改这两个寄存器值的代码。

PGF 的很多坐标命令会利用数学引擎做计算，然后将计算结果保存在尺寸寄存器中。

12.1　基本的点命令

\pgf@process{⟨*code*⟩}

这个命令的定义是：

```
\def\pgf@process#1{{#1\global\pgf@x=\pgf@x\global\pgf@y=\pgf@y}}
```

本命令的操作限制在一个组内。执行 ⟨*code*⟩ 后，就全局性地给尺寸寄存器 \pgf@x 和 \pgf@y 赋值，通常这两个寄存器用作点的 x 坐标和 y 坐标。

\pgfextract@process{⟨*macro*⟩}{⟨*code*⟩}

这个命令的定义是：

```
\def\pgfextract@process#1#2{%
    \pgf@process{#2}%
    \edef#1{\noexpand\global\pgf@x=\the\pgf@x
    ↪  \noexpand\relax\noexpand\global\pgf@y=\the\pgf@y\noexpand\relax}%
}
```

本命令先执行 ⟨*code*⟩, 然后将为 \pgf@x 和 \pgf@y 全局赋值的命令保存到宏 ⟨*macro*⟩ 中。

\pgfpoint{⟨*x coordinate*⟩}{⟨*y coordinate*⟩}

这个命令规定一个点，⟨*x coordinate*⟩ 和 ⟨*y coordinate*⟩ 用于计算点的坐标。

此命令的定义是：

```
\def\pgfpoint#1#2{%
  \pgfmathsetlength\pgf@x{#1}%
  \pgfmathsetlength\pgf@y{#2}\ignorespaces}
```

可见这个定义使用 \pgfmathsetlength→P. 112, 它的参数若不以“+”开头，则会被 \pgfmathparse 解析；解析时，如果 ⟨*x coordinate*⟩ 或 ⟨*y coordinate*⟩ 中的数值不带长度单位，那么就默认其长度单位是 pt. 给解析的结果带上长度单位 mu 或 pt 后再赋予 \pgf@x 和 \pgf@y.

注意 \pgfmathsetlength 做这两个赋值时并没有使用 \global.

下面例子表明，\pgfpoint 定义的点坐标是局部的：

10.0pt, 20.0pt

```
\pgfpoint{10}{20}
{\pgfpoint{30}{40}}
\makeatletter
\the\pgf@x, \the\pgf@y
\makeatother
```

注意下面的命令：

```
\edef\aaaa{\pgfpoint{1cm}{0cm}}% 或者
\edef\bbbb{\pgfmathparse{1cm}}
```

都会导致错误，原因可能与 \pgfmathparse 的定义有关，可以改为：

```
\edef\aaaa{\noexpand\pgfpoint{1cm}{0cm}}% 或者
\edef\bbbb{\noexpand\pgfmathparse{1cm}}
```

或者：

```
\pgfpoint{1cm}{0cm}
\edef\aaaa{\pgf@process{}}
```

\pgfqpoint{⟨*x dimen expression*⟩}{⟨*y dimen expression*⟩}

此命令的定义是：

```
\def\pgfqpoint#1#2{\global\pgf@x=#1\relax\global\pgf@y=#2\relax}
```

此命令直接把 ⟨*x dimen expression*⟩ 赋予 \pgf@x; 直接把 ⟨*y dimen expression*⟩ 赋予 \pgf@y. 此命令的赋值是全局的，所以 ⟨*x dimen expression*⟩ 和 ⟨*y dimen expression*⟩ 都应当是尺寸表达式。例如：

```
\pgfqpoint{0.2\linewidth\advance\pgf@x by 1cm}{0pt}
导致
\pgf@x 的值是 0.2\linewidth + 1cm
\pgf@y 的值是 0pt
```

\pgfpointorigin

原点，其定义是

```
\def\pgfpointorigin{\global\pgf@x=0pt \global\pgf@y=\pgf@x\ignorespaces}
```

与 \pgfpoint 不同，本命令全局地为 \pgf@x, \pgf@y 赋值。

\pgfpointpolar{⟨*degree*⟩}{⟨*radius*⟩ and ⟨*y-radius*⟩}

这个命令产生一个极坐标点，其中 ⟨*degree*⟩ 是一个数学表达式，代表角度制下的数值。其中的 and ⟨*y-radius*⟩ 这一部分是可选的，如果给出这一可选部分，则所确定的点位于中心在原点，横半轴长度为 ⟨*radius*⟩, 纵半轴长度为 ⟨*y-radius*⟩ 的椭圆上，点的角度是 ⟨*degree*⟩. 如果 ⟨*radius*⟩, ⟨*y-radius*⟩ 不带长度单位，那么默认其长度单位是 pt.

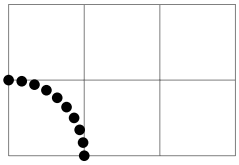
参数 ⟨*degree*⟩, ⟨*radius*⟩, ⟨*y-radius*⟩ 都会被命令 \pgfmathsetlength 处理。处理 ⟨*degree*⟩ 的结果用于计算角度的正弦值 s 和余弦值 c. 将处理 ⟨*radius*⟩ 的结果 (一个尺寸) 记为 f_x, 处理 ⟨*y-radius*⟩ 的结果 (一个尺寸) 记为 f_y，那么最后全局地做赋值 \pgf@x= cf_x, \pgf@y= sf_y, 从而确定一个点。

本命令的定义是：

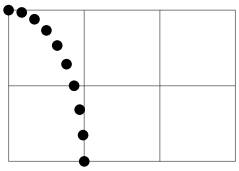
```
\def\pgfpointpolar#1#2{%
  \pgfutil@in@{and }{#2}%
  \ifpgfutil@in@%
    \pgf@polar@#2\@@%
  \else%
    \pgf@polar@#2 and #2\@@%
  \fi%
```

```
  \pgfmathparse{#1}%
  \let\pgfpoint@angle=\pgfmathresult%
  \pgfmathcos@{\pgfpoint@angle}%
  \global\pgf@x=\pgfmathresult\pgf@x%
  \pgfmathsin@{\pgfpoint@angle}%
  \global\pgf@y=\pgfmathresult\pgf@y%
}

\def\pgf@polar@#1and #2\@@{%
  \pgfmathsetlength{\pgf@y}{#2}%
  \pgfmathsetlength{\pgf@x}{#1}%
}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolar{\angle}{1cm}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolar{\angle}{1cm and 2cm}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfqpointpolar{⟨*degree*⟩}{⟨*radius*⟩}

这是 quick 版的极坐标命令。

参数 ⟨*degree*⟩ 是一个数学表达式，⟨*radius*⟩ 应当是可以直接赋予尺寸寄存器的尺寸表达式。

本命令的定义是：

```
\def\pgfqpointpolar#1#2{%
  \global\pgf@x=#2%
  \global\pgf@y=\pgf@x%
  \pgfmathcos@{#1}%
  \global\pgf@x=\pgfmathresult\pgf@x%
  \pgfmathsin@{#1}%
  \global\pgf@y=\pgfmathresult\pgf@y\relax%
}
```

参数 ⟨*degree*⟩ 被用作正弦、余弦函数的参数，计算正弦值 s 和余弦值 c.

参数 ⟨*radius*⟩ 被直接赋予寄存器，所以它应当是一个尺寸表达式。

本命令全局地做赋值 \pgf@x= c⟨*radius*⟩, \pgf@y= s⟨*radius*⟩, 从而确定一个点。

12.2 XYZ-坐标系统中的点

在文件《pgfcorepoints.code.tex》中有下面的规定：

```
\pgfsetxvec{\pgfpoint{1cm}{0cm}}
\pgfsetyvec{\pgfpoint{0cm}{1cm}}
\pgfsetzvec{\pgfpoint{-0.385cm}{-0.385cm}}
```

以上三行命令分别指定 3 个单位向量，作为 xyz-Coordinate System 的 x, y, z 轴的单位向量。

寄存器 (\pgf@xx, \pgf@xy), (\pgf@yx, \pgf@yy), (\pgf@zx, \pgf@zy) 看作 canvas 坐标系中的点，用作 xyz 坐标系轴的单位向量。

\pgfsetxvec{⟨*point*⟩}

参数 ⟨*point*⟩ 是 canvas 坐标系中的点，或者是为 \pgf@x, \pgf@y 赋值的代码 (看作点坐标)，本命令将 ⟨*point*⟩ 作为 *xyz* 坐标系统的 *x* 轴的单位向量。

本命令的定义是：

```
\def\pgfsetxvec#1{%
  \pgf@process{#1}%
  \pgf@xx=\pgf@x%
  \pgf@xy=\pgf@y%
  \ignorespaces}
```

\pgfsetyvec{⟨*point*⟩}

参数 ⟨*point*⟩ 是 canvas 坐标系中的点，或者是为 \pgf@x, \pgf@y 赋值的代码 (看作点坐标)，本命令将 ⟨*point*⟩ 作为 *xyz* 坐标系统的 *y* 轴的单位向量。

本命令的定义是：

```
\def\pgfsetyvec#1{%
  \pgf@process{#1}%
  \pgf@yx=\pgf@x%
  \pgf@yy=\pgf@y%
  \ignorespaces}
```

\pgfsetzvec{⟨*point*⟩}

参数 ⟨*point*⟩ 是 canvas 坐标系中的点，或者是为 \pgf@x, \pgf@y 赋值的代码 (看作点坐标)，本命令将 ⟨*point*⟩ 作为 *xyz* 坐标系统的 *z* 轴的单位向量。

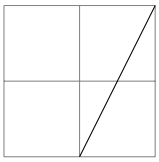
本命令的定义是：

```
\def\pgfsetzvec#1{%
  \pgf@process{#1}%
  \pgf@zx=\pgf@x%
  \pgf@zy=\pgf@y%
  \ignorespaces}
```

\pgfpointxy{⟨s_x⟩}{⟨s_y⟩}

本命令的两个参数 ⟨s_x⟩ 和 ⟨s_y⟩ 都会被命令 \pgfmathparse 解析，解析的结果再乘以 *xyz* 坐标轴的单位向量，从而确定一个点。

本命令最后全局地为 \pgf@x, \pgf@y 赋值，将这两个尺寸寄存器值作为 canvas 坐标系中的点。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

本命令的定义是：

```
\def\pgfpointxy#1#2{%
  \pgfmathparse{#1}%
  \let\pgftemp@x=\pgfmathresult%
  \pgfmathparse{#2}%
  \let\pgftemp@y=\pgfmathresult%
  \global\pgf@x=\pgftemp@x\pgf@xx%
  \global\advance\pgf@x by \pgftemp@y\pgf@yx%
  \global\pgf@y=\pgftemp@x\pgf@xy%
  \global\advance\pgf@y by \pgftemp@y\pgf@yy}
```

\pgfqpointxy{⟨s_x⟩}{⟨s_y⟩}

参数 ⟨s_x⟩ 和 ⟨s_y⟩ 都应是数值，不会被 \pgfmathparse 解析。

本命令的定义是：

```
\def\pgfqpointxy#1#2{%
  \global\pgf@x=#1\pgf@xx%
  \global\advance\pgf@x by #2\pgf@yx%
  \global\pgf@y=#1\pgf@xy%
  \global\advance\pgf@y by #2\pgf@yy}
```

\pgfpointpolarxy{⟨*degree*⟩}{⟨*radius*⟩ and ⟨*y-radius*⟩}

这个命令产生一个极坐标点。

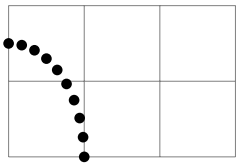
参数 ⟨*degree*⟩ 会被 \pgfmathparse 解析,结果作为角度。参数 ⟨*radius*⟩ 和 ⟨*y-radius*⟩ 会被 \pgfmathsetlength 处理。

参数中的 and ⟨*y-radius*⟩ 部分是可选的，如果不给出这一部分，就默认为 and ⟨*x-radius*⟩.

本命令利用正弦、余弦函数计算

$$a\cos\theta(\pgf@xx,\pgf@xy)+b\sin\theta(\pgf@yx,\pgf@yy)=(\pgf@x,\pgf@y).$$

本命令最后全局地为 \pgf@x, \pgf@y 赋值，将这两个尺寸寄存器值作为点的坐标。

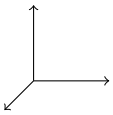


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolarxy{\angle}{1 and 1.5}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointxyz{⟨s_x⟩}{⟨s_y⟩}{⟨s_z⟩}

本命令的参数 ⟨s_x⟩, ⟨s_y⟩, ⟨s_z⟩ 都会被命令 \pgfmathparse 解析，解析的结果再乘以 xyz 坐标轴的单位向量，从而确定一个点。

本命令最后全局地为 \pgf@x, \pgf@y 赋值，将这两个尺寸寄存器值作为点的坐标。



```
\begin{pgfpicture}
  \pgfsetarrowsend{to}
  \pgfpathmoveto{\pgfpointorigin} \pgfpathlineto{\pgfpointxyz{0}{0}{1}}
  \pgfusepath{stroke}
  \pgfpathmoveto{\pgfpointorigin} \pgfpathlineto{\pgfpointxyz{0}{1}{0}}
  \pgfusepath{stroke}
  \pgfpathmoveto{\pgfpointorigin} \pgfpathlineto{\pgfpointxyz{1}{0}{0}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfqpointxyz{⟨s_x⟩}{⟨s_y⟩}{⟨s_z⟩}

参数 ⟨s_x⟩, ⟨s_y⟩, ⟨s_z⟩ 都应是数值，不会被 \pgfmathparse 解析。

本命令的定义是：

```
\def\pgfqpointxyz#1#2#3{%
  \global\pgf@x=#1\pgf@xx%
  \global\advance\pgf@x by #2\pgf@yx%
  \global\advance\pgf@x by #3\pgf@zx%
  \global\pgf@y=#1\pgf@xy%
  \global\advance\pgf@y by #2\pgf@yy%
  \global\advance\pgf@y by #3\pgf@zy}
```

\pgfpointcylindrical{⟨*degree*⟩}{⟨*radius*⟩}{⟨*height*⟩}

这个命令产生一个圆柱坐标系点，其定义是：

```
\def\pgfpointcylindrical#1#2#3{%
  \pgfpointpolarxy{#1}{#2}%
  \pgfmathparse{#3}%
  \global\advance\pgf@x by \pgfmathresult\pgf@zx%
  \global\advance\pgf@y by \pgfmathresult\pgf@zy}
```

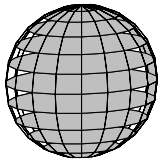
可见本命令等效于

```
\pgfpointadd{\pgfpointpolarxy{⟨degree⟩}{⟨radius⟩}}{\pgfpointxyz{0}{0}{⟨height⟩}}
```

本命令的三个参数都会被命令 \pgfmathparse 解析。本命令最后全局地为 \pgf@x, \pgf@y 赋值。

\pgfpointspherical{⟨*longitude*⟩}{⟨*latitude*⟩}{⟨*radius*⟩}

这个命令产生一个球坐标系点，⟨*longitude*⟩ 是经度，⟨*latitude*⟩ 是纬度，⟨*radius*⟩ 是半径。
本命令的三个参数都会被 \pgfmathparse 解析。本命令会利用三角函数做计算。
本命令最后全局地为 \pgf@x, \pgf@y 赋值。



```
\begin{tikzpicture}[x={(-135:0.25)},y={(1cm,0)},z={(0,1cm)}]
  \pgfsetfillcolor{lightgray}
  \foreach \latitude in {-90,-75,...,90}
  {
    \foreach \longitude in {0,20,...,360}
    {
      \pgfpathmoveto{\pgfpointspherical{\longitude}{\latitude}{1}}
      \pgfpathlineto{\pgfpointspherical{\longitude+20}{\latitude}{1}}
      \pgfpathlineto{\pgfpointspherical{\longitude+20}{\latitude+15}{1
      ↪  }}
      \pgfpathlineto{\pgfpointspherical{\longitude}{\latitude+15}{1}}
      \pgfpathclose
    }
    \pgfusepath{fill,stroke}
  }
\end{tikzpicture}
```

12.3 用已有坐标构建新的坐标

12.3.1 基本的坐标计算

\pgfpointtransformed{⟨*point*⟩}

本命令将当前的 canvas 变换矩阵用于点 ⟨*point*⟩, 然后将得到的点的坐标全局地保存在寄存器 \pgf@x 和 \pgf@y 中。
本命令的定义是：

```
\def\pgfpointtransformed#1{%
  \pgf@process{%
    #1%
    \pgf@pos@transform@glob%
  }%
}
```

\pgf@pos@transform{⟨*register x*⟩}{⟨*register y*⟩}

尺寸寄存器 ⟨*register x*⟩, ⟨*register y*⟩ 代表一个点的坐标，本命令用当前的 canvas 坐标系的矩阵变换这两个坐标，变换结果仍然保存到这两个寄存器中。这个变换结果是局部保存的。
见文件《pgfcoretransformations.code.tex》。

\pgf@pos@transform@glob

把当前的寄存器 \pgf@x, \pgf@y 看作点坐标，本命令用当前的 canvas 坐标系的矩阵变换这两个坐标，变换结果全局地保存到这两个寄存器中。

在文件《pgfcoretransformations.code.tex》中有：

```
\def\pgf@pos@transform@glob{%
  \ifpgf@pt@identity%
  \else%
    \pgf@pt@temp=\pgf@x%
    \global\pgf@x=\pgf@pt@aa\pgf@x%
    \global\advance\pgf@x by\pgf@pt@ba\pgf@y%
    \global\pgf@y=\pgf@pt@bb\pgf@y%
    \global\advance\pgf@y by\pgf@pt@ab\pgf@pt@temp%
  \fi%
  \global\advance\pgf@x by\pgf@pt@x%
  \global\advance\pgf@y by\pgf@pt@y%
}
```

\pgfpointadd{⟨v_1⟩}{⟨v_2⟩}

本命令得到向量 ⟨v_1⟩ 与 ⟨v_2⟩ 的和。

\pgfpointscale{⟨*factor*⟩}{⟨*coordinate*⟩}

本命令得到数值 ⟨*factor*⟩ 与向量 ⟨*coordinate*⟩ 的乘积。实际上本命令的第一个参数 ⟨*factor*⟩ 会被命令 \pgfmathparse 解析，所以 ⟨*factor*⟩ 可以带有长度单位。本命令的定义是：

```
\def\pgfpointscale#1#2{%
  \pgf@process{#2}%
  \pgfmathparse{#1}%
  \global\pgf@x=\pgfmathresult\pgf@x%
  \global\pgf@y=\pgfmathresult\pgf@y%
}
```

从这个定义看，下面代码也是可行的：

```
\pgfpoint{10}{20}
\pgfpointscale{2pt}{}
\makeatletter
\the\pgf@x, \the\pgf@y
\makeatother
```

20.0pt, 40.0pt

\pgfpointdiff{⟨*start*⟩}{⟨*end*⟩}

本命令得到向量 ⟨*end*⟩ 减去 ⟨*start*⟩ 的差，即“终点”减去“起点”。此命令的定义是：

```
\def\pgfpointdiff#1#2{%
  \pgf@process{#1}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{#2}%
  \global\advance\pgf@x by-\pgf@xa\relax%
  \global\advance\pgf@y by-\pgf@ya\relax\ignorespaces}
```

\pgfpointnormalised{⟨*point*⟩}

本命令将向量 ⟨*point*⟩ 单位化，即方向不变，长度变成 1pt. 如果 ⟨*point*⟩ 是 0 向量或者是长度极短的向量，那么本命令得到的是方向竖直向上，长度是 1pt 的向量。

本命令内部使用反正切函数、正弦函数、余弦函数做计算：

```
\def\pgfpointnormalised#1{%
  \pgf@process{#1}%
  \pgfmathatantwo{\the\pgf@y}{\the\pgf@x}%
  \let\pgf@tmp=\pgfmathresult%
  \pgfmathcos@{\pgf@tmp}%
  \pgf@x=\pgfmathresult pt\relax%
  \pgfmathsin@{\pgf@tmp}%
  \pgf@y=\pgfmathresult pt\relax%
}
```

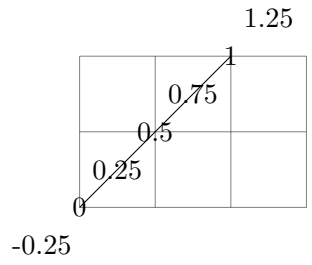
本命令最后（非全局地）为 \pgf@x, \pgf@y 赋值。

12.3.2 直线或曲线上的点

\pgfpointlineattime{⟨time t⟩}{⟨point p⟩}{⟨point q⟩}

参数 ⟨time t⟩ 会被 \pgfmathsetmacro 处理。

本命令假设 ⟨point p⟩ 是时刻 $t=0$ 时动点所处的位置 P, ⟨point q⟩ 是时刻 $t=1$ 时动点所处的位置 Q, 动点在这两点决定的直线上移动。本命令得到时刻 ⟨time t⟩ 时动点所处的位置，即点 $P+t(Q-P)$.

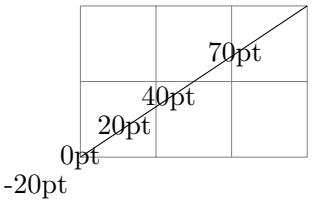



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{2cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \t in {-0.25,0,...,1.25}
  {\pgftext
  ↪  [at=\pgfpointlineattime{\t}{\pgfpointorigin}{\pgfpoint{2cm}{2cm}}]
  ↪  {\t}}
\end{tikzpicture}
```

\pgfpointlineatdistance{⟨distance⟩}{⟨start point⟩}{⟨end point⟩}

参数 ⟨distance⟩ 会被 \pgfmathsetlength 处理，处理结果作为一个“距离”。

点 ⟨start point⟩ 与 ⟨end point⟩ 决定一个方向，沿着这个方向，与点 ⟨start point⟩ 的距离为 ⟨distance⟩ 的点就是本命令确定的点。⟨distance⟩ 可以是负值尺寸。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \d in {-20pt,0pt,20pt,40pt,70pt}
  {\pgftext
  ↪  [at=\pgfpointlineatdistance{\d}{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}]
  ↪  {\d}}
\end{tikzpicture}
```

\pgfpointarcaxesattime{⟨time t⟩}{⟨center⟩}{⟨0-degree axis⟩}{⟨90-degree axis⟩}{⟨start angle⟩}{⟨end angle⟩}

记 ⟨time t⟩$=t$ 代表一个时刻，它会被 \pgfmathparse 处理。

点 ⟨center⟩$=C$ 代表椭圆的中心；向量 ⟨0-degree axis⟩$=V_x$ 代表椭圆的横半轴向量；向量 ⟨90-degree axis⟩$=V_n$ 代表椭圆的纵半轴向量；

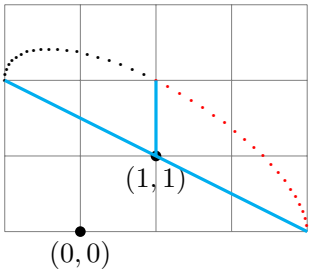
⟨start angle⟩$=\theta_s$ 代表椭圆弧的起始角度；⟨end angle⟩$=\theta_e$ 代表椭圆弧的终止角度；这两个参数会被 \pgfmathsetmacro 处理。

本命令的处理是：

1. 计算 $\theta=\theta_s+t(\theta_e-\theta_s)$

2. 若 $\theta_s > \theta_e$, 则记 $\delta = 1$; 若 $\theta_s \leqslant \theta_e$, 则记 $\delta = -1$; 计算 $T = \delta(\sin(\theta)V_z - \cos(\theta)V_n)$, 这就是椭圆弧在 θ 处的切向量。把 T 的坐标保存在 \pgf@xa 和 \pgf@ya 中。
3. 计算 $\cos(\theta)V_z + \sin(\theta)V_n + C$, 结果全局地保存在 \pgf@x 和 \pgf@y 中。

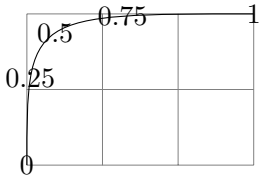
本命令计算了椭圆弧在时刻 t 的点，以及在这个点处的切向量。




```
\begin{tikzpicture}
  \draw [help lines] (-1,0) grid (3,3);
  \fill circle (2pt) node [below] {$(0,0)$};
  \fill (1,1) circle (2pt) node [below] {$(1,1)$};
  \foreach \t in {0,0.05,...,1}
  {
    \pgftext[at=\pgfpointarcaxesattime{\t}{\pgfpoint{1cm}{1cm}}
      {\pgfpoint{-2cm}{1cm}}{\pgfpoint{0cm}{1cm}}{0}{90}]{.}% 黑点号
    \pgftext[at=\pgfpointarcaxesattime{\t}{\pgfpoint{1cm}{1cm}}
      {\pgfpoint{-2cm}{1cm}}{\pgfpoint{0cm}{1cm}}{90}{180}]{\color{red
      ↪  }.}% 红点号
  }
  \draw [cyan,line width=1.2pt](1,1)--++(0,1) (1,1)--++(-2,1)
  ↪  (1,1)--++(2,-1);
\end{tikzpicture}
```

\pgfpointcurveattime{⟨*time t*⟩}{⟨*point p*⟩}{⟨*point s*$_1$⟩}{⟨*point s*$_2$⟩}{⟨*point q*⟩}

设想一个以点 ⟨*point p*⟩ 为始点，以 ⟨*point q*⟩ 为终点，以 ⟨*point s*$_1$⟩, ⟨*point s*$_2$⟩ 为控制点的控制曲线，假设一个动点在该曲线上运动，在时刻 $t = 0$ 时，动点位于始点 ⟨*point p*⟩, 在时刻 $t = 1$ 时，动点位于终点 {⟨*point q*⟩}, 那么在时刻 ⟨*time t*⟩ 时动点的位置 M 就是本命令确定的点。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto{\pgfpoint{0cm}{2cm}}{\pgfpoint{0cm}{2cm}}{\pgfpoint
  ↪  {3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \t in {0,0.25,0.5,0.75,1}
  {\pgftext[at=\pgfpointcurveattime{\t}
                                  {\pgfpointorigin}
                                  {\pgfpoint{0cm}{2cm}}
                                  {\pgfpoint{0cm}{2cm}}
                                  {\pgfpoint{3cm}{2cm}}]{\t}}
\end{tikzpicture}
```

设 ⟨*time t*⟩$= t$, ⟨*point p*⟩$= P$, ⟨*point s*$_1$⟩$= S_1$, ⟨*point s*$_2$⟩$= S_2$, ⟨*point q*⟩$= Q$, 本命令计算：

1. $X = (1-t)P + tS_1$
2. $A = (1-t)S_1 + tS_2$
3. $B = (1-t)S_2 + tQ$
4. $X = (1-t)X + tA$
5. $A = (1-t)A + tB$
6. $B = X$
7. $X = (1-t)X + tA$

所以

$$
\begin{aligned}
X(t) &= (1-t)^3P + 3t(1-t)^2S_1 + 3t^2(1-t)S_2 + t^3Q,\\
A(t) &= (1-t)^2S_1 + 2t(1-t)S_2 + t^2Q,\\
B(t) &= (1-t)^2P + 2t(1-t)S_1 + t^2S_2,\\
\frac{1}{3}X'(t) &= A(t) - B(t).
\end{aligned}
$$

本命令保存计算结果：

- 宏 \pgf@time@s 保存 t
- 宏 \pgf@time@t 保存 $1-t$
- 寄存器 (\pgf@x, \pgf@y) 全局地保存 $X(t)$
- 寄存器 (\pgf@xa, \pgf@ya) 保存 $A(t)$
- 寄存器 (\pgf@xb, \pgf@yb) 保存 $B(t)$
- 寄存器 (\pgf@xc, \pgf@yc) 保存 Q

执行本命令后，以上宏、寄存器是可用的。

12.3.3 矩形或椭圆边界上的点

\pgfpointborderrectangle{⟨*direction point*⟩}{⟨*corner*⟩}

参数 ⟨*direction point*⟩, ⟨*corner*⟩ 是 PGF 点，或者是为 \pgf@x, \pgf@y 赋值的代码。

设想一个以原点为中心，以点 ⟨*corner*⟩ 为右上角的矩形，以及一条以原点为起点，方向为向量 ⟨*direction point*⟩ 的射线，二者交于点 P, 这个点 P 就是本命令确定的点。

注意向量 ⟨*direction point*⟩ 的长度应当接近 1pt，如果向量 ⟨*direction point*⟩ 的长度不是 1pt，那么它会被“单位化”，使其长度是 1pt. 在“单位化”过程中可能会出现舍入误差，因此向量 ⟨*direction point*⟩ 的长度越是接近 1pt，舍入误差就越小。

\pgfpointborderellipse{⟨*direction point*⟩}{⟨*corner*⟩}

本命令的参数是 PGF 点，或者是为 \pgf@x, \pgf@y 赋值的代码。

设一个以原点为中心，以点 ⟨*corner*⟩ 为右上角的矩形，在该矩形内有一个内切椭圆，还有一条以原点为起点，方向为向量 ⟨*direction point*⟩ 的射线，射线与椭圆的交点是 P, 这个点 P 就是本命令确定的点。

12.3.4 直线与直线、圆与圆的交点

\pgfpointintersectionoflines{⟨p⟩}{⟨q⟩}{⟨s⟩}{⟨t⟩}

本命令的参数是 PGF 点，或者是为 \pgf@x, \pgf@y 赋值的代码。

设过点 ⟨p⟩ 和 ⟨q⟩ 的直线，与过点 ⟨s⟩ 和 ⟨t⟩ 的直线相交，交点是 P, 这个点 P 就是本命令确定的点。

如果两直线没有交点，或者 ⟨p⟩ 与 ⟨q⟩ 重合，或者 ⟨s⟩ 与 ⟨t⟩ 重合，那么本命令的计算结果是未知的。

\pgfpointintersectionofcircles{⟨p_1⟩}{⟨p_2⟩}{⟨r_1⟩}{⟨r_2⟩}{⟨*solution*⟩}

参数 ⟨p_1⟩, ⟨p_2⟩ 是 PGF 点，或者是为 \pgf@x, \pgf@y 赋值的代码。

参数 ⟨r_1⟩, ⟨r_2⟩ 会被 \pgfmathsetlength 处理。

本命令假设一个以点 ⟨p_1⟩ 为圆心、以 ⟨r_1⟩ 为半径的圆，与一个以点 ⟨p_2⟩ 为圆心、以 ⟨r_2⟩ 为半径的圆相交，本命令确定二者的交点。如果 ⟨*solution*⟩ 是 1，则本命令确定两圆的第一个交点；否则本命令确定两圆的第二个交点。

如果两个圆没有交点，可能会导致错误。

利用 intersections 库的命令可以计算两个路径的交点，参考 intersections 库。

12.4 获取点的坐标

\pgfextractx{⟨*dimension*⟩}{⟨*point*⟩}

⟨*dimension*⟩ 是个提前声明的 TEX 尺寸寄存器，本命令将点 ⟨*point*⟩ 的 x 分量，即一个长度尺寸赋予 ⟨*dimension*⟩.

56.9055pt
```
\newdimen\mydim
\pgfextractx{\mydim}{\pgfpoint{2cm}{4pt}}
\the\mydim
```

\pgfextracty{⟨*dimension*⟩}{⟨*point*⟩}

⟨*dimension*⟩ 是个已经声明的 TEX 尺寸寄存器，本命令将点 ⟨*point*⟩ 的 y 分量，即一个长度尺寸赋予 ⟨*dimension*⟩.

\pgfgetlastxy{⟨*macro for x*⟩}{⟨*macro for y*⟩}

将本命令之前最近出现的坐标点的 x 分量和 y 分量分别保存到宏 ⟨*macro for x*⟩ 和 ⟨*macro for y*⟩, 这两个宏不需要提前声明。

‘56.9055pt’ and ‘113.81102pt’.
```
\pgfpoint{2cm}{4cm}
\pgfgetlastxy{\macrox}{\macroy}
‘\macrox’ and ‘\macroy’.
```

此命令的定义是：

```
\def\pgfgetlastxy#1#2{%
    \edef#1{\the\pgf@x}%
    \edef#2{\the\pgf@y}%
}%
```

使用本命令时要注意那些定义坐标点的基本层命令之间的区别，例如，\pgfpoint →P. 250 不会全局地规定 \pgf@x 与 \pgf@y 的值，而 \pgfqpoint →P. 251 则会全局地规定这两个寄存器的值。

第十三章　坐标系统与变换矩阵

PGF 的变换有：

- 坐标变换，这种变换由 PGF 自己完成计算，所以 PGF 能跟踪变换结果。这种变换又分为：
  - 线性变换，这个变换利用矩阵来实现，又分为
    - 与 canvas 坐标系有关的变换，当说到“线性变换”时，一般指的是这种变换；当说到“线性矩阵、变换矩阵”时，一般指的是实现这种变换的矩阵。
    - 与 xyz 坐标系有关的变换，这是 (关于 canvas 坐标系的)“线性变换”的衍生形式。
  - 非线性变换，这种变换没有固定的形式，通常需要自定义变换代码，可参考 curvilinear 库。
- 画布变换，PGF 的画布变换命令会生成输出语言的变换句法 (protocoling)，然后交给编译引擎处理，PGF 并不计算这种变换，因此 PGF(目前) 无法追踪画布变换的结果。

PGF 的构建路径的命令 (如 `\pgfpathmoveto`, `\pgfpathlineto`) 会先对点坐标 (即寄存器 `\pgf@x` 和 `\pgf@y`) 做线性变换 (结果仍然保存在寄存器 `\pgf@x` 和 `\pgf@y` 中)，再对结果 (即寄存器 `\pgf@x` 和 `\pgf@y`) 做非线性变换 (结果仍然保存在寄存器 `\pgf@x` 和 `\pgf@y` 中)。

PGF 的变换所做的计算基本上都是利用 TEX 的尺寸寄存器完成的，有的地方需要纯数值运算，也要借助寄存器来实现。由于寄存器的单位是 pt, 所以变换过程中数据的单位通常也是 pt. 当为变换提供输入参数时，一定要考虑参数是否需要带上长度单位——如果需要带上长度单位，那一定要恰当使用单位，例如，输入的 1pt 可能被变换为一个单位长度 (这个单位长度可能不是 1pt)。

13.1　线性变换

文件《pgfcoretransformations.code.tex》。

为了方便，以下认为“坐标系”等价于“标架”。

假设矩阵 A 和向量 x, 对于运算 Ax 有 2 种视角：

(i) 假设有一个固定不变的标架，A 是变换，x 是标架中某个点的坐标，运算 Ax 的结果是标架中的另一个点的坐标，这是坐标变换导致的点变换；

(ii) 假设有一个标架 $\mathscr{C}$, 以 $\mathscr{C}$ 的向量 c_1, c_2 为基向量，以 $\mathscr{C}$ 的原点为原点，得到一个新的标架 $\mathscr{A}$ (标架 $\mathscr{A}$ 以 $\mathscr{C}$ 为参照); 以 $\mathscr{A}$ 的基向量为列向量组成矩阵 A; 某个点 P 在 $\mathscr{C}$ 中的坐标是 x (一个列向量)；某个点 Q 在 $\mathscr{A}$ 中的坐标也是 x (一个列向量)；运算 Ax 得到点 Q 在 $\mathscr{C}$ 中的坐标；此时

$$x \to Ax \text{ 诱导 } P \to Q,$$

这是标架变换导致的坐标变换和点变换。

这两个角度是等价的，它们是对同一代数式的不同几何解释。第 1 个角度容易理解，第 2 个角度复杂一些。两个角度都有用处，哪个角度容易解释问题，就采用哪个角度。第 2 个角度是说：认为变换命令的主要作用是改变标架——以当前标架为参照来改变当前标架，得到一个新的标架；用标架的基向量 (以及原点，见下文) 构成矩阵，来对坐标 (点) 做变换，这个矩阵就是“变换矩阵”、“线性矩阵”，所做的变换是

“线性变换”。有时候会提到“node 自身的坐标系”，“箭头路径自身的坐标系”，“装饰路径自身的坐标系”之类的说法，从第 2 个角度来看会比较顺畅一些。

PGF 有 2 种线性坐标系：xyz 坐标系和 canvas 坐标系，这 2 种坐标系实际上是 PGF 的 2 种矩阵，所以：坐标变换 $\rightarrow$ 标架变换 $\rightarrow$ 矩阵变换。

13.1.1 对应 canvas 坐标系的矩阵

首先假设有一个不变坐标系 $\mathscr{C}_0$, 它的原点记为 O, 它的 x 轴的正方向水平向右，y 轴的正方向竖直向上。以 $\mathscr{C}_0$ 的基向量和原点为行向量构成的齐次化矩阵是

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = T_0.$$

- $\mathscr{C}_0$ 中的 3 个向量

  $$C_0^1 = (x_0^1, y_0^1),\ C_0^2 = (x_0^2, y_0^2),\ P_0 = (p_0, q_0),$$

  构成一个 canvas 坐标系 $\mathscr{C}_1(C_0^1, C_0^2; P_0)$, 其中 P_0 是 $\mathscr{C}_1$ 的原点，C_0^1, C_0^2 分别是 $\mathscr{C}_1$ 的 x 轴和 y 轴的单位向量。以 $\mathscr{C}_1$ 的基向量和原点为行向量构成的矩阵是

  $$\begin{bmatrix} x_0^1 & y_0^1 & 0 \\ x_0^2 & y_0^2 & 0 \\ p_0 & q_0 & 1 \end{bmatrix} = T_1.$$

- 对于 $i = 1, 2, \ldots,$ 坐标系 $\mathscr{C}_i$ 中的 3 个向量

  $$C_i^1 = (x_i^1, y_i^1),\ C_i^2 = (x_i^2, y_i^2),\ P_i = (p_i, q_i),$$

  构成一个 canvas 坐标系 $\mathscr{C}_{i+1}(C_i^1, C_i^2; P_i)$, 其中 P_i 是 $\mathscr{C}_{i+1}$ 的原点，C_i^1, C_i^2 分别是 $\mathscr{C}_{i+1}$ 的 x 轴和 y 轴的单位向量。以 $\mathscr{C}_{i+1}$ 的基向量和原点为行向量构成的矩阵是

  $$\begin{bmatrix} x_i^1 & y_i^1 & 0 \\ x_i^2 & y_i^2 & 0 \\ p_i & q_i & 1 \end{bmatrix} = T_{i+1}.$$

- $\mathscr{C}_i$ 在 $\mathscr{C}_0$ 中的表达是如下的矩阵积

  $$T_i \cdots T_1 T_0 = W_i,$$

  第 i 个变换命令的参数往往是矩阵 T_i 的元素 (以 $\mathscr{C}_{i-1}$ 为参照)，而变换命令的计算结果则是 W_i.
- 在第 i 个变换命令完成计算后，当前的 canvas 坐标系就是 W_i. 如果页面上的点 A 在 $\mathscr{C}_i$ 中的坐标为 (a_1, a_2), 那么点 A 在 $\mathscr{C}_0$ 中的坐标是 $(a_1, a_2)W_i$, 这就是坐标变换。

在内部计算中，canvas 坐标系在不变坐标系 $\mathscr{C}_0$ 中的矩阵是：

$$\begin{bmatrix} \texttt{\textbackslash pgf@pt@aa} & \texttt{\textbackslash pgf@pt@ab} & 0 \\ \texttt{\textbackslash pgf@pt@ba} & \texttt{\textbackslash pgf@pt@bb} & 0 \\ \texttt{\textbackslash pgf@pt@x} & \texttt{\textbackslash pgf@pt@y} & 1 \end{bmatrix}$$

其中 \pgf@pt@aa, \pgf@pt@ab, \pgf@pt@ba, \pgf@pt@ba 是 4 个数值“宏”，\pgf@pt@x, \pgf@pt@y 是 2 个“尺寸寄存器”。对点 (x, y)(其中的 x, y 带有长度单位) 做的变换就是：

$$(x, y, 1) \begin{bmatrix} \texttt{\textbackslash pgf@pt@aa} & \texttt{\textbackslash pgf@pt@ab} & 0 \\ \texttt{\textbackslash pgf@pt@ba} & \texttt{\textbackslash pgf@pt@bb} & 0 \\ \texttt{\textbackslash pgf@pt@x} & \texttt{\textbackslash pgf@pt@y} & 1 \end{bmatrix} = \begin{bmatrix} \texttt{\textbackslash pgf@pt@aa} \cdot x + \texttt{\textbackslash pgf@pt@ba} \cdot y + \texttt{\textbackslash pgf@pt@x} \\ \texttt{\textbackslash pgf@pt@ab} \cdot x + \texttt{\textbackslash pgf@pt@bb} \cdot y + \texttt{\textbackslash pgf@pt@y} \\ 1 \end{bmatrix}^T$$

在初始之下，这个矩阵是单位矩阵。如果把这个矩阵看作是一个“标架”，那么向量

$$(\texttt{\textbackslash pgf@pt@aa}, \texttt{\textbackslash pgf@pt@ab}),$$
$$(\texttt{\textbackslash pgf@pt@ba}, \texttt{\textbackslash pgf@pt@bb}),$$

是标架的基向量，而 (\pgf@pt@x, \pgf@pt@y) 是标架的原点。

PGF 的构建路径的命令 (如 \pgfpathmoveto, \pgfpathlineto) 会利用这个矩阵对点的坐标 (寄存器 \pgf@x 和 \pgf@y) 做变换，即执行 \pgfpointtransformed→P. 255 做变换 (结果仍然保存在寄存器 \pgf@x 和 \pgf@y 中)。注意，路径中的点坐标有如下对应

$$(\texttt{1pt,0}) \to (\texttt{\textbackslash pgf@pt@aa}, \texttt{\textbackslash pgf@pt@ab}),$$
$$(\texttt{0,1pt}) \to (\texttt{\textbackslash pgf@pt@ba}, \texttt{\textbackslash pgf@pt@bb}),$$

这个矩阵的元素值是局部定义的，其作用效果受到 TEX 组的限制。

\ifpgf@pt@identity

在计算过程中，如果得到的 canvas 坐标系的矩阵是单位矩阵，那么应当把这个 TEX-if 的真值设置为 true. 也就是说，如果这个 TEX-if 的真值是 true, 那么当前 canvas 坐标系的矩阵是单位矩阵。

\pgftransformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{⟨point⟩}

参数 ⟨a⟩,⟨b⟩,⟨c⟩,⟨d⟩ 的 (彻底) 展开结果应当能作为 \pgfmathsetlength 的参数，例如

```
\pgfmathsetlength\pgf@x{彻底展开的 ⟨a⟩}%
```

因此这 4 个参数的彻底展开可以是数学表达式，或者是以“+”开头的尺寸表达式。

参数 ⟨point⟩ 是 PGF 的点，或者是为 \pgf@x 和 \pgf@y 赋值的代码。

记当前的变换矩阵是 $\begin{bmatrix} \texttt{\textbackslash pgf@pt@aa} & \texttt{\textbackslash pgf@pt@ab} & 0 \\ \texttt{\textbackslash pgf@pt@ba} & \texttt{\textbackslash pgf@pt@bb} & 0 \\ \texttt{\textbackslash pgf@pt@x} & \texttt{\textbackslash pgf@pt@y} & 1 \end{bmatrix} = T$, 本命令的处理是：

1. 处理 \pgf@process{⟨point⟩}, 并赋值寄存器 \pgf@xc=\pgf@x, \pgf@yc=\pgf@y.
2. 用 \edef 将 ⟨a⟩,⟨b⟩,⟨c⟩,⟨d⟩ 彻底展开，展开结果被 \pgfmathsetlength 处理，处理结果的数值部分分别记为 ⟨a'⟩,⟨b'⟩,⟨c'⟩,⟨d'⟩.
3. 重新计算变换矩阵为：

$$\begin{bmatrix} \langle a' \rangle & \langle b' \rangle & 0 \\ \langle c' \rangle & \langle d' \rangle & 0 \\ \texttt{\textbackslash pgf@xc} & \texttt{\textbackslash pgf@yc} & 1 \end{bmatrix} T = \begin{bmatrix} \texttt{\textbackslash pgf@pt@aa} & \texttt{\textbackslash pgf@pt@ab} & 0 \\ \texttt{\textbackslash pgf@pt@ba} & \texttt{\textbackslash pgf@pt@bb} & 0 \\ \texttt{\textbackslash pgf@pt@x} & \texttt{\textbackslash pgf@pt@y} & 1 \end{bmatrix}$$

并且全局定义 \pgf@x 和 \pgf@y (新标架的原点坐标) 为

$$(\texttt{\textbackslash pgf@xc}, \texttt{\textbackslash pgf@yc}, 1) \cdot T = (\texttt{\textbackslash pgf@x}, \texttt{\textbackslash pgf@y}, 1)$$

以上计算是利用寄存器完成的 (不是用数学引擎的命令)。

4. 重定义 \pgf@idtest:

```
\edef\pgf@idtest{\pgf@pt@aa,\pgf@pt@ba,\pgf@pt@ab,\pgf@pt@bb}%
```

5. 检查所得矩阵是否单位矩阵

```
\ifx\pgf@idtest\pgf@idmatrixtext%
  \pgf@pt@identitytrue%
\else%
  \pgf@pt@identityfalse%
\fi%
```

注意本命令对 \pgf@pt@aa, \pgf@pt@ab, \pgf@pt@ba, \pgf@pt@ba 这 4 个数值宏的赋值，以及对 \pgf@pt@x, \pgf@pt@y 这 2 个尺寸寄存器的赋值都 不是全局地；而对 \pgf@x, \pgf@y 这 2 个尺寸寄存器的赋值都 是全局地。

在文件《pgfcoretransformations.code.tex》中还有：

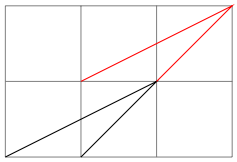
```
\def\pgf@idmatrixtext{1.0,0.0,0.0,1.0}
\def\pgf@zerozerotext{0.0,0.0}
\def\pgf@one@text{1.0}
\def\pgf@zero@text{0.0}
```

按 \pgf@process →P. 250 的定义，下面 2 种用法等价：

```
\pgftransformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{\pgfpoint{⟨x⟩}{⟨y⟩}}
等价于
\pgfpoint{⟨x⟩}{⟨y⟩}
\pgftransformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{}
```

\pgftransformshift{⟨*point*⟩}

按向量 ⟨*point*⟩ 做平移。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2,1) -- (1,0);
  \pgftransformshift{\pgfpoint{1cm}{1cm}}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

在文件《pgfcoretransformations.code.tex》中定义：

```
\def\pgftransformshift#1{\pgftransformcm{1}{0}{0}{1}{#1}}
```

可见命令 \pgftransformshift 调用命令 \pgftransformcm →P. 263 来对标架做平移。

\pgftransformxshift{⟨*dimensions*⟩}

在 x 轴方向做平移，其定义是：

```
\def\pgftransformxshift#1{\pgftransformcm{1}{0}{0}{1}{\pgfpoint{#1}{+0pt}}}
```

\pgftransformyshift{⟨*dimensions*⟩}

在 y 轴方向做平移，其定义是：

```
\def\pgftransformyshift#1{\pgftransformcm{1}{0}{0}{1}{\pgfpoint{+0pt}{#1}}}
```

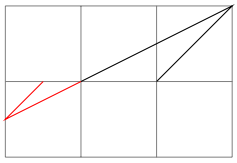
\pgftransformscale{⟨*factor*⟩}

以当前标架的原点为中心做位似变换，⟨*factor*⟩ 是放缩比例。如果 ⟨*factor*⟩ 是负值，则先以原点为中心做中心对称，再做位似变换。

本命令的定义是：

```
\def\pgftransformscale#1{\pgftransformcm{#1}{0}{0}{#1}{\pgfpointorigin}}
```

可见命令 \pgftransformscale 调用命令 \pgftransformcm →P. 263 来修改变换矩阵。



```
\begin{tikzpicture}
  \draw[help lines] (-1,-1) grid (2,1);
  \draw (0,0) -- (2,1) -- (1,0);
  \pgftransformscale{-.5}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformxscale{⟨*factor*⟩}

保持点的纵坐标不变，将点的横坐标乘上 ⟨*factor*⟩. 本命令的定义是：

```
\def\pgftransformxscale#1{\pgftransformcm{#1}{0}{0}{1.0}{\pgfpointorigin}}
```

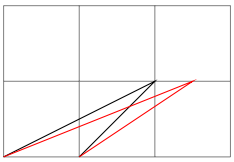
\pgftransformmyscale

保持点的横坐标不变，将点的纵坐标乘上 ⟨*factor*⟩. 本命令的定义是：

```
\def\pgftransformyscale#1{\pgftransformcm{1.0}{0}{0}{#1}{\pgfpointorigin}}
```

\pgftransformxslant{⟨*factor*⟩}

这个变换是 $(x, y) \to (x + y \cdot \texttt{<factor>}, y)$.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2,1) -- (1,0);
  \pgftransformxslant{.5}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

本命令的定义是：

```
\def\pgftransformxslant#1{\pgftransformcm{1.0}{0}{#1}{1.0}{\pgfpointorigin}}
```

\pgftransformyslant{⟨*factor*⟩}

这个变换是 $(x, y) \to (x, y + x \cdot \texttt{<factor>})$.
本命令的定义是：

```
\def\pgftransformyslant#1{\pgftransformcm{1.0}{#1}{0}{1.0}{\pgfpointorigin}}
```

\pgftransformrotate{⟨*angle*⟩}

参数 ⟨*angle*⟩ 是能被 \pgfmathparse 解析的表达式。本命令使得当前标架围绕其原点旋转 ⟨*angle*⟩ 角度。

本命令的定义是：

```
\def\pgftransformrotate#1{%
  \pgfmathparse{#1}%
  \let\pgftransform@angle=\pgfmathresult%
  \pgfmathsin@{\pgftransform@angle}%
  \let\pgftransform@sin=\pgfmathresult%
  \pgfmathcos@{\pgftransform@angle}%
  \let\pgftransform@cos=\pgfmathresult%
  \pgf@x=\pgftransform@sin pt%
  \pgf@xa=-\pgf@x%
  \pgftransformcm{\pgftransform@cos}{\pgftransform@sin}{\pgf@sys@tonumber{\pgf@xa
  ↪  }}{\pgftransform@cos}{\pgfpointorigin}%
}
```

可见旋转命令利用了三角函数。假设当前的变换矩阵是 $\left[\begin{smallmatrix} a_a & a_b & 0 \\ b_a & b_b & 0 \\ s & t & 1 \end{smallmatrix}\right]$, 本选项将其变成

$$\begin{bmatrix} \cos(\langle angle \rangle) & \sin(\langle angle \rangle) & 0 \\ -\sin(\langle angle \rangle) & \cos(\langle angle \rangle) & 0 \\ 0\text{pt} & 0\text{pt} & 1 \end{bmatrix} \begin{bmatrix} a_a & a_b & 0 \\ b_a & b_b & 0 \\ s & t & 1 \end{bmatrix}$$

例如，

```
\begin{tikzpicture}
    \pgftransformxscale{2}
    \pgftransformrotate{30}
    \draw [->,red](0,0)--(1,0);
    \draw [->,green](0,0)--(0,1);
\end{tikzpicture}
```

上面例子中，初始变换矩阵是 $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0\text{pt} & 0\text{pt} & 1 \end{bmatrix}$，两个变换命令做的变换是

$$\begin{bmatrix} \cos(30^\circ) & \sin(30^\circ) & 0 \\ -\sin(30^\circ) & \cos(30^\circ) & 0 \\ 0\text{pt} & 0\text{pt} & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0\text{pt} & 0\text{pt} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0\text{pt} & 0\text{pt} & 1 \end{bmatrix} = \begin{bmatrix} \sqrt{3} & \frac{1}{2} & 0 \\ -1 & \frac{\sqrt{3}}{2} & 0 \\ 0\text{pt} & 0\text{pt} & 1 \end{bmatrix}$$

这样，变换矩阵的基向量就由相互正交的 $(2,0)$ 与 $(0,1)$，变成了不相正交的 $(\sqrt{3},\frac{1}{2})$ 与 $(-1,\frac{\sqrt{3}}{2})$.

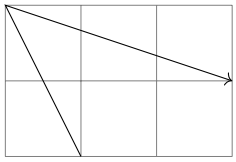
\pgftransformtriangle{⟨a⟩}{⟨b⟩}{⟨c⟩}

参数 ⟨a⟩, ⟨b⟩, ⟨c⟩ 是 PGF 点，或者是为 \pgf@x 和 \pgf@y 赋值的代码。

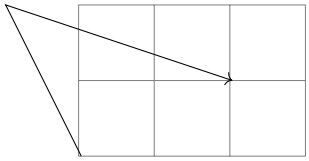
本命令得到一个新的标架，新的标架以原来标架中的点 ⟨a⟩ 为原点，以原来标架中的 $\overrightarrow{\langle a\rangle\langle b\rangle}$ 为“横轴”的单位向量，以原来标架中的 $\overrightarrow{\langle a\rangle\langle c\rangle}$ 为“纵轴”的单位向量。也就是说，“横轴”的 1pt 对应 $\overrightarrow{\langle a\rangle\langle b\rangle}$；“纵轴”的 1pt 对应 $\overrightarrow{\langle a\rangle\langle c\rangle}$.

本命令的定义是：

```
\def\pgftransformtriangle#1#2#3{%
  \pgf@process{#2}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{#3}%
  \pgf@xb=\pgf@x%
  \pgf@yb=\pgf@y%
  \pgf@process{#1}%
  \advance\pgf@xa by-\pgf@x%
  \advance\pgf@ya by-\pgf@y%
  \advance\pgf@xb by-\pgf@x%
  \advance\pgf@yb by-\pgf@y%
  \pgftransformcm%
  {\pgf@sys@tonumber\pgf@xa}{\pgf@sys@tonumber\pgf@ya}%
  {\pgf@sys@tonumber\pgf@xb}{\pgf@sys@tonumber\pgf@yb}%
  {\pgfpoint{\pgf@x}{\pgf@y}}%
}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformtriangle
    {\pgfpoint{1cm}{0cm}}
    {\pgfpoint{0cm}{2cm}}
    {\pgfpoint{3cm}{1cm}}
  \draw [->](0,0) -- (1pt,0pt) -- (0pt,1pt);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformtriangle
    {\pgfpoint{1}{0}}
    {\pgfpoint{0}{2}}
    {\pgfpoint{3}{1}}
  \draw [->](0,0) -- (1,0) -- (0,1);
\end{tikzpicture}
```

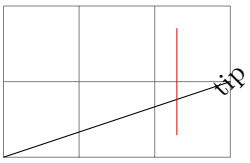
\pgftransformarrow{⟨start⟩}{⟨end⟩}

点 ⟨start⟩ 到点 ⟨end⟩ 构成一个向量，记这个向量的倾角是 θ，本命令将标架原点平移到点 ⟨end⟩ 处，并将标架旋转角度 θ，得到一个新的标架。

记 ⟨start⟩$=(s_1,s_2)$，⟨end⟩$=(e_1,e_2)$，$\frac{(e_1,e_2)-(s_1,s_2)}{\sqrt{(e_1-s_1)^2+(e_2-s_2)^2}}=(n_1,n_2)$. 本命令执行

```
\pgftransformshift{⟨end⟩}%
\pgftransformcm%
  {n1}{n2}%
  {−n2}{n1}{\pgfqpoint{0pt}{0pt}}%
```

本命令在当前 canvas 坐标系内做平移、旋转。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (3,1);
  \pgftransformarrow{\pgfpoint{1cm}{-1cm}}
    {\pgfpoint{3cm}{1cm}}
  \pgftext{tip}
  \draw [red](-1,0) -- (0,1);
\end{tikzpicture}
```

\pgftransformlineattime{⟨time⟩}{⟨start⟩}{⟨end⟩}

参数 ⟨time⟩ 应当能被 \pgfmathsetmacro 处理。

参数 ⟨start⟩ 和 ⟨end⟩ 是 PGF 点，或是为 \pgf@x, \pgf@y 赋值的代码。

记 $\langle start\rangle = (s_1, s_2)$, $\langle end\rangle = (e_1, e_2)$, $\frac{(e_1,e_2)-(s_1,s_2)}{\sqrt{(e_1-s_1)^2+(e_2-s_2)^2}} = (n_1, n_2)$.

本命令的处理是：

1. 调用 \pgfpointlineattime → P. 257 做平移

   ```
   \pgftransformshift{\pgfpointlineattime{⟨tima⟩}{⟨start⟩}{⟨end⟩}}%
   ```

   命令 \pgfpointlineattime 得到的点是 $(s_1, s_2) + \langle time\rangle \cdot ((e_1, e_2) - (s_1, s_2))$, 这一步将 canvas 坐标系的原点平移到这个点处。

2. 如果 \ifpgfresetnontranslationattime 的真值是 true, 则

   ```
   \pgftransformresetnontranslations
   ```

   将当前 canvas 坐标系的矩阵中的旋转、反射、放缩成分去掉，只保留平移作用。

3. 如果 \ifpgfslopedattime 的真值是 true,

   - 如果 \ifpgfallowupsidedownattime 的真值是 true, 则

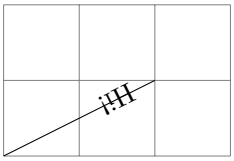
     ```
     \pgftransformcm%
       {n1}{n2}%
       {−n2}{n1}{\pgfqpoint{0pt}{0pt}}%
     ```

     此时 node 中的文字有可能出现“upside down”的状态 (上下颠倒)。

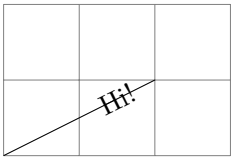
   - 如果 \ifpgfallowupsidedownattime 的真值是 false, 并且 $n_1 < 0$, 则

     ```
     \pgftransformcm%
       {−n1}{−n2}%
       {n2}{−n1}{\pgfqpoint{0pt}{0pt}}%
     ```

     此时 PGF 不会让 node 中的文字出现“upside down”的状态。



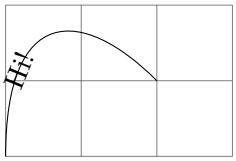

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgfallowupsidedownattimetrue% 允许 node 上下颠倒
  \pgftransformlineattime{.25}
    {\pgfpoint{2cm}{1cm}}{\pgfpointorigin}
  \pgftext{Hi!}
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2,1);
  \pgfslopedattimetrue
%  \pgfallowupsidedownattimetrue% 允许 node 上下颠倒
  \pgftransformlineattime{.25}
    {\pgfpoint{2cm}{1cm}}{\pgfpointorigin}
  \pgftext{Hi!}
\end{tikzpicture}
```

\pgftransformcurveattime{⟨*time*⟩}{⟨*start*⟩}{⟨*first support*⟩}{⟨*second support*⟩}{⟨*end*⟩}

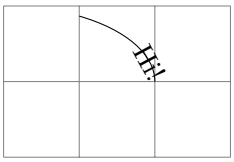
本命令类似 \pgftransformlineattime, 不过本命令调用 \pgfpointcurveattime 得到控制曲线上的一个点，然后将 canvas 坐标系的原点平移到这个点处，其余步骤一样。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgfslopedattimetrue
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}
    {\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

\pgftransformarcaxesattime{⟨*time t*⟩}{⟨*center*⟩}{⟨*0-degree axis*⟩}{⟨*90-degree axis*⟩}
{⟨*start angle*⟩}{⟨*end angle*⟩}

本命令类似 \pgftransformlineattime, 不过本命令调用 \pgfpointarcaxesattime 得到控制曲线上的一个点，然后将 canvas 坐标系的原点平移到这个点处，其余步骤一样。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpoint{2cm}{1cm}}
  \pgfpatharcaxes{0}{60}{\pgfpoint{2cm}{0cm}}
    {\pgfpoint{0cm}{1cm}}
  \pgfusepath{stroke}
  \pgfslopedattimetrue
  \pgftransformarcaxesattime{.25}{\pgfpoint{0cm}{1cm}}
    {\pgfpoint{2cm}{0cm}}{\pgfpoint{0cm}{1cm}}{0}{60}
  \pgftext{Hi!}
\end{tikzpicture}
```

\ifpgfslopedattime

这个 TEX-if 针对前面提到的带有“attime”的变换命令，它的默认值是 false. 如果它真值是 true, 那么 node 文字会沿着当前的 canvas 坐标系出现旋转，此时 node 文字可能会上下颠倒 (upside-down)。

\ifpgfallowupsidedownattime

这个 TEX-if 针对前面提到的带有“attime”的变换命令，它的默认值是 false, 也就是不允许 node 文字上下颠倒 (upside-down)。

\ifpgfresetnontranslationattime

这个 TEX-if 针对前面提到的带有“attime”的变换命令。如果它的真值是 true, 那么就会执行 \pgftransformresetnontranslations→P. 269.

\pgftransformreset

本命令将当前的 canvas 坐标系的矩阵变成单位矩阵，并设置 \ifpgf@pt@identity 的真值为 true.

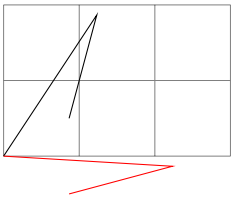
\pgftransformresetnontranslations

本命令将当前的 canvas 坐标系的矩阵变成平移矩阵 $\begin{pmatrix}1&0&s\\0&1&t\\0&0&1\end{pmatrix}$，也就是说，将旋转、反射、放缩成分去掉，只保留平移作用。


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{2}
  \pgftransformrotate{30}
  \pgftransformxshift{1cm}
  \pgftext{\color{red}rotated}
  \pgftransformresetnontranslations
  \pgftext{shifted only}
\end{tikzpicture}
```

\pgftransforminvert

本命令将当前的 canvas 坐标系的矩阵换成其逆矩阵。如果当前变换矩阵的条件数太小 (即接近奇异矩阵)，本命令的作用可能会有明显的误差。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \draw (0,0) -- (2,1) -- (1,0);
  \pgftransforminvert
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgfgettransform{⟨*macro*⟩}

本命令定义宏 ⟨*macro*⟩, 并把当前的变换矩阵的元素 (放在花括号里) 保存在宏 ⟨*macro*⟩ 中，之后可以用命令 \pgfsettransform 引用这个变换矩阵。

本命令的定义是：

```
\def\pgfgettransform#1{%
  \edef#1{{\pgf@pt@aa}{\pgf@pt@ab}{\pgf@pt@ba}{\pgf@pt@bb}{\the\pgf@pt@x}{
  ↪  \the\pgf@pt@y}}%
}
```

\pgfsettransform{⟨*macro*⟩}

⟨*macro*⟩ 是保存变换矩阵元素的宏。本命令调用保存在宏 ⟨*macro*⟩ 中的变换矩阵，并检查这个矩阵是否单位矩阵。

本命令的定义是：

```
\def\pgfsettransform#1{%
  \edef\pgf@temp{#1}%
  \expandafter\pgf@settransform\pgf@temp%
}
\def\pgf@settransform#1#2#3#4#5#6{%
  \def\pgf@pt@aa{#1}%
  \def\pgf@pt@ab{#2}%
  \def\pgf@pt@ba{#3}%
  \def\pgf@pt@bb{#4}%
  \pgf@pt@x=#5%
  \pgf@pt@y=#6%
  \edef\pgf@idtest{#1,#2,#3,#4}%
  \ifx\pgf@idtest\pgf@idmatrixtext%
    \pgf@pt@identitytrue%
  \else%
    \pgf@pt@identityfalse%
```

```
  \fi%
}
```

从定义看，宏 ⟨*macro*⟩ 的展开应当是这种形式：

```
{数值 a}{数值 b}{数值 c}{数值 d}{尺寸 x}{尺寸 y}
```

\pgfgettransformentries{⟨*macro for a*⟩}{⟨*macro for b*⟩}{⟨*macro for c*⟩}{⟨*macro for d*⟩}{⟨*macro for shift x*⟩}{⟨*macro for shift y*⟩}

本命令定义 6 个宏。设当前变换矩阵是 $\begin{pmatrix} a & b & s \\ c & d & t \\ 0 & 0 & 1 \end{pmatrix}$，本命令将 a, b, c, d, s, t 分别保存在这 6 个宏中。

\pgfsettransformentries{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}{⟨*d*⟩}{⟨*shiftx*⟩}{⟨*shifty*⟩}

本命令指定变换矩阵的 6 个元素，从而重定义变换矩阵，实际上等效于

```
\pgftransformreset
\pgftransformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{\pgfpoint{⟨shiftx⟩}{⟨shifty⟩}}
```

\pgftransformationadjustments

这个命令针对各种带有“scale”，“slant”的放缩、拉伸变换，但是最好只针对带有“scale”的放缩变换。当你使用了放缩变换后，你却可能希望路径命令中的某个特殊点不接受放缩变换，而你还要路径中的其它点仍然接受放缩变换，此时可以使用本命令。

本命令计算下面两个宏。

\pgfhorizontaltransformationadjustment

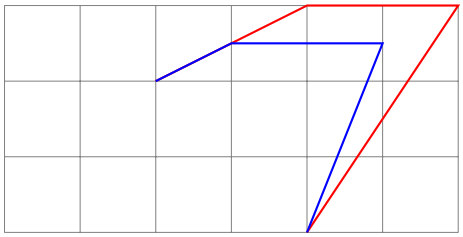
假设当前 canvas 坐标系的 x 轴的单位向量是 (a, b), 那么这个宏的值就是 $\frac{1}{\|(a,b)\|}$ (数值)。

这个宏的初始值等于 \pgf@one@text, 即数值 1.

\pgfverticaltransformationadjustment

假设当前 canvas 坐标系的 y 轴的单位向量是 (a, b), 那么这个宏的值就是 $\frac{1}{\|(a,b)\|}$ (数值)。

这个宏的初始值等于 \pgf@one@text, 即数值 1.



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (6,3);
  \begin{scope}[scale=2,thick]
    \draw [red] (1,1) -- ++(1,.5) -- ++(1,0) -- (2,0);
    \pgftransformationadjustments
    \draw [blue] (1,1) --
      ++(\pgfhorizontaltransformationadjustment,.5*\pgfverticaltransformationadjustment)
      -- ++(1,0) -- (2,0);
  \end{scope}
\end{tikzpicture}
```

13.1.2 对应 xyz 坐标系的矩阵

在文件《pgfcorepoints.code.tex》中定义了 3 个命令：

- \pgfsetxvec{⟨*point X*⟩}, 点 ⟨*point X*⟩ 是当前 canvas 坐标系中的点，它的坐标分量分别赋予寄存器 \pgf@xx, \pgf@xy

- \pgfsetyvec{⟨*point Y*⟩}, 点 ⟨*point Y*⟩ 是当前 canvas 坐标系中的点，它的坐标分量分别赋予寄存器 \pgf@yx, \pgf@yy
- \pgfsetzvec{⟨*point Z*⟩}, 点 ⟨*point Z*⟩ 是当前 canvas 坐标系中的点，它的坐标分量分别赋予寄存器 \pgf@zx, \pgf@zy

所以向量 v_x =(\pgf@xx,\pgf@xy), v_y =(\pgf@yx,\pgf@yy), v_z =(\pgf@zx,\pgf@zy) 是当前 canvas 坐标系中的向量，以这 3 个向量分别作为 x, y, z 轴的单位向量，以当前 canvas 坐标系的原点为原点，构成一个坐标系，这就是一个 xyz 坐标系。

文件《pgfcorepoints.code.tex》有下面的初始规定：

```
\pgfsetxvec{\pgfpoint{1cm}{0cm}}
\pgfsetyvec{\pgfpoint{0cm}{1cm}}
\pgfsetzvec{\pgfpoint{-0.385cm}{-0.385cm}}
```

PGF 的创建点的命令，例如 \pgfpointxy, \pgfpointpolarxy, 会利用 xyz 坐标系来变换点坐标。例如对于 $(x, y, z) \in$ 当前的 xyz 坐标系, 有变换

$$(x,y,z)\begin{bmatrix}\texttt{\textbackslash pgf@xx} & \texttt{\textbackslash pgf@xy}\\ \texttt{\textbackslash pgf@yx} & \texttt{\textbackslash pgf@yy}\\ \texttt{\textbackslash pgf@zx} & \texttt{\textbackslash pgf@zy}\end{bmatrix} = (\texttt{\textbackslash pgf@x}, \texttt{\textbackslash pgf@y}) \in \text{当前的 canvas 坐标系}$$

例如：

```
\pgfpathmoveto{\pgfpointxy{1}{0}}
```

会做 2 个变换：

- 首先 \pgfpointxy 会：当前的 xyz 坐标系 $\ni (1, 0) \to (a, b) \in$ 当前的 canvas 坐标系
- 之后 \pgfpathmoveto 会：当前的 canvas 坐标系 $\ni (a, b) \to (a', b') \in$ 不变坐标系

13.2 画布变换

PGF 的画布变换命令生成输出语言 (例如 PDF 或 PostScript) 的变换句法，然后交给编译引擎处理，PGF 自己并不计算画布变换，因此 (目前) PGF 不跟踪画布变换，当使用画布变换时，PGF 不再计算图形的边界盒子。

画布变换的有效范围受到 {pgfscope} 环境的限制，这是因为画布变换主要由后台驱动来实现，而不是由 TEX 或 PGF 来实现。

13.2.1 基本命令

\pgflowlevelsynccm

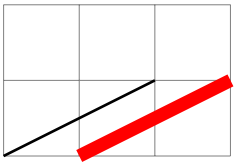
本命令利用当前线性变换矩阵的元素来声明一个画布变换；然后将当前线性变换矩阵改为单位矩阵；然后设置 \pgf@relevantforpicturesizefalse, 不再更新图形的边界盒子。

本命令的定义是：

```
\def\pgflowlevelsynccm{%
  \pgfsys@transformcm%
  {\pgf@pt@aa}{\pgf@pt@ab}%
  {\pgf@pt@ba}{\pgf@pt@bb}%
  {\pgf@pt@x}{\pgf@pt@y}%
  \pgftransformreset%
  \pgf@relevantforpicturesizefalse%
}
```

当输出语言为 PDF 时，文件《pgfsys-common-pdf.def》定义

```
\def\pgfsys@transformcm#1#2#3#4#5#6{%
  \pgfsysprotocol@literalbuffered{#1 #2 #3 #4}\pgf@sys@bp{#5}\pgf@sys@bp{#6}
  ↪  \pgfsysprotocol@literal{cm}}
```



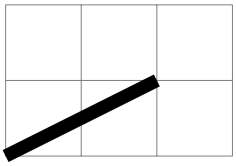
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgftransformscale{5}
  \draw (0,0) -- (0.4,.2);
  \pgftransformxshift{0.2cm}
  \pgflowlevelsynccm
  \draw[red] (0,0) -- (0.4,.2);
\end{tikzpicture}
```

\pgflowlevel{⟨*transformation code*⟩}

⟨*transformation code*⟩ 是线性变换命令。本命令在一个组内：先将线性变换矩阵改为单位矩阵，再执行 ⟨*transformation code*⟩ 得到一个线性变换矩阵，再将这个线性变换矩阵转成画布变换矩阵。由于组的限制，当前的线性变换矩阵不会被本命令改变。不过本命令会设置 \pgf@relevantforpicturesizefalse，不再更新图形的边界盒子。

本命令的定义是：

```
\def\pgflowlevel#1{%
  {%
    \pgftransformreset%
    #1%
    \pgflowlevelsynccm%
  }%
  \pgf@relevantforpicturesizefalse%
}
```



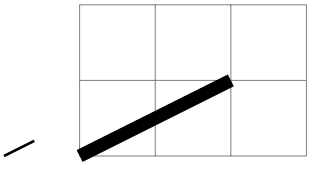
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgflowlevel{\pgftransformscale{5}}
  \draw (0,0) -- (0.4,.2);
\end{tikzpicture}
```

\pgflowlevelobj{⟨*transformation code*⟩}{⟨*code*⟩}

⟨*transformation code*⟩ 是坐标变换命令，⟨*code*⟩ 是绘图命令。本命令创建一个 {pgfscope} 环境，在这个环境中套嵌一个组，在组中先用命令 \pgflowlevel 来处理 ⟨*transformation code*⟩，然后执行 ⟨*code*⟩。

本命令的定义是：

```
\long\def\pgflowlevelobj#1#2{\pgfscope{\pgflowlevel{#1}#2}\endpgfscope}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgflowlevelobj{\pgftransformscale{5}}
    {\draw (0,0) -- (0.4,.2);}
  \pgflowlevelobj{\pgftransformxshift{-1cm}}
    {\draw (0,0) -- (0.4,.2);}
\end{tikzpicture}
```

\begin{pgflowlevelscope}{⟨*transformation code*⟩}

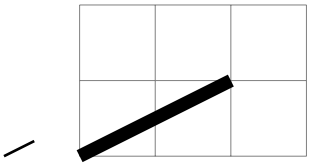
⟨*environment content*⟩

\end{pgflowlevelscope}

这个环境创建一个 {pgfscope} 环境，在环境内，先调用命令 \pgflowlevel 来处理 ⟨*transformation code*⟩, 然后执行 ⟨*environment contents*⟩.

本环境的定义是：

```
\def\pgflowlevelscope#1{\pgfscope\pgflowlevel{#1}}
\def\endpgflowlevelscope{\endpgfscope}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \begin{pgflowlevelscope}{\pgftransformscale{5}}
    \draw (0,0) -- (0.4,.2);
  \end{pgflowlevelscope}
  \begin{pgflowlevelscope}{\pgftransformxshift{-1cm}}
    \draw (0,0) -- (0.4,.2);
  \end{pgflowlevelscope}
\end{tikzpicture}
```

```
\pgflowlevelscope{⟨transformation code⟩}
  ⟨environment contents⟩
\endpgflowlevelscope
```

这是 Plain TeX 中的用法。

```
\startpgflowlevelscope{⟨transformation code⟩}
  ⟨environment contents⟩
\stoppgflowlevelscope
```

这是 ConTeXt 中的用法。

13.2.2 创建 View Boxes

环境 {pgfviewboxscope} 能利用画布变换创建一个 view box, 关于 view box 的设计思路参考 views 程序库。

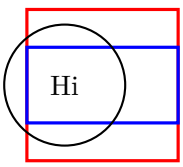
\begin{pgfviewboxscope}{⟨ll_1⟩}{⟨ur_1⟩}{⟨ll_2⟩}{⟨ur_2⟩}{⟨*meet or slice*⟩}
  ⟨*environment content*⟩
\end{pgfviewboxscope}

⟨ll_1⟩ 与 ⟨ur_1⟩ 对应 ⟨*to-be-viewed corner*⟩ rectangle ⟨*to-be-viewed corner*⟩.

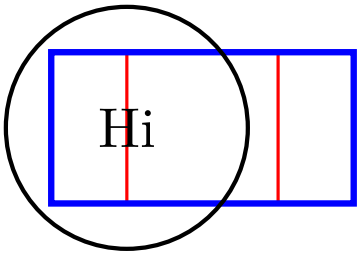
⟨ll_2⟩ 与 ⟨ur_2⟩ 对应 ⟨*window corner*⟩ rectangle ⟨*window corner*⟩.

⟨*meet or slice*⟩ 是选项 meet 或 slice.

本环境先调用 \pgfsys@viewboxmeet 或者 \pgfsys@viewboxslice 设置画布变换,再设置 \pgf@relevantforpict
再处理环境内容。




```
\tikz {
  \draw [red, very thick] (0,0) rectangle (20mm,20mm);
  \begin{pgfviewboxscope}
    {\pgfpoint{5mm}{5mm}}{\pgfpoint{25mm}{15mm}} % Source
    {\pgfpoint{0mm}{0mm}}{\pgfpoint{20mm}{20mm}} % Target
    {meet}
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [radius=8mm] node {Hi};
  \end{pgfviewboxscope} }
```



```
\tikz {
  \draw [red, very thick] (0,0) rectangle (20mm,20mm);
  \begin{pgfviewboxscope}
    {\pgfpoint{5mm}{5mm}}{\pgfpoint{25mm}{15mm}} % Source
    {\pgfpoint{0mm}{0mm}}{\pgfpoint{20mm}{20mm}} % Target
    {slice}
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [radius=8mm] node {Hi};
  \end{pgfviewboxscope} }
```

第十四章　创建路径

文件《pgfcorepathconstruct.code.tex》。

一个路径由 move-to(\pgfpathmoveto), line-to(\pgfpathlineto), curve-to(\pgfpathcurveto), close path(\pgfpathclose) 四种基本构造命令创建，这些命令的参数通常是 PGF 点，是路径的“构造点”。路径这个概念本身不包含线型、线宽、颜色、点标记类型等等内容。创建路径的命令会跟踪两个“边界盒子”，一个是当前路径的边界盒子，另一个是所有路径（即整个当前图形）的边界盒子。

“当前路径”是一个全局概念，不受 TEX 组的限制。因此在构建一个路径的过程中可以插入 TEX 组来完成某些工作。当遇到命令 \pgfusepath 时当前路径才会结束。

创建路径的命令都会使用命令 \pgfpointtransformed→P. 255, 将当前的线性变换矩阵作用于点坐标；如果还启用了非线性变换模块 nonlineartransformations, 并引入了非线性变换代码 (参考 \pgftransformnonlinear)，那么还会再用非线性变换来处理点坐标。

14.1　基本的构造命令

\pgf@path@lastx

这是个尺寸寄存器，一般情况下，当构造路径的命令处理一个构造点后，会把这个点的 x 坐标保存到这个寄存器中，代表“当前点的 x 坐标”。

\pgf@path@lasty

这是个尺寸寄存器，一般情况下，当构造路径的命令处理一个构造点后，会把这个点的 y 坐标保存到这个寄存器中，代表“当前点的 y 坐标”。

\pgfgetpath⟨*macro*⟩

本命令将当前的软路径保存到宏 ⟨*macro*⟩.

```
\let\pgfgetpath=\pgfsyssoftpath@getcurrentpath
```

\pgfsetpath⟨*soft path macro*⟩

本命令将保存在宏 ⟨*soft path macro*⟩ 中的软路径用作当前的软路径。

```
\let\pgfsetpath=\pgfsyssoftpath@setcurrentpath
```

14.1.1　move-to

Move-To 操作将当前点移动到指定位置，有的路径命令自带 move-to 功能，例如 \pgfpathrectangle, \pgfpathcircle. 路径应当以 move-to 操作开头。

\pgf@lt@moveto{⟨*dimension register 1*⟩}{⟨*dimension register 2*⟩}

本命令的参数是尺寸寄存器。这个命令代表“线性变换”的 move-to.

```
\def\pgf@lt@moveto#1#2{%
  \pgf@protocolsizes{#1}{#2}%
  \pgfsyssoftpath@moveto{\the#1}{\the#2}%
}
```

\pgf@nlt@moveto{⟨*dimension register 1*⟩}{⟨*dimension register 2*⟩}

本命令的参数是尺寸寄存器。这个命令代表“非线性变换”的 move-to, 不过其初始状态等于 \pgf@lt@moveto. 如果使用非线性变换，则 \pgf@nlt@moveto 会被重定义，参考 \pgftransformnonlinear.

```
\let\pgf@nlt@moveto\pgf@lt@moveto
```

\pgfpathmoveto{⟨*coordinate*⟩}

参数 ⟨*coordinate*⟩ 是 PGF 点。

本命令将当前点移动到 ⟨*coordinate*⟩ 位置，移动时不画线，其中 ⟨*coordinate*⟩ 可以使用 \pgfpoint 等命令指定。一般情况下，在路径的开头应当使用这个命令来开启一个路径创建过程，例如从原点开始创建路径：

```
\pgfpathmoveto{\pgfpointorigin}
```

如果当前路径为“空”，使用该命令可以开启当前路径的构造。如果在一个路径之中使用该命令，会把路径分为不相连的数个部分（子路径）。

本命令的定义是：

```
\def\pgfpathmoveto#1{%
  \pgfpointtransformed{#1}%
  \pgf@nlt@moveto{\pgf@x}{\pgf@y}%
  \global\pgf@path@lastx=\pgf@x%
  \global\pgf@path@lasty=\pgf@y%
}
```

本命令的处理是：

1. 执行 \pgfpointtransformed→P. 255 对参数 ⟨*coordinate*⟩ 做线性变换，变换结果 (两个尺寸) 全局地保存到寄存器 \pgf@x 和 \pgf@y 中。
2. 执行 \pgf@nlt@moveto, 它的默认处理是：
   (a) 用 \pgf@protocolsizes 刷新图形 (如若需要)、当前路径的边界盒子。
   (b) 执行 \pgfsyssoftpath@moveto 延伸软路径
   如果使用非线性变换，则 \pgf@nlt@moveto 会被重定义，参考 \pgftransformnonlinear.
3. 将当前点的坐标全局地保存到寄存器 \pgf@path@lastx 和 \pgf@path@lasty 中。

14.1.2 Line-To 路径操作

\pgf@lt@lineto{⟨*dimension register 1*⟩}{⟨*dimension register 2*⟩}

本命令的参数是尺寸寄存器。这个命令代表“线性变换”的 line-to.

```
\def\pgf@lt@lineto#1#2{%
  \pgf@protocolsizes{#1}{#2}%
  \pgfsyssoftpath@lineto{\the#1}{\the#2}%
}
```

\pgf@nlt@lineto{⟨*dimension register 1*⟩}{⟨*dimension register 2*⟩}

本命令的参数是尺寸寄存器。这个命令代表“非线性变换”的 line-to, 不过其初始状态等于 \pgf@lt@lineto. 如果使用非线性变换，则本命令会被重定义，参考 \pgftransformnonlinear.

```
\let\pgf@nlt@lineto\pgf@lt@lineto
```

\pgfpathlineto{⟨*coordinate*⟩}

这个命令将当前坐标点移动到 ⟨*coordinate*⟩ 位置，移动时画线，延伸当前路径。

注意本命令不用于开启一个路径（即不作为一个路径的开头），如果以该命令开头来创建一个路径 (而不以 move-to 操作开头)，那么编译器可能不会认为这是个错误，但有的阅读器在渲染图形时，可能对此产生一个错误信息。以本命令开头构建的“路径”不是真正的路径，但是这个假“路径”会被算入边界盒子内。

```
\begin{pgfpicture}
  \pgftransformrotate{-90}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

本命令的定义是：

```
\def\pgfpathlineto#1{%
  \pgfpointtransformed{#1}%
  \pgf@roundcornerifneeded%
  \pgf@nlt@lineto{\pgf@x}{\pgf@y}%
  \global\pgf@path@lastx=\pgf@x%
  \global\pgf@path@lasty=\pgf@y%
}
```

本命令的处理是：

1. 执行 \pgfpointtransformed → P. 255 对参数 ⟨*coordinate*⟩ 做线性变换，变换结果 (两个尺寸) 全局地保存到寄存器 \pgf@x 和 \pgf@y 中。
2. 执行 \pgf@roundcornerifneeded, 即添加圆角参数。
3. 执行 \pgf@nlt@lineto, 它的默认处理是：
   (a) 用 \pgf@protocolsizes 刷新图形 (如若需要)、当前路径的边界盒子。
   (b) 执行 \pgfsyssoftpath@lineto 延伸软路径
   如果使用非线性变换，则 \pgf@nlt@lineto 会被重定义，参考 \pgftransformnonlinear.
4. 将当前点的坐标全局地保存到寄存器 \pgf@path@lastx 和 \pgf@path@lasty 中。

14.1.3 Curve-To 路径操作

\pgf@lt@curveto{⟨*dimension register 1*⟩}{⟨*dimension register 2*⟩}{⟨*dimension register 3*⟩}{⟨*dimension register 4*⟩}{⟨*dimension register 5*⟩}{⟨*dimension register 6*⟩}

本命令的参数是尺寸寄存器。这个命令代表“线性变换”的 curve-to.

```
\def\pgf@lt@curveto#1#2#3#4#5#6{%
  \pgf@protocolsizes{#1}{#2}%
  \pgf@protocolsizes{#3}{#4}%
  \pgf@protocolsizes{#5}{#6}%
  \pgfsyssoftpath@curveto{\the#1}{\the#2}{\the#3}{\the#4}{\the#5}{\the#6}%
}
```

\pgf@nlt@curveto{⟨*dimension register 1*⟩}{⟨*dimension register 2*⟩}{⟨*dimension register 3*⟩}{⟨*dimension register 4*⟩}{⟨*dimension register 5*⟩}{⟨*dimension register 6*⟩}

本命令的参数是尺寸寄存器。这个命令代表“非线性变换”的 curve-to, 不过其初始状态等于 \pgf@lt@curveto.

如果使用非线性变换，则本命令会被重定义，参考 \pgftransformnonlinear.

```
\let\pgf@nlt@curveto\pgf@lt@curveto
```

\pgfpathcurveto{⟨*support 1*⟩}{⟨*support 2*⟩}{⟨*coordinate*⟩}

本命令的参数是 PGF 点。

本命令创建一段控制曲线来延伸当前路径。以本命令之前最近出现的点 (当前点) 为始点，以 ⟨*coordinate*⟩ 为终点，以 ⟨*support 1*⟩ 和 ⟨*support 2*⟩ 分别为第一和第二控制点，构建一段控制曲线。与 line-to 命令一样，本命令一般不用于路径开头。

本命令会先将当前的坐标变换矩阵用于 ⟨*coordinate*⟩, ⟨*support 1*⟩ 和 ⟨*support 2*⟩, 然后利用变换后的坐标来构建控制曲线。

一般情况下，本命令会刷新当前路径和图形的边界盒子，注意边界盒子会把代码中出现的起点、终点、控制点都包括在内，这可能会导致边界盒子的边界与控制曲线之间的距离过大。

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto
    {\pgfpoint{1cm}{1cm}}{\pgfpoint{2cm}{1cm}}
    {\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

本命令的定义是：

```
\def\pgfpathcurveto#1#2#3{%
  \pgfpointtransformed{#3}%
  \pgf@xb=\pgf@x%
  \pgf@yb=\pgf@y%
  \pgfpointtransformed{#2}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgfpointtransformed{#1}%
  \pgf@roundcornerifneeded%
  \pgf@nlt@curveto{\pgf@x}{\pgf@y}{\pgf@xa}{\pgf@ya}{\pgf@xb}{\pgf@yb}%
  \global\pgf@path@lastx=\pgf@xb%
  \global\pgf@path@lasty=\pgf@yb%
}
```

本命令的处理是：

1. 执行 \pgfpointtransformed →P. 255 对参数做线性变换，终点 ⟨*coordinate*⟩ 的变换结果 (两个尺寸) 被全局地保存到寄存器 \pgf@x 和 \pgf@y 中。
2. 执行 \pgf@roundcornerifneeded, 即添加圆角参数。
3. 执行 \pgf@nlt@curveto, 它的默认处理是：
   (a) 用 \pgf@protocolsizes 刷新图形 (如若需要)、当前路径的边界盒子。
   (b) 执行 \pgfsyssoftpath@curveto 延伸软路径
   如果使用非线性变换，则 \pgf@nlt@curveto 会被重定义，参考 \pgftransformnonlinear.
4. 将终点 ⟨*coordinate*⟩ 的变换结果 (两个尺寸) 全局地保存到寄存器 \pgf@path@lastx 和 \pgf@path@lasty 中。

\pgfpathquadraticcurveto{⟨*support*⟩}{⟨*coordinate*⟩}

本命令的参数是 PGF 点。

本命令的处理是：

1. 执行 \pgfpointtransformed →P. 255 对参数做线性变换，⟨*support*⟩ 的变换结果 (两个尺寸) 被全局地保存到寄存器 \pgf@x 和 \pgf@y 中。

记当前点 (\pgf@path@lastx, \pgf@path@lasty) = P_1, ⟨*support*⟩ 的变换结果是 S, ⟨*coordinate*⟩ 的变换结果是 P_4.

2. 计算 $P_2 = P_1 + \frac{2}{3}(S - P_1)$, $P_3 = P_4 + \frac{2}{3}(S - P_4)$.
3. 执行 \pgf@roundcornerifneeded, 即添加圆角参数。
4. 执行 \pgf@nlt@curveto, 它的默认处理是：
   (a) 用 \pgf@protocolsizes 刷新图形 (如若需要)、当前路径的边界盒子。
   (b) 执行 \pgfsyssoftpath@curveto 延伸软路径
   如果使用非线性变换，则 \pgf@nlt@curveto 会被重定义，参考 \pgftransformnonlinear.
5. 将终点 ⟨*coordinate*⟩ 的变换结果 (两个尺寸) 全局地保存到寄存器 \pgf@path@lastx 和 \pgf@path@lasty 中。

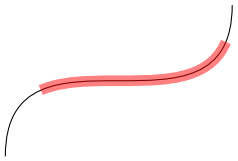
注意执行本命令后的边界盒子未必把点 S 包括在内。

\pgfpathcurvebetweentime{⟨*time* t_1⟩}{⟨*time* t_2⟩}{⟨*point* p⟩}{⟨*point* s_1⟩}{⟨*point* s_2⟩}{⟨*point* q⟩}

设想一个控制曲线 $C(t)$, $t \in [0,1]$, 以 ⟨*point* p⟩ 为始点，以 ⟨*point* q⟩ 为终点，以 ⟨*point* s_1⟩ 和 ⟨*point* s_2⟩ 分别为第一和第二控制点。

本命令构建的是 $C(t)$, $t \in [t_1, t_2]$ 这一部分曲线。

本命令将 move-to 操作作为自己的开头，即使得“画笔”移动到始点 ⟨*point* p⟩, 因此本命令可以用在路径的开端处。



```
\begin{tikzpicture}
  \draw [thin] (0,0) .. controls (0,2) and (3,0) .. (3,2);
  \pgfpathcurvebetweentime{0.25}{0.9}
    {\pgfpointxy{0}{0}}{\pgfpointxy{0}{2}}
    {\pgfpointxy{3}{0}}{\pgfpointxy{3}{2}}
  \pgfsetstrokecolor{red}
  \pgfsetstrokeopacity{0.5}
  \pgfsetlinewidth{4pt}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfpathcurvebetweentimecontinue{⟨*time* t_1⟩}{⟨*time* t_2⟩}{⟨*point* p⟩}{⟨*point* s_1⟩}{⟨*point* s_2⟩}{⟨*point* q⟩}

类似\pgfpathcurvebetweentime, 只是本命令不把 move-to 操作作为自己的开头。

14.1.4 Close 路径操作

\pgf@lt@closepath

这个命令代表“线性变换”的 closepath.

```
\let\pgf@lt@closepath\pgfsyssoftpath@closepath
```

\pgf@nlt@closepath

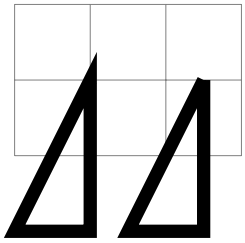
这个命令代表“非线性变换”的 closepath, 不过其初始状态等于 \pgf@lt@closepath. 如果使用非线性变换，本命令会被重定义，参考 \pgftransformnonlinear.

```
\let\pgf@nlt@closepath\pgf@lt@closepath
```

\pgfpathclose

这个命令用在路径的某个子路径之后，在该子路径的起止点之间画一个直线段，将这个子路径作成闭合的（即首尾相接），并且首尾连接处看起来比较自然。

注意起止点重合不等于“闭合”，二者在外观上有明显的不同，观察下面的例子：



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{5pt}
  \pgfpathmoveto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{0cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{-1cm}}
  \pgfpathclose  % 将前面的子路径作成闭合的
  \pgfpathmoveto{\pgfpoint{2.5cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1.5cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{2.5cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{2.5cm}{1cm}}
  ↪  % 这一段子路径首尾相接，但不“闭合”
  \pgfusepath{stroke}  % 画出路径
\end{tikzpicture}
```

本命令的定义是：

```
\def\pgfpathclose{%
  \pgf@roundcornerifneeded%
  \pgf@nlt@closepath%
}
```

本命令的处理是：

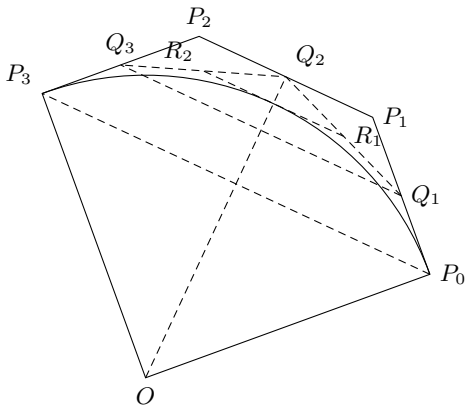
1. 执行 \pgf@roundcornerifneeded, 即添加圆角参数
2. 执行软路径操作 \pgfsyssoftpath@closepath, 此命令导致

```
\pgfsyssoftpath@addtocurrentpath{%
  \pgfsyssoftpath@closepathtoken⟨ 展开的 \pgfsyssoftpath@lastmoveto 坐标⟩}%
```

猜测上面命令的意思是，把一段“子路径”添加到当前的软路径中，这段子路径把当前点与最近出现的 moveto 点联系起来，并且封闭起来。

14.2 圆，椭圆，弧

设 $0 < \varphi \leqslant \frac{\pi}{2}$, 考虑单位圆上从点 $P_0 = (\cos\alpha, \sin\alpha)^T$ 到点 $P_3 = (\cos(\alpha+\varphi), \sin(\alpha+\varphi))^T$ 的一段圆弧 S. 现在用一段 3 次 Bézier 曲线来近似圆弧 S. 控制曲线的控制点是 P_0, P_1, P_2, P_3, 在最好的近似情况下，这 4 个点的位置应当如下图所示：



其中 Q_1 是 P_0P_1 的中点，R_1 是 Q_1Q_2 的中点……可以计算出 P_1, P_2 的坐标是：

$$P_1 = \frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot\begin{pmatrix}-\sin\alpha\\ \cos\alpha\end{pmatrix} + \begin{pmatrix}\cos\alpha\\ \sin\alpha\end{pmatrix},$$

$$P_2 = \frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot\begin{pmatrix}\sin(\alpha+\varphi)\\ -\cos(\alpha+\varphi)\end{pmatrix} + \begin{pmatrix}\cos(\alpha+\varphi)\\ \sin(\alpha+\varphi)\end{pmatrix}.$$

当 $\varphi=\frac{\pi}{2}$ 时，记系数 $t=\frac{4}{3}\cdot\tan(\frac{\varphi}{4})=\frac{4}{3}(\sqrt{2}-1)\approx 0.5522847498308$, 上述公式是：

$$P_0=(\cos\alpha,\sin\alpha)^T,\ P_3=(\cos(\alpha+\frac{\pi}{2}),\sin(\alpha+\frac{\pi}{2}))^T,$$
$$P_1=t\cdot P_3+P_0,\qquad P_2=t\cdot P_0+P_3.$$

当 $\varphi=-\frac{\pi}{2}$ 时，记系数 $-t=-\frac{4}{3}\cdot\tan(\frac{\varphi}{4})=-\frac{4}{3}(\sqrt{2}-1)\approx -0.5522847498308$, 上述公式形式不变。

上面公式是针对单位圆的，容易变化到下面两个情况：

(i) 对于圆心在原点，半径为 r 的圆，上述公式就是：

$$P_1=\frac{4}{3}\tan(\frac{\varphi}{4})\cdot r\cdot\begin{pmatrix}-\sin\alpha\\ \cos\alpha\end{pmatrix}+r\cdot\begin{pmatrix}\cos\alpha\\ \sin\alpha\end{pmatrix},$$
$$P_2=\frac{4}{3}\tan(\frac{\varphi}{4})\cdot r\cdot\begin{pmatrix}\sin(\alpha+\varphi)\\ -\cos(\alpha+\varphi)\end{pmatrix}+r\cdot\begin{pmatrix}\cos(\alpha+\varphi)\\ \sin(\alpha+\varphi)\end{pmatrix}.$$

记 $(r\cos\theta,r\sin\theta)^T=(\theta:r)$, 上述公式就是：

$$\begin{aligned}P_0&=(\alpha:r),\\ P_1&=\frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot(\alpha+\frac{\pi}{2}:r)+(\alpha:r),\\ P_2&=\frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot(\alpha+\varphi-\frac{\pi}{2}:r)+(\alpha+\varphi:r),\\ P_3&=(\alpha+\varphi:r).\end{aligned}$$

(ii) 对于椭圆 $\begin{pmatrix}a\cos\theta\\ b\sin\theta\end{pmatrix}$, 只需要对上面公式中的点做个仿射变换：

$$(r\cos\theta,r\sin\theta)^T\to(a\cos\theta,b\sin\theta)^T=(\theta:a\ \&\ b),$$

上述公式变成：

$$\begin{aligned}P_0&=(\alpha:a\ \&\ b),\\ P_1&=\frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot(\alpha+\frac{\pi}{2}:a\ \&\ b)+(\alpha:a\ \&\ b),\\ P_2&=\frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot(\alpha+\varphi-\frac{\pi}{2}:a\ \&\ b)+(\alpha+\varphi:a\ \&\ b),\\ P_3&=(\alpha+\varphi:a\ \&\ b).\end{aligned}$$

注意这个变换保持圆 (椭圆) 的横轴与纵轴相正交。

\pgfpatharc{⟨*start angle*⟩}{⟨*end angle*⟩}{⟨*radius*⟩ and ⟨*y-radius*⟩}

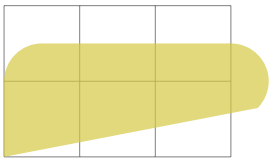
这个命令以当前坐标点为始点构建一段圆弧或者椭圆弧。and ⟨*y-radius*⟩ 这一部分是可选的。如果只给出 ⟨*radius*⟩ 则指示这是圆弧半径。如果还给出 and ⟨*y-radius*⟩, 则指示构建一段椭圆弧。

假设一个具有标准形式 $\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$ 的椭圆，椭圆的横半轴长度是 ⟨*radius*⟩, 纵半轴长度是 ⟨*y-radius*⟩, 在这个椭圆上取一段弧，弧的起点向量的方向角度是 ⟨*start angle*⟩, 弧的终点向量的方向角度是 ⟨*end angle*⟩. 这里规定：如果 ⟨*start angle*⟩ 小于 ⟨*end angle*⟩, 则按逆时针方向选取弧；如果 ⟨*start angle*⟩ 大于 ⟨*end angle*⟩, 则按顺时针方向选取弧。然后将当前的线性变换矩阵作用于这段弧，然后再做平移，使得弧的起点与当前坐标点重合，从而将弧添加到当前路径上。这就是本命令的作用。

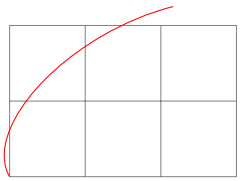
由于本命令需要一个提前给出的点作为弧的始点，所以本命令不能用作路径的开头，也不用作一个孤立的子路径的开头。

注意命令 `\pgfpatharc{0}{360}{1cm}` 给出的不是一个完整的圆，因为它不闭合，在它后面跟上命令 `\pgfpathclose` 使之闭合。

本命令会刷新边界盒子。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0cm}{1cm}}
  \pgfpatharc{180}{90}{.5cm}
  \pgfpathlineto{\pgfpoint{3cm}{1.5cm}}
  \pgfpatharc{90}{-45}{.5cm}
  \pgfsetfillcolor{yellow!80!black}
  \pgfsetfillopacity{0.7}
  \pgfusepath{fill}
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{180}{60}{2cm and 1cm}
  \pgfsetstrokecolor{red}
  \pgfusepath{draw}
\end{tikzpicture}
```

本命令的处理是：

1. 用 \pgfmathparse 解析 ⟨*start angle*⟩, ⟨*end angle*⟩, 结果作为起止角度。
2. 检查它的第三个参数是 ⟨*radius*⟩ 还是 ⟨*radius*⟩ and ⟨*y-radius*⟩. 如果是 ⟨*radius*⟩, 就变成 ⟨*radius*⟩ and ⟨*radius*⟩ 来处理，也就是说，总是针对椭圆的情况处理。参数 ⟨*radius*⟩, ⟨*y-radius*⟩ 会被 \pgfmathparse 解析, 结果作为椭圆的半轴长度。
3. 执行一个 \loop...\repeat 循环，这个循环对一个给定的角度区间做分解处理。
   循环处理 记当前的角度区间是 $[d_0, d_1]$, 检查角度差 $d_1 - d_0 = \delta$, 然后分三个情况处理：
   - 如果 $\delta \le 90$, 则退出循环。
   - 如果 $90 < \delta \le 115$, 则做分解 $[d_0, d_1] = [d_0, d_0 + 60] \cup [d_0 + 60, d_1]$.
     然后先对第一个区间 $[d_0, d_0 + 60]$ 执行命令 \pgf@arc, 此命令会在这个角度区间内画一段控制曲线来近似椭圆弧。
     然后回到“循环处理”，令 $d_0 = d_0 + 60$, $d_1 = d_1$, 重复循环。
   - 如果 $115 \le \delta$, 则做分解 $[d_0, d_1] = [d_0, d_0 + 90] \cup [d_0 + 90, d_1]$.
     然后先对第一个区间 $[d_0, d_0 + 90]$ 执行命令 \pgf@arc, 此命令会在这个角度区间内画一段控制曲线来近似椭圆弧。
     然后回到“循环处理”，令 $d_0 = d_0 + 90$, $d_1 = d_1$, 重复循环；如此循环，最终由 $\delta \le 90$ 退出循环。

   记 ⟨*start angle*⟩ $= \alpha$, ⟨*end angle*⟩ $= \alpha + \varphi$, 下面为了简便，约定 $\varphi > 0$. 上述循环处理的第一个角度区间就是 $[\alpha, \alpha + \varphi]$.
4. 在上述循环中，如果角度差 $\delta \le 90$, 则退出循环，再执行

```
\pgf@roundcornerifneeded%
\pgf@arc%
```

   命令 \pgf@arc 画一段画控制曲线，在当前区间之内近似椭圆弧。如果在执行 \pgfpatharc 之前执行了 \pgfsetcornersarced, 那么命令 \pgf@roundcornerifneeded 就会起作用。

\pgf@arc

这个命令依照前面的公式计算控制点，从而构造一段控制曲线。前述公式中的椭圆是标准椭圆：以原点为中心，长短半轴与坐标轴共线。在一般情况下需要对这个椭圆做一个仿射变换。

本命令的处理是：

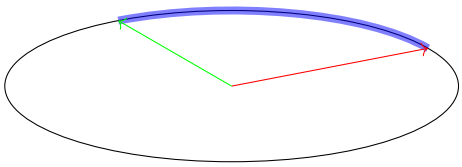
1. 依照前述公式，针对标准椭圆获取控制点 P_0, P_1, P_2, P_3;
2. 用当前的线性变换矩阵将控制点分别变成 P_0', P_1', P_2', P_3', 然后对这 4 个点做平移，得到点 P_0'', P_1'', P_2'', P_3'', 这个平移使得 P_0'' 的坐标是当前点的坐标 `\pgf@path@lastx`, `\pgf@path@lasty`(然后再对这些点做非线性变换，如果有非线性变换的话);
3. 以 P_0'', P_1'', P_2'', P_3'' 为控制点构造一段曲线 (执行`\pgf@nlt@curveto`);
4. 然后将 P_3'' 的坐标 (两个尺寸) 全局地保存到 `\pgf@path@lastx`, `\pgf@path@lasty` 中。

当然在上面的步骤中，本命令不对点 P_0, P_0', P_0'' 做计算，因为 P_0'' 是执行本命令时面临的当前点 `\pgf@path@lastx`, `\pgf@path@lasty`.

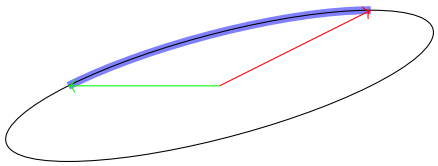
总结起来，当 ⟨*start angle*⟩ 与 ⟨*end angle*⟩ 相差较大时，命令 `\pgfpatharc` 画的弧线是由数段控制曲线 $C_1, \ldots, C_n$ 连结而成的。假如当前的线性变换矩阵 M 是正交矩阵，任意一段段控制曲线 C_i 的起止角度差有以下几种情况：

- 角度差 $\delta \in [0, 90]$
- 角度差 $\delta \in [-90, 0]$
- 角度差 $\delta \in [0, 60]$
- 角度差 $\delta \in [-60, 0]$

但如果当前的线性变换矩阵 M 不是正交矩阵，例如：



```
\begin{tikzpicture}
  \draw circle [x radius=3cm, y radius=1cm];
  \pgftransformxscale{3}
  \pgftransformrotate{30}
  \pgfscope
    \pgfpathmoveto{\pgfpoint{1cm}{0pt}}
    \pgfpatharc{0}{90}{1cm and 1cm}
    \pgfsetlinewidth{3pt}
    \pgfsetstrokeopacity{0.5}
    \pgfsetstrokecolor{blue}
    \pgfusepath{draw}
  \endpgfscope
  \draw [red,->](0,0)--(1,0);
  \draw [green,->](0,0)--(0,1);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \pgftransformcm{2}{1}{-2}{0}{0pt}{0pt}
  \pgfscope
    \pgfpathmoveto{\pgfpoint{1cm}{0pt}}
    \pgfpatharc{0}{90}{1cm and 1cm}
    \pgfsetlinewidth{3pt}
    \pgfsetstrokeopacity{0.5}
    \pgfsetstrokecolor{blue}
    \pgfusepath{draw}
  \endpgfscope
  \draw circle [x radius=1cm, y radius=1cm];
  \draw [red,->](0,0)--(1,0);
```

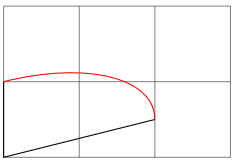
```
  \draw [green,->](0,0)--(0,1);
\end{tikzpicture}
```

\pgfpatharcaxes{⟨*start angle*⟩}{⟨*end angle*⟩}{⟨*first axis*⟩}{⟨*second axis*⟩}

本命令的定义是：

```
\def\pgfpatharcaxes#1#2#3#4{%
  {%
    \pgftransformtriangle{\pgfpointorigin}{#3}{#4}%
    \pgfpatharc{#1}{#2}{1pt}%
  }%
}
```

本命令先用 \pgftransformtriangle→P. 266 修改 canvas 坐标系，再用 \pgfpatharc 画弧。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2cm,5mm) (0,0) -- (0cm,1cm);
  \pgfpathmoveto{\pgfpoint{2cm}{5mm}}
  \pgfpatharcaxes{0}{90}{\pgfpoint{2cm}{5mm}}
                        {\pgfpoint{0cm}{1cm}}
  \pgfsetstrokecolor{red}
  \pgfusepath{draw}
\end{tikzpicture}
```

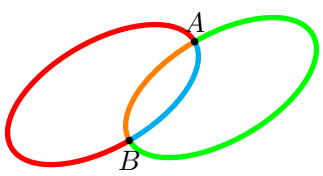
\pgfpatharcto{⟨*x-radius*⟩}{⟨*y-radius*⟩}{⟨*rotation*⟩}{⟨*large arc flag*⟩}{⟨*counterclockwise flag*⟩}{⟨*target point*⟩}

本命令的参数 ⟨*x-radius*⟩, ⟨*y-radius*⟩, ⟨*rotation*⟩, ⟨*large arc flag*⟩, ⟨*counterclockwise flag*⟩ 都会被命令 \pgfmathparse 处理，而参数 ⟨*target point*⟩ 会被命令 \pgfpointtransformed 处理。

这个命令以当前的坐标点为始点，构建一段椭圆弧，延伸当前路径。这里 ⟨*x-radius*⟩ 与 ⟨*y-radius*⟩ 都是带单位的尺寸，如果不带单位就默认单位为 pt; ⟨*rotation*⟩ 是个代表角度的数值，⟨*large arc flag*⟩ 与 ⟨*counterclockwise flag*⟩ 是整数，⟨*target point*⟩ 是个 PGF 点。

假设一个标准形式的椭圆 $\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$，它的横半轴长度 a 等于尺寸 ⟨*x-radius*⟩, 纵半轴长度 b 等于尺寸 ⟨*y-radius*⟩. 给定两个点 A, B, 点 A 是当前的坐标点，点 B 是 ⟨*target point*⟩.

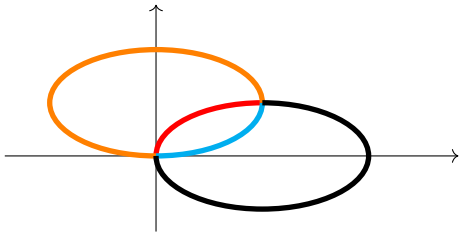
令 θ 等于角度 ⟨*rotation*⟩, 将椭圆旋转 θ 角度，然后再做一个平移变换，使得变换后的椭圆经过点 A, B. 如果能满足这个要求，那么一般会有两个 (形状一样的) 椭圆满足这个要求 (两个椭圆可能重合，但仍然看作是两个)。下图展示了这样的两个椭圆：



如上图所示，如果要在点 A, B 之间选择一段椭圆弧，那么就有 4 种选择。

本命令所确定的椭圆弧就是以上 4 种之一。如果参数 ⟨*large arc flag*⟩ 是非 0 整数（0 代表 false，非 0 值代表 true），则从当前坐标点到点 ⟨*target point*⟩ 的椭圆弧所张开的角度不小于 |180°|(加绝对值符号表示概括顺时针与逆时针两种角度方向). 如果参数 ⟨*counterclockwise flag*⟩ 是非 0 整数（0 代表 false，非 0 值代表 true），则从当前坐标点到点 ⟨*target point*⟩ 的椭圆弧是逆时针方向的。

本命令利用椭圆弧所张开的角度（绝对值）和椭圆弧的方向，从 4 个椭圆弧中选取一个弧。



```
\begin{tikzpicture}[scale=2]
  \draw[->] (-1,0) -- (2,0);
  \draw[->] (0,-.5) -- (0,1);
  \pgfsetlinewidth{2pt}
% Flags 0 0: 红色
  \pgfsetstrokecolor{red}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{0}{0}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
% Flags 0 1: 青色
  \pgfsetstrokecolor{cyan}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{0}{1}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
% Flags 1 0: 橙色
  \pgfsetstrokecolor{orange}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{1}{0}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
% Flags 1 1: 黑色
  \pgfsetstrokecolor{black}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{1}{1}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

注意：尽管本命令是个比较实用的工具，但实现本命令的数值计算过程可能不够稳定。椭圆弧的终点未必总是处于点 ⟨*target point*⟩ 上，有时会偏离数个 pt 距离，这是因为 TEX 在数值计算方面较弱。当椭圆弧的张角是 90° 的整数倍时，此命令的结果较好。

以点 $M=(m_x,m_y)$ 为中心的椭圆

$$C(\gamma)=M+(a\cos(\gamma),b\sin(\gamma)),\gamma\in[0^\circ,360^\circ],$$

假设 γ_0, γ_1 是两个角度，有椭圆弧

$$C(\gamma(t)),\gamma(t)=\gamma_0+t(\gamma_1-\gamma_0),t\in[0,1],\gamma(0)=\gamma_0,\gamma(1)=\gamma_1,$$

它的起点是 $C(\gamma_0)=M+(a\cos(\gamma_0),b\sin(\gamma_0))=P_0=(P_0^x,P_0^y)$, 终点是 $C(\gamma_1)=M+(a\cos(\gamma_1),b\sin(\gamma_1))=P_3=(P_3^x,P_3^y)$, 所以

$$\cos(\gamma_0)=\frac{P_0^x-m_x}{a},\quad \sin(\gamma_0)=\frac{P_0^y-m_y}{b},$$
$$\cos(\gamma_1)=\frac{P_3^x-m_x}{a},\quad \sin(\gamma_1)=\frac{P_3^y-m_y}{b},$$

将椭圆弧 $C(\gamma(t))$ 看作一段 3 次 Bézier 曲线，它的控制点是 P_0, P_1, P_2, P_3, 假设 P_0, P_3 给定，而 P_1, P_2 待定。有

$$C'(\gamma(0))=3(P_1-P_0)=\frac{\pi}{180}(\gamma_1-\gamma_0)(-a\sin(\gamma_0),b\cos(\gamma_0)),$$
$$C'(\gamma(1))=3(P_3-P_2)=\frac{\pi}{180}(\gamma_1-\gamma_0)(-a\sin(\gamma_1),b\cos(\gamma_1)),$$

因为参数 γ 是角度制的，而求导运算是弧度制的，所以其中有 $\frac{\pi}{180}$. 于是解出

$$P_1 = \frac{\pi}{27\times 20}(\gamma_1-\gamma_0)(-\frac{a}{b}(P_0^y-m_y),\frac{b}{a}(P_0^x-m_x))+P_0,$$
$$P_2 = \frac{-\pi}{27\times 20}(\gamma_1-\gamma_0)(-\frac{a}{b}(P_3^y-m_y),\frac{b}{a}(P_3^x-m_x))+P_3,$$

\pgfpatharctoprecomputed{⟨*center point*⟩}{⟨*start angle*⟩}{⟨*end angle*⟩}{⟨*end point*⟩}
{⟨*x-radius*⟩}{⟨*y-radius*⟩}{⟨*ratio x-radius/y-radius*⟩}{⟨*ratio y-radius/x-radius*⟩}

这个命令构建一段椭圆弧，它的计算速度以及稳定性都较好，但是它的参数比较复杂。本命令的参数针对椭圆 $\frac{(x+c_1)^2}{a^2}+\frac{(y+c_2)^2}{b^2}=1$ 上的弧：

- ⟨*center point*⟩ 是椭圆弧所在椭圆的中心点 $C=(c_1,c_2)$;
- ⟨*start angle*⟩ 是从点 ⟨*center point*⟩ 到起点的方向角 θ_s(一个数值，或保存一个数值的宏)；
- ⟨*end angle*⟩ 是从点 ⟨*center point*⟩ 到终点 ⟨*end point*⟩ 的方向角 θ_e(一个数值，或保存一个数值的宏)；
- ⟨*end point*⟩ 是椭圆弧的终点 E;
- ⟨*x-radius*⟩ 是椭圆横半轴的长度 a (带单位，如果不带就默认单位是 pt)；
- ⟨*y-radius*⟩ 是椭圆纵半轴的长度 b (带单位，如果不带就默认单位是 pt)；
- ⟨*ratio x-radius/y-radius*⟩ 是椭圆横半轴长度与纵半轴长度的比值 $\frac{a}{b}$;
- ⟨*ratio y-radius/x-radius*⟩ 是椭圆纵半轴长度与横半轴长度的比值 $\frac{b}{a}$;
- \pgfpatharctomaxstepsize, 这个宏应当保存一个正整数，其初始值为 45.

这个画弧的命令在一个组内执行其所有操作：

1. 它会对角度区间 $[\theta_s,\theta_e]$ 做分割,分割出来的一小段区间 $[\gamma_0,\gamma_1]$ 的长度是宏 \pgfpatharctomaxstepsize 的值，所以这个宏代表“角度”，这个宏的值越小，分割出来的小区间就越多。
2. 依次在各个小角度区间 $[\gamma_0,\gamma_1]$ 上执行 \pgfpatharcofellipse@, 这个命令利用前述公式，
   (a) 以当前点 (\pgf@path@lastx,\pgf@path@lasty) 为起点 P_0, 以 $(a\cos(\gamma_1),b\sin(\gamma_1))$ 为终点 P_3, 计算控制点 P_1, P_2,
   (b) 然后执行

   \pgfpathcurveto{P_1}{P_2}{P_3}

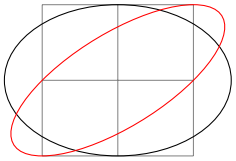
   这会导致对 P_1, P_2, P_3 做变换,得到 P_1', P_2', P_3', 并把 P_3' 的坐标作为当前点的坐标 \pgf@path@lastx, \pgf@path@lasty.

   注意，如果当前的变换矩阵不是单位矩阵，那么 \pgfpatharcofellipse@ 的这种操作会导致混乱。

\pgfpathellipse{⟨*center*⟩}{⟨*first axis*⟩}{⟨*second axis*⟩}

本命令构建一个椭圆，它可以开启一个路径，或者延伸当前路径。本命令构建的椭圆是闭的。本命令结束后，当前点是椭圆的中心点。

坐标点 ⟨*center*⟩ 指定椭圆的中心，向量 ⟨*first axis*⟩ 和 ⟨*second axis*⟩ 分别指定椭圆的“横半轴向量”和“纵半轴向量”，这两个向量可以不是正交的。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \pgfpathellipse{\pgfpoint{1cm}{1cm}}
                 {\pgfpoint{1.5cm}{0cm}}
                 {\pgfpoint{0cm}{1cm}}
  \pgfusepath{draw}
  \color{red}
  \pgfpathellipse{\pgfpoint{1cm}{1cm}}
                 {\pgfpoint{1cm}{1cm}}
                 {\pgfpoint{-1cm}{0cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

本命令的处理步骤是：

1. 用命令 \pgfpointtransformed 处理三个参数，规定几个寄存器值
2. 执行 \pgf@nlt@moveto{⟨$start_x$⟩}{⟨$start_y$⟩}，其中的 ⟨$start_x$⟩，⟨$start_y$⟩ 是椭圆的起点，即向量 ⟨*first axis*⟩ + ⟨*center*⟩ 的两个分量，从这个点开始画椭圆
3. 用命令 \pgf@nlt@curveto 画 4 段前后相接的控制曲线，使用的系数是 0.55228475，所以每一段曲线对应的角度最好都是 90°.
   第一段曲线的起点是 ⟨*first axis*⟩ + ⟨*center*⟩，终点是 ⟨*second axis*⟩ + ⟨*center*⟩.
   第二段曲线的起点是 ⟨*second axis*⟩ + ⟨*center*⟩，终点是 $-1\cdot($⟨*first axis*⟩ + ⟨*center*⟩$)$.
   第三段曲线的起点是 $-1\cdot($⟨*first axis*⟩ + ⟨*center*⟩$)$，终点是 $-1\cdot($⟨*second axis*⟩ + ⟨*center*⟩$)$.
   第四段曲线的起点是 $-1\cdot($⟨*second axis*⟩ + ⟨*center*⟩$)$，终点是 ⟨*first axis*⟩ + ⟨*center*⟩.
4. 最后执行：

```
\pgf@nlt@closepath%
\pgf@nlt@moveto{\pgf@xc}{\pgf@yc}%
```

   可见本命令创建的是闭合路径，而且本命令结束后，当前点是椭圆的中心点。

注意：

- 如果参数 ⟨*first axis*⟩，⟨*second axis*⟩ 不是方向为水平、竖直的向量，那么画的椭圆是倾斜的。
- 如果从向量 ⟨*first axis*⟩ 到 ⟨*second axis*⟩ 成左手系，那么按顺时针方向创建椭圆。
- 如果向量 ⟨*first axis*⟩ 与 ⟨*second axis*⟩ 非正交，那么近似的精度可能降低。

\pgfpathcircle{⟨*center*⟩}{⟨*radius*⟩}

本命令利用 \pgfpathellipse 构建一个圆，它可以开启一个路径，或者延伸当前路径，本命令构建的圆是闭的。本命令结束后，当前点是圆的中心点。坐标点 ⟨*center*⟩ 指定圆心，尺寸 ⟨*radius*⟩ 指定圆的半径。

本命令的定义是：

```
\def\pgfpathcircle#1#2{\pgfpathellipse{#1}{\pgfpoint{#2}{0pt}}{\pgfpoint{0pt}{#2}}
↪  }
```

14.3 矩形

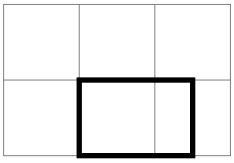
下面两个命令构建矩形，它们可以开启一个路径，或者延伸当前路径，它们构建的矩形是其所在路径的一个孤立部分，即所构建的矩形本身是闭的。

\pgfpathrectangle{⟨*corner*⟩}{⟨*diagonal vector*⟩}

以点 ⟨*corner*⟩ 和 ⟨*corner*⟩+⟨*diagonal vector*⟩ 为对角顶点确定矩形。

本命令接受坐标变换命令。

本命令刷新边界盒子。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{2pt}
  \pgfpathrectangle{\pgfpoint{1cm}{0cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

\pgfpathrectanglecorners{⟨*corner*⟩}{⟨*opposite corner*⟩}

以点 {⟨*corner*⟩} 和 {⟨*opposite corner*⟩} 为对角顶点确定矩形。

本命令接受坐标变换命令。

本命令刷新边界盒子。

14.4 网格

\pgfpathgrid[⟨*options*⟩]{⟨*first corner*⟩}{⟨*second corner*⟩}

参数 ⟨*first corner*⟩ 与 ⟨*second corner*⟩ 是 PGF 点。

本命令将网格添加到当前路径中。网格的范围由以点 ⟨*first corner*⟩ 与 ⟨*second corner*⟩ 为对角顶点的矩形限定。

本命令会检查选项 ⟨*options*⟩ 规定的网格线间距。如果竖直网格线的水平间距 $\leqslant 0.01$pt, 就不画竖直网格线。如果水平网格线的竖直间距 $\leqslant 0.01$pt, 就不画水平网格线。

本命令会对网格线的位置做调整，使得原点是一个格点 (如果扩展网格，让网格覆盖原点的话)，因此，网格线未必恰好通过点 ⟨*first corner*⟩ 与 ⟨*second corner*⟩.

本命令采用 \pgfutil@loop 循环画网格线。在一个循环中，先计算网格线的端点，然后对端点做变换 (\pgf@pos@transform)，以变换后的点为端点画线 (\pgf@nlt@lineto, 这会更新图形、路径的边界盒子)。

本命令以 \pgf@nlt@moveto 结束，move-to 的目标是变换 ⟨*second corner*⟩ 得到的点。

本命令不改变 \pgf@path@lastx 和 \pgf@path@lasty 的值。

⟨*options*⟩ 中可以使用以下选项：

/pgf/stepx=⟨*dimension*⟩ (no default, initially 1cm)

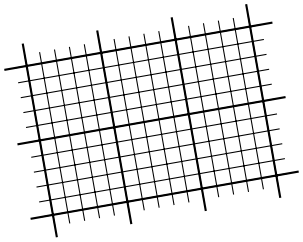
本选项指定网格的水平线之间的间距。在命令 \pgfpathgrid 中，它会被 \pgfmathsetlength 处理。

/pgf/stepy=⟨*dimension*⟩ (no default, initially 1cm)

本选项指定网格的竖直线之间的间距。在命令 \pgfpathgrid 中，它会被 \pgfmathsetlength 处理。

/pgf/step=⟨*vector*⟩ (no default)

将向量 ⟨*vector*⟩ 作为一个网格单元的对角向量。⟨*vector*⟩ 应当是 PGF 点，或者是为 \pgf@x, \pgf@y 赋值的代码。当使用这个选项时，必须提供选项值。本选项实际上为 /pgf/stepx, /pgf/stepy 赋值。



```
\begin{pgfpicture}
  \pgftransformrotate{10}
  \pgfsetlinewidth{0.8pt}
  \pgfpathgrid[step={\pgfpoint{1cm}{1cm}}]
    {\pgfpoint{-3mm}{-3mm}}{\pgfpoint{33mm}{23mm}}
  \pgfusepath{stroke}
  \pgfsetlinewidth{0.4pt}
  \pgfpathgrid[stepx=2mm,stepy=2mm]
    {\pgfpoint{-1.5mm}{-1.5mm}}{\pgfpoint{31.5mm}{21.5mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

14.5 抛物线

平面上开口向上或向下的抛物线的方程形式是

$$y = a(x-b)^2 + c,$$

它的顶点是 (b,c)，所以如果指定它的顶点和另外一个点，这个抛物线就确定了。下面的命令就是针对这种情况。

\pgfpathparabola{⟨*bend vector*⟩}{⟨*end vector*⟩}

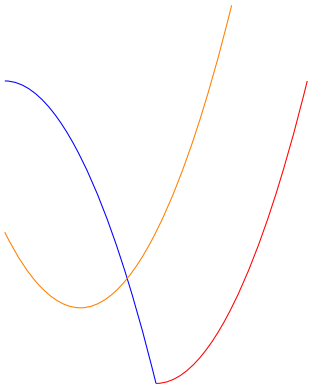
本命令的参数是 PGF 点，或者是为 \pgf@x, \pgf@y 赋值的代码。

本命令构建两段抛物线：

- 记原点 (0pt,0pt) 为 P_0，点 ⟨*bend vector*⟩ 为 $P_3 = (P_3^x, P_3^y)$. 对点 P_0, $P_1 = (0.1125P_3^x, 0.225P_3^y)$, $P_2 = (0.5P_3^x, P_3^y)$, P_3 做线性变换，得到 P_0', P_1', P_2', P_3', 然后做平移，得到 P_0'', P_1'', P_2'', P_3'', 这个平移使得 P_0'' 与当前点 (\pgf@path@lastx,\pgf@path@lasty) 重合 (然后再对这些点做非线性变换，如果有非线性变换的话)，以这些点为控制点创建曲线 (同时更新图形、路径的边界盒子)，就是第一段抛物线，其顶点是 P_3''.
  第一段抛物线完成后，当前点是 P_3''.
  如果 ⟨*bend vector*⟩=(0pt,0pt), 则没有第一段抛物线。
- 记原点 (0pt,0pt) 为 P_0，点 ⟨*end vector*⟩ 为 $P_3 = (P_3^x, P_3^y)$. 对点 P_0, $P_1 = (0.5P_3^x, 0P_3^y)$, $P_2 = (0.8875P_3^x, 0.775P_3^y)$, P_3 做线性变换，得到 P_0', P_1', P_2', P_3', 然后做平移，得到 P_0'', P_1'', P_2'', P_3'', 这个平移使得 P_0'' 与当前点 (\pgf@path@lastx,\pgf@path@lasty) 重合 (然后再对这些点做非线性变换，如果有非线性变换的话)，以这些点为控制点创建曲线 (同时更新图形、路径的边界盒子)，就是第二段抛物线，其顶点是 P_0''.
  第一段抛物线完成后，当前点是 P_3''.
  如果 ⟨*end vector*⟩=(0pt,0pt), 则没有第二段抛物线。

实际上本命令不计算 P_0, P_0', P_0'', 因为 P_0'' 就是执行本命令时的当前点 (\pgf@path@lastx,\pgf@path@lasty).

不能用本命令构建一段不含顶点的抛物线。



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpointorigin}{\pgfpoint{2cm}{4cm}}
  \color{red}
  \pgfusepath{stroke}

  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpoint{-2cm}{4cm}}{\pgfpointorigin}
  \color{blue}
  \pgfusepath{stroke}

  \pgfpathmoveto{\pgfpoint{-2cm}{2cm}}
  \pgfpathparabola{\pgfpoint{1cm}{-1cm}}{\pgfpoint{2cm}{4cm}}
  \color{orange}
  \pgfusepath{stroke}
\end{pgfpicture}
```

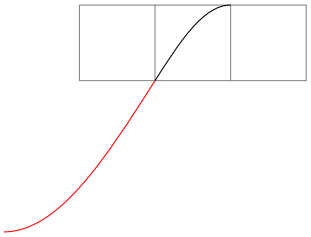
14.6 正弦，余弦

\pgfpathsine{⟨*vector*⟩}

本命令的参数是 PGF 点，或者是为 `\pgf@x`, `\pgf@y` 赋值的代码。
记原点 (`0pt,0pt`) 为 P_0，点 $\langle vector\rangle$ 为 $P_3 = (P_3^x, P_3^y)$. 对点 P_0，$P_1 = (0.3260P_3^x, 0.5120P_3^y)$，$P_2 = (0.6380P_3^x, P_3^y)$，$P_3$ 做线性变换，得到 P_0', P_1', P_2', P_3'，然后做平移，得到 P_0'', P_1'', P_2'', P_3''，这个平移使得 P_0'' 与当前点 (`\pgf@path@lastx,\pgf@path@lasty`) 重合 (然后再对这些点做非线性变换，如果有非线性变换的话)，以这些点为控制点创建曲线 (同时更新图形、路径的边界盒子)，对应的是函数

$$\sin x,\ x \in [0, \frac{\pi}{2}],$$

的图像，其中 $(\frac{\pi}{2}, \sin\frac{\pi}{2}) = (\frac{\pi}{2}, 1)$ 对应 $\langle vector\rangle$.
此命令完成后，当前点是 P_3''.
实际上本命令不计算 P_0, P_0', P_0'', 因为 P_0'' 就是执行本命令时的当前点 (`\pgf@path@lastx,\pgf@path@lasty`).



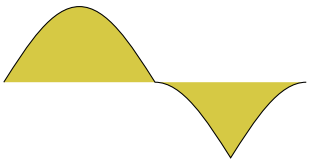
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,1);
  \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
  \pgfpathsine{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
  \color{red}
  \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
  \pgfpathsine{\pgfpoint{-2cm}{-2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfpathcosine{`$\langle vector\rangle$`}`

本命令的参数是 PGF 点，或者是为 `\pgf@x`, `\pgf@y` 赋值的代码。
记原点 (`0pt,0pt`) 为 P_0，点 $\langle vector\rangle$ 为 $P_3 = (P_3^x, P_3^y)$. 对点 P_0，$P_1 = (0.3620P_3^x, 0P_3^y)$，$P_2 = (0.6740P_3^x, 0.4880P_3^y)$，$P_3$ 做线性变换，得到 P_0', P_1', P_2', P_3'，然后做平移，得到 P_0'', P_1'', P_2'', P_3''，这个平移使得 P_0'' 与当前点 (`\pgf@path@lastx,\pgf@path@lasty`) 重合 (然后再对这些点做非线性变换，如果有非线性变换的话)，以这些点为控制点创建曲线 (同时更新图形、路径的边界盒子)，对应的是函数

$$\cos x,\ x \in [0, \frac{\pi}{2}],$$

的图像，其中 $(\frac{\pi}{2}, \cos\frac{\pi}{2}) = (\frac{\pi}{2}, 0)$ 对应 $\langle vector\rangle$.
此命令完成后，当前点是 P_3''.
实际上本命令不计算 P_0, P_0', P_0'', 因为 P_0'' 就是执行本命令时的当前点 (`\pgf@path@lastx,\pgf@path@lasty`).



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{0cm}}
  \pgfpathsine{\pgfpoint{1cm}{1cm}}
  \pgfpathcosine{\pgfpoint{1cm}{-1cm}}
  \pgfpathcosine{\pgfpoint{1cm}{-1cm}}
  \pgfpathsine{\pgfpoint{1cm}{1cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

14.7 Plot

具体参考 §112.

14.8 圆角（Rounded Corners）

当在一段连续路径内部有“拐角”时，例如，折线段、多边形、先后相继的控制曲线，拐角一般是尖角。可以把尖角改成圆角。

处理圆角的流程大体上是：

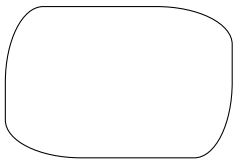
1. 命令 \pgfsetcornersarced 会设置 \ifpgf@arccorners 的真值以及圆角的尺寸；
2. 如果 \ifpgf@arccorners 的真值是 true, 那么构建路径的基本命令 (例如 \pgfpathlineto) 会调用 \pgf@roundcornerifneeded → P. 292 向软路径中添加记号 \pgfsyssoftpath@specialroundtoken;
3. 之后命令 \pgfusepath → P. 295 调用 \pgfprocessround 处理软路径中的尖角。

由于 TEX 组能限制 \ifpgf@arccorners 的真值，所以在一个路径中，可以用组来规定这个路径的哪一部分使用圆角，哪一部分使用尖角，也就是说，尖角和圆角可以共存于一个路径中。

\pgfsetcornersarced{⟨*point*⟩}

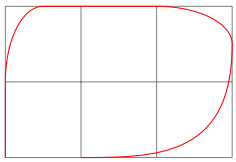
本命令将其后的所有尖角变成圆角。

假设线段或曲线 l 与 r 构成一段连续路径，二者在连接处有一个“尖的拐角”，并且 l 在前，r 在后。将 l 靠近连接点的一部分截去，也把 r 靠近连接点的一部分截去，然后用圆弧连接 l 与 r, 就把尖角变成了圆角。该命令的参数——点 ⟨*point*⟩ 的 x 分量决定 l 被截去的那一段有多长，点 ⟨*point*⟩ 的 y 分量决定 r 被截去的那一段有多长，从而影响圆弧的尺寸和形态。



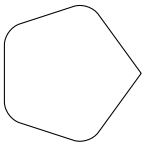
```
\begin{tikzpicture}
  \pgfsetcornersarced{\pgfpoint{5mm}{10mm}}
  \pgfpathrectanglecorners{\pgfpointorigin}
                          {\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

从上面例子看出，矩形是按逆时针方向构建的。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetcornersarced{\pgfpoint{10mm}{5mm}}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0cm}{2cm}}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfpathcurveto{\pgfpoint{3cm}{0cm}}
                 {\pgfpoint{2cm}{0cm}}
                 {\pgfpoint{1cm}{0cm}}
  \color{red}
  \pgfusepath{stroke}
\end{tikzpicture}
```

如果点 ⟨*point*⟩ 的 x 分量与 y 分量相同，并且原来的尖角是 90° 的，则本命令构建的圆角是极其接近圆弧的。如果点 ⟨*point*⟩ 的 x 分量与 y 分量不相同，则本命令构建的圆角与圆弧的接近程度要差一些。



```
\begin{pgfpicture}
  \pgfsetcornersarced{\pgfpoint{6pt}{6pt}}
  \pgfpathmoveto{\pgfpointpolar{0}{1cm}}
  \pgfpathlineto{\pgfpointpolar{72}{1cm}}
  \pgfpathlineto{\pgfpointpolar{144}{1cm}}
  \pgfpathlineto{\pgfpointpolar{216}{1cm}}
  \pgfpathlineto{\pgfpointpolar{288}{1cm}}
  \pgfpathlineto{\pgfpointpolar{0}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

在上面的例子中，表面上看路径是个多边形，但是它并不“封闭”，实际上是个折线段，所以它的起始边与终止边没有连接起来构成“角”，因此圆角命令不对这个地方起作用。使用命令 \pgfpathclose 可以纠正这一点。

使用圆角命令后，如果想改回“尖角”，则可以使用命令：

```
\pgfsetcornersarced{\pgfpointorigin}
```

注意，如果构成尖角的边的长度很短，那么使用圆角命令可能造成意外结果。
本命令的定义是：

```
\newif\ifpgf@arccorners

\def\pgfsetcornersarced#1{%
  \pgf@process{#1}%
  \edef\pgf@corner@arc{{\the\pgf@x}{\the\pgf@y}}%
  \pgf@arccornerstrue%
  \ifdim\pgf@x=0pt%
    \ifdim\pgf@y=0pt\relax%
      \pgf@arccornersfalse%
    \fi%
  \fi%
}
```

可见命令 \pgfsetcornersarced{⟨*code*⟩} 的参数 ⟨*code*⟩ 应当是能够决定尺寸寄存器 \pgf@x, \pgf@y 的值的代码。本命令设置 \ifpgf@arccorners 的真值。命令 \pgf@roundcornerifneeded 会检查这个真值。

\pgf@roundcornerifneeded

本命令的定义是：

```
\def\pgf@roundcornerifneeded{%
  \ifpgf@arccorners\expandafter\pgfsyssoftpath@specialround\pgf@corner@arc\fi%
}
```

本命令与 \pgfsetcornersarced 相配合。那些构建路径的基本命令会调用本命令，如 \pgfpathlineto, \pgfpathclose, \pgfpathcurveto, \pgfpathquadraticcurveto, \pgfpatharc, 例如

```
\def\pgfpathlineto#1{%
  \pgfpointtransformed{#1}%
  \pgf@roundcornerifneeded%
  \pgf@nlt@lineto{\pgf@x}{\pgf@y}%
  \global\pgf@path@lastx=\pgf@x%
  \global\pgf@path@lasty=\pgf@y%
}
```

其作用是：在路径的“转角”的前面执行 \pgf@roundcornerifneeded, 如果 \ifpgf@arccorners 的真值是 true, 则把命令

```
\expandafter\pgfsyssoftpath@specialround\pgf@corner@arc
```

放到“转角”的前面。命令 \pgfsyssoftpath@specialround → P. 226 见文件《pgfsyssoftpath.code.tex》，它会向当前的软路径中添加记号

```
\pgfsyssoftpath@specialroundtoken{⟨x⟩}{⟨y⟩}
```

14.9 跟踪路径或图形的边界盒子

在构建路径时，PGF 会跟踪当前路径的边界盒子以及整个图形的边界盒子。
文件《pgfcorepoints.code.tex》中声明以下 8 个尺寸寄存器：

1. \pgf@pathminx, \pgf@pathmaxx, \pgf@pathminy, \pgf@pathmaxy, 用于记录当前路径的左、右、下、

上侧边界的当前的尺寸坐标。例如，若 \pgf@pathminx=-10pt, \pgf@pathmaxx=10pt, 则当前路径的当前宽度是 20pt.

2. \pgf@picminx, \pgf@picmaxx, \pgf@picminy, \pgf@picmaxy, 用于记录当前图形的左、右、下、上侧边界的当前的尺寸坐标。

在 pgfpicture 环境的开头会设置

```
\pgf@picmaxx=-16000pt\relax%
\pgf@picminx=16000pt\relax%
\pgf@picmaxy=-16000pt\relax%
\pgf@picminy=16000pt\relax%
```

也会执行 \pgf@resetpathsizes.

在 pgfscope 环境的开头会执行 \pgf@resetpathsizes.

在 \pgfusepath 处理当前路径后，也会执行 \pgf@resetpathsizes.

通常情况下，构造路径的命令，如 \pgfpathmoveto, \pgfpathlineto, 会利用其参数来跟新这些寄存器的值，使得边界盒子将参数 (路径的构造点) 包含在内。

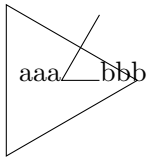
\pgfresetboundingbox

本命令在《pgfcorescopes.code.tex》中定义：

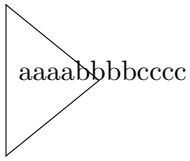
```
\def\pgfresetboundingbox{%
    \global\pgf@picmaxx=-16000pt\relax%
    \global\pgf@picminx=16000pt\relax%
    \global\pgf@picmaxy=-16000pt\relax%
    \global\pgf@picminy=16000pt\relax%
}%
```

本命令重设图形的边界盒子。本命令使得程序“忘记”已经计算出的图形的边界盒子，并从当下开始，跟踪之后的坐标点，重新计算边界盒子。如果本命令用于绘图环境的末尾，那么程序就当图形没有边界，并且使得图形的原点位于插入图形的位置。

本命令通常与 \pgfusepath{use as bounding box} 配合使用。




```
aaa\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointpolar{0}{0cm}}
  \pgfpathlineto{\pgfpointpolar{150}{2cm}}
  \pgfpathlineto{\pgfpointpolar{210}{2cm}}
  \pgfpathclose
  \pgfusepath{stroke}
  \pgfresetboundingbox  % 重设边界盒子
  \pgfpathmoveto{\pgfpointpolar{120}{1cm}}
  \pgfpathlineto{\pgfpointpolar{0}{-1cm}}
  \pgfpathlineto{\pgfpointpolar{0}{-0.5cm}}
  \pgfusepath{stroke}
\end{pgfpicture}bbb
```




```
aaaabbbb\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointpolar{0}{-0.5cm}}
  \pgfpathlineto{\pgfpointpolar{150}{2cm}}
  \pgfpathlineto{\pgfpointpolar{210}{2cm}}
  \pgfpathclose
  \pgfusepath{stroke}
  \pgfresetboundingbox  % 重设边界盒子
\end{pgfpicture}cccc
```

\pgf@protocolsizes{⟨*x-dimension*⟩}{⟨*y-dimension*⟩}

这里 ⟨*x-dimension*⟩ 和 ⟨*y-dimension*⟩ 都是尺寸，二者分别作为横坐标和纵坐标，决定一个坐标点。本命令利用这个点来更新边界盒子。

本命令的处理是：

1. 如果 \ifpgf@relevantforpicturesize 的真值是 true, 那么就用 ⟨*x-dimension*⟩ 和 ⟨*y-dimension*⟩ 更新图形的边界盒子；并且，如果 \ifpgf@size@hooked 的真值也是 true, 还会执行：

   ```
   \let\pgf@size@hook@x#1\let\pgf@size@hook@y#2\pgf@path@size@hook
   ```

   其中 \pgf@path@size@hook 的初始状态等于 \pgfutil@empty.
   也就是说，可以设置 \pgf@size@hookedtrue, 并且重定义 \pgf@path@size@hook, 使之能处理 \pgf@size@hook@x 和 \pgf@size@hook@y.
   如果 \ifpgf@relevantforpicturesize 的真值是 false, 那就不会更新图形的边界盒子。
2. 用 ⟨*x-dimension*⟩ 和 ⟨*y-dimension*⟩ 更新路径的边界盒子。

\ifpgf@relevantforpicturesize

在这个 TEX-if 的真值是 false 的情况下，图形的边界盒子不会被更新，也不执行 \pgf@path@size@hook.

\pgf@resetpathsizes

这个命令重置路径的边界盒子。

```
\def\pgf@resetpathsizes{%
  \global\pgf@pathmaxx=-16000pt\relax%
  \global\pgf@pathminx=16000pt\relax%
  \global\pgf@pathmaxy=-16000pt\relax%
  \global\pgf@pathminy=16000pt\relax%
}
```

\pgf@getpathsizes⟨*macro*⟩

本命令将路径的当前边界坐标保存到宏 ⟨*macro*⟩ 中。

```
\def\pgf@getpathsizes#1{%
  \edef#1{{\the\pgf@pathmaxx}{\the\pgf@pathminx}{\the\pgf@pathmaxy}{
  ↪  \the\pgf@pathminy}}%
}
```

\pgf@setpathsizes⟨*macro*⟩

本命令将保存在宏 ⟨*macro*⟩ 中的尺寸用作当前的路径边界坐标。

```
\def\pgf@setpathsizes#1{%
  \expandafter\pgf@@setpathsizes#1%
}
\def\pgf@@setpathsizes#1#2#3#4{%
  \global\pgf@pathmaxx=#1\relax%
  \global\pgf@pathminx=#2\relax%
  \global\pgf@pathmaxy=#3\relax%
  \global\pgf@pathminy=#4\relax%
}
```

第十五章　使用路径

15.1　Overview

构建一个路径后就可以使用它，“使用”的意思是，例如，你可以画出它，填充颜色，用于剪切其它路径，等等。也有很多图形参数能影响路径的外观，例如线宽，线型，颜色等。设置图形参数的命令都以 `\pgfset` 开头。

注意，影响路径的外观的选项都是对整个路径有效的，例如，对于一个路径来说，你不能把该路径的前一段画成红色（red），而把后一段画成绿色（green）。

路径的使用方式有以下几种：

1. 画出路径，stroke 或 draw.
2. 给路径添加箭头，arrow tips.
3. 填充颜色，fill.
4. 作为剪切路径来剪切之后的路径，clip.
5. 画阴影，shade.
6. 用作边界盒子，use as bounding box.

以上几种方式可以混合使用，当然同时使用 fill 和 shade 没有意义。

`\pgfusepath{`⟨*actions*⟩`}`

⟨*actions*⟩ 是一个或数个使用路径的方式选项，相邻选项之间用逗号分开。在 ⟨*actions*⟩ 中可以使用以下选项：

- `fill`

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{fill}
\end{pgfpicture}
```

- `stroke`

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

- `draw`, 等效于 `stroke`.
- `clip`

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke,clip}
  \pgfpathcircle{\pgfpoint{1cm}{1cm}}{0.5cm}
  \pgfusepath{fill}
\end{pgfpicture}
```

- discard, 丢弃当前路径，等效于 \pgfusepath{}.

\pgfusepath{stroke,fill} 等于 \pgfusepath{fill,stroke}, 这两个选项的次序可换。

使用命令 \pgfshadepath 给路径添加阴影效果。

15.2 画出路径

用命令 \pgfusepath{stroke} 画出当前路径。有许多图形参数能影响画出的路径的外观，多数参数的有效范围都受到绘图环境的限制，有的参数，例如判断区域内部或外部的“奇偶规则”、“非零规则”，以及箭头选项，都受到 TEX 组的限制。不过在将来的版本中，可能会改变这一状况。

15.2.1 图形参数：线宽 Line Width

\pgfsetlinewidth{⟨*line width*⟩}

参数 ⟨*line width*⟩ 应当能被 \pgfmathsetlength 处理，处理结果是 TEX 尺寸。本命令规定其后的 \pgfusepath{stroke} 命令画出的路径线条的线宽（限于当前 pgfscope 内）。

```
\begin{pgfpicture}
  \pgfsetlinewidth{1mm}
  \pgfpathmoveto{\pgfpoint{0mm}{0mm}}
  \pgfpathlineto{\pgfpoint{2cm}{0mm}}
  \pgfusepath{stroke}
  \pgfsetlinewidth{2\pgflinewidth}  % 线宽加倍
  \pgfpathmoveto{\pgfpoint{0mm}{5mm}}
  \pgfpathlineto{\pgfpoint{2cm}{5mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

在文件《pgfcoregraphicstate.code.tex》中有定义：

```
\def\pgfsetlinewidth#1{%
  \pgfmathsetlength\pgflinewidth{#1}%
  \global\pgflinewidth=\pgflinewidth%
  \pgfsys@setlinewidth{\the\pgflinewidth}%
  \ignorespaces}
```

参考 \pgfmathsetlength→P. 112 的定义，如果 ⟨*line width*⟩ 以加号 + 开头，则直接把 ⟨*line width*⟩ 赋予 \pgflinewidth. 如果 ⟨*line width*⟩ 不以加号 + 开头，则 ⟨*line width*⟩ 会被 \pgfmathparse 解析。本命令全局地为寄存器 \pgflinewidth 赋值。

《pgfcorepathconstruct.code.tex》《pgfcorescopes.code.tex》中都有：

```
\pgfsetlinewidth{0.4pt}
```

\pgflinewidth

这是个 TEX 尺寸寄存器，它保存的是当前的线宽。你可以全局地设置它的值，然后在某个绘图环境中局部地修改它。

```
0.4pt \tikz;
      \the\pgflinewidth
```

15.2.2 图形参数：线冠 Caps 与交接 Joins

\pgfsetbuttcap

规定线冠为 butt cap.

```
\def\pgfsetbuttcap{\pgfsys@buttcap\ignorespaces}
```

\pgfsetroundcap

规定线冠为 round cap.

```
\def\pgfsetroundcap{\pgfsys@roundcap\ignorespaces}
```

\pgfsetrectcap

规定线冠为 square cap.

```
\def\pgfsetrectcap{\pgfsys@rectcap\ignorespaces}
```

\pgfsetroundjoin

规定交接为 round join.

```
\def\pgfsetroundjoin{\pgfsys@roundjoin\ignorespaces}
```

\pgfsetbeveljoin

规定交接为 bevel join.

```
\def\pgfsetbeveljoin{\pgfsys@beveljoin\ignorespaces}
```

\pgfsetmiterjoin

规定交接为 miter join.

```
\def\pgfsetmiterjoin{\pgfsys@miterjoin\ignorespaces}
```

\pgfsetmiterlimit{⟨*miter limit factor*⟩}

设置交接的极限因子，参数 ⟨*miter limit factor*⟩ 是数值。

```
\def\pgfsetmiterlimit#1{%
  \ifdim#1pt<1pt\pgferror{miter limit cannot be less than 1}\fi%
  \pgfsys@setmiterlimit{#1}%
  \ignorespaces}
```

15.2.3 图形参数：线型 Dashing

\pgfsetdash{⟨*list of even length of dimensions*⟩}{⟨*phase*⟩}

这个命令规定线型，注意其中的花括号结构，例如：

```
\pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0.2cm}
```

这个命令规定的线型是，0.5cm 的实线，后接 0.5cm 的虚线，后接 0.1cm 的实线，后接 0.2cm 的虚线，这是一个“周期”，画线时不断重复这个周期。最后一个花括号规定“相位”。

使用命令 `\pgfsetdash{}{0pt}` 得到实线。

本命令的所有参数都会被 `\pgfmathsetlength` 处理。本命令最后调用 `\pgfsys@setdash` 设置线型。

15.2.4 图形参数：线条颜色

\pgfsetstrokecolor{⟨*color*⟩}

本命令设置之后画出的路径线条的颜色，⟨*color*⟩ 是 LaTeX 的颜色格式 red 或 black!20!red. 这个命令的作用范围受到当前 pgfscope 环境的限制，而不是受到 TeX 组的限制。如果使用 LaTeX 的颜色命令 \color{⟨*color*⟩} 设置颜色 (这个命令受到 TeX 组的限制)，那么所有颜色 (包括线条和填充) 都被修改。

```
\begin{pgfpicture}
  \pgfsetlinewidth{1pt}
  \color{red}  % 设置各种颜色设为 red
  \pgfpathcircle{\pgfpoint{0cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
  \pgfsetstrokecolor{black}  % 设置线条颜色为 black
  \pgfpathcircle{\pgfpoint{1cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
  \color{red}  % 设置各种颜色设为 red
  \pgfpathcircle{\pgfpoint{2cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
\end{pgfpicture}
```

注意上面图形中的第三个圆，其边界线条还是黑色，与手册描述不一样。上面例子是在 xelatex 下编译的，如果在 pdflatex 下编译，那么结果与手册描述的一样。

本命令需要 xcolor 宏包的支持。宏包 xxcolor 能影响这个命令。

\pgfsetcolor{⟨*color*⟩}

这个命令同时设置线条颜色、填充颜色，其有限范围受到绘图环境的限制（而不是 TeX 组）。

本命令需要 xcolor 宏包的支持。宏包 xxcolor 能影响这个命令。

15.2.5 线条透明度

使用命令 \pgfsetstrokeopacity{⟨*value*⟩} 设置线条透明度。

15.2.6 双线的内线

如果给双线加箭头，那么会先画出外线，再画内线，然后画箭头，而箭头的尺寸首先参照内线的线宽。

\pgfsetinnerlinewidth{⟨*dimension*⟩}

本命令的定义是：

```
\def\pgfsetinnerlinewidth#1{%
  \pgfmathsetlength\pgf@x{#1}%
  \edef\pgfinnerlinewidth{\the\pgf@x}%
}
\def\pgfinnerlinewidth{0pt}
```

本命令规定内线的线宽。使用这个命令后，宏 \pgfinnerlinewidth 的值应当不是 0pt, 这会导致启用双线功能。宏 \pgfinnerlinewidth 的初始值是 0pt, 在这个尺寸下根本不会有画内线的动作。

这个命令的有限范围受到 TeX 组的限制，而不是受到 pgfscope 的限制。双线的内线不能用作剪切路径来剪切其它路径，如果把双线用作剪切路径，那么内线就消失，外线也没有剪切作用。在将来的版本中可能会改变这一点。通常的线宽属于图形状态，但内线线宽不属于图形状态。

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfsetlinewidth{2pt}
  \pgfsetinnerlinewidth{1pt}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetinnerstrokecolor{⟨color⟩}

这个命令设置双线功能中内线的颜色，其有效范围受到 TEX 组的限制。

```
\def\pgfsetinnerstrokecolor#1{\def\pgfinnerstrokecolor{#1}}
\def\pgfinnerstrokecolor{white}
```

15.3 给路径加箭头

参考《pgfcorearrows.code.tex》。

\pgfsetarrowsstart{⟨start arrow tip specification⟩}

本命令指定路径始端的箭头类型，其有效范围受到 TEX 组的限制。

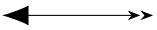
\pgfsetarrowsend{⟨end arrow tip specification⟩}

本命令指定路径末端的箭头类型，其有效范围受到 TEX 组的限制。

```
\begin{pgfpicture}
  \pgfsetarrowsstart{Latex[length=10pt]}
  \pgfsetarrowsend{Computer Modern Rightarrow}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetarrows{⟨argument⟩}

这个命令指定箭头的样式，⟨argument⟩ 是指定箭头样式的那些句法格式，参考 §16.

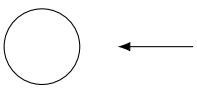


```
\begin{pgfpicture}
  \pgfsetarrows{Latex[length=10pt]-{Stealth[] . Stealth[]}}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetshortenstart{⟨dimension⟩}

这个命令可以在路径开头处将路径截去一段，也就是将路径起点沿着路径方向移动一段，使得路径缩短，这一段的长度就是 ⟨dimension⟩. 如果对某个路径同时使用这个命令以及加箭头的命令，程序先将路径如此缩短，然后再加箭头，而添加箭头时程序还会自动将路径再裁掉一段。

使用 /pgf/arrow keys/sep →P. 795 选项也能达到本命令的效果。



```
\begin{pgfpicture}
  \pgfpathcircle{\pgfpointorigin}{5mm}
  \pgfusepath{stroke}
  \pgfsetarrows{Latex-}
  \pgfsetshortenstart{5mm}
  \pgfpathmoveto{\pgfpoint{5mm}{0cm}}
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetshortenend{⟨dimension⟩}

这个命令可以在路径结尾处将路径截去一段，也就是将路径终点沿着反路径方向移动一段，使得路径缩短，这一段的长度就是 ⟨dimension⟩.

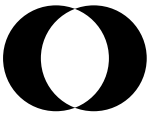
15.4 填充路径

填充路径的意思是给路径“内部”的点用某种颜色来着色。只有闭合路径才有“内部”与“外部”的区分。在填充路径时，程序会检查路径是否闭合，如果路径不是闭合的就把它当成是闭合的，即在路径的始点与终点之间使用闭合操作（但不画线）。对于一个闭合路径和某个点，需要某种规则来判断这个点是否属于该路径的“内部”。有两种规则，奇偶规则、非零规则，默认使用非零规则。

15.4.1 图形参数：判断内部点的规则

\pgfseteorule

这个命令指定奇偶规则，其有效范围受到 TEX 组的限制。

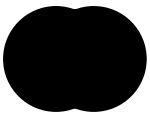


```
\begin{pgfpicture}
  \pgfseteorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

```
\def\pgfseteorule{\pgfsys@eoruletrue\ignorespaces}
```

\pgfsetnonzerorule

这个命令指定非零规则，其有效范围受到 TEX 组的限制。这是默认的规则。



```
\begin{pgfpicture}
  \pgfsetnonzerorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

```
\def\pgfsetnonzerorule{\pgfsys@eorulefalse\ignorespaces}
```

15.4.2 图形参数：填充色

\pgfsetfillcolor{⟨*color*⟩}

这个命令指定填充用的颜色。本命令需要 xcolor 宏包的支持。宏包 xxcolor 能影响这个命令。

15.4.3 图形参数：填充色的不透明度

用命令 \pgfsetfillopacity{⟨*value*⟩} 设置填充色的不透明度。

15.5 剪切路径

当使用选项 clip 时，当前路径就成为剪切路径，对之后的路径起到剪切作用。剪切时，只保留路径内部的图形，所以这里仍然需要判断路径内部或外部的规则。在一个剪切操作之内可以套嵌一个剪切操作，内部剪切操作的剪切范围，不会超出外部剪切操作的剪切范围。

剪切路径的剪切作用会一直持续到 pgfscope 环境结束。

15.6 将路径用作边界盒子

使用 use as bounding box 选项后，当前路径会被计入整个图形的边界盒子，但是当前路径之后的路径不计入边界盒子。本选项受到 TEX 组的限制。命令 \pgfresetboundingbox 通常与 \pgfusepath{use as bounding box} 配合使用。

Left right.

```
Left
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{use as bounding box} % 将矩形用作边界盒子
  \pgfpathcircle{\pgfpointorigin}{2ex}
  \pgfusepath{stroke}
\end{pgfpicture}
right.
```

15.7 文件《pgfcorepathusage.code.tex》

在命令 \pgfusepath{⟨*actions*⟩} 中，参数 ⟨*actions*⟩ 列出的是“选项”，即“键 (key)”，文件中有：

```
\pgfkeys{
  /pgf/stroke/.code=\let\pgf@up@stroke\pgf@up@stroke@text\pgf@up@path@neededtrue,
  /pgf/draw/.code=\let\pgf@up@stroke\pgf@up@stroke@text\pgf@up@path@neededtrue,
  /pgf/fill/.code=\let\pgf@up@fill\pgf@up@fill@text\pgf@up@path@neededtrue,
  /pgf/clip/.code=\let\pgf@up@clip\pgf@up@clip@text\pgf@up@path@neededtrue,
  /pgf/discard/.code=,
  /pgf/use as bounding box/.code=\let\pgf@up@bb\pgf@do@up@bb,
}
```

文件还定义了关于箭头的选项：

```
\pgfkeys{
  /pgf/arrow keys/flex/.code=\pgferror{You need to say \string\usetikzlibrary{bending}
  ↪  for flexing and bending arrows},
  /pgf/arrow keys/flex'/.code=\pgferror{You need to say \string\usetikzlibrary{bending
  ↪  } for flexing and bending arrows},
  /pgf/arrow keys/bend/.code=\pgferror{You need to say \string\usetikzlibrary{bending}
  ↪  for flexing and bending arrows},
  /pgf/arrow keys/quick/.code=\pgfarrowsaddtooptions{\let\pgf@arrow@flex@mode
  ↪  \pgf@arrow@mode@is@quick},
}
```

其中的 /pgf/arrow keys/flex 等会被文件《tikzlibrarybending.code.tex》重新定义。

定义 tip mode 选项：

```
\pgfkeys{
  /pgf/.cd,
  tips/.is choice,
  tips/.default=true,
  tips/proper/.code=\let\pgf@tips@mode\pgf@tips@mode@onproper,
  tips/on proper draw/.code=\let\pgf@tips@mode\pgf@tips@mode@onproperdraw,
  tips/true/.code=\let\pgf@tips@mode\pgf@tips@mode@true,
  tips/on draw/.code=\let\pgf@tips@mode\pgf@tips@mode@ondraw,
  tips/never/.code=\let\pgf@tips@mode\pgf@tips@mode@false,
  tips/false/.code=\let\pgf@tips@mode\pgf@tips@mode@false,
}
\def\pgf@tips@mode@onproper{4}
\def\pgf@tips@mode@onproperdraw{3}
```

```
\def\pgf@tips@mode@true{2}
\def\pgf@tips@mode@ondraw{1}
\def\pgf@tips@mode@false{0}
\let\pgf@tips@mode\pgf@tips@mode@ondraw
```

文件还有初始化定义：

```
% The following can only be set to "true" using the options in the
% module "bending"
\newif\ifpgf@precise@shortening
\def\pgf@arrow@mode@is@quick{0}
\let\pgf@arrow@flex@mode\pgf@arrow@mode@is@quick

\newif\ifpgf@up@path@needed
\newif\ifpgf@up@draw@arrows
\def\pgf@up@stroke@text{stroke}
\def\pgf@up@fill@text{fill}
\def\pgf@up@clip@text{clip}
\def\pgf@do@up@bb{\pgf@relevantforpicturesizefalse}
%...
\def\pgfsetshortenstart#1{\pgfmathsetlength\pgf@shorten@start@additional{#1}}
\def\pgfsetshortenend#1{\pgfmathsetlength\pgf@shorten@end@additional{#1}}

\newdimen\pgf@shorten@end@additional
\newdimen\pgf@shorten@start@additional
```

`\ifpgf@up@path@needed` 指示是否需要当前的软路径，如果不需要，就把当前软路径做成空的。

命令 `\pgfusepath` 的定义是：

```
\def\pgfusepath#1{%
  \pgf@up@path@neededfalse%
  \pgf@up@draw@arrowsfalse%
  %......
}
```

这个命令的定义比较长。命令 `\pgfusepath{`⟨*actions*⟩`}` 的处理过程如下：

1. 设置真值

   ```
   \pgf@up@path@neededfalse%
   \pgf@up@draw@arrowsfalse%
   ```

2. 清空

   ```
   \let\pgf@up@stroke\pgfutil@empty%
   \let\pgf@up@fill\pgfutil@empty%
   \let\pgf@up@clip\pgfutil@empty%
   \let\pgf@up@bb\pgfutil@empty%
   ```

3. 执行选项 `\pgfset`→ P. 202`{`⟨*actions*⟩`}`.

4. 定义 `\pgf@up@action`

   ```
   \expandafter\def\expandafter\pgf@up@action\expandafter{\csname pgfsys@\pgf@up@fill
   ↪  \pgf@up@stroke\endcsname}%
   ```

   按选项 ⟨*actions*⟩ 的设置，宏 `\pgf@up@action` 保存的可能是

   - `\pgfsys@`
   - `\pgfsys@fill`
   - `\pgfsys@stroke`
   - `\pgfsys@fillstroke`

5. 检查宏 \pgf@tips@mode 保存的整数值，
   - 如果它的值是 2，即设置了选项 /pgf/tips=true(这是默认的)，或者
   - 如果它的值是 4，即设置了选项 /pgf/tips=proper

   那么设置真值 \pgf@up@path@neededtrue.
6. 检查 \ifpgf@up@path@needed 的真值，如果它的真值是 false, 则

   ```
   \pgfsyssoftpath@setcurrentpath\pgfutil@empty%
   ```

   即把 \pgfutil@empty 作为当前的软路径，或者说，清空软路径。
7. 如果 \pgf@up@stroke 和 \pgf@up@fill 都等于 \pgfutil@empty, 则

   ```
   \let\pgf@up@action=\pgfutil@empty%
   ```

   并且如果 \pgf@up@clip 不等于 \pgfutil@empty （仅仅剪切）, 则

   ```
   \let\pgf@up@action=\pgfsys@discardpath%
   ```

8. 执行

   ```
   \pgfsyssoftpath@getcurrentpath\pgf@last@processed@path
   ```

   把当前的软路径保存到宏 \pgf@last@processed@path 中。
9. 执行

   ```
   \pgfprocessround{\pgf@last@processed@path}{\pgf@last@processed@path}%
   ```

   这是 (在必要下) 把软路径中的尖角变成圆角，再把软路径保存到宏 \pgf@last@processed@path 中。
10. 执行

    ```
    \pgfsyssoftpath@setcurrentpath\pgf@last@processed@path%
    ```

    把宏 \pgf@last@processed@path 中的软路径用作当前的软路径。
11. 如果 \ifpgf@relevantforpicturesize 的真值是 true, 并且 \pgf@up@stroke 不等于 \pgfutil@empty, 并且 \pgf@picmaxx 的值不等于 -16000pt, 则将当前 picture 的边界向外围扩展 0.5 个线宽（\pgflinewidth）。
12. 做一些列条件检查
    - 如果 \pgf@up@clip 等于 \pgfutil@empty
      - 如果 \pgf@up@stroke 等于 \pgfutil@empty
        - 如果 \pgf@up@action 等于 \pgfutil@empty, 则检查 \pgf@tips@mode 保存的整数值，
          - 如果它的值是 2，即设置了选项 /pgf/tips=true(这是默认的)，或者
          - 如果它的值是 4，即设置了选项 /pgf/tips=proper

          那么执行 \pgf@up@draw@arrows@only→P. 304, 此时的路径是不画出的，可能只需要处理箭头。
        - 如果 \pgf@up@action 不等于 \pgfutil@empty, 则执行

          ```
          \pgfsyssoftpath@invokecurrentpath→P. 220%
          \pgf@up@action%
          ```

          命令 \pgfsyssoftpath@invokecurrentpath→P. 220 将软路径放到当前位置，注意此时的 \pgf@up@action 可能等效于 \pgfsys@fill.
      - 如果 \pgf@up@stroke 不等于 \pgfutil@empty, 则
        (a) 执行 \pgfgetpath\pgf@arrowpath, 将当前的软路径保存到 \pgf@arrowpath.
        (b) 执行 \pgf@path@setup@tempswa→P. 305,
        (c) 执行

        ```
        \pgfprocesscheckclosed→P. 305{\pgf@arrowpath}{\pgfutil@tempswafalse}
        ```

这是检查软路径 \pgf@arrowpath 是否封闭路径,如果是,则设置真值 \pgfutil@tempswafalse.

(d) 执行 \pgf@path@check@proper→P. 305,

(e) 执行

```
\ifpgfutil@tempswa%
  \pgf@check@for@arrow@and@animation→P. 305%
  \pgf@prepare@end@of@path→P. 306%
  \begingroup%
    \pgf@prepare@start@of@path%
\fi%
```

在 \ifpgfutil@tempswa 的真值是 true 时，准备路径上的箭头，先准备起点处的箭头。

(f) 执行 \pgfsyssoftpath@invokecurrentpath→P. 220, 插入当前的软路径，

(g) 执行 \pgf@up@action, 此时的 \pgf@up@action 可能等效于 \pgfsys@fillstroke 或者 \pgfsys@stroke 或者 \pgfsys@fill.

(h) 执行

```
\ifdim\pgfinnerlinewidth>0pt\relax%
  \pgf@stroke@inner@line%
\fi%
```

检查画双线。

(i) 执行

```
\ifpgfutil@tempswa%
    \pgf@add@arrow@at@start%
  \endgroup%
  \pgf@add@arrow@at@end→P. 306%
\fi%
```

画出起点与终点处的箭头。先在一个组内画起点箭头，再画终点箭头。

- 如果 \pgf@up@clip 不等于 \pgfutil@empty, 则执行
  (a) \pgfsyssoftpath@invokecurrentpath, 插入当前软路径。
  (b) \pgfsys@clipnext, 对之后的路径做剪切。
  (c) \pgf@up@action, 此时的 \pgf@up@action 可能等效于 \pgfsys@fillstroke, \pgfsys@fill, \pgfsys@stroke.
  (d) 设置真值 \pgf@relevantforpicturesizefalse.

13. \pgf@up@bb, 相等于 \pgf@relevantforpicturesizefalse 或者 \pgfutil@empty.
14. 执行 \pgfsyssoftpath@setcurrentpath\pgfutil@empty, 清空软路径。
15. \pgf@resetpathsizes, 重置记录路径边界的寄存器值，见文件《pgfcorepathconstruct.code.tex》:

```
\def\pgf@resetpathsizes{%
  \global\pgf@pathmaxx=-16000pt\relax%
  \global\pgf@pathminx=16000pt\relax%
  \global\pgf@pathmaxy=-16000pt\relax%
  \global\pgf@pathminy=16000pt\relax%
}
```

16. \ignorespaces

\pgf@up@draw@arrows@only

此命令的定义是:

```
\def\pgf@up@draw@arrows@only{%
  \pgfgetpath\pgf@arrowpath%
  \pgfutil@tempswatrue%
  \pgfprocesscheckclosed→P. 305{\pgf@arrowpath}{\pgfutil@tempswafalse}%
  \pgf@path@check@proper→P. 305%
  \ifpgfutil@tempswa%
    \pgf@check@for@arrow@and@animation→P. 305%
    \pgf@prepare@end@of@path%
    \begingroup%
      \pgf@prepare@start@of@path%
      \pgf@add@arrow@at@start%
    \endgroup%
    \pgf@add@arrow@at@end%
  \fi%
}
```

可见此命令只是处理箭头。

\pgfprocesscheckclosed{⟨*soft path*⟩}{⟨*code*⟩}

参数 ⟨*soft path*⟩ 是软路径，如果这个软路径包含 close path token, 则执行 ⟨*code*⟩.

\pgf@check@for@arrow@and@animation

本命令检查路径是否带有箭头和动画选项。如果同时使用了动画和箭头选项，则报错。

\ifpgfutil@tempswa

如果需要给路径加箭头，则对应真值 \pgfutil@tempswatrue.

在文件《pgfutil-common.tex》中有：

```
\newif\ifpgfutil@tempswa
```

\pgf@path@setup@tempswa

此命令的定义是：

```
\def\pgf@path@setup@tempswa{%
  \ifnum\pgf@tips@mode>0\relax
    \pgfutil@tempswatrue%
  \else
    \pgfutil@tempswafalse%
  \fi%
}
```

此命令检查 \pgf@tips@mode 保存的整数值，再决定 \ifpgfutil@tempswa 的真值。

\pgf@path@check@proper

本命令处理软路径 \pgf@arrowpath, 其定义是：

```
\def\pgf@path@check@proper{%
  \ifpgfutil@tempswa%
    \ifnum\pgf@tips@mode>2\relax%
      \pgf@path@check@proper@%
    \fi%
  \fi%
}
```

可见在 \ifpgfutil@tempswa 的真值是 true, 并且 \pgf@tips@mode 的值 >2 的情况下 (用选项 /pgf/tips=on proper draw 或 /pgf/tips=proper)，本命令才有效。

\pgf@path@check@proper@

本命令处理软路径 \pgf@arrowpath, 影响本命令结果的是软路径的最后一个 subpath, 对于这个 subpath 来说,如果它的所有构造点的坐标都一样 (所有构造点都重合),则此命令设置真值 \pgfutil@tempswafalse, 否则设置真值 \pgfutil@tempswatrue.

\ifpgf@worry

如果需要对路径的始端或末端做裁剪，则对应 \pgf@worrytrue.

\pgf@prep@curveend

这个命令在文件《pgfmodulebending.code.tex》中定义：

```
\def\pgf@prep@curveend{
  \pgftransformreset%
  \pgfsetcurvilinearbeziercurve{\pgfpointlastonpath}{\pgfpointsecondlastonpath}{
  ↪  \pgfpointthirdlastonpath}{\pgfpointfourthlastonpath}
}%
```

参考 \pgfsetcurvilinearbeziercurve→P. 556.

\pgf@prepare@end@of@path

本命令处理软路径 \pgf@arrowpath, 对路径的末端做裁剪。

1. 设置 \pgf@precise@shorteningfalse
2. 计算 \pgf@arrow@compute@shortening→P. 343\pgf@end@tip@sequence→P. 325, 计算结果是,\pgf@xa 保存箭头序列对路径的裁剪长度，\pgf@xb 保存箭头序列的总长度。
3. 执行

   ```
   \advance\pgf@xa by\pgf@shorten@end@additional%
   \advance\pgf@xb by\pgf@shorten@end@additional%
   \ifdim\pgf@xa=0pt\relax\else\pgf@worrytrue\fi
   ↪  % Also, worry if shortening is requested
   \edef\pgf@path@shortening@distance{\the\pgf@xa}%
   \edef\pgf@arrow@tip@total@length{\the\pgf@xb}%
   ```

   也就是把 \pgf@shorten@end@additional 加到 \pgf@xa 和 \pgf@xb 中，然后分别保存。

   - \pgf@path@shortening@distance
   - \pgf@arrow@tip@total@length
4. 假设需要对路径做裁剪，则
   (a) 将软路径分裂开，参考 \pgfprocesssplitpath→P. 226, \pgfprocesssplitsubpath→P. 227,
   (b) 单独分析最后的那一段子路径，
   (c) 分直线和曲线两种情况做裁剪。
   注意，如果此时“最后的那一段子路径”的长度小于 \pgf@path@shortening@distance, 可能会导致意外情况。
   (d) 将裁剪后的软路径再保存到 \pgf@arrowpath.

\pgf@add@arrow@at@end

本命令在路径末端画出箭头。

```
\def\pgf@add@arrow@at@end{%
  \ifx\pgf@arrowpath\pgfutil@empty\else%
    \ifx\pgf@end@tip@sequence\pgfutil@empty\else%
      \pgf@do@draw@end%
    \fi%
  \fi%
}
```

在执行 \pgf@parse@end 时，\pgf@do@draw@end 可能被 let 为

```
\let\pgf@do@draw@end\pgf@do@draw@straightend%
```

或者

```
\let\pgf@do@draw@end\pgf@do@draw@curvedend%
```

\pgf@do@draw@straightend

本命令在一个直线段末端直接加箭头。本命令调用 \pgf@arrow@draw@arrow→P. 343.

```
\def\pgf@do@draw@straightend{%
  {%
    \pgftransformreset%
    \pgftransformarrow{\pgfqpoint{\pgf@xc}{\pgf@yc}}{\pgfqpoint{\pgf@xb}{\pgf@yb}}
    ↪  %
    \pgf@arrow@draw@arrow\pgf@end@tip@sequence\pgf@arrow@tip@total@length%
  }%
}
```

\pgf@do@draw@curvedend

这个命令在文件《pgfmodulebending.code.tex》中定义。

第十六章　定义新的箭头

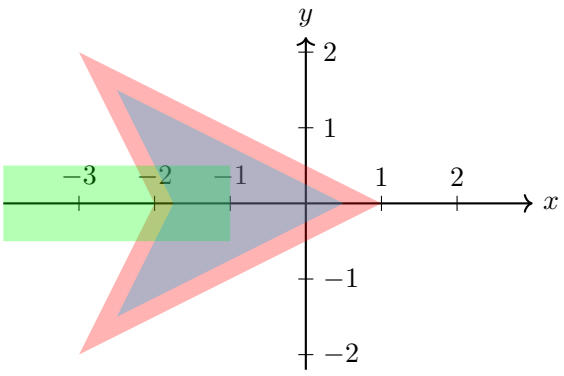
16.1　Overview

§16 介绍了 Ti*k*Z 中箭头的用法，不过 Ti*k*Z 并不对箭头做任何事情，处理箭头的是 PGF. 在 PGF 中，箭头是“meta-arrows”，就像在 TEX 中字体是“meta-fonts”. 一个 meta-font 并不是一种通常意义上的具体的字体，实际上它是个“蓝图”（blueprint），或者说是一组程序，它能将实际输出的符号调整到指定的尺寸、形态。举例说，“Berlin”是正常尺寸（10pt）的单词，而“**Berlin**”是 tiny 尺寸（5pt）的单词放大 1 倍后的效果，显然，同一个单词从 tiny 尺寸变到正常尺寸，不是简单地放大 1 倍。

PGF 的 meta-arrows 的作用也是类似的。箭头的形态受到多个参数的影响，其中添加箭头的路径的线宽是个因素。5pt 线宽路径的箭头的尺寸，并不是 1pt 线宽路径的箭头尺寸的 5 倍，而是小于 5 倍。你可以想象实际的处理过程比较复杂。

16.2　有关术语

路径和路径上的箭头有几个特征点，参照下图：



上图的意思是给路径 `(-4,0)--(1,0)` 的端点加箭头。绿色线代表需要添加箭头的路径，红色线代表箭头的轮廓线，青色区域是箭头内部的填充色。定义箭头时需要用绘图代码规定箭头的轮廓，这些绘图代码所在的坐标系叫做“箭头坐标系”（arrow tip’s coordinate system）。程序先开启箭头坐标系，按代码画出箭头，然后经过一些必要的变换，再添加到路径上，一般会使得箭头的尖端与路径原来的端点重合。注意绘制路径的代码所在的坐标系与“箭头坐标系”是不同的。上面图形中，把路径与箭头画在同一个坐标系内，只是为了方便而已。不过下面与箭头有关的“特征点”都是在“箭头坐标系”中描绘的。

- 点 $(1,0)$ 叫作 tip end，即箭头的尖端，总是假定该点位于箭头坐标系的 x 轴上。
- 点 $(-3,0)$ 叫作 back end，总是假定该点位于箭头坐标系的 x 轴上。
- 点 $(-1,0)$ 叫作 line end，即路径端点，总是假定该点位于箭头坐标系的 x 轴上。原来的端点是 $(1,0)$，添加箭头后路径的端点有所移动，使得路径被裁去一段。如果不裁去一段，因为该路径线宽是 10mm，那么路径会突出箭头边界之外。

- 点 $(-2,0)$ 叫作 visual back end，总是假定该点位于箭头坐标系的 x 轴上。
- 与 visual back end 相对应，也有一个 visual tip end，总是假定该点位于箭头坐标系的 x 轴上。上图中 visual tip end 与 tip end 重合。
- 顶点，convex，上图箭头的凸顶点有 3 个，$(1,0)$，$(-3,2)$，$(-3,-2)$，这三个点可以决定箭头的“尺寸”。一般情况下，PGF 会追踪画出的任何路径的边界盒子。但是 PGF 缓存（cache）箭头的有关代码时，并不能确定箭头的尺寸。因此必须显式地提供箭头的凸顶点来确定箭头尺寸。

还要注意箭头线宽 (arrow line width), 即箭头轮廓线的线宽，上图中就是红色线的线宽。注意箭头的特征点都位于箭头轮廓线的外缘上，就像 node 的多数 anchor 位置都位于其背景路径 (形状路径) 的线宽的外缘上一样。因此在自定义箭头时，这一点会增加计算量。

16.3 pgf 处理箭头的一般过程

1. 首先定义箭头。定义箭头的命令是 `\pgfdeclarearrow{`⟨*config*⟩`}`, 在 ⟨*config*⟩ 中需要设置关于箭头的 `name`，`parameters`，`setup code`，`drawing code`，`defaults`, 等等。这个命令保存箭头的定义，但不画出箭头，也不会进一步处理定义代码。本命令的参数 `name=foo` 声明一个箭头名称，这个名称在本命令之后是可以用的。

2. 定义箭头后就可以使用箭头，即给路径的端点添加箭头。假设定义的箭头名称是 `name=foo`, 例如所添加的箭头是 `foo[length=5pt,open]`, 这里不仅有箭头名称，也有箭头选项。箭头名称、箭头选项（键值对）会被当作一个“整体组合”；如果两个组合的箭头名称、箭头选项相同，但选项的值不同，也会被当作不同的组合。

3. 当一个箭头及其选项的组合，例如，`foo[length=5pt,open]` 第一次被使用时，PGF 首先会解析箭头及其选项，参考 `\pgfarrows@initial@parser` →P. 328, 将声明箭头时保存的代码调出来，将箭头的选项所保存的代码单独保存。在绘制箭头时，执行 `\pgf@arrow@drawer` →P. 343, 或者 `\pgf@do@draw@curvedend` →P. 307, 导致先执行 `defaults` 选项保存的默认的箭头选项，再执行箭头选项 `[length=5pt,open]`, 再执行 `setup code`, 见 `\pgfarrows@getid` →P. 340. 此时 `setup code` 有两个作用：

   - 第一，计算箭头的特征点；
   - 第二，为画出箭头做准备性的计算。

   `setup code` 并不画出箭头，只是为“画出箭头”做一些准备性工作。`setup code` 的准备结果应当转存到 `\pgf@arrows@saves` →P. 339 中。

4. 然后，PGF 会利用箭头名称、选项值等信息构造一个箭头组合的“全名”（这个全名就代表这个箭头组合），并把这个全名做成一个控制序列，这个控制序列全局地保存箭头组合的序号（每个箭头组合都有自己的序号）。

5. 缓存 `drawing code`, 即绘制箭头的路径命令，此时 `drawing code` 会接收之前的默认箭头选项，箭头选项，`setup code` 的计算结果。`drawing code` 会被放入一个“沙盒”（sandbox）中来执行，其中使用底层（basic layer）绘图命令，并将绘图结果“缓存”（cache）。

   不过在两种例外情况下，`drawing code` 只是被保存，而不是被缓存为底层的代码：

   - 第一，在声明箭头时使用了 `cachable=false` 选项。如果 `drawing code` 中含有底层绘图命令不能接受的内容（例如含有 `\pgftext` 添加的文本），就需要使用这个选项；
   - 第二，在声明箭头时使用了 `bend` 选项，因为需要计算路径的曲率来实现 bend 效果，而各个路径的曲率又是不固定的，所以需要针对各个路径来分别执行 `drawing code`.

6. 然后绘制箭头，PGF 先做变换，如平移（shift）、倾斜（slant）、颠倒（yscale=-1），然后再执行 \pgf@arrows@saves→P. 339, 再执行之前保存的绘制箭头的代码。此时如果能修改 \pgf@arrows@saves→P. 339 这个宏，就可以修改 drawing code.

7. 当第二次使用同一个箭头名称、箭头选项的组合时，PGF 仍然会解析箭头、选项（包括默认选项），执行选项保存的代码，然后构造一个“全名”，然后再检查这个“全名”是否已经被定义为一个控制序列。如果这个全名是之前定义过的，那么就不必再次执行 setup code. 不过 PGF 仍然会做变换，如平移（shift）、倾斜（slant）、颠倒（yscale=-1），再执行 \pgf@arrows@saves→P. 339, 再执行之前保存的绘制箭头的代码。

16.4 自定义箭头

参考《pgfcorearrows.code.tex》。

使用命令 \pgfdeclarearrow 声明（定义）箭头。箭头分 4 种：

1. 点号“.”代表的箭头，这是一种特殊箭头。
2. single char 箭头，即单字符箭头，箭头名称是某个字符。
3. shorthand 箭头，即用已定义的一个或数个箭头组合成一种箭头，例如

```
\pgfdeclarearrow{ name=goo, means = {Bar[length=0.5cm]} }
```

这样定义的箭头 goo 是一种 shorthand 箭头，其中使用选项 means, 不使用选项 drawing code.

4. meta 箭头，例如

```
\pgfdeclarearrow{
  name = foo, % 箭头名称为 foo
  parameters = { ... },
  setup code = { ... },
  drawing code = { ... },
  defaults = { length = 4cm }
}
```

这样定义的箭头 foo 是一种 meta 箭头，其中使用选项 drawing code, 不使用选项 means.

注意命令 \pgfdeclarearrow 不允许对“同一箭头名称”做重复声明。

如果箭头 ⟨*end name*⟩ 是 meta 箭头，那么控制序列 \csname pgf@ar@code@⟨*end name*⟩\endcsname 就不等于 \relax, 此时就不能再次对 ⟨*end name*⟩ 做声明。

如果箭头 ⟨*end name*⟩ 是 shorthand 箭头,那么控制序列 \csname pgf@ar@means@⟨*end name*⟩\endcsname 就不等于 \relax, 此时就不能再次对 ⟨*end name*⟩ 做声明。

16.4.1 新定义一种 meta 箭头

定义一种 meta 箭头的一般方式是：

```
\pgfdeclarearrow{
  name = ⟨name⟩,
  parameters = { ⟨parameters⟩ },
  setup code = { ⟨setup code⟩ },
  drawing code = { ⟨drawing code⟩ },
  defaults = { ⟨default arrow options⟩ },
  cache=⟨true or false⟩,
  bending mode=⟨mode⟩
}
```

\pgfdeclarearrow 的参数是以下选项：

- /pgf/@arrows decl/name→P. 321, 声明箭头的名称，其值可以是
  (i) -⟨*end name*⟩
  (ii) ⟨*start name*⟩-⟨*end name*⟩
- /pgf/@arrows decl/parameters→P. 323, 这个选项的参数用于构造箭头组合的“全名”，使得这个全名更加具有独特性。
- /pgf/@arrows decl/setup code→P. 322, 为 drawing code 做准备。

  在 ⟨*setup code*⟩ 中可以使用下面的命令：
  - \pgfarrowssettipend→P. 340
  - \pgfarrowssetbackend→P. 340
  - \pgfarrowssetlineend→P. 340
  - \pgfarrowssetvisualbackend→P. 340
  - \pgfarrowssetvisualtipend→P. 340
  - \pgfarrowshullpoint→P. 339
  - \pgfarrowsupperhullpoint→P. 339
  - \pgfarrowssave→P. 339
  - \pgfarrowssavethe→P. 339

  在 ⟨*drawing code*⟩ 中可能要用到由 ⟨*setup code*⟩ 计算出来的尺寸，或者在 ⟨*setup code*⟩ 中定义的宏，此时，最好使用 \pgfarrowssavethe→P. 339 或者 \pgfarrowssave→P. 339 来转存将要用到的尺寸或宏定义。
- /pgf/@arrows decl/drawing code→P. 322, 绘制箭头的代码，其中可以使用一些变量，这些变量能够由 setup code, 箭头选项决定。

  ⟨*drawing code*⟩ 所绘制的箭头应该是“沿着 x 轴指向右方的”，在实际画出箭头时，PGF 会对绘制的箭头做画布变换，使之旋转到某个角度并添加到路径端点处。通常箭头的指向沿着路径的方向。

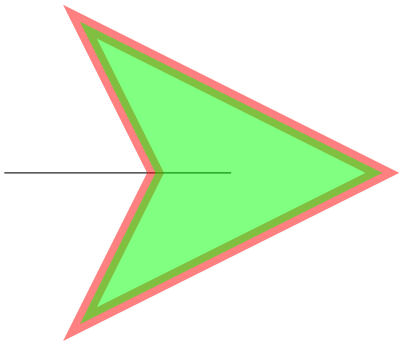
  在 ⟨*drawing code*⟩ 中需要注意：
  (i) 在 ⟨*drawing code*⟩ 中不要使用 \pgfusepath 命令，此命令会试图给箭头添加箭头，导致一种循环，导致错误。你可以使用“quick”版的命令，例如 \pgfusepathqstroke→P. 389, 使用这种命令不会出现给箭头添加箭头的情况。
  (ii) 如果在 ⟨*drawing code*⟩ 中使用 \pgfusepathqstroke, 你可能需要先设置线型为实线，并设置线冠和线结合的样式，使得箭头外观总是保持一致。
  (iii) 当 ⟨*drawing code*⟩ 被执行时，⟨*drawing code*⟩ 会被放入一个底层的域（scope）中，不会对域外的处理产生副作用。
  (iv) 当第一次执行 ⟨*drawing code*⟩ 时，高层的坐标变换矩阵会被设为单位矩阵。
- /pgf/@arrows decl/defaults→P. 323, 这个选项保存一些默认的箭头选项，这些箭头选项的路径应该是 /pgf/arrow keys.
- /pgf/@arrows decl/cache→P. 322, 这个选项决定是否对 ⟨*drawing code*⟩ 做最底层的缓存。
- /pgf/@arrows decl/bending mode→P. 322, 这个选项为箭头设置一种 bending mode.

对于前面的例子来说，可以用下面的代码来实现：

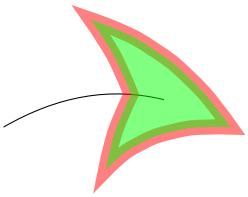
```
\pgfdeclarearrow{
  name = foo,  % 箭头名称为 foo
```

```
  parameters = { \the\pgfarrowlength },  % 只有一个长度参数宏
  setup code = {
% 用长度宏指定特征点
    \pgfarrowssettipend{.25\pgfarrowlength}
    \pgfarrowssetlineend{-.25\pgfarrowlength}
    \pgfarrowssetvisualbackend{-.5\pgfarrowlength}
    \pgfarrowssetbackend{-.75\pgfarrowlength}
% 指定凸顶点
    \pgfarrowshullpoint{.25\pgfarrowlength}{0pt}
    \pgfarrowshullpoint{-.75\pgfarrowlength}{.5\pgfarrowlength}
    \pgfarrowshullpoint{-.75\pgfarrowlength}{-.5\pgfarrowlength}
% 另存长度宏
    \pgfarrowssavethe\pgfarrowlength
  },
% 绘制箭头路径的底层命令
  drawing code = {
    \pgfpathmoveto{\pgfpoint{.25\pgfarrowlength}{0pt}}  % 使用坐标命令 \pgfpoint
    \pgfpathlineto{\pgfpoint{-.75\pgfarrowlength}{.5\pgfarrowlength}}
    \pgfpathlineto{\pgfpoint{-.5\pgfarrowlength}{0pt}}
    \pgfpathlineto{\pgfpoint{-.75\pgfarrowlength}{-.5\pgfarrowlength}}
    \pgfpathclose
    \pgfsetstrokecolor{red}  % 箭头边界路径颜色为红色
    \pgfsetlinewidth{2mm}
    \pgfsetstrokeopacity{.5}
    \pgfusepathqstroke  % 画出箭头边界路径
    \pgfpathmoveto{\pgfqpoint{.25\pgfarrowlength}{0pt}}
    ↪  % 使用坐标命令 \pgfqpoint，与 \pgfpoint 等效
    \pgfpathlineto{\pgfqpoint{-.75\pgfarrowlength}{.5\pgfarrowlength}}
    \pgfpathlineto{\pgfqpoint{-.5\pgfarrowlength}{0pt}}
    \pgfpathlineto{\pgfqpoint{-.75\pgfarrowlength}{-.5\pgfarrowlength}}
    \pgfpathclose
    \pgfsetfillcolor{green}  % 箭头填充颜色为绿色
    \pgfsetfillopacity{.5}
    \pgfusepathqfill  % 填充箭头
  },
% 设置长度选项的默认值
  defaults = { length = 4cm }
}
```

这样定义后，就可以使用箭头 foo 了，如下：



```
\tikz\draw[-{foo[color=blue]}] (0,0)--(5,0);
% 这里的选项 color=blue 没有作用
```



```
\tikz \draw [-{foo[length=2cm,bend]}]
  (0,0) to [bend left] (3,0);
```

16.4.2 定义一个 Shorthand 箭头

可以用已有的箭头来定义一个 Shorthand 箭头，用到命令 \pgfdeclarearrow, 以及该命令的选项 name 和 means.

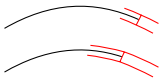
```
\pgfdeclarearrow{name=⟨shorthand arrow name⟩, means=⟨end arrow specification⟩}
```

- 首先，使用选项 name 指定 shorthand 箭头的名称。
- 然后使用选项 means=⟨*end arrow specification*⟩ 规定箭头。⟨*end arrow specification*⟩ 中使用已定义的箭头以及箭头选项来指定一种箭头样式，所指定的是路径末端的箭头样式。如果 shorthand 箭头用于路径始端，那么所添加的箭头是末端的 ⟨*end arrow specification*⟩ 样式的翻转。例如：

  ```
  \pgfdeclarearrow{ name=goo, means = bar[length=2cm+\mydimen] }
  ```

  这样定义后，当给路径添加箭头 goo 时，实际添加的箭头是代码 bar[length=2cm+\mydimen] 规定的箭头，其中 bar 应当是某个已定义的箭头名称。

  当命令 \pgfdeclarearrow 被执行时，⟨*end arrow specification*⟩ 会被立即执行，得到箭头选项的缓存 (caches). 注意：第一，⟨*end arrow specification*⟩ 中使用的箭头必须是已定义的；第二，⟨*end arrow specification*⟩ 中使用的箭头选项会被立即执行,例如,对于上面的例子来说,bar[length=2cm+\mydimen] 会被立即执行，所以 \mydimen 会被展开，因此选项 length 是个具体的尺寸，这个尺寸可以看作是箭头 goo 的“默认长度”。之后虽然可以改变 \mydimen 的值，但这并不能影响这个“默认长度”。当然使用 goo[length=] 仍然可以改变箭头 goo 的实际长度。



```
\pgfdeclarearrow{ name=goo, means = {Bar[length=0.5cm]} }
\tikz \draw [-{goo[bend,color=red]}]
  (0,0) to [bend left] (2,0);\par
\tikz \draw [-{goo[bend,color=red,length=1cm]}]
  (0,0) to [bend left] (2,0);
```

利用手柄 /.tip 也可以定义 shorthand 箭头。

16.5 关于箭头的选项

给路径添加箭头时，箭头可能会带有选项，即 key，这些选项影响箭头的外观、特征尺寸。在定义箭头的命令 \pgfdeclarearrow 中，parameters，setup code，drawing code 都会涉及与箭头选项有对应关联的参数宏。当使用选项时，例如使用长度选项 foo[length=1cm]，选项的值 1cm 会被赋予尺寸寄存器 \pgfarrowslength, 从而影响对 parameters，setup code，drawing code 的计算结果。

16.5.1 尺寸选项

有的箭头选项，如 length, width, 会设定某个 TEX 寄存器的值。例如，与 length 对应的 TEX 寄存器是 \pgfarrowslength, length 的值会变成 \pgfarrowslength 的值。

以下是 TEX 尺寸对应的选项：

- 寄存器 \pgfarrowslength 对应选项 length 和 angle.
- 寄存器 \pgfarrowswidth 对应选项 width, width' 和 angle.
- 寄存器 \pgfarrowsinset 对应选项 inset 和 inset'.
- 寄存器 \pgfarrowslinewidth 对应选项 line width 和 line width'.

如果 setup code 或 drawing code 中用到了某个寄存器，或者希望在箭头选项中使用相应的选项，就应当把相应的寄存器写入 parameters 的列表中。例如：

```
parameters={\the\pgfarrowlength \the\pgfarrowwidth}  或者
parameters={\the\pgfarrowlength, \the\pgfarrowwidth}
```

16.5.2 True-False 选项

有的箭头选项的值是 true 或 false, 例如选项 reversed, 它们都设定某个对应的 TEX-if 条件判断宏的真值。

- \ifpgfarrowreversed 对应选项 reversed.
- \ifpgfarrowswap 对应选项 swap 和 right.
- \ifpgfarrowharpoon 对应选项 harpoon, left 和 right.
- \ifpgfarrowroundcap 的 true 值对应选项 line cap=round, round; 其 false 值对应选项 line cap=butt, sharp.
- \ifpgfarrowroundjoin 的 true 值对应选项 line join=round, round; 其 false 值对应选项 line join=miter, sharp.
- \ifpgfarrowopen 的 true 值对应选项 fill=none, open; 其 false 值对应选项 fill=⟨*color*⟩, color.

如果 setup code 或 drawing code 中用到了某个 TEX-if 条件判断宏，或者希望在箭头选项中使用相应的选项，就应当把相应的宏写入 parameters 的列表中。例如：

```
parameters = { \the\pgfarrowlength,...,
               \ifpgfarrowharpoon h\fi\
               \ifpgfarrowroundjoin j\fi}
```

在上面的格式中 \ifpgfarrowharpoon h\fi 的作用是，若 \ifpgfarrowharpoon 的值为 true, 则得到一个字符 h, 这个字符 h 将用于构成箭头组合的“全名”。

注意在这个格式中，不同的 TEX-if 条件判断宏应该对应不同的字符，使它们相互区别。

选项 reversed 的作用是通过（在箭头坐标系中）将箭头做关于 y 轴对称的变换实现的；swap 的作用是通过（在箭头坐标系中）将箭头做关于 x 轴对称的变换实现的。

当选项 reversed 的作用实现后，原箭头的各种特征点的横坐标都变成对应点的相反数，箭头的 back end 与 tip end 互换并取相反数，箭头的 visual back end 与 visual tip end 互换并取相反数。但是注意，line end 也变成原值的相反数，这可能导致路径线宽突出箭头轮廓之外。因此在定义箭头时，要仔细考虑各种情况，合理设置各个特征点的位置，此时，你可能需要（在 setup code 中）用一个条件句，根据 \ifpgfarrowreversed 的不同值来分别设置 line end 的位置。

16.5.3 setup code 中不能引用的选项

有的选项影响箭头的外观，但不能在 setup code 中列出。

- quick, flex, flex', bend, 这些选项会影响箭头的旋转，见 §16.3.8.
- color=⟨*color*⟩, fill=⟨*color*⟩.
- sep.

16.5.4 自定义箭头选项

预定义的箭头选项已经很多了，但有时你可能需要自己设计某个箭头选项来方便地实现某种效果。

假设你需要定义一个选项 depth, 那么你应当引入一个寄存器或者宏来保存该选项的值，例如：

```
\newdimen\pgfarrowdepth  % 使用一个尺寸寄存器
```

而且你还需要用命令 \pgfkeys 定义一个 key:

```
/pgf/arrow keys/depth
```

为了使得选项 depth 能够修改命令 \pgfarrowdepth 的值，并且能让这个命令对箭头的位置、形态产生影响，需要仔细地设置一些代码。PGF 的 key 操作的功能很强，但所需的代价也较高，所以应尽量减少 key 操作的次数。因此有“选项缓存”，即对于每一组“选项、选项值”，如 foo[⟨*options*⟩], ⟨*options*⟩ 只被执行一次，并将执行结果缓存起来，以备后来使用，参考 \pgf@arrows@options→P. 332. 为了能让选项 depth 进入“缓存”（即添加到 \pgf@arrows@options→P. 332 中），要使用 \pgfarrowsaddtooptions→P. 332, \pgfarrowsaddtolateoptions→P. 333.

定义选项时可以使用下面的命令：

- \pgfarrowsaddtooptions→P. 332, 例如对于前面需要定义的选项 depth, 可以设置：

  ```
  \pgfkeys{/pgf/arrow keys/depth/.code=
    \pgfarrowsaddtooptions{\pgfmathsetlength{\pgfarrowdepth}{#1}}
  ```

  这样设置后，每当使用选项 depth 时，程序都会执行 \pgfmathsetlength 来为 \pgfarrowdepth 赋值，这样选项 depth 与命令 \pgfarrowdepth 就对应起来了。

  不过执行命令 \pgfmathsetlength 所需的代价较高，为了减少执行这个命令的次数，可以使用下面的代码：

  ```
  \pgfkeys{/pgf/arrow keys/depth/.code=
    \pgfmathsetlength{\somedimen}{#1}
    \pgfarrowsaddtooptions{\pgfarrowdepth=\somedimen}
  ```

  其中的 \somedimen 应该是某个寄存器。这样设置的想法是：执行命令 \pgfmathsetlength, 把选项 depth 的值赋予寄存器 \somedimen, 再把寄存器 \pgfarrowdepth 和 \somedimen 联系起来，直接在两个寄存器宏之间进行操作，运行起来就会快一些。

  尽管这样的想法很好，但是这样的设置仍然不可用，因为在读取并执行选项缓存时，缓存中的寄存器 \somedimen 的值可能已经发生变化 (不再等于选项 depth 的值)。为此可以使用下面的代码：

  ```
  \pgfkeys{/pgf/arrow keys/depth/.code=
    \pgfmathsetlength{\somedimen}{#1}
    \expandafter\pgfarrowsaddtooptions\expandafter{\expandafter\pgfarrowdepth
    ↪  \expandafter=\the\somedimen}
  ```

  上面代码利用 \expandafter 把寄存器 \somedimen 的展开值（而不是直接把寄存器 \somedimen）赋予寄存器 \pgfarrowdepth, 这样缓存中的寄存器 \somedimen 的值就是一个确定的尺寸了。

- \pgfarrowsaddtolateoptions→P. 333, 这个命令类似上面的 \pgfarrowsaddtooptions.

  有的箭头选项，例如 width', 该选项的使用格式是

  ```
  width'=⟨dimension⟩ ⟨length factor⟩ ⟨line width factor⟩
  ```

  其中的 ⟨*length factor*⟩ 是要用与箭头长度 length 相乘的，因此，如果存在这个 ⟨*length factor*⟩, 那么在程序处理箭头长度 length 的值之前（当然应当事先规定默认长度值），选项 width' 是不能被执

行的。所以，选项 width' 就在其它（没有 late 特征的）选项（包括 length）被执行之后才被执行。

所以，对于没有 late 特征的选项最好指定其默认值，这样在定义具有 late 特征的选项时就可引用这些选项对应的宏。

- \pgfarrowsaddtolengthscalelist → P. 332
- \pgfarrowsaddtowidthscalelist → P. 332
- \pgfarrowsthreeparameters → P. 333
- \pgfarrowslinewidthdependent → P. 333
- \pgfarrowslengthdependent → P. 333

选项 /pgf/arrow keys/length 的定义是（见文件 pgflibraryarrows.meta.code）：

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  length/.code={%
    \pgfarrowsthreeparameters{#1}%
    \expandafter\pgfarrowsaddtooptions\expandafter{
    ↪  \expandafter\pgfarrowslinewidthdependent\pgfarrowstheparameters\pgfarrowlength\pgf@x
    ↪  }%
  },
  ......
}
```

仿照这个定义，可以定义依赖那三个参数的选项 depth, 代码如下：

```
\pgfkeys{/pgf/arrow keys/depth/.code={%
  \pgfarrowsthreeparameters{#1}%
  \expandafter\pgfarrowsaddtolateoptions% 使用有 late 特征的命令
  \expandafter{\expandafter\pgfarrowslinewidthdependent\pgfarrowstheparameters
  ↪  \pgfarrowdepth\pgf@x}%
}
```

当执行 \pgfkeys{/pgf/arrow keys/depth=⟨x⟩ ⟨y⟩ ⟨z⟩} 时，导致以下操作步骤：

1. 执行 \pgfarrowsthreeparameters{⟨x⟩ ⟨y⟩ ⟨z⟩}, 把解析 ⟨x⟩, ⟨y⟩, ⟨z⟩ 的结果保存到 \pgfarrowstheparameters 中。
2. 执行

   ```
   \pgfarrowsaddtolateoptions{\pgfarrowslinewidthdependent{⟨x⟩}{⟨y⟩}{⟨z⟩}
   ↪  \pgfarrowdepth\pgf@x}
   ```

   命令 \pgfarrowslinewidthdependent{⟨x⟩}{⟨y⟩}{⟨z⟩} 计算 \pgf@x 的值，然后把 \pgf@x 的值赋予 \pgfarrowdepth.

16.6 文件《pgfcorearrows.code.tex》

16.6.1 声明箭头

\pgfdeclarearrow{⟨*config*⟩}

这个命令有两个用处，新定义一个箭头，以及用已有的箭头来定义一个“shorthand”.

注意在这个命令内部不能有空行，并且这个命令的有效范围受到 TEX 分组的限制。

此命令的定义是：

```
\def\pgfdeclarearrow#1{%
  \let\pgf@decl@arrow@defaults\pgfutil@empty%
  \let\pgf@decl@arrow@means\relax%
  \let\pgf@decl@arrow@code\relax%
  \pgfkeys{/pgf/@arrows decl/.cd,
    name=,
    setup code=,
    cache=true,
    bending mode=orthogonal,
    parameters=,%
    #1%
  }%
  \ifx\pgf@decl@arrow@name@end\pgfutil@empty%
    \pgferror{Declaring unnamed arrow}%
  \else%
    \pgf@arrow@letter{name@end}%
    \pgf@arrow@letter{name@start}%
    \expandafter\let\csname pgf@ar@start@\pgf@decl@arrow@name@start
    ↪  \endcsname\pgf@decl@arrow@name@end%
    \expandafter\let\expandafter\pgf@old@code\csname pgf@ar@code@
    ↪  \pgf@decl@arrow@name@end\endcsname%
    \expandafter\let\expandafter\pgf@old@means\csname pgf@ar@means@
    ↪  \pgf@decl@arrow@name@end\endcsname%
    \ifx\pgf@decl@arrow@code\relax%
      \ifx\pgf@decl@arrow@means\relax%
        \pgferror{You must set either the 'means' or the drawing code' keys for
        ↪  arrow tip '\pgf@decl@arrow@name@end'}%
      \else%
        \ifx\pgf@old@code\relax%
          \let\pgf@arrow@tip@sequence\pgfutil@empty%
          \let\pgf@arrow@translate\relax%
          \expandafter\let\expandafter\pgf@arrows@direct\csname
          ↪  pgf@arrows@direct@name@end@\pgf@decl@arrow@means\endcsname%
          \ifx\pgf@arrows@direct\relax%
            \let\pgf@arrows@options\pgfutil@empty%
            \expandafter\pgfarrows@initial@parser\pgf@decl@arrow@means[]\pgf@stop%
          \else%
            \let\pgf@arrow@tip@sequence\pgf@arrows@direct%
          \fi%
          \pgf@arrow@letter@\pgf@arrow@tip@sequence{means}%
          \expandafter\let\csname pgf@ar@code@\pgf@decl@arrow@name@end
          ↪  \endcsname\relax%
        \else%
          \pgferror{Arrow tip '\pgf@decl@arrow@name@end' was already
            defined with key 'drawing code' previously and you cannot change this}
            ↪  %
        \fi%
      \fi%
    \else%
      \ifx\pgf@decl@arrow@means\relax%
        \ifx\pgf@old@means\relax%
          \pgf@arrow@letter{code}%
          \pgf@arrow@letter{defaults}%
          \pgf@arrow@letter{setup}%
          \pgf@arrow@letter{bending@mode}%
```

```
          \pgf@arrow@letter{par}%
          \expandafter\let\csname pgf@ar@do@cache@\pgf@decl@arrow@name@end
          ↪  \endcsname\pgf@decl@arrow@cache%
        \else%
          \pgferror{Arrow tip '\pgf@decl@arrow@name@end' was already
            defined with key 'means' previously and you cannot change this}%
        \fi%
      \else%
        \pgferror{You cannot set both the 'means' and also the 'drawing code'
          keys for arrow tip '\pgf@decl@arrow@name@end'}%
      \fi%
    \fi%
  \fi%
}
```

可见在 ⟨*config*⟩ 中使用的应该是前缀为 `/pgf/@arrows decl` 的选项，如果下文的 `name`，`parameters`，`setup code`，`drawing code` 等，并且应该首先使用选项 `name`.

命令 `\pgfdeclarearrow{`⟨*config*⟩`}` 的处理是：

1. 清空 `\pgf@decl@arrow@defaults`

   ```
   \let\pgf@decl@arrow@defaults\pgfutil@empty%
   ```

   宏 `\pgf@decl@arrow@defaults` 与选项 `/pgf/@arrows decl/defaults` 相关。
2. 清理

   ```
   \let\pgf@decl@arrow@means\relax%
   \let\pgf@decl@arrow@code\relax%
   ```

3. 处理选项

   ```
   \pgfkeys{/pgf/@arrows decl/.cd,
     name=,
     setup code=,
     cache=true,
     bending mode=orthogonal,
     parameters=,%
     ⟨config⟩%
   }%
   ```

   选项 `name=` 导致 `\pgf@decl@arrow@name@end`, `\pgf@decl@arrow@name@start` 都等于 `\pgfutil@empty`.
4. 检查 `\pgf@decl@arrow@name@end` 的值，这个宏由选项 `/pgf/@arrows decl/name` → P. 321 定义，这个选项设置箭头名称 ⟨*start name*⟩ 和 ⟨*end name*⟩.
   - 如果 `\pgf@decl@arrow@name@end` 等于 `\pgfutil@empty`, 即没有设置箭头名称，则报错

     ```
     \pgferror{Declaring unnamed arrow}%
     ```

   - 如果 `\pgf@decl@arrow@name@end` 不等于 `\pgfutil@empty`, 即设置了箭头名称，则

     (a) 执行

     ```
     \pgf@arrow@letter{name@end}%
     导致
     \expandafter\let\csname pgf@ar@name@end@⟨end name⟩\endcsname
     ↪  \pgf@decl@arrow@name@end%
     ```

     用 `\csname pgf@ar@name@end@`⟨*end name*⟩`\endcsname` 保存 ⟨*end name*⟩.

     (b) 执行

```
\pgf@arrow@letter{name@start}%
```
导致
```
\expandafter\let\csname pgf@ar@name@start@⟨end name⟩\endcsname
↪  \pgf@decl@arrow@name@start%
```
用 \csname pgf@ar@name@start@⟨*end name*⟩\endcsname 保存 ⟨*start name*⟩

(c) 执行

```
\expandafter\let\csname pgf@ar@start@⟨start name⟩\endcsname
↪  \pgf@decl@arrow@name@end%
```
用 \csname pgf@ar@start@⟨*end name*⟩\endcsname 保存 ⟨*end name*⟩.

(d) 执行

```
\expandafter\let\expandafter\pgf@old@code\csname pgf@ar@code@⟨end name⟩
↪  \endcsname%
```
用 \pgf@old@code 保存 \csname pgf@ar@code@⟨*end name*⟩\endcsname.
从后文看，控制序列 \csname pgf@ar@code@⟨*end name*⟩\endcsname 的作用是用来检查箭头 ⟨*end name*⟩ 是否已经被声明过的 meta 箭头。如果之前已经用选项 drawing code 成功声明过箭头 ⟨*end name*⟩, 那么这个控制序列不等于 \relax, 此时就不能再次对箭头 ⟨*end name*⟩ 做声明。

(e) 执行

```
\expandafter\let\expandafter\pgf@old@means\csname pgf@ar@means@⟨end name⟩
↪  \endcsname%
```
用 \pgf@old@means 保存 \csname pgf@ar@means@⟨*end name*⟩\endcsname.
如果控制序列 \csname pgf@ar@means@⟨*end name*⟩\endcsname 是已定义的,则它等于 \pgf@arrow@tip@ 见后续的处理。从后文看，控制序列 \csname pgf@ar@means@⟨*end name*⟩\endcsname 的作用是用来检查箭头 ⟨*end name*⟩ 是否已经被声明过的 shorthand 箭头。如果之前已经用选项 means 成功声明过箭头 ⟨*end name*⟩, 那么这个控制序列不等于 \relax, 此时就不能再次对箭头 ⟨*end name*⟩ 做声明。

(f) 检查 \pgf@decl@arrow@code 的值,这个宏由选项 /pgf/@arrows decl/drawing code →P. 322 定义。

- 如果 \pgf@decl@arrow@code 等于 \relax, 即没有用选项 drawing code 设置绘制箭头的代码,再检查 \pgf@decl@arrow@means 的值,这个宏由选项 /pgf/@arrows decl/means →P. 322 定义。
  - 如果 \pgf@decl@arrow@means 等于 \relax, 即没有用选项 means 指定一种 shorthand 箭头，则报错

    ```
    \pgferror{You must set either the 'means' or the drawing code' keys
    ↪  for arrow tip '\pgf@decl@arrow@name@end'}%
    ```
    也就是说，在声明一个新箭头时，至少要用选项 drawing code 或 means 之一，如果都不用就报错。
  - 如果 \pgf@decl@arrow@means 不等于 \relax, 即打算用选项 means 声明一种 shorthand 箭头，再检查 \pgf@old@code 的值，即检查箭头 ⟨*end name*⟩ 是否已经被（选项 drawing code）声明过的 meta 箭头。
    ▶ 如果 \pgf@old@code 等于 \relax, 即箭头 ⟨*end name*⟩ 是尚未被（选项 drawing code）声明过的 meta 箭头，则声明一种 shorthand 箭头 ⟨*end name*⟩, 如下操作

```
\let\pgf@arrow@tip@sequence\pgfutil@empty%
\let\pgf@arrow@translate\relax%
\expandafter\let\expandafter\pgf@arrows@direct\csname
↪  pgf@arrows@direct@name@end@\pgf@decl@arrow@means\endcsname%
\ifx\pgf@arrows@direct\relax%
  \let\pgf@arrows@options\pgfutil@empty%
  \expandafter\pgfarrows@initial@parser\pgf@decl@arrow@means[]
  ↪  \pgf@stop%
\else%
  \let\pgf@arrow@tip@sequence\pgf@arrows@direct%
\fi%
\pgf@arrow@letter@\pgf@arrow@tip@sequence{means}%
\expandafter\let\csname pgf@ar@code@\pgf@decl@arrow@name@end
↪  \endcsname\relax%
```

以上操作定义的控制序列与 shorthand 箭头有关。

如果 ⟨*end name*⟩ 是一种尚未被声明的 shorthand 箭头，则执行 \pgfarrows@initial@parser → P. 328 解析它，将它解析为 meta 箭头以及相应的选项，解析结果是 \pgf@arrow@tip@sequence → P. 324 和 \pgf@arrows@options → P. 332.

▷ \csname pgf@ar@means@⟨*end name*⟩\endcsname 被 let 为 \pgf@arrow@tip@sequence → P. 324

▷ \csname pgf@ar@code@⟨*end name*⟩\endcsname 被 let 为 \relax.

▶ 如果 \pgf@old@code 不等于 \relax, 即箭头 ⟨*end name*⟩ 是已经被（选项 drawing code）声明过的 meta 箭头，则报错

```
\pgferror{Arrow tip '\pgf@decl@arrow@name@end' was already
  defined with key 'drawing code' previously and you cannot change
  ↪  this}%
```

– 如果 \pgf@decl@arrow@code 不等于 \relax, 即用选项 drawing code 设置了绘制箭头的代码，则检查 \pgf@decl@arrow@means 的值，即检查是否还使用了选项 means 来做声明。

∗ 如果 \pgf@decl@arrow@means 等于 \relax, 即没有使用选项 means, 则再检查 \pgf@old@means 的值，即检查箭头 ⟨*end name*⟩ 是否已经被（选项 means）声明过的 shorthand 箭头。

▶ 如果 \pgf@old@means 等于 \relax, 即箭头 ⟨*end name*⟩ 是尚未被（选项 means）声明过的 shorthand 箭头，则声明 meta 箭头 ⟨*end name*⟩, 如下定义控制序列：

▷ \csname pgf@ar@code@⟨*end name*⟩\endcsname

```
\expandafter\let\csname pgf@ar@code@⟨end name⟩\endcsname
↪  \pgf@decl@arrow@code%
```

▷ \csname pgf@ar@defaults@⟨*end name*⟩\endcsname

```
\expandafter\let\csname pgf@ar@defaults@⟨end name⟩\endcsname
↪  \pgf@decl@arrow@defaults%
```

▷ \csname pgf@ar@setup@⟨*end name*⟩\endcsname

```
\expandafter\let\csname pgf@ar@setup@⟨end name⟩\endcsname
↪  \pgf@decl@arrow@setup%
```

▷ \csname pgf@ar@bending@mode@⟨*end name*⟩\endcsname

```
\expandafter\let\csname pgf@ar@bending@mode@⟨end name⟩\endcsname
↪  \pgf@decl@arrow@bending@mode%
```

▷ \csname pgf@ar@par@⟨*end name*⟩\endcsname

```
\expandafter\let\csname pgf@ar@par@⟨end name⟩\endcsname
↪  \pgf@decl@arrow@par%
```

▷ \csname pgf@ar@do@cache@⟨end name⟩\endcsname

```
\expandafter\let\csname pgf@ar@do@cache@⟨end name⟩\endcsname
↪  \pgf@decl@arrow@cache%
```

▶ 如果 \pgf@old@means 不等于 \relax, 即箭头 ⟨end name⟩ 是已经被（选项 means）声明过的 shorthand 箭头，则报错

```
\pgferror{Arrow tip '\pgf@decl@arrow@name@end' was already
  defined with key 'means' previously and you cannot change this}%
```

∗ 如果 \pgf@decl@arrow@means 不等于 \relax, 即使用了选项 means, 则报错

```
\pgferror{You cannot set both the 'means' and also the 'drawing code'
  keys for arrow tip '\pgf@decl@arrow@name@end'}%
```

也就是说，不能同时使用选项 drawing code 和 means 来声明箭头。

\pgf@arrow@letter{⟨string⟩}

这个命令使得

\csname pgf@ar@⟨string⟩@⟨arrow name⟩\endcsname

等于

\csname pgf@decl@arrow@⟨string⟩\endcsname

16.6.1.1 命令 \pgfdeclarearrow 的参数

命令 \pgfdeclarearrow →P. 316 的参数 ⟨config⟩ 是个键值列表，其中可以使用以下键（key）。

/pgf/@arrows decl/name=⟨end name⟩ 或者 ⟨start name⟩-⟨end name⟩ (no default)

本选项用作 \pgfdeclarearrow 的参数中。选项值 ⟨start name⟩, ⟨end name⟩ 都是即将被声明的箭头名称，⟨end name⟩ 是用于路径末端 (end) 的箭头名称，⟨start name⟩ 是用于路径始端 (start) 的箭头名称。

此选项的定义是：

```
\pgfkeys{
  /pgf/@arrows decl/.cd,
  name/.code={%
    \pgfutil@in@-{#1}%
    \ifpgfutil@in@%
      \pgf@arrows@name@decomp#1\pgf@stop%
    \else%
      \def\pgf@decl@arrow@name@end{#1}\let\pgf@decl@arrow@name@start
      ↪  \pgf@decl@arrow@name@end%
    \fi},
}
\def\pgf@arrows@name@decomp#1-#2\pgf@stop{%
  \def\pgf@decl@arrow@name@start{#1}%
  \def\pgf@decl@arrow@name@end{#2}%
}
```

本选项定义 \pgf@decl@arrow@name@end, \pgf@decl@arrow@name@start 这两个宏。

如果是 name=⟨end name⟩ 这个形式，就

```
\def\pgf@decl@arrow@name@end{⟨end name⟩}
\let\pgf@decl@arrow@name@start\pgf@decl@arrow@name@end%
```

如果是 name=⟨*start name*⟩-⟨*end name*⟩ 这个形式，就

```
\def\pgf@decl@arrow@name@start{⟨start name⟩}%
\def\pgf@decl@arrow@name@end{⟨end name⟩}%
```

/pgf/@arrows decl/means=⟨*end arrow specification*⟩ (no default)

这个选项用于定义 Shorthand 箭头。此选项的定义是：

```
\pgfkeys{
  /pgf/@arrows decl/.cd,
  ...
  means/.store in=\pgf@decl@arrow@means,
  ...
}
```

参数 ⟨*end arrow specification*⟩ 保存到宏 \pgf@decl@arrow@means 中。

/pgf/@arrows decl/setup code=⟨*code*⟩ (no default)

此选项的定义是：

```
\pgfkeys{
  /pgf/@arrows decl/.cd,
  ...
  setup code/.store in=\pgf@decl@arrow@setup,
  ...
}
```

参数 ⟨*code*⟩ 保存到宏 \pgf@decl@arrow@setup 中。

/pgf/@arrows decl/drawing code=⟨*code*⟩ (no default)

此选项的定义是：

```
\pgfkeys{
  /pgf/@arrows decl/.cd,
  ...
  drawing code/.store in=\pgf@decl@arrow@code,
  ...
}
```

参数 ⟨*code*⟩ 保存到宏 \pgf@decl@arrow@code 中。

/pgf/@arrows decl/cache=true|false (initially true)

此选项的定义是：

```
\pgfkeys{
  /pgf/@arrows decl/.cd,
  ...
  cache/.store in=\pgf@decl@arrow@cache,
  ...
}
```

真值 true|false 保存到宏 \pgf@decl@arrow@cache 中。本选项的初始值由命令 \pgfdeclarearrow →P. 316 设置。

/pgf/@arrows decl/bending mode=none|orthogonal|polar (initially orthogonal)

此选项的定义是：

```
\pgfkeys{
  /pgf/@arrows decl/.cd,
  ...
```

```
  bending mode/.is choice,
  bending mode/none/.code=\let\pgf@decl@arrow@bending@mode\pgfutil@empty,
  bending mode/orthogonal/.code=\def\pgf@decl@arrow@bending@mode{
  ↪  \pgfpointcurvilinearbezierorthogonal},
  bending mode/polar/.code=\def\pgf@decl@arrow@bending@mode{
  ↪  \pgfpointcurvilinearbezierpolar},
  ...
}
```

本选项有 3 个可用值，本选项定义宏 \pgf@decl@arrow@bending@mode. 本选项的初始值由命令 \pgfdeclarearrow→P. 316 设置。

/pgf/@arrows decl/parameters=⟨*parameters*⟩ (no default)

此选项的定义是：

```
\pgfkeys{
  /pgf/@arrows decl/.cd,
  ...
  parameters/.store in=\pgf@decl@arrow@par,
  ...
}
```

参数 ⟨*parameters*⟩ 保存到宏 \pgf@decl@arrow@par 中。

控制序列 \csname pgf@ar@par@⟨*end name*⟩\endcsname 等于宏 \pgf@decl@arrow@par, 所以本选项的参数用于构成箭头的“全名”，见 \pgf@arrow@fullname→P. 340.

/pgf/@arrows decl/defaults=⟨*options*⟩ (no default)

此选项的定义是：

```
\pgfkeys{
  /pgf/@arrows decl/.cd,
  ...
  defaults/.code={%
    \let\pgf@arrows@options\pgfutil@empty%
    \pgfkeys{/pgf/arrow keys/.cd,#1}%
    \let\pgf@decl@arrow@defaults\pgf@arrows@options%
  }
  ...
}
```

参数 ⟨*options*⟩ 是以 /pgf/arrow keys 为前缀路径的选项列表,执行这些选项,然后定义宏 \pgf@decl@arrow@defa

16.6.2 在路径的始端、末端设置箭头

\ifpgf@arrows@translate

这个 TEX-if 的真值与路径末端、始端的箭头设置有关，即 \pgf@arrows@translatefalse 对应路径末端的箭头设置，而 \pgf@arrows@translatetrue 对应路径始端的箭头设置。

\pgf@arrow@swapper

仅当 \pgf@arrows@translatetrue 时，才有可能执行这个命令。其定义是：

```
\def\pgf@arrow@swapper{%
  \expandafter\let\expandafter\pgf@temp\csname pgf@ar@start@\pgf@temp\endcsname%
  \ifx\pgf@temp\relax%
    \def\pgf@temp{undefined}%
  \fi%
}
```

它的作用是，令 \pgf@temp 等于 \csname pgf@ar@start@⟨正在解析的箭头名称⟩\endcsname, 注意这个控制序列对应路径始端的箭头设置。然后做检查，如果这个控制序列尚未定义，就定义

```
\def\pgf@temp{undefined}
```

\pgf@arrow@tip@sequence

这个宏通常是在运行其他命令时被定义的，例如 \pgfsetarrowsend→P. 325 一开始就

```
\let\pgf@arrow@tip@sequence\pgfutil@empty
```

这个宏用于保存一个列表（无需分隔符号），列表项是处理箭头及其选项的命令，它通常由 \pgf@arrows@append@to@
重定义。它保存的是以下 4 种形式的命令：

- \pgf@arrow@handle@dot
- \pgf@arrow@handle{⟨*meta arrow name*⟩}{ 被 \expandafter 展开的 \pgf@arrows@options }
- \pgf@arrow@handle@shorthand@empty{ 被 \expandafter 展开的 \csname pgf@ar@means@⟨*shorthand arrow name*⟩\endcsname}
- \pgf@arrow@handle@shorthand{\csname pgf@ar@means@⟨*shorthand arrow name*⟩\endcsname}{ 被 \expandafter 展开的 \pgf@arrows@options}

\pgf@arrows@append@to@tips#1

其定义是：

```
\def\pgf@arrows@append@to@tips#1{%
  \let\pgf@tempa\pgf@arrow@tip@sequence%
  \def\pgf@tempb{#1}%
  \ifpgf@arrows@translate%
    \let\pgf@tempa\pgf@tempb%
    \let\pgf@tempb\pgf@arrow@tip@sequence%
  \fi%
  \expandafter\expandafter\expandafter\def%
  \expandafter\expandafter\expandafter\pgf@arrow@tip@sequence%
  \expandafter\expandafter\expandafter{\expandafter\pgf@tempa\pgf@tempb}%
}
```

可见本命令的作用是重定义宏 \pgf@arrow@tip@sequence.

本命令将参数 #1 添加到宏 \pgf@arrow@tip@sequence 中，即

```
\def\pgf@arrow@tip@sequence{
  被 \expandafter 展开一次的 \pgf@tempa
  被 \expandafter 展开一次的 \pgf@tempb
}
```

如果有 \pgf@arrows@translatetrue, 即在路径始端设置箭头，那么将 #1 添加到 \pgf@arrow@tip@sequence 的左侧。

如果有 \pgf@arrows@translatefalse, 即在路径末端设置箭头，那么将 #1 添加到 \pgf@arrow@tip@sequence 的右侧。

本命令通常被解析箭头的命令调用。

- 对于 meta 箭头 ⟨*meta arrow name*⟩，本命令添加的是

  ```
  \pgf@arrow@handle{⟨meta arrow name⟩}{ 被 \expandafter 展开的 \pgf@arrows@options }
  ```

  在不同情况下，其中的 \pgf@arrow@handle 通常会被 let 为其他命令。参考 \pgf@arrows@meta@parsed→P. 330.

- 对于点号箭头“.”, 本命令添加的是 \pgf@arrow@handle@dot, 在不同情况下,\pgf@arrow@handle@dot 会被 let 为其他命令。参考 \pgf@arrows@dot@parsed → P. 329.
- 对于 shorthand 箭头 ⟨*shorthand arrow name*⟩,
  - 如果 ⟨*shorthand arrow name*⟩ 不带方括号选项，那么本命令添加的是

```
\pgf@arrow@handle@shorthand@empty{ 被 \expandafter 展开的 \csname
↪  pgf@ar@means@⟨shorthand arrow name⟩\endcsname}
```

  - 如果 ⟨*shorthand arrow name*⟩ 带方括号选项，那么本命令添加的是

```
\pgf@arrow@handle@shorthand{\csname pgf@ar@means@⟨shorthand arrow name⟩
↪  \endcsname}{ 被 \expandafter 展开的 \pgf@arrows@options}
```

  参考 \pgf@arrows@shorthand@parsed → P. 330.

  \pgf@arrow@handle@shorthand@empty#1

  其定义是：

```
\def\pgf@arrow@handle@shorthand@empty#1{#1}
```

\pgf@start@tip@sequence

这个宏临时保存路径始端的、被解析后的箭头序列设置，它的初始值是 \pgfutil@empty. 当箭头序列被解析过后，解析结果通常保存在 \pgf@arrow@tip@sequence → P. 324 中，然后

```
\let\pgf@start@tip@sequence\pgf@arrow@tip@sequence
```

\pgf@end@tip@sequence

这个宏临时保存路径末端的、被解析后的箭头序列设置，它的初始值是 \pgfutil@empty. 当箭头序列被解析过后，解析结果通常保存在 \pgf@arrow@tip@sequence → P. 324 中，然后

```
\let\pgf@end@tip@sequence\pgf@arrow@tip@sequence
```

16.6.2.1 在路径末端设置箭头

\pgfsetarrowsend{⟨*arrow specification*⟩}

参数 ⟨*arrow specification*⟩ 的格式参考 §16.4。例如

```
\pgfsetarrowsend{[width=2pt]stealth[length=1pt] . stealth[length=2pt]}
\pgfsetarrowsend{>.>}
```

这个命令的定义是：

```
\def\pgfsetarrowsend#1{%
  \let\pgf@arrow@tip@sequence\pgfutil@empty%
  \pgf@arrows@translatefalse%
  \expandafter\let\expandafter\pgf@arrows@direct\csname
  ↪  pgf@arrows@direct@name@end@#1\endcsname%
  \ifx\pgf@arrows@direct\relax%
    \edef\pgf@marshal{\noexpand\pgfarrows@initial@parser#1[]\noexpand\pgf@stop}%
    \pgf@marshal%
  \else%
    \let\pgf@arrow@tip@sequence\pgf@arrows@direct%
  \fi%
  \let\pgf@end@tip@sequence\pgf@arrow@tip@sequence%
}
```

它的处理是：

1. 清空 \let\pgf@arrow@tip@sequence\pgfutil@empty

2. 设置真值 \pgf@arrows@translatefalse
3. 参数 ⟨*arrow specification*⟩ 可能是（保存箭头设置的）宏，将 ⟨*arrow specification*⟩ 展开后的结果记为 ⟨*Arrow Specification*⟩. 这一步 let 控制序列 \pgf@arrows@direct 等于控制序列

   \csname pgf@arrows@direct@name@end@⟨*Arrow Specification*⟩\endcsname
4. 检查 \pgf@arrows@direct 是否等于 \relax, 即检查控制序列

   \csname pgf@arrows@direct@name@end@⟨*Arrow Specification*⟩\endcsname

   是否已定义，
   - 如果 \pgf@arrows@direct 等于 \relax, 即控制序列

     \csname pgf@arrows@direct@name@end@⟨*Arrow Specification*⟩\endcsname

     未定义，则

     ```
     \edef\pgf@marshal{\noexpand\pgfarrows@initial@parser⟨arrow specification⟩[]
     ↪  \noexpand\pgf@stop}%
     \pgf@marshal%
     ```

     即解析 ⟨*Arrow Specification*⟩.

     注意以上代码在 ⟨*Arrow Specification*⟩ 后面添加了一对方括号，这对方括号会被之后的解析 ⟨*Arrow Specification*⟩ 的命令吃掉。
   - 如果 \pgf@arrows@direct 不等于 \relax, 即控制序列

     \csname pgf@arrows@direct@name@end@⟨*Arrow Specification*⟩\endcsname

     已定义，则

     ```
     \let\pgf@arrow@tip@sequence\pgf@arrows@direct
     ```

     这就不必再次解析 ⟨*Arrow Specification*⟩.
5. 将解析后的箭头序列保存在 \pgf@end@tip@sequence 中

   ```
   \let\pgf@end@tip@sequence\pgf@arrow@tip@sequence
   ```

其中的 \pgfarrows@initial@parser 解析 ⟨*Arrow Specification*⟩.

16.6.2.2 在路径始端设置箭头

\pgfsetarrowsstart⟨*arrow specification*⟩

类似 \pgfsetarrowsend, 只是箭头的次序会被反序，而且使用箭头的”reversed names”. 也就是说，如果

```
\pgfsetarrowsstart{>[]Stealth[]}
```

那么被保存的箭头是反序的 Stealth[]>[].

本命令的定义是：

```
\def\pgfsetarrowsstart#1{%
  \let\pgf@arrow@tip@sequence\pgfutil@empty%
  \pgf@arrows@translatetrue%
  \expandafter\let\expandafter\pgf@arrows@direct\csname
  ↪  pgf@arrows@direct@name@start@#1\endcsname%
  \ifx\pgf@arrows@direct\relax%
    \edef\pgf@marshal{\noexpand\pgfarrows@initial@parser#1[]\noexpand\pgf@stop}%
    \pgf@marshal%
  \else%
    \let\pgf@arrow@tip@sequence\pgf@arrows@direct%
  \fi%
  \let\pgf@start@tip@sequence\pgf@arrow@tip@sequence%
```

```
}
```

它的处理是：

1. 清空 \let\pgf@arrow@tip@sequence\pgfutil@empty
2. 设置真值 \pgf@arrows@translatetrue
3. 参数 ⟨*arrow specification*⟩ 可能是（保存箭头设置的）宏，将 ⟨*arrow specification*⟩ 展开后的结果记为 ⟨*Arrow Specification*⟩. 这一步 let 控制序列 \pgf@arrows@direct 等于控制序列

   \csname pgf@arrows@direct@name@start@⟨*Arrow Specification*⟩\endcsname
4. 检查 \pgf@arrows@direct 是否等于 \relax, 即检查控制序列

   \csname pgf@arrows@direct@name@start@⟨*Arrow Specification*⟩\endcsname

   是否已定义，
   - 如果 \pgf@arrows@direct 等于 \relax, 即控制序列

     \csname pgf@arrows@direct@name@start@⟨*Arrow Specification*⟩\endcsname

     未定义，则

     ```
     \edef\pgf@marshal{\noexpand\pgfarrows@initial@parser⟨arrow specification⟩[]
     ↪  \noexpand\pgf@stop}%
     \pgf@marshal%
     ```

     即解析 ⟨*Arrow Specification*⟩.

     注意以上代码在 ⟨*Arrow Specification*⟩ 后面添加了一对方括号，这对方括号会被之后的解析 ⟨*Arrow Specification*⟩ 的命令吃掉。
   - 如果 \pgf@arrows@direct 不等于 \relax, 即控制序列

     \csname pgf@arrows@direct@name@end@⟨*Arrow Specification*⟩\endcsname

     已定义，则

     ```
     \let\pgf@arrow@tip@sequence\pgf@arrows@direct
     ```

     这就不必再次解析 ⟨*Arrow Specification*⟩.
5. 将解析后的箭头序列保存在 \pgf@start@tip@sequence 中

   ```
   \let\pgf@start@tip@sequence\pgf@arrow@tip@sequence
   ```

16.6.2.3 在路径始端、末端设置箭头

\pgfsetarrows{⟨*arrow specification*⟩}

参数 ⟨*arrow specification*⟩ 可以是：

- ⟨*start arrow specification*⟩-⟨*end arrow specification*⟩
- [⟨路径为 */pgf/arrow keys* 的选项⟩], 如果是这个形式，那么这些选项保存的代码将会被添加到 \pgf@arrows@options@scope 中。
- [-⟨*something*⟩], 如果是这个形式，则

  ```
  \pgfsetarrowsstart{[}%
  \pgfsetarrowsend{⟨something⟩}%
  ```

如果在 ⟨*arrow specification*⟩ 中没有短横线-, 则会导致错误。

```
\def\pgfsetarrows#1{\pgf@arrows#1\pgf@stop}
\def\pgf@arrows{\pgfutil@ifnextchar[\pgf@arrows@test@opt\pgf@arrows@}%}
\def\pgf@arrows@#1-#2\pgf@stop{%
  \pgfsetarrowsstart{#1}%
  \pgfsetarrowsend{#2}%
```

```
}
```

\pgf@arrows@options@scope

这个宏的初始值被 let 为 \pgfutil@empty.

当执行 \pgfsetarrows{⟨*arrow options*⟩} 时，会导致

```
\let\pgf@arrows@options\pgf@arrows@options@scope%
\pgfkeys{/pgf/arrow keys/.cd,⟨arrow options⟩}% 向 \pgf@arrows@options 中添加代码
\let\pgf@arrows@options@scope\pgf@arrows@options%
```

也就是把 ⟨*arrow options*⟩ 保存的代码添加到 \pgf@arrows@options@scope 中。

16.6.2.4 解析 ⟨*arrow specification*⟩

下面解析 ⟨*arrow specification*⟩ 的命令的参数中有方括号，也就是说，这些命令会吃掉方括号，之前的 \pgfsetarrowsend 添加的方括号可以最终保证这些命令“有的吃”。

\pgfarrows@initial@parser⟨*arrow specification*⟩

有定义：

```
\def\pgfarrows@initial@parser{%
  \pgfutil@ifnextchar[\pgfarrows@initial@parser@{\pgfarrows@initial@parser@[]}%}

\def\pgfarrows@initial@parser@[#1]{%
  \let\pgf@arrows@options\pgfutil@empty%
  \pgfkeys{/pgf/arrow keys/.cd,#1}%
  \let\pgf@arrows@options@initial\pgf@arrows@options%
  \pgfarrows@parser%
}
```

如果 ⟨*arrow specification*⟩ 是 [⟨*options*⟩]⟨*Arrow Specification*⟩ 这种以方括号开头的格式，那么 ⟨*options*⟩ 在此被执行，不过，选项保存的命令一般不会被立即执行，而被添加到 \pgf@arrows@options 中。而 ⟨*Arrow Specification*⟩ 会被 \pgfarrows@parser 解析。

\pgfarrows@parser#1[

注意此命令的参数格式是 #1[, 此命令的处理是：

1. 将 #1 保存到 \pgf@temp 中：\pgfkeys@spdef\pgf@temp{#1}, 参考 \pgfkeys@spdef→P. 47.
2. 检查 \pgf@temp 保存的是什么，即 #1 是什么。
   - 如果 \pgf@temp 等于 \pgfutil@empty, 即 #1 是空的，则执行 \pgf@arrows@empty#1[, 这导致 \pgf@arrows@parser@done 或者 \pgfarrows@parser
   - 如果 \pgf@temp 不等于 \pgfutil@empty,
     - 如果 \pgf@temp 等于 \pgf@dot@text, 即 #1 是点号“.”, 则执行 \pgf@arrows@dot@parsed#1[
     - 如果 \pgf@temp 不等于 \pgf@dot@text,

       (a) 执行

       ```
       \let\pgf@untranslated\pgf@temp%
       \ifpgf@arrows@translate%
         \pgf@arrow@swapper→P. 323%
       \fi%
       ```

       (b) 然后再检查 #1 是哪一种箭头，

       其一： 如果 \csname pgf@ar@code@\pgf@temp\endcsname 等于 \relax,

       且 \csname pgf@ar@means@\pgf@temp\endcsname 等于 \relax, 也就是这两个控制序列都尚未定义，则执行 \pgf@arrows@single@char@parser#1[

其二： 如果 \csname pgf@ar@code@\pgf@temp\endcsname 等于 \relax,
但 \csname pgf@ar@means@\pgf@temp\endcsname 不等于 \relax, 这说明 #1 是 shorthand 箭头，则执行 \pgf@arrows@shorthand@parsed#1[

其三： 如果 \csname pgf@ar@code@\pgf@temp\endcsname 不等于 \relax, 这说明 #1 是 meta 箭头，则执行 \pgf@arrows@meta@parsed#1[

\pgf@arrows@empty[#1]

其定义是：

```
\def\pgf@arrows@empty[#1]{%
  \pgfutil@ifnextchar\pgf@stop\pgf@arrows@parser@done\pgfarrows@parser%
}
```

\pgf@arrows@parser@done#1

本命令吃掉一个参数。

```
\def\pgf@arrows@parser@done#1{%
}
```

\pgf@arrows@unknown#1[]#2[#3]

本命令吃掉方括号定界的参数，发出错误信息。

```
\def\pgf@arrows@unknown#1[]#2[#3]{%
  \pgferror{Unknown arrow tip kind '#1#2'}%
}
```

\pgf@arrows@dot@parsed#1[#2]

其定义是：

```
\def\pgf@arrows@dot@parsed#1[#2]{%
  \pgf@arrows@append@to@tips{\pgf@arrow@handle@dot}%
  \pgfutil@ifnextchar\pgf@stop\pgf@arrows@parser@done\pgfarrows@parser%
}
```

\pgf@arrows@single@char@parser#1

此命令的处理是：

1. 将 #1 保存到 \pgf@temp 中：\def\pgf@temp{#1}
2. 检查 \pgf@temp 保存的是什么，即 #1 是什么。
   - 如果 \pgf@temp 等于 \pgf@dot@text, 即 #1 是点号“.”, 则执行 \pgf@arrows@dot@parsed#1[]
   - 如果 \pgf@temp 不等于 \pgf@dot@text,
     (a) 执行

```
\let\pgf@untranslated\pgf@temp%
\ifpgf@arrows@translate%
  \pgf@arrow@swapper%
\fi%
```

→ P. 323

     (b) 然后再检查 #1 是哪一种箭头，
     – 如果 \csname pgf@ar@code@\pgf@temp\endcsname 等于 \relax,
       且 \csname pgf@ar@means@\pgf@temp\endcsname 等于 \relax, 也就是这两个控制序列都尚未定义，则执行 \pgf@arrows@unknown#1[], 发出错误信息
     – 如果 \csname pgf@ar@code@\pgf@temp\endcsname 等于 \relax,
       但 \csname pgf@ar@means@\pgf@temp\endcsname 不等于 \relax, 这说明 #1 是 shorthand 箭头，则执行 \pgf@arrows@shorthand@parsed#1[]

– 如果 \csname pgf@ar@code@\pgf@temp\endcsname 不等于 \relax，这说明 #1 是 meta 箭头，则执行 \pgf@arrows@meta@parsed#1[]

根据以上解析命令，可知 ⟨*arrow specification*⟩ 的形式应当如 §16.4 所描述。

\pgf@arrows@meta@parsed⟨*meta arrow name*⟩[⟨*arrows options*⟩]

本命令：

1. 执行 ⟨*arrows options*⟩，即

   \pgfkeys{/pgf/arrow keys/.cd,⟨*arrows options*⟩}

   不过，选项保存的命令一般不会被立即执行，而被添加到 \pgf@arrows@options 中。
2. 将

   \pgf@arrow@handle{⟨*meta arrow name*⟩}{⟨*arrows options*⟩ 保存的代码}

   这个 3 元序列添加到 \pgf@arrow@tip@sequence 中。
3. 继续解析后面的 arrow specification.

其定义是：

```
\def\pgf@arrows@meta@parsed#1[#2]{%
  % Ok, run the keys #2. This will add commands to \pgf@arrows@options:
  %
  \let\pgf@arrows@options\pgf@arrows@options@initial%
  \pgfkeys{/pgf/arrow keys/.cd,#2}%
  % Append the arrow and its options to the arrow tip sequence:
  \expandafter\pgf@arrows@meta@set\expandafter{\pgf@arrows@options}{#1}{
  ↪  \pgf@arrow@handle}%
  \pgfutil@ifnextchar\pgf@stop\pgf@arrows@parser@done\pgfarrows@parser%
}
```

\pgf@arrows@meta@set#1#2#3

本命令调用 \pgf@arrows@append@to@tips → P. 324 处理 #1#2#3.

```
\def\pgf@arrows@meta@set#1#2#3{%
  \pgf@arrows@append@to@tips{#3{#2}{#1}}%
}
```

\pgf@arrows@shorthand@parsed⟨*shorthand arrow name*⟩[⟨*options*⟩]

本命令：

1. \let\pgf@arrows@options\pgf@arrows@options@initial
2. 执行 ⟨*arrows options*⟩，即

   \pgfkeys{/pgf/arrow keys/.cd,⟨*arrows options*⟩}

   不过，选项保存的命令一般不会被立即执行，而被添加到 \pgf@arrows@options 中。
3. 做检查 \ifx\pgf@arrows@options\pgfutil@empty
   - 如果 \pgf@arrows@options 等于 \pgfutil@empty，则

     ```
     \pgf@arrows@append@to@tips{%
       \pgf@arrow@handle@shorthand@empty{
         被 \expandafter 展开一次的 \csname pgf@ar@means@⟨shorthand arrow name⟩
         ↪  \endcsname}
     }
     ```

   - 如果 \pgf@arrows@options 不等于 \pgfutil@empty，则

```
\pgf@arrows@meta@set
  {被 \expandafter 展开一次的 \pgf@arrows@options}
  {被 \expandafter 展开一次的 \csname pgf@ar@means@⟨shorthand arrow name⟩
  ↪  \endcsname}
  {\pgf@arrow@handle@shorthand}%
```

即调用 \pgf@arrows@append@to@tips →P. 324 来处理这 3 个参数。

4. 然后继续解析 arrow specification:

```
\pgfutil@ifnextchar\pgf@stop\pgf@arrows@parser@done\pgfarrows@parser
```

16.6.2.5 解析 shorthand 箭头的选项

\pgf@arrows@nested@options

这个宏保存 nested options, 它的初始值被 let 为 \pgfutil@empty.

本命令用在 \pgf@arrow@handle@shorthand 中。

本命令包含的是 shorthand 箭头的选项，例如

```
\pgfdeclarearrow{ name=goo, means = bar[length=2cm+\mydimen] }
\begin{pgfpicture}
\pgfsetarrowsend{goo[⟨nested options⟩]}
%...
\end{pgfpicture}
```

其中选项 ⟨*nested options*⟩ 保存的代码将转存到 \pgf@arrows@nested@options. 如果 shorthand 箭头是用其他的 shorthand 箭头定义的，那么它们的选项是被收集在一起的。

\pgf@arrow@handle@shorthand{⟨*tip sequence*⟩}{⟨*options code*⟩}

本命令的结果是重定义 \pgf@arrows@nested@options. 本命令的处理是：

1. \def\pgf@temp@opt{⟨*options code*⟩}
2. 定义

```
\def\pgf@arrows@nested@options@temp%
{
  被 \expandafter 展开的 \pgf@temp@opt
  被 \expandafter 展开的 \pgf@arrows@nested@options
}%
```

3. 用 \expandafter 将 \pgf@arrows@nested@options 展开为 ⟨*nested options A*⟩
4. 定义

```
\let\pgf@arrows@nested@options\pgf@arrows@nested@options@temp
```

5. 执行 ⟨*tip sequence*⟩
6. 定义

```
\def\pgf@arrows@nested@options{⟨nested options A⟩}%
```

16.6.3 箭头选项

有预定义项目

```
\newdimen\pgfarrowsep

\newif\ifpgfarrowswap
\newif\ifpgfarrowreversed
```

```
\newif\ifpgfarrowharpoon
\newif\ifpgfarrowopen

\let\pgf@arrows@stroke@color\pgfutil@empty
\let\pgf@arrows@fill@color\pgfutil@empty

\def\pgfarrows@length@scale{1}
\def\pgfarrows@width@scale{1}
\def\pgfarrows@slant{0}

\let\pgfarrows@scale@list\pgfutil@empty

\def\pgf@nonetext{none}
```

\pgfarrows@scale@list

这个宏保存一个序列（无需分隔符），序列的元素是

⟨*dimension register*⟩\pgfarrows@length@scale⟨*dimension register*⟩

或者

⟨*dimension register*⟩\pgfarrows@width@scale⟨*dimension register*⟩

这样的为 ⟨*dimension register*⟩ 赋值的句子,其中宏 \pgfarrows@length@scale 和 \pgfarrows@width@scale 保存数值。

\pgfarrowsaddtolengthscalelist⟨*dimension register*⟩

这个命令重定义宏 \pgfarrows@scale@list, 将赋值句子

⟨*dimension register*⟩\pgfarrows@length@scale⟨*dimension register*⟩

添加到 \pgfarrows@scale@list 中。

\pgfarrowsaddtowidthscalelist⟨*dimension register*⟩

这个命令重定义宏 \pgfarrows@scale@list, 将赋值句子

⟨*dimension register*⟩\pgfarrows@width@scale⟨*dimension register*⟩

添加到 \pgfarrows@scale@list 中。

\pgf@arrows@options

这个宏用于保存箭头的选项所保存的命令。在执行解析箭头的命令时通常会定义这个宏，它的初始值通常被 let 为 \pgfutil@empty. 它被 \pgfarrowsaddtooptions 或 \pgfarrowsaddtolateoptions 重定义。

\pgfarrowsaddtooptions{⟨*code*⟩}

本命令重定义 \pgf@arrows@options, 把某些代码 ⟨*code*⟩ 添加到宏 \pgf@arrows@options 中。

```
\def\pgfarrowsaddtooptions#1{\expandafter\def\expandafter\pgf@arrows@options
↪  \expandafter{\pgf@arrows@options#1}}
```

\pgf@arrows@late@options

这个宏用于保存箭头的 late 选项所保存的命令。它的初始值通常被 let 为 \pgfutil@empty. 它被 \pgf@arrows@add@to@late@options 重定义。

\pgf@arrows@add@to@late@options{⟨*code*⟩}

本命令重定义 \pgf@arrows@late@options, 把某些代码 ⟨*code*⟩ 添加到宏 \pgf@arrows@late@options 中。

```
\def\pgf@arrows@add@to@late@options#1{
↪  \expandafter\def\expandafter\pgf@arrows@late@options\expandafter{
↪  \pgf@arrows@late@options#1}}
```

\pgfarrowsaddtolateoptions{⟨*code*⟩}

本命令重定义 \pgf@arrows@options, 把命令 \pgf@arrows@add@to@late@options{⟨*code*⟩} 添加到宏 \pgf@arrows@options 中。

```
\def\pgfarrowsaddtolateoptions#1{\expandafter\def\expandafter\pgf@arrows@options
↪  \expandafter{\pgf@arrows@options\pgf@arrows@add@to@late@options{#1}}}
```

\pgfarrowsthreeparameters{⟨*parameters*⟩}

参数 ⟨*parameters*⟩ 可以是以下形式：

- ⟨*math expression 1*⟩
- ⟨*math expression 1*⟩␣⟨*math expression 2*⟩
- ⟨*math expression 1*⟩␣⟨*math expression 2*⟩␣⟨*math expression 3*⟩

以上参数 ⟨*math expression 1*⟩, ⟨*math expression 2*⟩, ⟨*math expression 3*⟩ 都会被 \pgfmathparse 解析：

- 解析 ⟨*math expression 1*⟩ 的结果赋予寄存器 \pgf@x, 这是个尺寸
- 解析 ⟨*math expression 2*⟩ 的结果赋予宏 \pgf@temp@a, 这是个数值
- 解析 ⟨*math expression 3*⟩ 的结果赋予宏 \pgf@temp@b, 这是个数值

所以本命令处理 3 个表达式，如果省略后面的表达式，就默认为 0.

本命令最后将计算结果保存到 \pgfarrowstheparameters.

```
\edef\pgfarrowstheparameters{{\the\pgf@x}{\pgf@temp@a}{\pgf@temp@b}}%
```

\pgfarrowslinewidthdependent{⟨*dimension*⟩}{⟨*factor 1*⟩}{⟨*factor 2*⟩}

本命令给出一种与线宽参数 \pgflinewidth 和 \pgfinnerlinewidth （这两个线宽是被添加箭头的路径的线宽，未必也是画出箭头轮廓的线宽）有关的计算方法，计算结果是一个尺寸，保存在寄存器 \pgf@x 中。

如果使用双线，且内线线宽满足 \pgfinnerlinewidth>0pt, 那么 \pgf@x 的值是

⟨*dimension*⟩ + ⟨*factor 1*⟩*(w - (w-n)*⟨*factor 2*⟩/2)

其中 w 是外线宽，n 是内线宽。

如果不使用双线，那么 \pgf@x 的值是

⟨*dimension*⟩ + ⟨*factor 1*⟩*w

其中 w 是外线宽。

由 \pgfarrowsthreeparameters 得到的 \pgfarrowstheparameters 有可能会被本命令处理，得到尺寸 \pgf@x, 然后再加以利用。

\pgfarrowslengthdependent{⟨*dimension*⟩}{⟨*num 1*⟩}{⟨*num 2*⟩}⟨*dimension register*⟩

本命令使得寄存器 ⟨*dimension register*⟩ 保存下面的尺寸：

⟨*dimension*⟩ + ⟨*num 1*⟩*\pgfarrowlength + ⟨*num 2*⟩*\pgflinewidth

其中的 \pgfarrowlength 是文件《pgflibraryarrows.meta.code.tex》声明的尺寸寄存器。

/pgf/arrow keys/scale={⟨*math expression*⟩} (no default)

本选项的定义是：

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  scale/.code={%
```

```
    \pgfmathparse{#1}%
    \expandafter\pgfarrowsaddtooptions\expandafter{
    ↪  \expandafter\def\expandafter\pgfarrows@length@scale\expandafter{
    ↪  \pgfmathresult}}%
    \expandafter\pgfarrowsaddtooptions\expandafter{
    ↪  \expandafter\def\expandafter\pgfarrows@width@scale\expandafter{
    ↪  \pgfmathresult}}%
  },
}
```

本选项利用 \pgfmathparse 解析 ⟨*math expression*⟩, 解析结果作为 \pgfarrows@length@scale 和 \pgfarrows@width@scale 的值。

本选项尚未实现对 \pgfarrows@length@scale 和 \pgfarrows@width@scale 的修改，只是保存这种修改。本选项只是调用 \pgfarrowsaddtooptions 来重定义 \pgf@arrows@options, 也就是把

```
\def\pgfarrows@length@scale{展开的 \pgfmathresult}
\def\pgfarrows@width@scale{展开的 \pgfmathresult}
```

添加到 \pgf@arrows@options 中。

/pgf/arrow keys/scale length={⟨*math expression*⟩} (initially 1)

本选项利用 \pgfmathparse 解析 ⟨*math expression*⟩, 解析结果作为 \pgfarrows@length@scale 的值。

本选项尚未实现对 \pgfarrows@length@scale 的修改,只是保存这种修改。本选项只是调用 \pgfarrowsaddtoopti 来重定义 \pgf@arrows@options, 也就是把

```
\def\pgfarrows@length@scale{展开的 \pgfmathresult}
```

添加到 \pgf@arrows@options 中。

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  scale length/.code={\pgfmathparse{#1}
  ↪  \expandafter\pgfarrowsaddtooptions\expandafter{
  ↪  \expandafter\def\expandafter\pgfarrows@length@scale\expandafter{
  ↪  \pgfmathresult}}},
}
```

/pgf/arrow keys/scale width={⟨*math expression*⟩} (initially 1)

本选项利用 \pgfmathparse 解析 ⟨*math expression*⟩, 解析结果作为 \pgfarrows@width@scale 的值。

本选项尚未实现对 \pgfarrows@width@scale 的修改,只是保存这种修改。本选项只是调用 \pgfarrowsaddtooptio 来重定义 \pgf@arrows@options, 也就是把

```
\def\pgfarrows@width@scale{展开的 \pgfmathresult}
```

添加到 \pgf@arrows@options 中。

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  scale width/.code={\pgfmathparse{#1}
  ↪  \expandafter\pgfarrowsaddtooptions\expandafter{
  ↪  \expandafter\def\expandafter\pgfarrows@width@scale\expandafter{
  ↪  \pgfmathresult}}},
}
```

/pgf/arrow keys/slant={⟨*math expression*⟩} (initially 0)

本选项利用 \pgfmathparse 解析 ⟨*math expression*⟩, 解析结果作为 \pgfarrows@slant 的值。

本选项尚未实现对 \pgfarrows@slant 的修改,只是保存这种修改。本选项只是调用 \pgfarrowsaddtooptions

来重定义 \pgf@arrows@options, 也就是把

```
\def\pgfarrows@slant{展开的 \pgfmathresult}
```

添加到 \pgf@arrows@options 中。

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  slant/.code={\pgfmathparse{#1}\expandafter\pgfarrowsaddtooptions\expandafter{
  ↪  \expandafter\def\expandafter\pgfarrows@slant\expandafter{\pgfmathresult}}},
}
```

/pgf/arrow keys/sep={⟨*parameters*⟩} (default 0.88pt .3 1)

参数 ⟨*parameters*⟩ 就是 \pgfarrowsthreeparameters→P. 333 的参数，本选项的定义是：

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  sep/.code={
    \pgfarrowsthreeparameters{#1}%
    \expandafter\pgfarrowsaddtooptions\expandafter{
    ↪  \expandafter\pgfarrowslinewidthdependent\pgfarrowstheparameters\pgfarrowsep\pgf@x
    ↪  }%
  },
}
```

本选项将

```
\pgfarrowslinewidthdependent{展开的 \the\pgf@x}{展开的 \pgf@temp@a}{展开的
↪  \pgf@temp@b}\pgfarrowsep\pgf@x
```

添加到 \pgf@arrows@options 中，其中 \pgfarrowsep 是个尺寸寄存器，上一命令的意思是，先用 \pgfarrowslinewidthdependent→P. 333 计算一个尺寸并保存到寄存器 \pgf@x 中，然后，将寄存器 \pgfarrowsep 的值设为 \pgf@x.

在本选项的默认值下，\pgfarrowsep 的值是 0.88pt + (w + n)*3/20.

/pgf/arrow keys/swap (no value)

本选项执行 \pgfarrowsaddtooptions→P. 332, 将

```
\ifpgfarrowswap\pgfarrowswapfalse\else\pgfarrowswaptrue\fi
```

添加到 \pgf@arrows@options→P. 332 中。

本选项的定义是：

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  swap/.code=\pgfarrowsaddtooptions{
  ↪  \ifpgfarrowswap\pgfarrowswapfalse\else\pgfarrowswaptrue\fi},
}
```

/pgf/arrow keys/reversed (no value)

本选项执行 \pgfarrowsaddtooptions→P. 332, 将

```
\ifpgfarrowreversed\pgfarrowreversedfalse\else\pgfarrowreversedtrue\fi
```

添加到 \pgf@arrows@options→P. 332 中。

本选项的定义是：

```
\pgfkeys{
  /pgf/arrow keys/.cd,
```

```
  reversed/.code=\pgfarrowsaddtooptions{
  ↪  \ifpgfarrowreversed\pgfarrowreversedfalse\else\pgfarrowreversedtrue\fi},
}
```

/pgf/arrow keys/harpoon=true|false (default true)

本选项执行 \pgfarrowsaddtooptions→P. 332, 将 \pgfarrowharpoontrue 或 \pgfarrowharpoonfalse 添加到 \pgf@arrows@options→P. 332 中。

本选项的定义是：

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  harpoon/.is choice,
  harpoon/.default=true,
  harpoon/true/.code=\pgfarrowsaddtooptions{\pgfarrowharpoontrue},
  harpoon/false/.code=\pgfarrowsaddtooptions{\pgfarrowharpoonfalse},
}
```

/pgf/arrow keys/left (style)

本选项的定义是：

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  left/.style={harpoon},
}
```

/pgf/arrow keys/right (style)

本选项的定义是：

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  right/.style={harpoon,swap},
}
```

/pgf/arrow keys/color=⟨*color*⟩ (no default)

这个选项设置箭头轮廓线的颜色。

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  color/.code=\pgfarrowsaddtooptions{\def\pgf@arrows@stroke@color{#1}},
}
```

把

```
\def\pgf@arrows@stroke@color{⟨color⟩}
```

添加到 \pgf@arrows@options 中。

/pgf/arrow keys/fill=⟨*color*⟩

这个选项设置箭头内部的填充色。

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  fill/.code={%
    \def\pgf@temp{#1}%
    \ifx\pgf@temp\pgf@nonetext%
      \pgfarrowsaddtooptions{\pgfarrowopentrue}
    \else
      \pgfarrowsaddtooptions{\pgfarrowopenfalse\def\pgf@arrows@fill@color{#1}}
    \fi
```

```
  },
  fill/.value required,
  open/.style={fill=none},
}
```

如果 ⟨*color*⟩ 不是 none, 则把

```
\pgfarrowopenfalse
\def\pgf@arrows@fill@color{⟨color⟩}
```

添加到 \pgf@arrows@options 中。

如果 ⟨*color*⟩ 是 none, 则把

```
\pgfarrowopentrue
```

添加到 \pgf@arrows@options 中。

在使用选项 fill 时，必须给出选项值。

/pgf/arrow keys/.unknown

本选项的定义是：

```
\pgfkeys{
  /pgf/arrow keys/.cd,
  .unknown/.code={
    \expandafter\pgfutil@in@\expandafter!\expandafter{\pgfkeyscurrentname}%
    \ifpgfutil@in@%
      % this is a color!
      \expandafter\pgfarrowsaddtooptions\expandafter{
      ↪  \expandafter\def\expandafter\pgf@arrows@stroke@color\expandafter{
      ↪  \pgfkeyscurrentname}}%
    \else%
      \pgfutil@doifcolorelse{\pgfkeyscurrentname}
      {%
        \expandafter\pgfarrowsaddtooptions\expandafter{
        ↪  \expandafter\def\expandafter\pgf@arrows@stroke@color\expandafter{
        ↪  \pgfkeyscurrentname}}%
      }
      {%
        \edef\pgf@temp{\pgfkeyscurrentname}%
        \pgfkeys{/errors/unknown key/.expand once=\expandafter{\pgf@temp}{#1}}%
      }
      \fi
    }
}
```

如果写出的某个箭头选项 ⟨*an unknown key*⟩ 不是已定义的选项，那么就检查这个选项是否是一个颜色。

- 如果 ⟨*an unknown key*⟩ 中含有感叹号 !, 那么就认为它是一个颜色，于是把

  ```
  \def\pgf@arrows@stroke@color{⟨color⟩}
  ```

  添加到 \pgf@arrows@options 中。
- 如果 ⟨*an unknown key*⟩ 中不含有感叹号 !, 那就再检查它是否其他颜色名称。
  在文件《pgfutil-plain.def》以及其他.def 文件中有：

  \pgfutil@doifcolorelse#1#2#3

  其定义是：

```
\def\pgfutil@doifcolorelse#1#2#3{%
  \expandafter\ifx\csname\string\color@#1\endcsname\relax%
    \let\pgf@next=\pgfutil@secondoftwo%
  \else
    \let\pgf@next=\pgfutil@firstoftwo%
  \fi%
  \pgf@next{#2}{#3}%
}
```

本命令检查 \csname\string\color@#1\endcsname 是否已定义，如果已定义则执行 #2, 如果未定义则执行 #3.

如果 \csname\string\color@⟨*an unknown key*⟩\endcsname 已定义，就认为 ⟨*an unknown key*⟩ 是其他颜色名称，此时把

```
\def\pgf@arrows@stroke@color{⟨color⟩}
```

添加到 \pgf@arrows@options 中；否则发出错误信息。

16.6.4 箭头的特征尺寸

箭头特征尺寸的初始定义

```
\def\pgf@arrows@copy@tipend{\pgf@arrows@the@tipend}
\def\pgf@arrows@copy@backend{\pgf@arrows@the@backend}
\def\pgf@arrows@copy@lineend{\pgf@arrows@the@lineend}
\def\pgf@arrows@zeropt{0pt}%

\let\pgf@arrows@saves\pgfutil@empty%
\let\pgf@arrows@convexhull\pgfutil@empty%
\let\pgf@arrows@the@tipend\pgf@arrows@zeropt%
\let\pgf@arrows@the@backend\pgf@arrows@zeropt%
\let\pgf@arrows@the@lineend\pgf@arrows@zeropt%
\let\pgf@arrows@the@visualtipend\pgf@arrows@copy@tipend%
\let\pgf@arrows@the@visualbackend\pgf@arrows@copy@backend%
```

代表特征点的宏：

- \pgf@arrows@the@tipend
- \pgf@arrows@the@backend
- \pgf@arrows@the@lineend
- \pgf@arrows@the@visualtipend
- \pgf@arrows@the@visualbackend

它们的初始值都是 0pt, 注意这些宏保存的应当总是尺寸。

\pgf@arrows@handle@reverse

这个命令：

1. \pgf@arrows@the@visualtipend 的值变成原 \pgf@arrows@the@visualbackend 的值的相反数
2. \pgf@arrows@the@visualbackend 的值变成原 \pgf@arrows@the@visualtipend 的值的相反数
3. \pgf@arrows@the@tipend 的值变成原 \pgf@arrows@the@backend 的值的相反数
4. \pgf@arrows@the@backend 的值变成原 \pgf@arrows@the@tipend 的值的相反数
5. \pgf@arrows@the@lineend 的值变成它原来值的相反数
6. 然后执行

```
\let\pgf@temp\pgf@arrows@convexhull%
\let\pgf@arrows@convexhull\pgfutil@empty%
\let\pgf@arrow@hull@point\pgf@arrow@hull@point@reverse%
\pgf@temp%
```

\pgf@arrows@saves

这个宏保存一个序列（无需分隔符号），序列的元素是为尺寸寄存器赋值的语句，以及定义宏的定义命令。它的初始值被 let 为 \pgfutil@empty, 它被 \pgfarrowssavethe 或者 \pgfarrowssave 重定义。

这个宏的内容会被全局地转存到控制序列 \csname pgf@ar@saves@\pgf@arrow@id\endcsname 中。

\pgfarrowssavethe⟨*dimen registor*⟩

本命令重定义宏 \pgf@arrows@saves, 将为寄存器 ⟨*dimen registor*⟩ 赋值的句子添加到 \pgf@arrows@saves 中。本命令执行：

```
\def\pgf@arrows@saves%
{
  被 \expandafter 展开的\pgf@arrows@saves
  ⟨dimen registor⟩被 \expandafter 展开的 \the⟨dimen registor⟩\relax% 赋值句子
}
```

\pgfarrowssave\⟨*macro*⟩

本命令重定义宏 \pgf@arrows@saves, 将定义宏 \⟨*macro*⟩ 的命令添加到 \pgf@arrows@saves 中。本命令执行：

```
\def\pgf@arrows@saves%
{
  被 \expandafter 展开的\pgf@arrows@saves
  \def\⟨macro⟩{被 \expandafter 展开的 \⟨macro⟩}% 定义命令
}
```

\pgf@arrows@convexhull

这个宏保存一个序列（无需分隔符号），序列的元素是“hull point”, 其形式为

\pgf@arrow@hull@point{⟨*x dimension*⟩}{⟨*y dimension*⟩}

它的初始值被 let 为 \pgfutil@empty, 它被 \pgfarrowshullpoint 或者 \pgfarrowsupperhullpoint 重定义。在处理 hull point 时，\pgf@arrow@hull@point 会被 let 为其他命令。

\pgfarrowshullpoint{⟨*x dimension*⟩}{⟨*y dimension*⟩}

这个命令将

\pgf@arrow@hull@point{⟨*x dimension*⟩}{⟨*y dimension*⟩}

添加到 \pgf@arrows@convexhull 中。

\pgfarrowsupperhullpoint{⟨*x dimension*⟩}{⟨*y dimension*⟩}

本命令的处理是：

- 如果 ⟨*y dimension*⟩>0pt, 那么将

  \pgf@arrow@hull@point{⟨*x dimension*⟩}{⟨*y dimension*⟩}

  添加到 \pgf@arrows@convexhull 中；如果 \ifpgfarrowharpoon 的真值是 false, 那么还会把

  \pgf@arrow@hull@point{⟨*x dimension*⟩}{⟨*-y dimension*⟩}

  添加到 \pgf@arrows@convexhull 中。
- 如果 ⟨*y dimension*⟩<=0pt, 那么将

  \pgf@arrow@hull@point{⟨*x dimension*⟩}{⟨*y dimension*⟩}

  添加到 \pgf@arrows@convexhull 中。

\pgfarrowssettipend{⟨*dimension*⟩}

本命令重定义 \pgf@arrows@the@tipend.

```
\def\pgfarrowssettipend#1{\pgf@x#1\edef\pgf@arrows@the@tipend{\the\pgf@x}}
```

\pgfarrowssetbackend{⟨*dimension*⟩}

本命令重定义 \pgf@arrows@the@backend.

```
\def\pgfarrowssetbackend#1{\pgf@x#1\edef\pgf@arrows@the@backend{\the\pgf@x}}
```

\pgfarrowssetlineend{⟨*dimension*⟩}

本命令重定义 \pgf@arrows@the@lineend.

```
\def\pgfarrowssetlineend#1{\pgf@x#1\edef\pgf@arrows@the@lineend{\the\pgf@x}}
```

\pgfarrowssetvisualtipend{⟨*dimension*⟩}

本命令重定义 \pgf@arrows@the@visualtipend.

```
\def\pgfarrowssetvisualtipend#1{\pgf@x#1\edef\pgf@arrows@the@visualtipend{
↪  \the\pgf@x}}
```

\pgfarrowssetvisualbackend{⟨*dimension*⟩}

本命令重定义 \pgf@arrows@the@visualbackend.

```
\def\pgfarrowssetvisualbackend#1{\pgf@x#1\edef\pgf@arrows@the@visualbackend{
↪  \the\pgf@x}}
```

16.6.5 箭头的全名及 id

\pgf@arrow@fullname{⟨*arrow name*⟩}

本命令定义一种“全名”：

```
\def\pgf@arrow@fullname#1{pgf@@ar@#1@\the\pgflinewidth @\pgfinnerlinewidth @
↪  \ifpgfarrowreversed r\fi @\pgf@arrows@stroke@color @\pgf@arrows@fill@color @
↪  \csname pgf@ar@par@#1\endcsname}
```

这个“全名”包含了很多信息：

- 前缀 pgf@@ar@
- 箭头名称 ⟨*arrow name*⟩
- 路径的外线宽值 \the\pgflinewidth
- 路径的内线宽值 \pgfinnerlinewidth
- 如果 \ifpgfarrowreversed 的值是 true, 就使用字母 r, 见选项 /pgf/arrow keys/reversed → P. 335
- 箭头轮廓线的颜色 \pgf@arrows@stroke@color, 见选项 /pgf/arrow keys/color → P. 789
- 箭头填充色的颜色 \pgf@arrows@fill@color, 见选项 /pgf/arrow keys/fill → P. 789
- 声明箭头 ⟨*arrow name*⟩ 时，选项 /pgf/@arrows decl/parameters → P. 323 的值

这个命令由 \pgfarrows@getid, \pgf@arrows@instantiate 调用。当 \pgf@arrows@instantiate 调用这个命令时，把全名变成控制序列，这个控制序列全局地保存箭头的序号（从 1 开始）。

\pgf@arrow@id@count

这个宏保存一个整数值，用于构造箭头的 id.

\pgfarrows@getid{⟨*arrow name*⟩}{⟨被 *\expandafter* 展开的 *\pgf@arrows@options*⟩}

本命令的第一个参数 ⟨*arrow name*⟩ 是箭头序列中的一个箭头名称；第二个参数是第一个参数 ⟨*arrow name*⟩ 的选项所保存的命令。参考 \pgf@end@tip@sequence → P. 325.

本命令被 \pgf@arrow@drawer →P. 343 调用。

本命令的处理是：

1. \let\pgf@arrows@late@options\pgfutil@empty
2. \csname pgf@ar@defaults@⟨*arrow name*⟩\endcsname
3. 执行 \pgf@arrows@options@scope →P. 328
4. 被 \expandafter 展开的 \pgf@arrows@options →P. 332\relax
5. \pgf@arrows@nested@options
6. \pgf@arrows@late@options
7. \pgfarrows@scale@list
8. 全局地将 \pgf@arrow@id let 为 \csname\pgf@arrow@fullname{⟨*arrow name*⟩}\endcsname
9. 检查 \pgf@arrow@id 是否等于 \relax, 即是否已定义。如果未定义就执行

   \pgf@arrows@instantiate{⟨*arrow name*⟩}

本命令的主要作用是给出构造箭头“全名”所需的信息，然后检查这个全名是否已经被定义为一个（不等于 \relax 的）控制序列。

\pgf@arrows@instantiate{⟨*arrow name*⟩}

这个命令定义 \pgf@arrow@id 以及相关的一些控制序列。

1. 将宏 \pgf@arrow@id@count 的值全局地加 1.
2. 将控制序列

   \csname\pgf@arrow@fullname{⟨*arrow name*⟩}\endcsname

   全局地 let 为 \pgf@arrow@id@count
3. 将 \pgf@arrow@id 全局地 let 为 \pgf@arrow@id@count
4. 用 { 开启一个组
5. \pgfclearid, 在《pgfcorescopes.code.tex》中有：

```
\def\pgfclearid{%
  \pgfsys@clear@id%
}
```

6. 执行 \csname pgf@ar@setup@⟨*arrow name*⟩\endcsname, 这个控制序列由 \pgfdeclarearrow →P. 316 定义。
7. 是否翻转箭头

```
\ifpgfarrowreversed%
  % Compute transformation:
  \pgf@arrows@handle@reverse →P. 338%
\fi%
```

8. 将 \csname pgf@ar@saves@\pgf@arrow@id\endcsname 全局地 let 为 \pgf@arrows@saves →P. 339

```
\expandafter\global\expandafter\let\csname pgf@ar@saves@\pgf@arrow@id
↪  \endcsname\pgf@arrows@saves%
```

9. 将 \csname pgf@ar@hull@\pgf@arrow@id\endcsname 全局地 let 为 \pgf@arrows@convexhull →P. 339

```
\expandafter\global\expandafter\let\csname pgf@ar@hull@\pgf@arrow@id\endcsname
↪  \pgf@arrows@convexhull%
```

10. 全局定义 \csname pgf@ar@ends@\pgf@arrow@id\endcsname

```
\expandafter\xdef\csname pgf@ar@ends@\pgf@arrow@id\endcsname{{
↪  \pgf@arrows@the@tipend}{\pgf@arrows@the@backend}{\pgf@arrows@the@lineend}}
↪  %
```

参考 \pgfarrowssettipend P. 340, \pgfarrowssetbackend P. 340, \pgfarrowssetlineend P. 340.

11. 全局定义 \csname pgf@ar@visual@\pgf@arrow@id\endcsname

```
\expandafter\xdef\csname pgf@ar@visual@\pgf@arrow@id\endcsname{{
↪  \pgf@arrows@the@visualtipend}{\pgf@arrows@the@visualbackend}{}}%
```

参考 \pgfarrowssetvisualtipend P. 340, \pgfarrowssetvisualbackend P. 340.

12. 定义 \pgf@arrow@code

```
\edef\pgf@arrow@code{%
  \ifpgfarrowreversed\noexpand\pgftransformxscale{-1}\fi%
  \expandafter\noexpand\csname pgf@ar@code@#1\endcsname%
}%
```

命令 \pgfdeclarearrow P. 316 定义了 \csname pgf@ar@code@⟨*arrow name*⟩\endcsname.

13. 全局定义 \csname pgf@ar@inst@code@\pgf@arrow@id\endcsname

```
\expandafter\global\expandafter\let\csname pgf@ar@inst@code@\pgf@arrow@id
↪  \endcsname\pgf@arrow@code%
```

14. 如果 \csname pgf@ar@do@cache@⟨*arrow name*⟩\endcsname 等于 \pgf@truetext, 即保存单词 true, 也就是选项 /pgf/@arrows decl/cache P. 322=true, 则

```
\pgfsysprotocol@getcurrentprotocol\pgf@arrow@temp%
{% 开启一个组
  \pgfinterruptpath%
    \let\pgfusepath=\pgf@nousepath@here%
    \pgf@relevantforpicturesizefalse%
    \pgftransformreset%
    \pgfsysprotocol@setcurrentprotocol\pgfutil@empty%
    \pgfsysprotocol@bufferedtrue%
    \pgfscope%
    \pgf@arrow@code%
    \endpgfscope%
    \pgfsysprotocol@getcurrentprotocol\pgf@@arrow@temp%
    \expandafter\gdef\expandafter\pgf@arrow@code\expandafter{\expandafter
    ↪  \pgfsysprotocol@literal\expandafter{\pgf@@arrow@temp}}%
  \endpgfinterruptpath%
}% 结束组
\pgfsysprotocol@setcurrentprotocol\pgf@arrow@temp%
```

其中 \pgf@nousepath@here 的定义是:

```
\def\pgf@nousepath@here#1{\pgferror{The definition of an arrow may not use
↪  \string\pgfusepath}}
```

以上最后的结果是全局定义 \pgf@arrow@code, 可参考 \pgfsysprotocol@literal P. 199.

15. 将 \csname pgf@ar@cache@\pgf@arrow@id\endcsname 全局地 let 为 \pgf@arrow@code

```
\expandafter\global\expandafter\let\csname pgf@ar@cache@\pgf@arrow@id
↪  \endcsname\pgf@arrow@code%
```

16. 用 } 结束组

16.6.6 计算路径的裁剪长度，箭头序列的总长度

单个箭头的长度就是 tip end 与 back end 的间距，箭头序列的总长度就是各个箭头的长度之和再加上箭头之间的 sep.

为了添加箭头会对路径做裁剪，路径被裁掉的长度是从最头上的箭头的 tip end 到路径的 line end 的间距。

\pgf@arrow@compute@shortening\pgf@end@tip@sequence

本命令处理箭头序列，得到两个结果：

- 寄存器 \pgf@xa 是需要裁掉的路径长度
- 寄存器 \pgf@xb 是箭头序列的总长度

参考 \pgf@end@tip@sequence→P. 325.

\pgfarrowtotallength{⟨*arrow specification*⟩}

本命令得到 2 个结果：

- \pgf@x 保存 ⟨*arrow specification*⟩ 箭头的总长度。
- \pgf@xa 保存路径被裁掉的长度。

16.6.7 绘制箭头的过程

\pgfarrowdraw{⟨*arrow specification*⟩}

本命令画出 ⟨*arrow specification*⟩.

```
\def\pgfarrowdraw#1{%
  {%
    \pgfsetarrowsend{#1}%
    \pgf@arrow@compute@shortening\pgf@end@tip@sequence%
    \pgf@arrow@draw@arrow\pgf@end@tip@sequence\pgf@xb%
  }%
}
```

参考 \pgf@end@tip@sequence→P. 325.

\pgf@arrow@draw@arrow\pgf@end@tip@sequence\pgf@xb

本命令被 \pgfarrowdraw 调用。其中参数 \pgf@end@tip@sequence→P. 325 是 \pgfarrowdraw 的一个结果，\pgf@xb 是 \pgf@arrow@compute@shortening 的一个计算结果，即箭头序列的总长度。

```
\def\pgf@arrow@draw@arrow#1#2{%
  {%
    \pgf@xb=#2%
    \pgftransformxshift{-\pgf@xb}%
    \let\pgf@arrow@handle\pgf@arrow@drawer
    \let\pgf@arrow@handle@dot\relax%
    #1%
  }%
}
```

本命令创建一个组，在组内执行所有操作。

本命令实际上被 \pgfusepath→P. 295, \pgf@do@draw@straightend→P. 307 调用。

\pgf@arrow@drawer{⟨*arrow name*⟩}{⟨被 *\expandafter* 展开的 *\pgf@arrows@options*⟩}

本命令的参数就是 \pgfarrows@getid→P. 340 的参数。本命令处理单个的箭头 ⟨*arrow name*⟩ 及其选项。

```
\def\pgf@arrow@drawer#1#2{%
  % Prepare:
  {%
    \pgfarrows@getid{#1}{#2}%
    % Do shift:
```

```
    \expandafter\expandafter\expandafter\pgf@arrow@drawer@shift\csname
    ↪  pgf@ar@ends@\pgf@arrow@id\endcsname%
    % Do slant:
    \ifdim\pgfarrows@slant pt=0pt%
    \else%
      \pgftransformxslant{\pgfarrows@slant}%
    \fi%
    % do swap:
    \ifpgfarrowswap%
      \pgftransformyscale{-1}%
    \fi%
    {%
      \csname pgf@ar@saves@\pgf@arrow@id\endcsname%
      \pgfscope%
        \pgf@arrows@color@setup%
        \pgflowlevelsynccm\csname pgf@ar@cache@\pgf@arrow@id\endcsname%
      \endpgfscope%
      \pgf@arrows@rigid@hull%
    }%
  \expandafter}%
  % Transform to next tip:
  \expandafter\pgftransformxshift\expandafter{\the\pgf@xc}%
}
```

可见本命令

- 在一个花括号组内处理箭头及其选项，
- 在一个套嵌的花括号组内执行控制序列 `\csname pgf@ar@saves@\pgf@arrow@id\endcsname`,
- 在套嵌的 `pgfscope` 内处理绘制箭头的代码。
- 其中 `\pgf@arrows@rigid@hull` 会检查 `\ifpgf@relevantforpicturesize` 的真值，从而决定是否用箭头刷新 `pgfpicture` 环境的边界盒子。

16.6.8 预先声明的箭头

由文件《pgfcorearrows.code.tex》预先声明的箭头有：

- 点号 `.`, 这是个 meta 箭头，它什么也没有，

```
\pgfdeclarearrow{
  name = .,
  drawing code =
}
```

- 下划线 `_`, 这是个 meta 箭头，它只有默认的 `/pgf/arrow keys/sep` → P. 795 选项，

```
\pgfdeclarearrow{
  name = _,
  defaults = {sep},
  drawing code =
}
```

- `To`, 这是个 shorthand 箭头，

```
\pgfdeclarearrow{ name = To,  means = } % forward declaration
之后
\pgfkeys{To /.tip = to}
```

参考 /.tip.

- Bar, 这是个 shorthand 箭头，它什么也没有，

```
\pgfdeclarearrow{ name = Bar, means = } % forward declaration
之后
\pgfkeys{Bar /.tip = @bar}
```

参考 /.tip.

- <, >, |, 这是 3 个 shorthand 箭头，

```
\pgfkeys{
  <-> /.tip           = {To},
  >-< /.tip           = {>[reversed]},
  | /.tip             = {Bar}
}
```

参考 /.tip.

16.6.9 Stealth 的定义

下面是文件《pgflibraryarrows.meta.code.tex》中箭头 Stealth 的定义：

```
\pgfdeclarearrow{
  name = Stealth,
  defaults = {
    length  = +3pt 4.5 .8,
    width'  = +0pt .75,
    inset'  = +0pt 0.325,
    line width = +0pt 1 1,
  },
  setup code = {
    % Cap the line width at 1/4th distance from inset to tip
    \pgf@x\pgfarrowlength
    \advance\pgf@x by-\pgfarrowinset
    \pgf@x.25\pgf@x
    \ifdim\pgf@x<\pgfarrowlinewidth
      \pgfarrowlinewidth\pgf@x
    \fi
    % Compute front miter length:
    \pgfmathdivide@{\pgf@sys@tonumber\pgfarrowlength}{\pgf@sys@tonumber\pgfarrowwidth}
    ↪  %
    \let\pgf@temp@quot\pgfmathresult%
    \pgf@x\pgfmathresult pt%
    \pgf@x\pgfmathresult\pgf@x%
    \pgf@x4\pgf@x%
    \advance\pgf@x by1pt%
    \pgfmathsqrt@{\pgf@sys@tonumber\pgf@x}%
    \pgf@xc\pgfmathresult\pgfarrowlinewidth% xc is front miter
    \pgf@xc.5\pgf@xc
    \pgf@xa\pgf@temp@quot\pgfarrowlinewidth% xa is extra harpoon miter
    % Compute back miter length:
    \pgf@ya.5\pgfarrowwidth%
    \csname pgfmathatan2@\endcsname{\pgfmath@tonumber\pgfarrowlength}{
    ↪  \pgfmath@tonumber\pgf@ya}%
    \pgf@yb\pgfmathresult pt%
```

```
    \csname pgfmathatan2@\endcsname{\pgfmath@tonumber\pgfarrowinset}{\pgfmath@tonumber\pgf@ya}%
    \pgf@ya\pgfmathresult pt%
    \advance\pgf@yb by-\pgf@ya%
    \pgf@yb.5\pgf@yb% half angle in yb
    \pgfmathtan@{\pgf@sys@tonumber\pgf@yb}%
    \pgfmathreciprocal@{\pgfmathresult}%
    \pgf@yc\pgfmathresult\pgfarrowlinewidth%
    \pgf@yc.5\pgf@yc%
    \advance\pgf@ya by\pgf@yb%
    \pgfmathsincos@{\pgf@sys@tonumber\pgf@ya}%
    \pgf@ya\pgfmathresulty\pgf@yc% ya is the back miter
    \pgf@yb\pgfmathresultx\pgf@yc% yb is the top miter
    \ifdim\pgfarrowinset=0pt%
      \pgf@ya.5\pgfarrowlinewidth% easy: back miter is half linewidth
    \fi
    % Compute inset miter length:
    \pgfmathdivide@{\pgf@sys@tonumber\pgfarrowinset}{\pgf@sys@tonumber\pgfarrowwidth}%
    \let\pgf@temp@quot\pgfmathresult%
    \pgf@x\pgfmathresult pt%
    \pgf@x\pgfmathresult\pgf@x%
    \pgf@x4\pgf@x%
    \advance\pgf@x by1pt%
    \pgfmathsqrt@{\pgf@sys@tonumber\pgf@x}%
    \pgf@yc\pgfmathresult\pgfarrowlinewidth% yc is inset miter
    \pgf@yc.5\pgf@yc%
    % Inner length (pgfutil@tempdima) is now arrowlength - front miter - back miter
    \pgfutil@tempdima\pgfarrowlength%
    \advance\pgfutil@tempdima by-\pgf@xc%
    \advance\pgfutil@tempdima by-\pgf@ya%
    \pgfutil@tempdimb.5\pgfarrowwidth%
    \advance\pgfutil@tempdimb by-\pgf@yb%
    % harpoon miter correction
    \ifpgfarrowroundjoin
      \pgfarrowssetbackend{\pgf@ya\advance\pgf@x by-.5\pgfarrowlinewidth}
    \else
      \pgfarrowssetbackend{0pt}
    \fi
    \ifpgfarrowharpoon
      \pgfarrowssetlineend{\pgfarrowinset\advance\pgf@x
        by\pgf@yc\advance\pgf@x by.5\pgfarrowlinewidth}
    \else
      \pgfarrowssetlineend{\pgfarrowinset\advance\pgf@x by\pgf@yc\advance\pgf@x by-.25\pgfarrowlinewidth}
      \ifpgfarrowreversed
        \ifdim\pgfinnerlinewidth>0pt
          \pgfarrowssetlineend{\pgfarrowinset}
        \else
          \pgfarrowssetlineend{\pgfutil@tempdima\advance\pgf@x by\pgf@ya\advance\pgf@x by-.25\pgfarrowlinewidth}
        \fi
      \fi
    \fi
    \ifpgfarrowroundjoin
      \pgfarrowssettipend{\pgfutil@tempdima\advance\pgf@x by\pgf@ya\advance\pgf@x by.5\pgfarrowlinewidth}
```

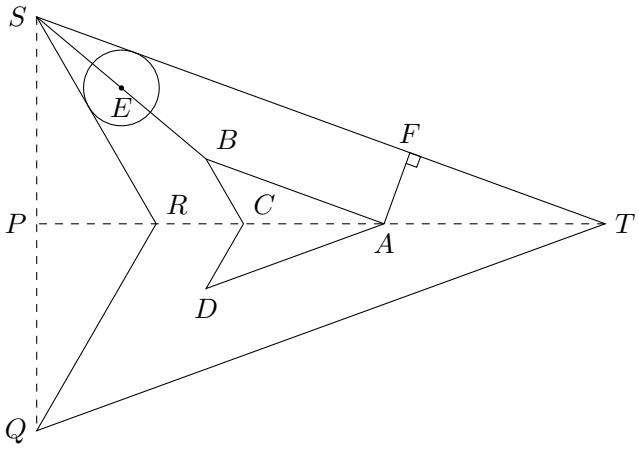
```
84      \else
85        \pgfarrowssettipend{\pgfarrowlength\ifpgfarrowharpoon\advance\pgf@x by\pgf@xa\fi}
86      \fi
87      % The hull:
88      \pgfarrowshullpoint{\pgfarrowlength\ifpgfarrowroundjoin\else\ifpgfarrowharpoon\advance\pgf@x by\pgf@xa\fi\fi}{\ifpgfarrowharpoon-.5\pgfarrowlinewidth\else0pt\fi}%
89      \pgfarrowsupperhullpoint{0pt}{.5\pgfarrowwidth}%
90      \pgfarrowshullpoint{\pgfarrowinset}{\ifpgfarrowharpoon-.5\pgfarrowlinewidth\else0pt\fi}%
91      % Adjust inset
92      \pgfarrowssetvisualbackend{\pgfarrowinset}
93      \advance\pgfarrowinset by\pgf@yc%
94      % The following are needed in the code:
95      \pgfarrowssavethe\pgfutil@tempdima
96      \pgfarrowssavethe\pgfutil@tempdimb
97      \pgfarrowssavethe\pgfarrowlinewidth
98      \pgfarrowssavethe\pgf@ya
99      \pgfarrowssavethe\pgfarrowinset
100   },
101   drawing code = {
102     \pgfsetdash{}{+0pt}
103     \ifpgfarrowroundjoin\pgfsetroundjoin\else\pgfsetmiterjoin\fi
104     \ifdim\pgfarrowlinewidth=\pgflinewidth\else\pgfsetlinewidth{+\pgfarrowlinewidth}\fi
105     \pgfpathmoveto{\pgfqpoint{\pgfutil@tempdima\advance\pgf@x by\pgf@ya}{0pt}}
106     \pgfpathlineto{\pgfqpoint{\pgf@ya}{\pgfutil@tempdimb}}
107     \pgfpathlineto{\pgfqpoint{\pgfarrowinset}{0pt}}
108     \ifpgfarrowharpoon \else
109     \pgfpathlineto{\pgfqpoint{\pgf@ya}{-\pgfutil@tempdimb}}
110     \fi
111     \pgfpathclose
112     \ifpgfarrowopen\pgfusepathqstroke\else\ifdim\pgfarrowlinewidth>0pt\pgfusepathqfillstroke\else\pgfusepathqfill\fi\fi
113   },
114   parameters = {
115     \the\pgfarrowlinewidth,%
116     \the\pgfarrowlength,%
117     \the\pgfarrowwidth,%
118     \the\pgfarrowinset,%
119     \ifpgfarrowharpoon h\fi%
120     \ifpgfarrowopen o\fi%
121     \ifpgfarrowroundjoin j\fi%
122   },
123 }%
```

上面定义，第 2 行规定箭头名称为 Stealth.

第 114 行规定所用的参数，有：

- 箭头线宽 \the\pgfarrowlinewidth, 参考选项 /pgf/arrow keys/line width→P. 790. 记为 a_{lw}pt.
- 箭头长度 \the\pgfarrowlength, 参考选项 /pgf/arrow keys/length→P. 786. 记为 a_{l}pt.
- 箭头宽度 \the\pgfarrowwidth, 参考选项 /pgf/arrow keys/width→P. 787. 记为 a_{w}pt.
- 箭头内凹尺寸 \the\pgfarrowinset, 参考选项 /pgf/arrow keys/inset→P. 787. 记为 a_{i}pt.

- 条件判断 \ifpgfarrowharpoon h\fi, 参考选项 /pgf/arrow keys/harpoon→P. 788, 参考选项 /pgf/arrow keys/left→P. 789.
- 条件判断 \ifpgfarrowopen o\fi, 参考选项 /pgf/arrow keys/open→P. 789.
- 条件判断 \ifpgfarrowroundjoin j\fi, 参考选项 /pgf/arrow keys/line join→P. 790.



a_{lw}pt 对应上面图形中线段 AF 的长度，圆 E 的直径等于 AF.

第 9 行，开始 setup code, 计算各种参数，设置特征点。

第 11 到 16 行，检查 a_{lw}pt 是否恰当，如果 $a_{lw} > \frac{a_l - a_i}{4}$, 则令 $a_{lw} = \frac{a_l - a_i}{4}$.

第 18 行，命令 \pgf@sys@tonumber 将 \pgfarrowlength 的长度单位转换为 pt, 然后把单位 pt 去掉，返回长度的数值部分。这一行计算比例 $\frac{a_l}{a_w}$.

第 19 行，将比例 $\frac{a_l}{a_w}$ 保存到 \pgf@temp@quot 中。

第 20 到 23 行，把 \pgf@x 的值变成 $\left((\frac{2a_l}{a_w})^2 + 1\right)$pt.

第 24 到 26 行，把 \pgf@xc 的值变成 $\frac{a_{lw}}{2}\sqrt{(\frac{2a_l}{a_w})^2 + 1}$pt, 即上图中的 $\frac{AT}{2}$ 的长度。

第 27 行，把 \pgf@xa 的值变成 $\frac{a_l}{a_w}a_{lw}$pt.

第 29 行，把 \pgf@ya 的值变成 $\frac{a_w}{2}$pt.

第 30, 31 行，把 \pgf@yb 的值变成 $\arctan(\frac{2a_l}{a_w})$pt.

第 32, 33 行，把 \pgf@ya 的值变成 $\arctan(\frac{2a_i}{a_w})$pt.

第 34, 35 行，把 \pgf@yb 的值变成 $\frac{\arctan(\frac{2a_l}{a_w}) - \arctan(\frac{2a_i}{a_w})}{2}$pt.

第 36 到 39 行，把 \pgf@yc 的值变成 $\frac{a_{lw}}{2\tan\frac{\arctan(\frac{2a_l}{a_w}) - \arctan(\frac{2a_i}{a_w})}{2}}$pt.

第 40 到 43 行，把 \pgf@ya 的值变成 $\sin(\frac{\arctan(\frac{2a_l}{a_w}) + \arctan(\frac{2a_i}{a_w})}{2}) \cdot \frac{a_{lw}}{2\tan\frac{\arctan(\frac{2a_l}{a_w}) - \arctan(\frac{2a_i}{a_w})}{2}}$pt.

把 \pgf@yb 的值变成 $\cos(\frac{\arctan(\frac{2a_l}{a_w}) + \arctan(\frac{2a_i}{a_w})}{2}) \cdot \frac{a_{lw}}{2\tan\frac{\arctan(\frac{2a_l}{a_w}) - \arctan(\frac{2a_i}{a_w})}{2}}$pt.

第 44 行，如果 $a_i = 0$, 则把 \pgf@ya 的值变成 $\frac{a_{lw}}{2}$.

第 48 到 53 行，把 \pgf@x 的值变成 $\left((\frac{2a_i}{a_w})^2 + 1\right)$pt.

第 54 到 56 行，把 \pgf@yc 的值变成 $\frac{a_{lw}}{2}\sqrt{(\frac{2a_i}{a_w})^2 + 1}$pt.

第 58 到 60 行，把 \pgfutil@tempdima 的值变成 $\left(a_l - \frac{a_{lw}}{2}\sqrt{(\frac{2a_l}{a_w})^2 + 1} - \text{\pgf@ya}\right)$pt.

第 61 到 62 行，把 \pgfutil@tempdimb 的值变成 $\left(\frac{a_w}{2} - \text{\pgf@yb}\right)$pt.

.....

第 95 到 99 行，用命令 \pgfarrowssavethe 将数个寄存器值做“特别保存”，在 101 行的 drawing code 中使用这些寄存器值。

这个箭头定义涉及的计算有点复杂！

第十七章　图样 Patterns

17.1　Overview

用图样填充一个区域的做法类似用瓷砖铺地面，可以用“图样砖”（tile）铺成图样（tiling pattern）。图样砖本身应该带有图形，需要在“图样砖坐标系”内画出它，然后在水平方向和竖直方向铺放从而得到图样 (铺好的地面)。你可以假设图样是“足够大”的，当用图样填充路径时，路径会剪切图样。为了解释如何用图样填充一个区域，约定以下：

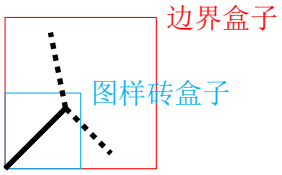
- 图样砖坐标系，这个坐标系用于绘制“图样砖图形”，指定“边界盒子”、“图样砖盒子”的对角点。
- 图样砖图形，在“图样砖坐标系”中绘制的图形。
- 图样砖图形的边界盒子，是“图样砖图形”本身的（自然的）边界盒子。
- 边界盒子，在“图样砖坐标系”中，这个盒子的左下角点、右上角点都需要指定，它的作用是剪切“图样砖图形”。
- 图样砖盒子，在“图样砖坐标系”中，这个盒子的左下角点是原点，右上角点需要指定，注意它与“图样砖图形的边界盒子”不同。被“边界盒子”剪切后的“图样砖图形”会被放到“图样砖盒子”中。

与单块瓷砖对应的是“图样砖盒子”，也就是说，在水平方向和竖直方向铺放的是“图样砖盒子”。一个“图样砖盒子”附带着一个 (被剪切的)“图样砖图形”(用“边界盒子”剪切“图样砖图形”得到的图形)。所以，若“边界盒子”、“图样砖盒子”、“图样砖图形的边界盒子”这三个盒子相差过大，就会出现 (被剪切的)“图样砖图形”相重叠的情况。

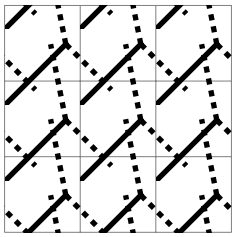
例如，下面代码定义图样 SlopeLine:

```
\pgfdeclarepatternformonly{SlopeLine}{\pgfpointorigin}{\pgfpoint{2cm}{2cm}}{\pgfpoint
↪  {1cm}{1cm}}
{
  \pgfsetlinewidth{2pt}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0.8cm}{0.8cm}}
  \pgfusepath{draw}
  \pgfsetdash{{0.8mm}{0.8mm}}{0mm}
  \pgfsetlinewidth{2pt}
  \pgfpathmoveto{\pgfpoint{0.8cm}{0.8cm}}
  \pgfpathlineto{\pgfpoint{0.6cm}{1.8cm}}
  \pgfpathmoveto{\pgfpoint{0.8cm}{0.8cm}}
  \pgfpathlineto{\pgfpoint{1.4cm}{0.2cm}}
  \pgfusepath{draw}
}
```

上面代码规定的“图样单元”如下所示：



其中的图样图形是三个线段，其尺寸超出图样砖盒子，而边界盒子又把该图形包含在内，所以使用这个图样时，会出现虚线线段与实线线段交叠的情况，如下：



```
\tikz{
  \draw [help lines](0,0) grid (3,3);
  \path [pattern=SlopeLine] (0,0) rectangle (3,3);
}
```

还要注意，同样的图样定义在 standalone 文类下的外观，与在 article 文类或 ctexart 文类下的外观可能不相同。

图样本身可能缺少“可修改性”，例如，对于多数图样来说，一旦定义（声明）它后，就不能再改变它的形状、线宽，也不能对它做比例变换。图样的“可修改性”主要与 PGF 本身有关（而非绘图语言），这取决于如何定义图样。有的图样具有某些“可修改性”，当然处理这样的图样会困难一些。

PGF 会直接把图样映射到图形语言（PostScript, pdf, svg）的图样机制中。当用图样填充路径时，PostScript, pdf 与 svg 的处理有共同点，也有不同点。它们都是先确定第一个图样砖的位置，然后在水平方向和竖直方向不断重复铺放图样砖，从而得到图样，然后再用被填充路径剪切图样。所以，第一个图样砖的位置——它的“图样坐标系”原点的位置，对图样的填充外观 (相位) 有一定的影响。被填充路径有自己的边界盒子，SVG 规定第一个图样砖的“原点”位于被填充路径的边界盒子的左上角。而 PostScript 和 pdf 则在整个图形中选定了一个固定点，将第一个图样砖的“原点”放在这个固定点上，所以在整个图形中，图样的位置不会因为被填充路径的变化而变化。

17.2 声明一个图样

先声明一个图样，然后使用它。图样分为两种类型：inherently colored patterns 和 form-only patterns，前者有固定颜色，其颜色不可变；后者只有固定的轮廓线条，没有固定颜色，其颜色可变。每种类型的图样又可以分为两类：mutable 类和 unmutable 类，mutable 类的图样具有可修改性。

同一个图样名称不能被声明两次。

\pgf@declarepatternmutable{⟨*name*⟩}{⟨*variables*⟩}{⟨*bottom left*⟩}{⟨*top right*⟩}{⟨*tile size*⟩}{⟨*code*⟩}{⟨*type number*⟩}

本命令用于声明 mutable 类型的图样。

⟨*name*⟩ 是图样的名称。

⟨*variables*⟩ 是能修改图样外观的变量，见 \pgfsetfillpattern →P. 355 调用的 \pgf@pattern@check@vars.

在“图样砖坐标系”中以坐标点 ⟨*bottom left*⟩ 和 ⟨*top right*⟩ 为对角点构建一个矩形的“边界盒子”，这个边界盒子用来“剪切”图样砖中的图形。

以“图样砖坐标系”的原点和坐标点 ⟨*tile size*⟩ 为对角点构建一个矩形的“图样砖盒子”，这个图样砖盒子相当于铺地面的单块瓷砖。

⟨*code*⟩ 是在“图样砖坐标系”内绘制图样图形的代码，必须是能够被“protocolled”的 PGF 绘图命令。注意其中不能使用命令 \color, 此命令不能被“protocolled”。

参数 ⟨*type number*⟩ 是数字 2 (formonly) 或者 3 (inherentlycolored).

本命令首先检查控制序列 \csname pgf@pattern@name@⟨*name*⟩\endcsname 是否已定义，即检查图样 ⟨*name*⟩ 是否已被声明，如果已被声明就报错；如果未被声明，就全局地保存它的参数。

本命令的定义是：

```
\def\pgf@declarepatternmutable#1#2#3#4#5#6#7{%
  \pgfutil@ifundefined{pgf@pattern@name@#1}{%
    \expandafter\gdef\csname pgf@pattern@name@#1\endcsname{#1}%
    \expandafter\gdef\csname pgf@pattern@variables@#1\endcsname{#2}%
    \expandafter\gdef\csname pgf@pattern@lowerleft@#1\endcsname{#3}%
    \expandafter\gdef\csname pgf@pattern@upperright@#1\endcsname{#4}%
    \expandafter\gdef\csname pgf@pattern@tilesize@#1\endcsname{#5}%
    \expandafter\long\expandafter\gdef\csname pgf@pattern@code@#1\endcsname{#6}%
    \expandafter\gdef\csname pgf@pattern@type@#1\endcsname{#7}%
  }{\pgferror{The pattern `#1' is already defined}}%
}
```

\pgf@declarepattern{⟨*name*⟩}{⟨*bottom left*⟩}{⟨*top right*⟩}{⟨*tile size*⟩}{⟨*code*⟩}{⟨*type number*⟩}

本命令用于声明 unmutable 类型的图样。

本命令的参数如前一命令的参数，参数 ⟨*type number*⟩ 是 0 (可变色的图样，formonly) 或者 1 (不可变色的图样，inherentlycolored)。

本命令首先检查控制序列 \csname pgf@pattern@name@⟨*name*⟩\endcsname 是否已定义，即检查图样 ⟨*name*⟩ 是否已被声明，如果已被声明就报错；如果未被声明，就如下操作：

```
\pgfsysprotocol@getcurrentprotocol\pgf@pattern@temp%
{%
  \pgfinterruptpath%
    \pgfpicturetrue%
    \pgf@relevantforpicturesizefalse%
    \pgftransformreset%
    \pgfsysprotocol@setcurrentprotocol\pgfutil@empty%
    \pgfsysprotocol@bufferedtrue%
    \pgfsys@beginscope%
    \pgfsetarrows{-}%
    #5%
    \pgfsys@endscope%
    \pgfsysprotocol@getcurrentprotocol\pgf@pattern@code%
    \global\let\pgf@pattern@code=\pgf@pattern@code%
  \endpgfinterruptpath%
  \pgf@process{#2}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{#3}%
  \pgf@xb=\pgf@x%
  \pgf@yb=\pgf@y%
  \pgf@process{#4}%
  \pgf@xc=\pgf@x%
  \pgf@yc=\pgf@y%
  % Now, build a name for the pattern
  \pgfutil@tempcnta=\pgf@pattern@number%
  \advance\pgfutil@tempcnta by1\relax%
  \xdef\pgf@pattern@number{\the\pgfutil@tempcnta}%
  \expandafter\xdef\csname pgf@pattern@name@#1\endcsname{\the\pgfutil@tempcnta}%
  \expandafter\gdef\csname pgf@pattern@type@#1\endcsname{#6}%
  \xdef\pgf@marshal{\noexpand\pgfsys@declarepattern
```

```
32      {\csname pgf@pattern@name@#1\endcsname}% name
33      {\the\pgf@xa}{\the\pgf@ya}{\the\pgf@xb}{\the\pgf@yb}% bbox
34      {\the\pgf@xc}{\the\pgf@yc}% x/y step
35      \pgfsys@patternmatrix% transformation matrix
36      {\pgf@pattern@code}{#6}}%
37  }%
38  \pgfsysprotocol@setcurrentprotocol\pgf@pattern@temp%
39  \expandafter\global\expandafter\let\csname pgf@pattern@instantiate@#1\endcsname=
    ↪  \pgf@marshal%
```

1 行 保存当前的 protocol 缓存内容

2 行 开启一个组

3 行 中断当前路径

5 行 取消针对整个图形的边界盒子的计算

6 行 设置单位矩阵

7 行 清空当前的 protocol 缓存

8 行 开启 protocol 缓存模式

10 行 取消箭头

11 行 执行绘制图样的代码 ⟨*code*⟩, 并做 protocol 缓存

13 行 将绘制图样的代码的 protocol 缓存保存到宏 `\pgf@pattern@code`

29 行 定义控制序列 `\csname pgf@pattern@name@`⟨*name*⟩`\endcsname`, 保存一个编号

30 行 定义控制序列 `\csname pgf@pattern@type@`⟨*name*⟩`\endcsname`, 保存图样的类型，即数字 0 (不可变色的图样) 或者 1 (可变色的图样)。

31 行 保存用于创建图样的系统层命令

37 行 结束之前开启的组

38 行 恢复之前的 protocol 缓存

39 行 全局定义 `\csname pgf@pattern@instantiate@`⟨*name*⟩`\endcsname`, 保存创建图样的系统层命令

`\pgfdeclarepatternformonly[`⟨*variables*⟩`]{`⟨*name*⟩`}{`⟨*bottom left*⟩`}{`⟨*top right*⟩`}{`⟨*tile size*⟩`}{`⟨*code*⟩`}`

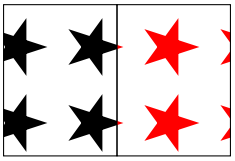
本命令的声明是全局有效的。本命令的参数中可以含有 `\par`.

如果不写出 `[`⟨*variables*⟩`]` 或 ⟨*variables*⟩ 是空的，那么本命令声明 unmutable 的图样，此时导致：

```
\pgf@declarepattern{⟨name⟩}{⟨bottom left⟩}{⟨top right⟩}{⟨tile size⟩}{⟨code⟩}{0}
```

如果 ⟨*variables*⟩ 非空，那么本命令声明 mutable 的图样，此时导致：

```
\pgf@declarepatternmutable
↪  {⟨name⟩}{⟨variables⟩}{⟨bottom left⟩}{⟨top right⟩}{⟨tile size⟩}{⟨code⟩}{2}
```



```
\pgfdeclarepatternformonly{stars}{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}{\pgfpoint
↪  {1cm}{1cm}}
{
  \pgftransformshift{\pgfpoint{.5cm}{.5cm}}
  \pgfpathmoveto{\pgfpointpolar{0}{4mm}}
  \pgfpathlineto{\pgfpointpolar{144}{4mm}}
  \pgfpathlineto{\pgfpointpolar{288}{4mm}}
  \pgfpathlineto{\pgfpointpolar{72}{4mm}}
```

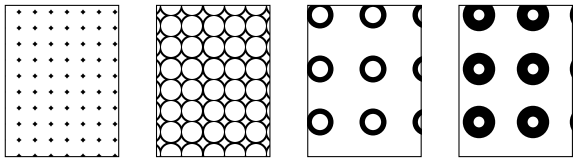
```
  \pgfpathlineto{\pgfpointpolar{216}{4mm}}
  \pgfpathclose%
  \pgfusepath{fill}
}
\begin{tikzpicture}
  \filldraw[pattern=stars] (0,0) rectangle (1.5,2);
  \filldraw[pattern=stars,pattern color=red] (1.5,0) rectangle (3,2);
\end{tikzpicture}
```

本命令的可选选项 ⟨*variables*⟩ 是一个用逗号分隔的列表，列表项可以是宏（macro），寄存器（registers），键（key）。如果要列出某个 key，则需要写出它的完整路径。注意所列出的宏和键应当是“简单”的，即它们仅仅用来保存一个值（values），而不是展开为复杂的命令组合或运算。例如，如果列表项是一个键，那么应当可以使用 \pgfkeysvalueof 获取这个键的值。

见 \pgfsetfillpattern →P. 355 调用的 \pgf@pattern@check@vars.

⟨*variables*⟩ 中列出的各个项目用在定义图样的 ⟨*code*⟩ 中，其中的绘图命令将这些项目作为变量，使得所定义的图样具有“可修改性”。

当 ⟨*variables*⟩ 非空时，本命令实际上并不创建图样，只是保存各个变量，因此 ⟨*variables*⟩ 中的各个项目不必提前定义。当使用图样填充路径时，在使用图样做填充之前，需要为各个变量赋值。PGF 能够自动区分宏、寄存器、键。



```
\pgfdeclarepatternformonly[/tikz/radius,\thickness,\size]{rings}
  {\pgfpoint{-0.5*\size}{-0.5*\size}}
  {\pgfpoint{0.5*\size}{0.5*\size}}
  {\pgfpoint{\size}{\size}}
  {
    \pgfsetlinewidth{\thickness}
    \pgfpathcircle\pgfpointorigin{\pgfkeysvalueof{/tikz/radius}}
    \pgfusepath{stroke}
  }
\newdimen\thickness
\tikzset{
  radius/.initial=4pt,
  size/.store in=\size, size=20pt,
  thickness/.code={\thickness=#1},
  thickness=0.75pt
}
\begin{tikzpicture}[rings/.style={pattern=rings}]
  \filldraw [rings, radius=2pt, size=6pt] (0,0) rectangle +(1.5,2);
  \filldraw [rings, radius=2pt, size=8pt, pattern color=red] (2,0) rectangle
  ↪  +(1.5,2);
  \filldraw [rings, radius=6pt, thickness=2pt] (4,0) rectangle +(1.5,2);
  \filldraw [rings, radius=8pt, thickness=4pt, pattern color=green] (6,0)
  ↪  rectangle +(1.5,2);
\end{tikzpicture}
```

⟨*variables*⟩ 只是被单纯地保存到 \pgf@pattern@tempvars 中，并不会被立即处理。只有在执行 \pgfsetfillpattern →P. 355 时，才会被 \pgf@pattern@check@vars 处理。

\pgfdeclarepatterninherentlycolored[⟨*variables*⟩]{⟨*name*⟩}{⟨*lower left*⟩}{⟨*upper right*⟩}
{⟨*tile size*⟩}{⟨*code*⟩}

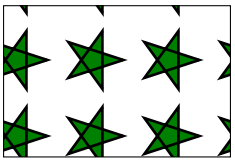
本命令的声明是全局有效的。本命令的参数中可以含有 \par.
如果不写出 [⟨variables⟩] 或 ⟨variables⟩ 是空的，那么本命令声明 unmutable 的图样，此时导致：

```
\pgf@declarepattern{⟨name⟩}{⟨bottom left⟩}{⟨top right⟩}{⟨tile size⟩}{⟨code⟩}{1}
```

如果 ⟨variables⟩ 非空，那么本命令声明 mutable 的图样，此时导致：

```
\pgf@declarepatternmutable
↪   {⟨name⟩}{⟨variables⟩}{⟨bottom left⟩}{⟨top right⟩}{⟨tile size⟩}{⟨code⟩}{3}
```

本命令声明一个 inherently colored pattern, 各个参数的意思与上一个命令类似，只是需要在 ⟨code⟩ 中使用 PGF 的颜色命令来设置图样砖图形的颜色。注意不能使用命令 \color, 此命令不能被 “protocolled”。



```
\pgfdeclarepatterninherentlycolored{green stars}
{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
{\pgfpoint{1cm}{1cm}}
{
  \pgfsetfillcolor{green!50!black}
  \pgftransformshift{\pgfpoint{.5cm}{.5cm}}
  \pgfpathmoveto{\pgfpointpolar{0}{4mm}}
  \pgfpathlineto{\pgfpointpolar{144}{4mm}}
  \pgfpathlineto{\pgfpointpolar{288}{4mm}}
  \pgfpathlineto{\pgfpointpolar{72}{4mm}}
  \pgfpathlineto{\pgfpointpolar{216}{4mm}}
  \pgfpathclose%
  \pgfusepath{stroke,fill}
}
\begin{tikzpicture}
  \filldraw[pattern=green stars] (0,0) rectangle (3,2);
\end{tikzpicture}
```

以上命令可以总结为：
根据 \pgfdeclarepatternformonly → P. 352 与 \pgfdeclarepatterninherentlycolored → P. 353 的可选项 ⟨variables⟩ 是否非空，分别如下调用：

操作	formonly	inherently colored
mutable ⟨variables⟩ 非空	2 \pgf@declarepattern-formonly@mutable 调用 \pgf@declarepatternmutable	3 \pgf@declarepatterninherently-colored@mutable 调用 \pgf@declarepatternmutable
unmutable ⟨variables⟩ 空	0 \pgf@declarepattern-formonly 调用 \pgf@declarepattern	1 \pgf@declarepatterninherently-colored 调用 \pgf@declarepattern

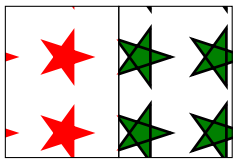
17.3 使用图样

声明一个图样后，就可以用命令 \pgfsetfillpattern 使用它：

- 如果 ⟨*name*⟩ 是 mutable 类型的图样，那么图样名称 ⟨*name*⟩ 以及与此图样有关的变量 ⟨*variables*⟩ 的值会被看作是一个组合。当第一次使用这个组合时，会解析变量 ⟨*variables*⟩ 的值，并把这个 mutable 类型的图样改型为 unmutable 类型的图样，然后用图样填充路径。当再次使用这个组合时，就不必重复“解析、改型”的操作，而是可以直接使用。
- 如果 ⟨*name*⟩ 是 unmutable 类型的图样，那么这个图样会被直接用于填充路径。

\pgfsetfillpattern{⟨*name*⟩}{⟨*color*⟩}

本命令指定图样 ⟨*name*⟩，提供给之后的填充路径的命令使用。如果 ⟨*name*⟩ 是 inherently colored pattern，则忽略 ⟨*color*⟩；如果 ⟨*name*⟩ 是 form-only pattern，则用颜色 ⟨*color*⟩ 填充图样 ⟨*name*⟩，然后再用图样填充路径。



```
\begin{tikzpicture}
  \pgfsetfillpattern{stars}{red}
  \filldraw (0,0) rectangle (1.5,2);
  \pgfsetfillpattern{green stars}{red}
  \filldraw (1.5,0) rectangle (3,2);
\end{tikzpicture}
```

这个命令首先检查控制序列 \csname pgf@pattern@name@⟨*name*⟩\endcsname 是否已定义，即检查名称为 ⟨*name*⟩ 的图样是否已定义，如果未定义就报错；如果已定义，则再检查控制序列 \csname pgf@pattern@type@⟨*name*⟩\endcsname 的值是否 > 1，即检查图样 ⟨*name*⟩ 是否 mutable 类型的：

- 如果图样 ⟨*name*⟩ 是 mutable 类型的，
  1. 先清空\pgf@pattern@tempvars，
  2. 调用 \pgf@pattern@check@vars 解析

     \csname pgf@pattern@variables@⟨*name*⟩\endcsname

     保存的 ⟨*variables*⟩. 所保存的 ⟨*variables*⟩ 应当是一个用逗号分隔的列表，对各列表项逐一解析，解析步骤是：首先根据列表项的第一个记号来判断其类别，
     - 如果列表项的第一个记号是“/”，那就认为是它是一个“完整的键”，此时做完全展开定义：

       ```
       \edef\pgf@pattern@tempvar{(\pgfkeysvalueof{⟨ 完整键⟩})}%
       ```

       注意其中用圆括号包裹键值。
     - 如果列表项的第一个记号的类代码与 \relax 相同，就认为它是寄存器，此时做展开定义：

       ```
       \edef\pgf@pattern@tempvar{(\the⟨ 寄存器⟩)}%
       ```

       注意其中用圆括号包裹寄存器值。
     - 如果列表项不是以上 2 个情况，就认为它是宏，此时做完全展开定义：

       ```
       \edef\pgf@pattern@tempvar{(⟨ 宏⟩)}%
       ```

       注意其中用圆括号包裹宏的值。

     所以 ⟨*variables*⟩ 中的列表项只能是以上能被解析、并顺利做定义的对象。然后

     ```
     \edef\pgf@pattern@tempvars{\pgf@pattern@tempvars\pgf@pattern@tempvar}%
     ```

     在对各个列表项做如上解析后，各个列表项的最后执行结果 (分别用圆括号括起来) 就都保存到了 \pgf@pattern@tempvars 中。
  3. 做完全展开定义：

```
\edef\pgf@pattern@nametemp{pgf@pattern@#1@\pgf@pattern@tempvars}%
```

注意这个宏保存了图样的 (带前缀的) 名称和变量值。

4. 再检查控制序列 \csname\pgf@pattern@nametemp\endcsname 是否已定义
   - 如果已定义，则什么也不做
   - 如果未定义，则检查当前声明的图样是否 inherently colored 类型，也就是检查控制序列 \csname pgf@pattern@type@⟨*name*⟩\endcsname 保存的值是否奇数。如果是则执行

```
\pgfdeclarepatterninherentlycolored
  {\pgf@pattern@nametemp}%
  {\csname pgf@pattern@lowerleft@⟨name⟩\endcsname}%
  {\csname pgf@pattern@upperright@⟨name⟩\endcsname}%
  {\csname pgf@pattern@tilesize@⟨name⟩\endcsname}%
  {\csname pgf@pattern@code@⟨name⟩\endcsname}%
```

这导致 \pgf@declarepattern → P. 351 进而导致 \pgfsys@declarepattern → P. 209, 正式声明一个图样，也会定义控制序列

\csname\pgf@pattern@nametemp\endcsname

如果不是则执行

```
\pgfdeclarepatternformonly
  {\pgf@pattern@nametemp}%
  {\csname pgf@pattern@lowerleft@⟨name⟩\endcsname}%
  {\csname pgf@pattern@upperright@⟨name⟩\endcsname}%
  {\csname pgf@pattern@tilesize@⟨name⟩\endcsname}%
  {\csname pgf@pattern@code@⟨name⟩\endcsname}%
```

这导致 \pgf@declarepattern → P. 351 进而导致 \pgfsys@declarepattern → P. 209, 正式声明一个图样，也会定义控制序列

\csname\pgf@pattern@nametemp\endcsname

5. 引用图样

```
\expandafter\pgf@set@fillpattern\expandafter{\pgf@pattern@nametemp}{⟨color⟩}
↪  %
```

6. 全局定义

```
\expandafter\gdef\csname\pgf@pattern@nametemp\endcsname{⟨name⟩}%
```

- 如果图样 ⟨*name*⟩ 不是 mutable 类型的，则直接引用图样

```
\pgf@set@fillpattern{⟨name⟩}{⟨color⟩}%
```

文件《pgflibrarypatterns.meta.code.tex》会重定义 \pgfsetfillpattern → P. 355.

\pgf@set@fillpattern{⟨*name*⟩}{⟨*color*⟩}

这个命令首先检查控制序列 \csname pgf@pattern@name@⟨*name*⟩\endcsname 是否已定义，即检查名称为 ⟨*name*⟩ 的图样是否已定义，如果未定义就报错；如果已定义，则

1. 执行

\csname pgf@pattern@instantiate@⟨*name*⟩\endcsname

参考 \pgf@declarepattern → P. 351.

2. 然后删除这个控制序列

```
\expandafter\global\expandafter\let\csname pgf@pattern@instantiate@⟨name⟩
↪  \endcsname=\relax%
```

3. 检查当前的图样是否 inherently colored 类型，也就是检查控制序列

\csname pgf@pattern@type@⟨*name*⟩\endcsname

保存的值是否奇数。

- 如果是则执行

```
\pgfsys@setpatterncolored{\csname pgf@pattern@name@#1\endcsname}%
```

创建图样。

- 如果不是，那么：如果使用了 PGF 的 xxcolor 宏包，就用 \applycolormixins 处理颜色 ⟨*color*⟩; 然后将这个颜色转为 rgb 颜色模式 (数据 ⟨*r*⟩, ⟨*g*⟩, ⟨*b*⟩)；然后执行

```
\pgfsys@setpatternuncolored{\csname pgf@pattern@name@⟨name⟩\endcsname}{⟨r⟩}{
↪  ⟨g⟩}{⟨b⟩}%
```

创建图样。

第十八章　图层

18.1　Overview

PGF 提供分层绘图机制。设想有两块玻璃板，每一块玻璃板上都画有图形，把两块玻璃板上下叠放，那么上层玻璃板的图形会（全部或部分地）遮挡下层玻璃板的图形。下面的玻璃板可以看作是“背景图层”（background layer），上面的玻璃板可以看作是“前端图层”（foreground layer）。在 PGF 中，你可以声明多个图层并规定它们的上下叠放次序，每个图层上都可以画出属于“本图层”的图形，按图层的叠放次序，这些图形有其“遮挡次序”。

参考文件《pgfcorelayers.code.tex》。

18.2　声明图层

PGF 的图层都有自己的名称，预定义的图层是 `main`. 每个图层名称都对应自己的盒子，一个图层内的代码都被放入相应的盒子里。

`\pgfdeclarelayer{`⟨*name*⟩`}`

这个命令声明一个图层，其名称为 ⟨*name*⟩. 你可以多次使用本命令声明多个图层。如果 ⟨*name*⟩ 是某个已经被声明的图层名称，那么本命令什么也不做。

这个命令实际上使用 `\pgf@newbox` (等效于 `\newbox`) 声明 2 个盒子：

- `\csname pgf@layerbox@`⟨*name*⟩`\endcsname`
- `\csname pgf@layerboxsaved@`⟨*name*⟩`\endcsname`

在文件《pgfcorelayers.code.tex》中没有用命令 `\pgfdeclarelayer{main}` 声明关于 `main` 的盒子，但在文件《pgfcorescopes.code.tex》中有

```
\newbox\pgf@layerbox@main
```

注意没有名称为 `pgf@layerboxsaved@main` 的盒子。

PGF 对 `main` 图层的处理与其他图层有所不同。

`\pgfsetlayers{`⟨*layer list*⟩`}`

⟨*layer list*⟩ 是由已经声明的图层名称构成的列表，名称之间用逗号分隔，其中必须包含图层 `main`. 本命令按照这个列表次序，规定各个图层 (自下而上) 的叠放次序。例如：

```
\pgfdeclarelayer{background layer}
\pgfdeclarelayer{foreground layer}
\pgfsetlayers{background layer,main,foreground layer}
```

这个命令可以用在绘图环境之外，也可以用在绘图环境内。如果本命令用在绘图环境内，那么要确保本命令与 `\end{pgfpicture}` 之间没有 TEX 分组的结束标志，也不能有其它以 `\end` 开头的控制语句。当然你可以把本命令放在 `\end{pgfpicture}` 的前面，这也是有效的。

⟨*layer list*⟩ 中的图层名称先被彻底展开，名称首尾的空格会被忽略，然后本命令将图层名称列表保存到宏 `\pgf@layerlist` 中。文件《pgfcorelayers.code.tex》中有命令

```
\pgfsetlayers{main}
```

这就定义了 `\pgf@layerlist`, 当用户再执行 `\pgfsetlayers` 时，就会重定义宏 `\pgf@layerlist`.

18.3 在图层上绘图

在绘图环境中，如果不指明一个绘图命令属于哪个图层，那么这个绘图命令就属于 `main` 层。

`\begin{pgfonlayer}{`⟨*layer name*⟩`}`
⟨*environment content*⟩
`\end{pgfonlayer}`

本环境用在绘图环境 `{pgfpicture}` 内或者子绘图环境 `{pgfscope}` 内。

⟨*layer name*⟩ 是已经被 `\pgfsetlayers` 排序的某一个图层名称，⟨*environment contents*⟩ 是绘图命令。本环境指定 ⟨*environment contents*⟩ 属于图层 ⟨*layer name*⟩.

如果在绘图环境内针对某个图层多次使用这个环境，那么 ⟨*environment contents*⟩ 会在该图层上累计。注意最好不要针对 `main` 使用本环境，下面代码：

```
\begin{pgfonlayer}{main}
⟨environment contents⟩
\end{pgfonlayer}
```

实际上等价于

```
\begingroup
⟨environment contents⟩
\endgroup
```

环境内容 ⟨*environment contents*⟩ 会被一个组包裹起来，这个组会成为盒子 `\pgf@layerbox@main` 的内容的一部分。

```
\pgfonlayer{⟨layer name⟩}
  ⟨environment contents⟩
\endpgfonlayer
```

这是 plain TeX 中的用法。

```
\startpgfonlayer{⟨layer name⟩}
  ⟨environment contents⟩
\stoppgfonlayer
```

这是 ConTeXt 中的用法。


```
\pgfdeclarelayer{background layer}
\pgfdeclarelayer{foreground layer}
\pgfsetlayers{background layer,main,foreground layer}
\begin{tikzpicture}
  \fill[blue] (0,0) circle (1cm);  % 在 main 层上
  \begin{pgfonlayer}{background layer}  % 在 background layer 层上
    \fill[yellow] (-1,-1) rectangle (1,1);
  \end{pgfonlayer}
  \begin{pgfonlayer}{foreground layer}  % 在 foreground layer 层上
```

```
    \node[white] {foreground};
  \end{pgfonlayer}
  \begin{pgfonlayer}{background layer}  % 在 background layer 层上
    \fill[black] (-.8,-.8) rectangle (.8,.8);
  \end{pgfonlayer}
  \fill[blue!50] (-.5,-1) rectangle (.5,1);  % 在 main 层上
\end{tikzpicture}
```

18.4 关于环境 pgfonlayer

18.4.1 命令 \pgfonlayer

命令 \begin{pgfonlayer}{⟨*layer name*⟩} 导致执行命令 \pgfonlayer{⟨*layer name*⟩}, 其处理是：

检查盒子 pgf@layerbox@⟨*layer name*⟩ 是否已被声明，如果未声明，则执行 \pgferror 报错，然后设置 2 个套嵌的组：

```
\bgroup
  \begingroup
```

这与 \endpgfonlayer 对应；如果已声明，则执行以下步骤：

1. 用 \begingroup 开启一个组
2. 定义 \edef\pgf@temp{⟨*layer name*⟩}
3. 执行一个 \ifx 语句，检查 \pgf@temp 的值，也就是检查 ⟨*layer name*⟩,
   - 如果 \pgf@temp 的值与 \pgfonlayer@name 相同，即 ⟨*layer name*⟩ 就是 main, 则定义

     ```
     \def\pgf@temp{%
       \bgroup
         \begingroup
     }%
     ```

   - 如果 \pgf@temp 的值与 \pgfonlayer@name 不相同，即 ⟨*layer name*⟩ 不是 main, 则

     (a) 执行 \let\pgfonlayer@name=\pgf@temp, 即规定其值为 ⟨*layer name*⟩

     (b) 执行 \pgfonlayer@assert@is@active, 此命令执行一个 \ifx 语句：

     – 如果 ⟨*layer name*⟩ 是 discard, 则什么也不做

     – 如果 ⟨*layer name*⟩ 不是 discard, 则

     i. 用 \begingroup 开启一个组

     ii. 定义 \def\pgfonlayer@isactive{0}

     iii. 执行 \pgf@assert@layer@is@active@loop 对 \pgf@layerlist 保存的图层名称列表做循环处理：逐个检查 \pgf@layerlist 中的图层名称，若被检查的图层名称是 ⟨*layer name*⟩, 就定义

     ```
     \def\pgfonlayer@isactive{1}
     ```

     并且结束循环处理，否则继续检查下一个图层名称；如果名称 ⟨*layer name*⟩ 不在列表中，那么什么也不做。

     iv. 如果 \pgfonlayer@isactive 的值是 0, 则执行 \pgfonlayer@assert@fail, 这会给出错误信息，提示名称 ⟨*layer name*⟩ 不在 \pgf@layerlist 保存的列表中

     v. 用 \endgroup 结束前面开启的组

     (c) 定义

```
\def\pgf@temp{%
  \expandafter\global\expandafter%
  \setbox\csname pgf@layerbox@⟨layer name⟩\endcsname=\hbox to 0pt%
    \bgroup%
      \expandafter\box\csname pgf@layerbox@⟨layer name⟩\endcsname%
      \begingroup%
}%
```

4. 设置线宽 \pgfsetlinewidth{0.4pt}
5. 执行 \pgf@temp,
   - 在 ⟨*layer name*⟩ 等于 main 时，这导致 (见前面的代码)

     ```
     \bgroup
       \begingroup
     ```

   - 在 ⟨*layer name*⟩ 不等于 main 时，这导致对盒子 \pgf@layerbox@⟨*layer name*⟩ 的定义，

     ```
     \expandafter\global\expandafter%
       \setbox\csname pgf@layerbox@⟨layer name⟩\endcsname=\hbox to 0pt%
         \bgroup%
           \expandafter\box\csname pgf@layerbox@⟨layer name⟩\endcsname%
           \begingroup%
     ```

     这是对盒子的重定义，这个盒子的第一个内容是原来的盒子内容，新内容被放到一个组中。
     此盒子定义在命令 \endpgfonlayer 那里结束。
     注意这个盒子定义是全局地。

18.4.2 命令 \endpgfonlayer

命令 \end{pgfonlayer} 导致执行命令 \endpgfonlayer, 此命令的定义是：

```
\def\endpgfonlayer{%
      \endgroup%
      \hss
    \egroup%
  \endgroup
}
```

其中：

1. 第一个 \endgroup 结束盒子 \pgf@layerbox@⟨*layer name*⟩ 定义之内的一个组，或者是 (在 ⟨*layer name*⟩ 等于 main 时) 开启的组
2. 用 \hss 插入弹性空白
3. 用 \egroup 结束盒子 \pgf@layerbox@⟨*layer name*⟩ 的定义，或者是 (在 ⟨*layer name*⟩ 等于 main 时) 开启的组
4. 第二个 \endgroup 结束命令 \pgfonlayer 开启的组

18.4.3 图层的内容

从 \pgfonlayer 与 \endpgfonlayer 的定义看，如果多次向 (不是 main 的) 图层 ⟨*layer name*⟩ 中添加内容：

```
\pgfonlayer{⟨layer name⟩}
  ⟨code 1⟩
\endpgfonlayer
\pgfonlayer{⟨layer name⟩}
```

```
  ⟨code 2⟩
\endpgfonlayer
......
```

那么会定义盒子 \pgf@layerbox@⟨layer name⟩ 为：

```
\begingroup
  \expandafter\global\expandafter%
  \setbox\csname pgf@layerbox@⟨layer name⟩\endcsname=\hbox to 0pt%
    \bgroup%
      \begingroup%
         ⟨code 1⟩
       \endgroup\hss
      \begingroup%
         ⟨code 2⟩
       \endgroup\hss
       ......
    \egroup%
\endgroup
```

18.4.4 图层与 pgfpicture 环境的配合

图层与 pgfpicture 环境的配合是通过下面的命令实现的。

参考文件《pgfcorelayers.code.tex》。

假设 \pgf@layerlist 中保存的图层名称列表是 ⟨name 1⟩,⟨name 2⟩,....

\pgf@insertlayers

本命令的定义是：

```
\def\pgf@insertlayers{%
  \expandafter\pgf@dolayer\pgf@layerlist,,\relax%
}
```

\pgf@dolayer#1,#2,\relax

这两个命令的定义是：

```
\def\pgf@maintext{main}%
\def\pgf@dolayer#1,#2,\relax{%
  \def\pgf@test{#1}%
  \ifx\pgf@test\pgf@maintext%
    \box\pgf@layerbox@main%
  \else%
    \pgfsys@beginscope%
      \expandafter\box\csname pgf@layerbox@#1\endcsname%
    \pgfsys@endscope%
  \fi%
  \def\pgf@test{#2}%
  \ifx\pgf@test\pgfutil@empty%
  \else%
    \pgf@dolayer#2,\relax%
  \fi%
}
```

命令 \pgf@insertlayers 会依次插入盒子 pgf@layerbox@⟨name i⟩, $i = 1, 2, \cdots$, 并且，如果 ⟨name i⟩ 不是 main, 那么会用环境 \pgfsys@beginscope, \pgfsys@endscope 把盒子 pgf@layerbox@⟨name i⟩ 包裹起来。

注意在文件《pgfcorescopes.code.tex》中有定义：

```
\def\pgf@insertlayers{%
  \box\pgf@layerbox@main%
}
```

可见这与文件《pgfcorelayers.code.tex》对命令 \pgf@insertlayers 的定义不一样，但在文件《pgfcore.code.tex》中规定：先载入文件《pgfcorescopes.code.tex》，再载入文件《pgfcorelayers.code.tex》。

\pgf@savelayers

本命令的定义是：

```
\def\pgf@savelayers{%
  \expandafter\pgf@dosavelayer\pgf@layerlist,,\relax%
}
```

\pgf@dosavelayer#1,#2,\relax

这两个命令的定义是：

```
\def\pgf@dosavelayer#1,#2,\relax{%
  \def\pgf@test{#1}%
  \ifx\pgf@test\pgf@maintext%
  \else%
    \setbox\csname pgf@layerboxsaved@#1\endcsname=\box\csname pgf@layerbox@#1
    ↪  \endcsname%
  \fi%
  \def\pgf@test{#2}%
  \ifx\pgf@test\pgfutil@empty%
  \else%
    \pgf@dosavelayer#2,\relax%
  \fi%
}
```

命令 \pgf@savelayers 的作用是把盒子 pgf@layerbox@⟨*name i*⟩ ($i = 1, 2, \cdots$, 且 ⟨*name i*⟩≠main) 中的内容转移到盒子 pgf@layerboxsaved@⟨*name i*⟩ 中，然后清空前者。

\pgf@restorelayers

本命令的定义是：

```
\def\pgf@restorelayers{%
  \expandafter\pgf@dorestorelayer\pgf@layerlist,,\relax%
}
```

\pgf@dorestorelayer#1,#2,\relax

这两个命令的定义是：

```
\def\pgf@dorestorelayer#1,#2,\relax{%
  \def\pgf@test{#1}%
  \ifx\pgf@test\pgf@maintext%
  \else%
    \global\setbox\csname pgf@layerbox@#1\endcsname=\box\csname
    ↪  pgf@layerboxsaved@#1\endcsname%
  \fi%
  \def\pgf@test{#2}%
  \ifx\pgf@test\pgfutil@empty%
  \else%
    \pgf@dorestorelayer#2,\relax%
  \fi%
}
```

命令 \pgf@restorelayers 的作用是把盒子 pgf@layerboxsaved@⟨*name i*⟩ ($i = 1, 2, \cdots$, 且 ⟨*name i*⟩≠main) 中的内容转移到盒子 pgf@layerbox@⟨*name i*⟩ 中，然后清空前者。

在执行命令 \pgfpicture 时，盒子 \pgf@layerbox@main 会被定义。在执行命令 \endpgfpicture 的过程中，命令 \pgf@insertlayers 会被执行，以插入各个图层盒子。

第十九章　颜色渐变

19.1　Overview

颜色渐变就是在某个区域中，一种或数种颜色逐渐变化、平滑过渡。有多种渐变方式：横向渐变，纵向渐变，辐射渐变，函数渐变。PGF 会把渐变放入一个盒子中，这个盒子可以直接放在文档中，不必放在绘图环境中。先用声明命令——如声明横向渐变的命令 \pgfdeclarehorizontalshading——声明一个渐变样式，然后用命令 \pgfuseshading 使用这个渐变样式，即在一个盒子中实现这个样式并将这个盒子插入文档中。例如：

```
\pgfdeclarehorizontalshading{myshadingA}{1cm}
  {rgb(0cm)=(1,0,0); color(2cm)=(green); color(4cm)=(blue)}
\pgfuseshading{myshadingA}
```

上面例子中声明一个横向渐变。设想有一个“渐变坐标系”，参数 rgb(0cm)=(1,0,0) 指定渐变坐标系横轴的 0cm 点处的颜色是 rgb 颜色模式下的红色，其颜色值是 (1,0,0)；参数 color(2cm)=(green) 指定横轴的 2cm 点处是绿色。参数 color(4cm)=(blue) 指定横轴的 4cm 点处是蓝色。这样渐变盒子的宽度就是 4cm，高度被指定为 1cm. 这个渐变盒子表现为一个“颜色条”，呈现“红、绿、蓝”三色渐变。注意在指定各个位置的颜色时所用的标点符号，位置和颜色要用圆括号括起来，各位置颜色之间用分号分隔，各位置的次序必须是单调的。

如果想在环境 {pgfpicture} 中画出渐变盒子，就需要把渐变盒子用作命令 \pgftext 的参数。在绘图环境中，你可以旋转一个渐变盒子，也可以用某个路径剪切它。

命令 \pgfshadepath 必须用在绘图环境中，它为路径作出颜色渐变效果，即用某个渐变样式填充路径。

19.1.1　颜色渐变中使用的颜色模式

颜色渐变中可用的颜色模式是宏包 xcolor 的 rgb，cmyk，gray. 宏包 xcolor 的自然色 (natural) 默认为 RGB 模式。作为颜色空间，cmyk 是 rgb 的真子集，如果不注意颜色所属的模式就可能出现色差，例如：cmyk 模式下的绿色 (green) 并不等于 rgb 模式下的绿色 (green).

在 xcolor 宏包手册中有关于颜色的知识，手册的末尾也有参考文献。

文件《pgfcoreshade.code.tex》中定义了下面的命令：

\pgf@setup@shading@model

声明颜色渐变的命令，如 \pgfdeclarehorizontalshading, 一般都会先调用这个命令。本命令决定颜色渐变的颜色模式，并做相关的设置。

本命令的处理是：

1. 本命令利用 xcolor 宏包的内部命令：

```
\XC@sdef\pgf@mod@test{\XC@tgt@mod{natural}}%
```

获取 xcolor 的 natural 模式的内涵 (保存在 \pgf@mod@test)，然后检查这个内涵究竟是哪个颜色模式：

- 如果是 cmyk, 就做相应的定义：

```
\def\pgf@shading@device{/DeviceCMYK}%
\def\pgf@shading@ps@device{setcmykcolor}%
\def\pgf@shading@functional@range{0 1 0 1 0 1 0 1}% 颜色数据的范围
\def\pgf@shading@model{cmyk}%
\pgfshadingmodelrgbfalse
\pgfshadingmodelcmyktrue
```

- 如果是 gray, 就做相应的定义：

```
\def\pgf@shading@device{/DeviceGray}%
\def\pgf@shading@ps@device{setgray}%
\def\pgf@shading@functional@range{0 1}% 颜色数据的范围
\def\pgf@shading@model{gray}%
\pgfshadingmodelrgbfalse
\pgfshadingmodelgraytrue
```

- 如果不是以上两个模式，就认为是 rgb 模式，定义：

```
\pgfshadingmodelrgbtrue
\pgfshadingmodelcmykfalse
\pgfshadingmodelgrayfalse
\def\pgf@shading@device{/DeviceRGB}%
\def\pgf@shading@ps@device{setrgbcolor}%
\def\pgf@shading@functional@range{0 1 0 1 0 1}% 颜色数据的范围
\def\pgf@shading@model{rgb}%
```

2. 检查当前的驱动文件，并规定颜色模式：

```
\edef\pgf@sys@driver@dvisvgm{pgfsys-dvisvgm.def}%
\ifx\pgfsysdriver\pgf@sys@driver@dvisvgm
    \def\pgf@shading@model{rgb}%
\fi
\edef\pgf@sys@driver@texforht{pgfsys-tex4ht.def}%
\ifx\pgfsysdriver\pgf@sys@driver@texforht
    \def\pgf@shading@model{rgb}%
\fi
```

\pgf@shading@model

这个宏由 \pgf@setup@shading@model 定义，它保存当前的颜色渐变所采用的颜色模式。

19.2 声明渐变样式

那些声明一个颜色渐变的命令只是保存其参数，并不计算颜色值。以下命令中的可选项 ⟨*color list*⟩ 中的颜色名称可以是尚未被定义的颜色。只有在使用命令 \pgfuseshading 创建颜色渐变时，PGF 才会检查并利用所需要的颜色值，因此在执行 \pgfuseshading 前要确保所需要的颜色都有颜色值。

19.2.1 横向渐变与纵向渐变

\pgfdeclarehorizontalshading[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*shading height*⟩}
{⟨*color specification*⟩}

这个命令声明一个横向渐变，将这个渐变定义在一个矩形之内，即定义一个横向变化的“颜色条”。

⟨*shading name*⟩ 是渐变的名称。

⟨*shading height*⟩ 是颜色条的高度。

⟨*color specification*⟩ 是用分号分隔的列表，是沿着横轴的颜色设置，如前例所示，其中起点与终点的距离决定颜色条的宽度。在指定 ⟨*color specification*⟩ 时应当设想有一个“渐变坐标系”，参照这个坐标系指定渐变样式。⟨*color specification*⟩ 中的列表项可以是以下形式：

- `rgb(`⟨*horizontal coordinate*⟩`)=(`⟨*r*⟩`,`⟨*g*⟩`,`⟨*b*⟩`)`，其中 ⟨*r*⟩, ⟨*g*⟩, ⟨*b*⟩ 是 $[0,1]$ 中的数值
- `gray(`⟨*horizontal coordinate*⟩`)=(`⟨*gray*⟩`)`，其中 ⟨*gray*⟩ 是 $[0,1]$ 中的数值
- `cmyk(`⟨*horizontal coordinate*⟩`)=(`⟨*c*⟩`,`⟨*m*⟩`,`⟨*y*⟩`,`⟨*k*⟩`)`，其中 ⟨*c*⟩, ⟨*m*⟩, ⟨*y*⟩, ⟨*k*⟩ 是 $[0,1]$ 中的数值
- `color(`⟨*horizontal coordinate*⟩`)=(`⟨*color*⟩`)`，其中 ⟨*color*⟩ 是 xcolor 宏包能解析的颜色名称或颜色表达式

以上 4 个形式都会被转为

⟨`\pgf@shading@model` 的值，即目标模式⟩(⟨*horizontal coordinate*⟩)=(⟨ 颜色数据 (数值)⟩)

并依次保存到宏 `\pgf@conv` 中。

可选项 ⟨*color list*⟩ 是个颜色列表，其中的颜色可以是尚未定义的颜色，其作用见下面的例子：

```
\pgfdeclarehorizontalshading[mycolorA,mycolorB]{myshadingA}
  {1cm}{rgb(0cm)=(1,0,0); color(2cm)=(mycolorA); color(4cm)=(mycolorB)}
\definecolor{mycolorA}{rgb}{0.5,0.1,0.3}
\colorlet{mycolorB}{cyan!10!yellow}
\pgfuseshading{myshadingA}\par\vspace{3mm}
\colorlet{mycolorB}{cyan!80!orange}
\pgfuseshading{myshadingA}
```

本命令处理是：

1. 决定颜色模式，执行 `\pgf@setup@shading@model`
2. 保存可选的颜色列表 ⟨*color list*⟩:

   ```
   \expandafter\def\csname pgf@deps@pgfshading⟨shading name⟩!\endcsname{⟨color list⟩}%
   ```

3. 检查 `\csname pgf@deps@pgfshading`⟨*shading name*⟩`!\endcsname` 是否空的，
   - 如果是空的，即 ⟨*color list*⟩ 是空的，就
     (a) 清空 `\csname pgf@num@pgfshading`⟨*shading name*⟩`!\endcsname`

     ```
     \global\expandafter\let\csname pgf@num@pgfshading#2!\endcsname
     ↪  \pgfutil@empty
     ```

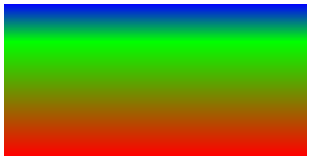
     (b) 执行系统命令

     `\pgfsys@horishading{`⟨*shading name*⟩`}{`⟨*shading height*⟩`}{`⟨*color specification*⟩`}`
   - 如果不是空的，即 ⟨*color list*⟩ 不是空的，就定义：

     ```
     \global\advance\pgf@shadingcount 1\relax
     \global\expandafter\edef\csname pgf@num@pgfshading⟨shading name⟩!\endcsname{
     ↪  \pgf@shadingnum}%
     \expandafter\def\csname pgf@func@pgfshading⟨shading name⟩!\endcsname{
     ↪  \pgfsys@horishading}%
     \expandafter\def\csname pgf@args@pgfshading⟨shading name⟩!\endcsname{
     ↪  {⟨shading height⟩}{⟨color specification⟩}}%
     \expandafter\let\csname @pgfshading⟨shading name⟩!\endcsname=\pgfutil@empty%
     ```

`\pgfdeclareverticalshading[`⟨*color list*⟩`]{`⟨*shading name*⟩`}{`⟨*shading width*⟩`}{`⟨*color specification*⟩`}`

这个命令声明一个纵向渐变，确定一个纵向变化的颜色条，⟨*shading name*⟩ 是渐变的名称，⟨*shading width*⟩ 是颜色条的宽度，⟨*color specification*⟩ 沿着纵轴指定颜色。可选选项 ⟨*color list*⟩ 的用处与前一个命令相同。



```
\pgfdeclareverticalshading{myshadingC}{4cm}
  {rgb(0cm)=(1,0,0); rgb(1.5cm)=(0,1,0); rgb(2cm)=(0,0,1)}
\pgfuseshading{myshadingC}
```

19.2.2 辐射渐变

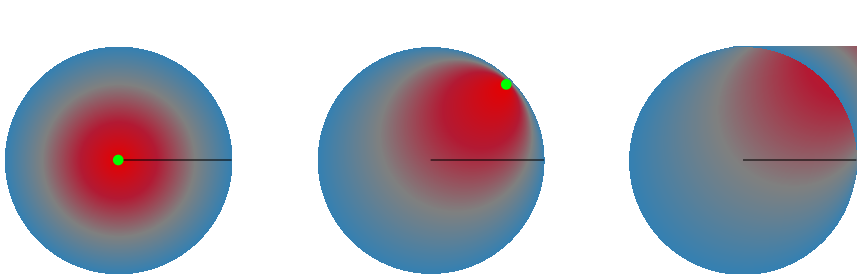
辐射渐变的意思是，颜色以某个点为中心，向外围渐变。

\pgfdeclareradialshading[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*center point*⟩}{⟨*color specification*⟩}

这个命令声明一个辐射渐变。⟨*shading name*⟩ 是所声明的辐射渐变的名称。⟨*color list*⟩ 的作用与前面的命令类似。

⟨*color specification*⟩ 是沿着横轴给出的“点——颜色”列表，它指定一个线段以及该线段上某些点处的颜色，这个线段就作为半径构造一个“辐射圆”（圆心是线段起点），用于呈现辐射渐变。

点 ⟨*center point*⟩ 是辐射渐变的中心，如果这个点不是“辐射圆”的圆心，那么可能出现意外效果。



```
\pgfkeys{
  fushejianbian/.code={
    \pgfdeclareradialshading{sphere}{\pgfpoint{#1cm}{#1cm}}%
      {rgb(0cm)=(0.9,0,0);
       rgb(0.5cm)=(0.7,0.1,0.2);
       rgb(1cm)=(0.5,0.5,0.5);
       rgb(1.5cm)=(0.2,0.5,0.7)}
    \tikz{\pgftext{\pgfuseshading{sphere}}
      \draw (0,0)--(1.5,0);
      \fill [green](#1,#1)circle(2pt);}
  }
}
\pgfkeys{fushejianbian={0}} \hspace{.5cm}
\pgfkeys{fushejianbian={1}} \hspace{.5cm}
\pgfkeys{fushejianbian={2}}
```

19.2.3 函数渐变

\pgfdeclarefunctionalshading[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*lower left corner*⟩}
{⟨*upper right corner*⟩}{⟨*init code*⟩}{⟨*type 4 function*⟩}

注意，这种渐变可能给渲染器带来很大负担，也可能无法正确显示。

这个命令创建一个函数渐变。⟨*shading name*⟩ 是渐变的名称。点 ⟨*lower left corner*⟩ 和点 ⟨*upper right corner*⟩ 确定一个矩形，本命令计算这个矩形内各点处的颜色（针对像素点）。

可选选项 ⟨*color list*⟩ 的用处与前面的命令类似。

⟨*init code*⟩ 通常与 ⟨*color list*⟩ 配合使用，见后文的例子。

这里的函数 ⟨*type 4 function*⟩ 指的是《PDF Reference》(version 1.7) 的 §3 中所说的 type 4 类型的函数，这类函数是 PostScript 语言中的函数 (见后文)。实际运行 ⟨*type 4 function*⟩ 函数计算的是 PDF 渲染器，而不是 PGF 或 TEX, 函数中的错误也只有渲染器能注意到。

使用 type 4 类型函数的句法属于 PostScript 语言，PDF 渲染器能处理这一部分句法。type 4 类型函数包括：`abs`, `add`, `atan`, `ceiling`, `cos`, `cvi`, `cvr`, `div`, `exp`, `floor`, `idiv`, `ln`, `log`, `mod`, `mul`, `neg`, `round`, `sin`, `sqrt`, `sub`, `truncate`, `and`, `bitshift`, `eq`, `false`, `ge`, `gt`, `le`, `lt`, `ne`, `not`, `or`, `true`, `xor`, `if`, `ifelse`, `copy`, `dup`, `exch`, `index`, `pop`.

这里输入给 type 4 类型函数的值都必须是不带单位的实数值，函数的输出也是不带单位的数值。函数利用点的坐标值进行计算，得到该点处的颜色数据值。函数计算一个坐标点的颜色时，先把该点的两个坐标分量的长度单位转换成 `bp`, 然后去掉长度单位作成两个实数，然后再输入给函数做计算；经过函数计算后，输出的是：

- rgb 模式下的 3 个颜色参数，或者
- cmyk 模式下的 4 个颜色参数，或者
- gray 模式下的 1 个颜色参数

type 4 类型函数的输入和输出都是小数，不是整数，这可能使得 Apple 的 PDF 渲染器不能正常显示（或许可以用函数 `cvr` 补救）。

本命令处理是：

1. 确定颜色模式，执行 `\pgf@setup@shading@model`
2. 保存可选的颜色列表 ⟨*color list*⟩:

```
\expandafter\def\csname pgf@deps@pgfshading⟨shading name⟩!\endcsname{⟨color list⟩}%
```

3. 检查 `\csname pgf@deps@pgfshading`⟨*shading name*⟩`!\endcsname` 是否空的，
   - 如果是空的，即 ⟨*color list*⟩ 是空的，就
     (a) 清空 `\csname pgf@num@pgfshading`⟨*shading name*⟩`!\endcsname`

```
\global\expandafter\let\csname pgf@num@pgfshading⟨shading name⟩!
↪  \endcsname=\pgfutil@empty
```

     (b) 执行命令

```
\pgfshade@functionaldo
↪  {⟨shading name⟩}{⟨lower left corner⟩}{⟨upper right corner⟩}{⟨init code⟩}{⟨type 4 function⟩}
```

     按 `\pgfshade@functionaldo` 的定义，它导致

```
\begingroup
  ⟨init code⟩%
  \pgfsys@functionalshading
  ↪  {⟨shading name⟩}{⟨lower left corner⟩}{⟨upper right corner⟩}{⟨type 4 function⟩}
  ↪  %
  \expandafter\pgfmath@smuggleone\csname @pgfshading⟨shading name⟩!
  ↪  \endcsname
\endgroup
```

   - 如果不是空的，即 ⟨*color list*⟩ 不是空的，就定义：

```
\global\advance\pgf@shadingcount 1\relax
\global\expandafter\edef\csname pgf@num@pgfshading⟨shading name⟩!\endcsname{
↪  \pgf@shadingnum}%
\expandafter\def\csname pgf@func@pgfshading⟨shading name⟩!\endcsname{
↪  \pgfshade@functionaldo}%
\expandafter\def\csname pgf@args@pgfshading⟨shading name⟩!\endcsname{
↪  {⟨lower left corner⟩}{⟨upper right corner⟩}{⟨init code⟩}{⟨type 4 function⟩}}%
\expandafter\let\csname @pgfshading⟨shading name⟩!\endcsname=\pgfutil@empty%
```

下面是个例子：

```
\pgfdeclarefunctionalshading{twospots}
  {\pgfpointorigin}{\pgfpoint{4cm}{4cm}}{}
  {
    2 copy
    45 sub dup mul exch
    40 sub dup mul 0.5 mul add sqrt
    dup mul neg 1.0005 exch exp 1.0 exch sub
    3 1 roll
    70 sub dup mul .5 mul exch
    70 sub dup mul add sqrt
    dup mul neg 1.002 exch exp 1.0 exch sub
    1.0 3 1 roll
  }
\pgfuseshading{twospots}
```

看一下上面例子中的 type 4 函数计算的是什么。

上面例子中，点 \pgfpointorigin 和 \pgfpoint{4cm}{4cm} 确定一个矩形，type 4 函数逐个计算这个矩形内（像素）点的颜色。假设矩形内任意一个（像素）点的坐标是 (x, y)，将坐标分量作成数值栈 x y 提供给函数，然后由函数中的算子按次序进行处理。函数对 x y 的处理是：

```
x y 2 copy
    45 sub dup mul exch
    40 sub dup mul 0.5 mul add sqrt
    dup mul neg 1.0005 exch exp 1.0 exch sub
    3 1 roll
    70 sub dup mul .5 mul exch
    70 sub dup mul add sqrt
    dup mul neg 1.002 exch exp 1.0 exch sub
    1.0 3 1 roll
```

下面分析前几个算子的运算过程，为了方便其中使用了数学公式。

x␣y␣2␣copy $\to$ x␣y␣x␣y

45␣sub $\to$ x␣y␣x␣y-45

dup $\to$ x␣y␣x␣y-45␣y-45

mul $\to$ x␣y␣x␣$(y-45)^2$

exch $\to$ x␣y␣$(y-45)^2$␣x

40␣sub $\to$ x␣y␣$(y-45)^2$␣x-40

dup $\to$ x␣y␣$(y-45)^2$␣x-40␣x-40

mul $\to$ x␣y␣$(y-45)^2$␣$(x-40)^2$

0.5␣mul $\to$ x␣y␣$(y-45)^2$␣$0.5(x-40)^2$

add $\to$ x␣y␣$(y-45)^2+0.5(x-40)^2$

sqrt $\to$ x␣y␣$\sqrt{(y-45)^2+0.5(x-40)^2}$

dup mul $\to$ x␣y␣$(y-45)^2+0.5(x-40)^2$

neg $\to$ x␣y␣$-(y-45)^2-0.5(x-40)^2$

1.0005␣exch $\to$ x␣y␣1.0005␣$-(y-45)^2-0.5(x-40)^2$

exp $\to$ x␣y␣$1.0005^{-(y-45)^2-0.5(x-40)^2}$

1.0␣exch $\to$ x␣y␣1.0␣$1.0005^{-(y-45)^2-0.5(x-40)^2}$

sub $\to$ x␣y␣$1.0-1.0005^{-(y-45)^2-0.5(x-40)^2}$

3␣1␣roll $\to$ $1.0-1.0005^{-(y-45)^2-0.5(x-40)^2}$␣x␣y

最后的结果就是：

$$1.0\text{␣}1.0-1.0005^{-(y-45)^2-0.5(x-40)^2}\text{␣}1.0-1.002^{-0.5(y-70)^2-(x-70)^2}$$

这三个数值在 rgb 颜色模式下决定一个颜色，可见这种 ⟨*type 4 function*⟩ 函数的运算并不直观。

在 ⟨*type 4 function*⟩ 中不能直接使用颜色名称，可以把颜色的数值参数保存在某个宏中，然后再把这个宏提供给 ⟨*type 4 function*⟩. 使用下面的 3 个命令：

\pgfshadecolortorgb, \pgfshadecolortocmyk, \pgfshadecolortogray

把颜色的数值参数保存在某个宏中，这个宏可以作为 ⟨*type 4 function*⟩ 的参数。这 3 个命令可以用在 ⟨*init code*⟩ 中。

\pgfshadecolortorgb{⟨*color name*⟩}\⟨*macro*⟩

本命令的作用是：

- 将名称为 ⟨*color name*⟩ 的颜色对应的、rgb 模式下的颜色参数保存在宏 \⟨*macro*⟩ 中，这里 ⟨*color name*⟩ 必须是某个“已经被定义”的颜色名称。宏 \⟨*macro*⟩ 中保存的是三个 (由空格分隔的) 0 到 1 之间的数值。\⟨*macro*⟩ 可以用作 ⟨*type 4 function*⟩ 的参数。
- 将宏 \⟨*macro*⟩ 中的第 1 个数值，即 r 分量保存到宏 \⟨*macro*⟩red
- 将宏 \⟨*macro*⟩ 中的第 2 个数值，即 g 分量保存到宏 \⟨*macro*⟩green
- 将宏 \⟨*macro*⟩ 中的第 3 个数值，即 b 分量保存到宏 \⟨*macro*⟩blue

```
\pgfshadecolortorgb{orange}{\mycol}
\mycol\\
\mycolred\\
\mycolgreen\\
\mycolblue
```

1.0 0.5 0.0
1.0
0.5
0.0

下面的例子展示了可选选项 ⟨*color list*⟩ 与 ⟨*init code*⟩ 的配合，二者通过本命令联系起来。

试试 

```
\pgfdeclarefunctionalshading[mycol]{sweep}{\pgfpoint{-1cm}{-1cm}}
{\pgfpoint{1cm}{1cm}}{\pgfshadecolortorgb{mycol}{\myrgb}}  % 将 mycol 的颜色值保存在 \myrgb 中
{
  2 copy
  2 copy abs exch abs add 0.0001 ge { atan } { pop } ifelse
  3 1 roll
  dup mul exch
  dup mul add sqrt
  30 mul
  add
  sin
  1 add 2 div
  dup
  \myrgb  % 引入名称 mycol 对应的颜色值
  5 4 roll
  mul
  3 1 roll
  3 index
  mul
  3 1 roll
  4 3 roll
  mul
  3 1 roll
}
试试\colorlet{mycol}{white}% 定义颜色 mycol
\pgfuseshading{sweep}%
\colorlet{mycol}{red}% 重定义颜色 mycol
\pgfuseshading{sweep}
```

如果把上面例子中的“试试”两个字去掉……

\pgfshadecolortocmyk{⟨*color name*⟩}\⟨*macro*⟩

把 ⟨*color name*⟩ 对应的 cmyk 模式下的 4 个颜色参数值 (在 0 到 1 之间) 保存到宏 \⟨*macro*⟩ 中，还定义 4 个后缀为 `cyan`, `magenta`, `yellow`, `black` 的宏分别保存这 4 个颜色参数值。

```
\pgfshadecolortocmyk{orange}{\mycol}
\mycol\\
\mycolcyan\\
\mycolmagenta\\
\mycolyellow\\
\mycolblack
```

0.0 0.5 1.0 0.0
0.0
0.5
1.0
0.0

\pgfshadecolortogray{⟨*color name*⟩}\⟨*macro*⟩

把 ⟨*color name*⟩ 对应的 gray 模式下的 1 个颜色参数值 (在 0 到 1 之间) 保存到宏 \⟨*macro*⟩ 中。

一般情况下，PostScript 代码 (上面的 ⟨*type 4 function*⟩) 输出的颜色数据必定属于 3 种模式 rgb, cmyk, gray 之一：

- ⟨*type 4 function*⟩ 输出一个 3 元数列 ⟨*x y z*⟩ 代表 rgb 模式下的一个颜色；
- 输出一个 4 元数列 ⟨*p q r s*⟩ 代表 cmyk 模式下的一个颜色；
- 出一个数值 ⟨*g*⟩ 代表 gray 模式下的一个颜色。

输使用下面的命令，可以在 ⟨*x y z*⟩, ⟨*p q r s*⟩, ⟨*g*⟩ 之间相互转换。参考 xcolor 宏包。

例如：

```
\pgfdeclarefunctionalshading[black]{portabletwospots}{\pgfpointorigin}{\pgfpoint{3.5cm
↪  }{3.5cm}}{}{
  2 copy
  45 sub dup mul exch
  40 sub dup mul 0.5 mul add sqrt
  dup mul neg 1.0005 exch exp 1.0 exch sub
  3 1 roll
  70 sub dup mul .5 mul exch
  70 sub dup mul add sqrt
  dup mul neg 1.002 exch exp 1.0 exch sub
  1.0 3 1 roll
  \ifpgfshadingmodelcmyk
    \pgffuncshadingrgbtocmyk
  \fi
  \ifpgfshadingmodelgray
    \pgffuncshadingrgbtogray
  \fi
}
```

\pgffuncshadingrgbtocmyk

用在命令 `\pgfdeclarefunctionalshading` 中，将 ⟨*type 4 function*⟩ 输出的 3 元数列 ⟨*x y z*⟩ 映射为 4 元数列 ⟨*p q r s*⟩.

\pgffuncshadingrgbtogray

用在命令 `\pgfdeclarefunctionalshading` 中，将 ⟨*type 4 function*⟩ 输出的 3 元数列 ⟨*x y z*⟩ 映射为 1 个数值 ⟨*g*⟩.

在 xcolor 宏包手册中有下面的公式：

$$gray := 0.3 \cdot red + 0.59 \cdot green + 0.11 \cdot blue$$

在文件《pgfcoreshade.code.tex》中有：

```
\def\pgffuncshadingrgbtogray{%
  0.11 mul exch 0.59 mul add exch 0.3 mul add
}
```

\pgffuncshadingcmyktorgb

用在命令 \pgfdeclarefunctionalshading 中，将 ⟨*type 4 function*⟩ 输出的 4 元数列 ⟨*p q r s*⟩ 映射为 3 元数列 ⟨*x y z*⟩.

\pgffuncshadingcmyktogray

用在命令 \pgfdeclarefunctionalshading 中，将 ⟨*type 4 function*⟩ 输出的 4 元数列 ⟨*p q r s*⟩ 映射为 1 个数值 ⟨*g*⟩.

\pgffuncshadinggraytorgb

用在命令 \pgfdeclarefunctionalshading 中，将 ⟨*type 4 function*⟩ 输出的 1 个数值 ⟨*g*⟩ 映射为 3 元数列 ⟨*x y z*⟩.

\pgffuncshadinggraytocmyk

用在命令 \pgfdeclarefunctionalshading 中，将 ⟨*type 4 function*⟩ 输出的 1 个数值 ⟨*g*⟩ 映射为 4 元数列 ⟨*p q r s*⟩.

\ifpgfshadingmodelrgb

用在命令 \pgfdeclarefunctionalshading 中，检查“创建颜色渐变时”的颜色模式是否是 rgb.

\ifpgfshadingmodelcmyk

用在命令 \pgfdeclarefunctionalshading 中，检查“创建颜色渐变时”的颜色模式是否是 cmyk.

\ifpgfshadingmodelgray

用在命令 \pgfdeclarefunctionalshading 中，检查“创建颜色渐变时”的颜色模式是否是 gray.

\pgfaliasshading{⟨*new shading name*⟩}{⟨*old shading name*⟩}

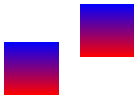
本命令将 ⟨*new shading name*⟩ 声明为 ⟨*old shading name*⟩ 的别名，两个名称都是可用的。

19.3 使用颜色渐变

在使用颜色渐变 ⟨*shading name*⟩ 前，需要确保 (声明这个渐变时的可选参数)⟨*color list*⟩ 中的各个颜色有定义 (有效)。使用渐变 ⟨*shading name*⟩ 时，PGF 会把“⟨*shading name*⟩”以及 ⟨*color list*⟩ 中的所有颜色看作一个组合。当第一次使用这个组合时，PGF 会对 ⟨*color list*⟩ 中的各个颜色作某些处理，然后再用处理结果创建渐变盒子。当再次使用这个组合时，PGF 就不会再对 ⟨*color list*⟩ 中的各个颜色作处理，而是直接使用第一次的处理结果，创建渐变盒子。

\pgfuseshading{⟨*shading name*⟩}

这个命令在一个盒子中创建渐变 ⟨*shading name*⟩. 本命令可以用在 {pgfpicture} 环境中，也可以用在这个环境之外。将本命令用作 \pgftext 的参数，可以放到 {pgfpicture} 环境中。



```
\begin{tikzpicture}
  \pgfdeclareverticalshading{myshadingD}
    {20pt}{color(0pt)=(red); color(20pt)=(blue)}
  \pgftext[at=\pgfpoint{1cm}{0cm}] {\pgfuseshading{myshadingD}}
  \pgftext[at=\pgfpoint{2cm}{0.5cm}]{\pgfuseshading{myshadingD}}
\end{tikzpicture}
```

本命令的处理是：首先检查名称为 ⟨*shading name*⟩ 的渐变是否已被声明，如果未声明就报错；如果已声明，就继续以下步骤：

1. 执行 \pgf@setup@shading@model →P. 365
2. 用一个循环处理

   \csname pgf@deps@pgfshading⟨*shading name*⟩!\endcsname

   中保存的列表 (即声明渐变的命令的可选参数 ⟨*color list*⟩)，对于其中的任一颜色，检查是否需要用 xxcolor 宏包的命令来处理。
3. 定义 \pgf@shadingxname, 它保存的是“⟨*shading name*⟩⟨*color list*⟩”这个组合 (作为一串记号)。
4. 检查这个组合是否已经被做成控制序列名称，如果没有，就定义这个控制序列。
5. 然后执行 \pgf@invokeshading 创建渐变。

\pgfshadepath{⟨*shading name*⟩}{⟨*angle*⟩}

这个命令必须用在 {pgfpicture} 环境中。当创建一个路径后，可以使用本命令来填充当前路径，填充内容就是名称为 ⟨*shading name*⟩ 的颜色渐变。

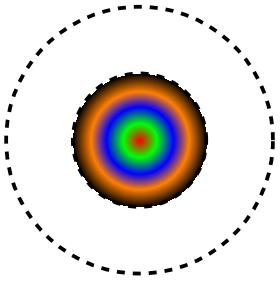
本命令会调用 \pgfuseshading.

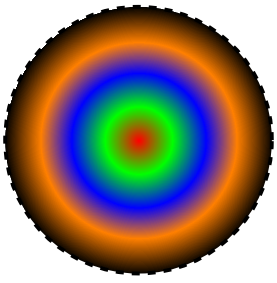
下面介绍一下这个命令的工作过程。首先 PGF 会设置一个 local scope. 在这个 local scope 中将路径用作剪切路径来剪切 ⟨*shading name*⟩ 的渐变盒子从而得到填充效果。在这个 local scope 之后，这个剪切路径仍然是“可用的”，即可画出该路径。在这个局部域内，考虑以 (0bp,0bp) 和 (100bp,100bp) 为对角点的矩形 R, PGF 调整底层的变换矩阵：

1. 做平移，使得渐变盒子的中心处于 R 的中心 (50bp,50bp) 处，此时把渐变盒子与矩形 R 固定在一起 (固定的意思是，矩形 R 有什么变化，渐变盒子就有什么变化)；
2. 做平移、伸缩变换，使得
   - (50bp,50bp) 对应路径边界盒子的中心；
   - (25bp,25bp) 对应路径边界盒子的左下角点；
   - (75bp,75bp) 对应路径边界盒子的右上角点；

   注意此时路径边界盒子只能覆盖矩形 (0bp,0bp) rectangle (100bp,100bp) 面积的 $\frac{1}{4}$.
3. 如果本命令的参数 ⟨*angle*⟩ 非空，则将渐变盒子绕中心点旋转 ⟨*angle*⟩ 角度。
4. 之后，如果本命令前面还使用了宏 \pgfsetadditionalshadetransform, 则针对渐变盒子执行这个宏所保存的变换。
5. 再之后，用路径剪切渐变盒子，得到填充效果。

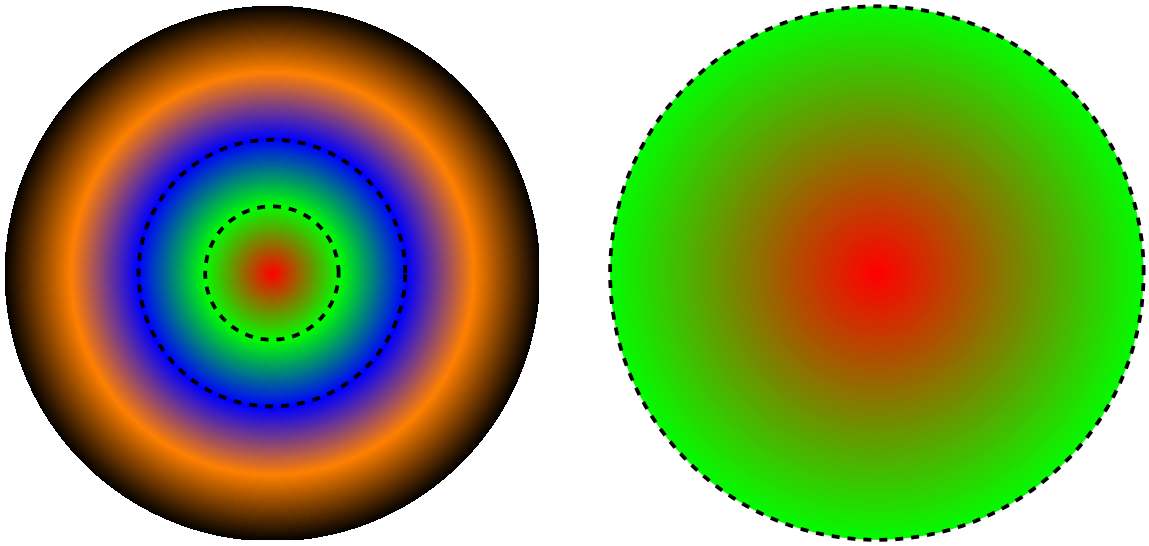
按以上机制,如果 ⟨*shading name*⟩ 的渐变盒子的原始尺寸不能覆盖矩形 (0bp,0bp) rectangle (100bp,100bp), 那么可能导致渐变不能填满路径的边界盒子；如果 ⟨*shading name*⟩ 的渐变盒子的原始尺寸远大于矩形 (0bp,0bp) rectangle (100bp,100bp), 那么渐变盒子中的很多颜色将被裁掉。





```
\pgfdeclareradialshading{test size A}{\pgfpoint{0cm}{0cm}}%
{color(0bp)=(red);
color(6.25bp)=(green);
color(12.5bp)=(blue);
color(18.25bp)=(orange);
color(25bp)=(black)
}
\begin{tikzpicture}
\pgftext{\pgfuseshading{test size A}}
```

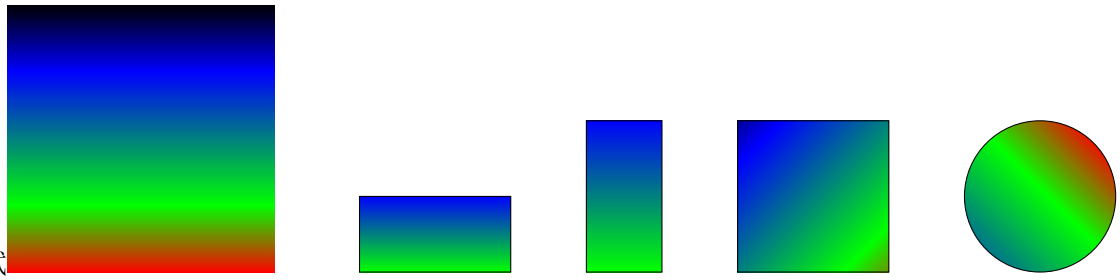
```
\draw [dashed,line width=0.5mm](0,0) circle [radius=25bp];
\draw [dashed,line width=0.5mm](0,0) circle [radius=50bp];
\draw [dashed,line width=0.5mm,shading=test size A](8,0) circle [radius=50bp];
\end{tikzpicture}
```



```
\pgfdeclareradialshading{test size B}{\pgfpoint{0cm}{0cm}}%
{color(0bp)=(red);
color(25bp)=(green);
color(50bp)=(blue);
color(75bp)=(orange);
color(100bp)=(black)
}
\begin{tikzpicture}
\pgftext{\pgfuseshading{test size B}}
\draw [dashed,line width=0.5mm](0,0) circle [radius=25bp];
\draw [dashed,line width=0.5mm](0,0) circle [radius=50bp];
\draw [dashed,line width=0.5mm,shading=test size B](8,0) circle [radius=100bp];
\end{tikzpicture}
```

\pgfsetadditionalshadetransform{⟨*transformation*⟩}

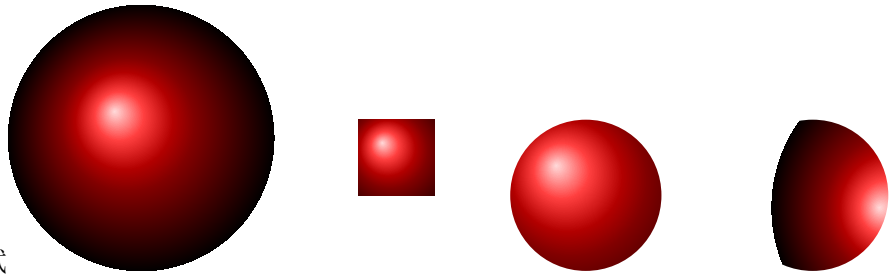
这个命令用在 \pgfshadepath 之前，这个命令的参数 ⟨*transformation*⟩ 是某些变换命令，是针对渐变盒子的。在用渐变填充路径之前，本命令对渐变盒子做变换。

试试 

```
\pgfdeclareverticalshading{myshadingE}{100bp}
  {color(0bp)=(red); color(25bp)=(green); color(75bp)=(blue); color(100bp)=(black)}
试试\pgfuseshading{myshadingE}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingE}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{3cm}{0cm}}{\pgfpoint{1cm}{2cm}}
  \pgfshadepath{myshadingE}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{5cm}{0cm}}{\pgfpoint{2cm}{2cm}}
  \pgfshadepath{myshadingE}{45}
```

```
  \pgfusepath{stroke}
  \pgfpathcircle{\pgfpoint{9cm}{1cm}}{1cm}
  \pgfsetadditionalshadetransform{\pgftransformrotate{90}\pgftransformyshift{0.8cm}}
  \pgfshadepath{myshadingE}{45}
  \pgfusepath{stroke}
\end{pgfpicture}
```

下面是辐射渐变的例子。



试试

```
\pgfdeclareradialshading{ballshading}{\pgfpoint{-10bp}{10bp}}{
  color(0bp)=(red!15!white); color(9bp)=(red!75!white);
  color(18bp)=(red!70!black); color(25bp)=(red!50!black); color(50bp)=(black)}
试试\pgfuseshading{ballshading}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
  \pgfshadepath{ballshading}{0}
  \pgfusepath{}
  \pgfpathcircle{\pgfpoint{3cm}{0cm}}{1cm}
  \pgfshadepath{ballshading}{0}
  \pgfusepath{}
  \pgfpathcircle{\pgfpoint{6cm}{0cm}}{1cm}
  \pgfsetadditionalshadetransform{\pgftransformshift{\pgfpoint{0.8cm}{-1cm}}}
  \pgfshadepath{ballshading}{45}
  \pgfusepath{}
\end{pgfpicture}
```

```
\pgfdeclareverticalshading{rainbow}{100bp}{
  color(0bp)=(red); color(25bp)=(red); color(35bp)=(yellow);
  color(45bp)=(green); color(55bp)=(cyan); color(65bp)=(blue);
  color(75bp)=(violet); color(100bp)=(violet)}
\begin{tikzpicture}[shading=rainbow]
  \shade (0,0) rectangle node[white] {\textsc{pride}} (2,1);
  \shade[shading angle=90] (3,0) rectangle +(1,2);
\end{tikzpicture}
```

19.4 关于 type 4 函数的补充

以下内容来自《PDF Reference》(version 1.7) 的 §3.

PDF 提供多种类型的“函数对象”(function objects)，它们是参数化的函数，函数的构造涉及数学表达式，样本点等。在 PDF 中函数有多种用途，例如提供光栅信息用于高质量输出（halfton spot functions 和 transfer functions)，在某个颜色空间中进行颜色变换，用于创建在颜色渐变。

PDF 中的函数是数值变换。例如加法函数以两个数值为输入，一个数值为输出：

$$f(x_0, x_1) = x_0 + x_1$$

两数的算术平均和几何平均函数是：

$$f(x_0, x_1) = \frac{x_0 + x_1}{2},\ \sqrt{x_0 x_1}$$

一般，一个函数可以有 m 个输入数值，产生 n 个输出值：

$$f(x_0, \cdots, x_{m-1}) = y_0, \cdots, y_{n-1}$$

PDF 函数的输入和输出都是数值，函数本身只是实现数值变换，没有其它作用。

每个函数都有自己的定义域，有的函数还有值域。如果输入给函数的数值不在其定义域内，函数就会从定义域中选择一个最接近输入值的数来代替它，然后执行计算。如果函数计算出来的输出值不在其值域内，函数就从其值域内选择一个最接近输出值的数来代替它，然后输出替换值。例如，假设函数

$$f(x) = x + 2,$$

它的定义域是 $[-1, 1]$，如果调用此函数，且输入值是 6，那么这个输入值就被替换为 1，然后计算 $f(1)$，得到输出值 3.

假设函数

$$f(x_0, x_1) = 3x_0 + x_1,$$

的值域是 $[0, 100]$，输入 -6 和 4 （假设在定义域内），计算结果是 -14，结果不在值域内，于是输出 0.

可用的函数有 4 种类型：

type 0 样本函数（sampled function），是个表格，用于插值计算。

type 2 指数插值函数（exponential interpolation function）。

type 3 缝合函数（stitching function）。

type 4 某些 PostScrip 计算函数，用的是 PostScript 语言。

type 4 类函数的表达式中只涉及整数、实数、布尔值，没有字符串、数组，也没有其它指令、变量、名称。能够用于构造这类函数的算子如下表所示：

type 4 类函数中的算子

算子类型	算子				
计算算子	abs	cvi	floor	mod	sin
	add	cvr	idiv	mul	sqrt
	atan	div	ln	neg	sub
	ceiling	exp	log	round	truncate
	cos				
真值算子	and	false	le	not	true
	bitshift	ge	lt	or	xor
	eq	gt	ne		
条件算子	if	ifelse			
栈算子	copy	exch	pop		
	dup	index	roll		

下面解释一下这些算子，参考《PostScript LANGUAGE REFERENCE》（third edition）的第 8 章。

设想有一个数值栈（stack），将某些数值按次序存放到这个栈中，位置靠前的数值处于“下部”，位置靠后的数值处于“上部”，第一个数值处于“底部”，最后一个数值处于“顶端”（topmost）。将一个算子放在数值栈之后，算子针对栈中的某些数值做运算，也就是说，算子都是“后置”算子（数学中的算子多数是“前置算子”）。以加法算子 `add` 为例，这个算子是二元算子，即它需要两个输入值（运算对象 operand），假设写出一串用空格分隔的数值，`1 2 3 4`，然后写上 `add`：

```
1 2 3 4 add
```

执行这一行代码，算子 add 将前面的 3 4 求和，得到的是数值列表：

```
1 2 7
```

下面用例子说明这些算子的作用。

abs 绝对值运算，输出值的类型与输入值相同；若输入是最小的整数，则输出是实数。

```
0 2 -1.0 abs 输出 0 2 1.0
```

add 求两数和，若两个输入值都是整数则输出整数，否则输出实数。

```
1 2 3.0 add 输出 1 5.0
```

and 逻辑与运算，输入可以是布尔值：

```
true false and 输出 false
```

输入也可以是整数，此时是“按位运算”：

```
2 99 1 and 输出 2 1，这里只对 99 和 1 做运算。
```

atan 变异的反正切函数，输入值是两个数，输出值是 0 到 360 之间的实数，代表角度。

```
13 -1 1 atan 输出 13 135.0，输出值是从向量 (1,0) 沿着逆时针方向转到向量 (-1,1) 所转
↪ 过的角度。
注意：
1 0 atan 输出 90.0
-100 0 atan 输出 270.0
```

bitshift 移位算子，以两个整数为输入值，第一个输入整数最好是正整数，本算子输出一个整数。

```
0 7 3 bitshift 输出 0 56，将 7 的二进制表示向左移动 3 位
142 -3 bitshift 输出 17，将 142 的二进制表示向右移动 3 位
```

ceiling 向上取整，例如

```
11 3.2 ceiling 输出 11 4.0
12 -4.8 ceiling 输出 12 -4.0
```

copy 复制算子，本算子从数值栈中选取靠近“顶部”的某些数值，复制这些数值并将它们添加到原来的栈中，即用复制的方法增加栈中的数值数量，增加栈长度（高度）。本算子选取数值的“标准”由栈的“顶端”数值决定，如下面的例子所示。

```
(a) (b) (c) 2 copy 输出 (a) (b) (c) (b) (c)
(a) (b) (c) 0 copy 输出 (a) (b) (c)
```

cos 角度的余弦值，本算子需要一个输入，输出 -1 到 1 之间的实数。

```
-1 60 cos 输出 -1 0.5
```

cvi 向零取整，是 convert to integer 的缩写。

cvr 将输入值转化为实数类型的数，是 convert to real 的缩写。

div 两数除法，需要两个输入值，计算前数除以后数的商并输出，输出值总是实数，其符号与第一个输入值相同。

```
3 2 div 输出 1.5
4 2 div 输出 2.0
```

dup 复制顶端值，本算子的输入只是栈顶端的数值，复制顶端值并添加到栈中。

```
1 2 dup 输出 1 2 2
```

eq 判断两数是否相等，需要两个输入值。

```
0 2 2 eq 输出 0 true
```

exch 换位，本算子需要两个输入值，交换它们在栈中的位置。

```
1 2 3 4 exch 输出 1 2 4 3
```

exp 幂运算，

```
1 2 3 exp 输出 1 8，这里计算 2^3
```

false 布尔值 false，本算子没有输入，只有输出值 false.

```
1 2 false 输出 1 2 false
```

floor 向下取整.

ge 判断是否不小于，需要两个输入值，判断前数是否不小于后数。

```
1 2 3 ge 输出 1 false
```

gt 判断是否大于，需要两个输入值，判断前数是否大于后数。

```
1 2 3 gt 输出 1 false
```

idiv 取整除法，需要两个整数作为输入，计算前数除以后数的商，并输出商的整数部分，输出值的符号与第一个输入值相同。

```
3 2 idiv 输出 1
4 2 idiv 输出 2
-5 2 idiv 输出 -2
```

if 条件算子，它只需要两个输入，本身没有输出。

⟨*bool*⟩ ⟨*pro*⟩ `if` 如果 ⟨*bool*⟩ 为 `true` 则执行 ⟨*pro*⟩，否则继续后面的处理。

ifelse 条件算子，它只需要三个输入，本身没有输出。

⟨*bool*⟩ ⟨*pro1*⟩ ⟨*pro2*⟩ `ifelse` 如果 ⟨*bool*⟩ 为 `true` 则执行 ⟨*pro1*⟩，否则执行 ⟨*pro2*⟩.

index 倒序索引，从中栈中选出某个数值，复制它并添加到栈中，使之处于“顶端”位置。本算子选择数值的标准由原栈的“顶端”数值决定。

⟨*anyn*⟩ ⋯ ⟨*any0*⟩ n index 输出 ⟨*anyn*⟩ ⋯ ⟨*any0*⟩ ⟨*anyn*⟩
(a) (b) (c) (d) 0 index 输出 (a) (b) (c) (d) (d)
(a) (b) (c) (d) 3 index 输出 (a) (b) (c) (d) (a)

le 判断是否不大于，需要两个输入值，判断前数是否不大于后数。

1 2 le 输出 true

ln 计算自然对数值，需要一个输入值。

10 ln 输出 2.30259
100 ln 输出 4.60517

log 计算常用对数值（以 10 为底），

10 log 输出 1.0
100 log 输出 2.0

lt 判断是否小于，需要两个输入值，判断前数是否小于后数。

1 2 lt 输出 true

mod 余数运算，需要两个整数作为输入值，计算前数除以后数的余数并输出之，余数的符号与第一个输入值相同。

5 3 mod 输出 2
-5 3 mod 输出 -2

mul 计算两数乘积，需要两个输入值。

ne 判断是否不等于，需要两个输入值。

neg 取相反数，需要一个输入值。

not 逻辑非，需要一个输入值。

or 逻辑或，需要两个输入值，输入值可以是布尔值，也可以是整数（此时是按位运算）。

pop 删除顶端值，这个算子只有一个输入，没有输出。它删除栈中的顶端数值。

1 2 3 pop 输出 1 2
1 2 3 pop pop 输出 1

roll 轮换，它的前面应当是两个正整数，本算子把这两个正整数与栈中的其它数值区别开来，参照这两个正整数，将栈中的其余数值做轮换。

$$any_{n-1}\cdots any_0\ n\ j\ \texttt{roll}\ 输出\ any_{(j-1)\mod n}\cdots any_0\ any_{n-1}\cdots any_{j\mod n}$$

(a) (b) (c) 3 1 roll 输出 (c) (a) (b)

round 四舍五入，需要一个输入值。

sin 计算角度的正弦值，需要一个输入值，输出为实数。

sqrt 计算非负数的平方根。

sub 计算两数的差，需要两个输入值，计算前数减去后数的差。

true 逻辑值 true.

truncate 去掉输入值的小数部分，保留整数部分并输出之，输出数值类型与输入值相同。

xor 异或运算，需要两个输入值，输入值可以是布尔值，也可以是整数（此时是按位运算）。

第二十章　透明度

20.1　指定不透明度

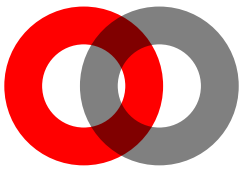
可以为画出的线条（stroke）、填充的颜色（fill）指定不透明度。

\pgfsetstrokeopacity{⟨*value*⟩}

参数 ⟨*value*⟩ 会被 \pgfmathparse 处理。本命令的定义是：

```
\def\pgfsetstrokeopacity#1{%
  \pgfmathparse{#1}%
  \expandafter\pgfsys@stroke@opacity\expandafter{\pgfmathresult}}
```

本命令为画线操作指定不透明度，⟨*value*⟩ 是 0 到 1 之间的数值，1 代表“完全不透明”，0 代表“完全透明”。



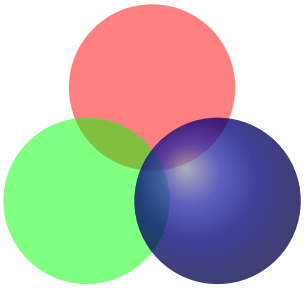
```
\begin{pgfpicture}
  \pgfsetlinewidth{5mm}
  \color{red}
  \pgfpathcircle{\pgfpoint{0cm}{0cm}}{8mm} \pgfusepath{stroke}
  \color{black}
  \pgfsetstrokeopacity{0.5}
  \pgfpathcircle{\pgfpoint{1cm}{0cm}}{8mm} \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetfillopacity{⟨*value*⟩}

参数 ⟨*value*⟩ 会被 \pgfmathparse 处理。本命令的定义是：

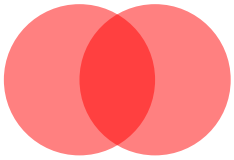
```
\def\pgfsetfillopacity#1{%
  \pgfmathparse{#1}%
  \expandafter\pgfsys@fill@opacity\expandafter{\pgfmathresult}}
```

本命令为填充颜色操作指定不透明度，⟨*value*⟩ 是 0 到 1 之间的数值。这个命令指定的不透明度不仅对填充色有效，对路径上的文字、插入的外部图形、颜色渐变都有效。



```
\begin{tikzpicture}
  \pgfsetfillopacity{0.5}
  \fill[red] (90:1cm) circle (11mm);
  \fill[green] (210:1cm) circle (11mm);
  \fill[shading=ball] (-30:1cm) circle (11mm);
\end{tikzpicture}
```

注意，在默认下，同一区域内的不透明度是叠加。也就是说，如果两个绘图命令都带有不透明度设置，并且两个命令画的图有重合部分，那么重合部分的不透明度是两个命令的不透明度的叠加。如果不希望叠加不透明度，可以通过“透明度组”来设置，见后文。

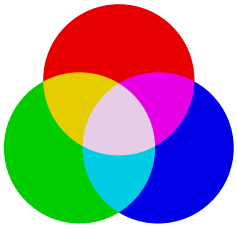


```
\begin{tikzpicture}
  \pgfsetfillopacity{0.5}
  \fill[red] (0,0) circle (1);
  \fill[red] (1,0) circle (1);
\end{tikzpicture}
```

20.2 指定混色模式

\pgfsetblendmode{⟨*mode*⟩}

混色模式见 §23.3. 混色模式是 PDF 的高级特性，未必总能得到正确显示。



```
\tikz [transparency group] {
  \pgfsetblendmode{screen}
  \fill[red!90!black] ( 90:.6) circle (1);
  \fill[green!80!black] (210:.6) circle (1);
  \fill[blue!90!black] (330:.6) circle (1);
}
```

20.3 Fading 效果

Fading 效果涉及“亮度”（luminosity）这个概念，这个概念可以简单理解为：一个像素点上，一秒钟内发出的可见光能量有多少。假设在一个白色背景上有个红色点，红色点的亮度越高，该点的红色就越鲜艳、越刺眼；红色点的亮度越低，该点的红色可见光就越少，白色背景就会显露出来，可见亮度与透明度有一定关系。如果一个点处没有出现任何光线，那么在理论上这个点没有颜色，在视觉上这个点是黑色的。因此没有颜色的点和黑色像素点的亮度都规定为 0，白色像素点的亮度是 1. 无论亮度怎么变化，红色都不会变成“暗红色”，因为红色与暗红色不是同一种颜色。

假设在一个原本没有颜色（黑色）的 TEX 盒子里写下单词 TikZ, 并且文字颜色是白色，此时盒子里的点只有两种：一种是构成文字的白色点，亮度为 1，另一种是没有颜色（黑色）的点，亮度为 0. 然后把这盒子想象成一个特别的“印章”，把另外某个盒子看作是“纸”，把“印章”盖到“纸”上，同时把印章的亮度值转换成不透明度值“印”在“纸”上，这样“纸”上的点就有了自己的不透明度值。如果这张“纸”原本是红色的，被盖章后，被白色文字 TikZ 覆盖的部分的不透明度是 1，其余部分的不透明度都是 0，于是原本的红色“纸”就变了，纸上有红色的文字 TikZ, 除文字外，纸的其余部分都没有颜色（完全透明，没有可见光能量）。也就是说，“印章”是一种“映射”，或者说赋值技术。

\pgfdeclarefading⟨*name*⟩{⟨*contents*⟩}

这个命令声明一个名称为 ⟨*name*⟩ 的 fading 样式，以供之后引用。本命令的声明全局有效。

⟨*contents*⟩ 是 TEX 内容，会被放入一个 TEX 盒子里。⟨*contents*⟩ 可以是文字，表格环境，数学公式，`{pgfpicture}` 环境，颜色渐变等，用于制作“印章”。注意 ⟨*contents*⟩ 中的颜色最好只涉及黑、白、灰 3 种颜色，灰色用 `black!20` 之类的颜色表达式表示。如果 ⟨*contents*⟩ 中涉及彩色可能不太容易控制。

为了方便，称盛放“印章”的盒子为“fading 盒子”。


```
\pgfdeclarefading{fading1}{\color{white}Ti\emph{k}Z}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading1}{\pgftransformshift{\pgfpoint{1cm}{1cm}}}
  \fill [red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

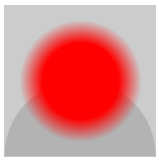
上面例子中，白色印章 fading1 印在红色纸上，得到红色文字 TikZ.

pgftransparent 这个预定义颜色就是 black, 在 fading 中对应“完全透明”。pgftransparent!20 表示一个灰色，pgftransparent!0 表示白色。fading 图形的内容的灰度默认为 pgftransparent!0, 所以上面例子中的 \color{white} 并不必要。

下面用一个颜色渐变图形制作一个“fading 盒子”。

```
\pgfdeclarefading{fading2}{
  \tikz \shade[left color=pgftransparent!0,
      right color=pgftransparent!100] (0,0) rectangle (2,2);}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading2}{\pgftransformshift{\pgfpoint{1cm}{1cm}}}
  \fill [red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

下面先声明一个辐射渐变，然后将这个辐射渐变用于制作 fading 盒子。



```
\pgfdeclareradialshading{myshading}{\pgfpointorigin}
{
  color(0mm)=(pgftransparent!0);
  color(5mm)=(pgftransparent!0);
  color(8mm)=(pgftransparent!100);
  color(15mm)=(pgftransparent!100)
}
\pgfdeclarefading{fading3}{\pgfuseshading{myshading}}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading3}{\pgftransformshift{\pgfpoint{1cm}{1cm}}}
  \fill [red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

\pgfsetfading{⟨*name*⟩}{⟨*transformations*⟩}

声明一个 fading 盒子后，就可以用本命令使用来使用它。在前文已经有本命令的例子。

本命令对之后的路径起作用，为之后的路径设置“状态参数”，本命令的有效范围一直延续到绘图环境（不是 TEX 组）结束，或者遇到下一个 \pgfsetfading.

“fading 盒子”是个 TEX 盒子。在使用 fading 盒子时，这个盒子的中心会被放在绘图环境坐标系的原点上，并且程序不会自动调整这个盒子的尺寸。

命令参数 ⟨*transformations*⟩ 是 PGF 的变换命令，用于对 fading 盒子做某些变换，例如平移、旋转、放缩等。对 fading 盒子做完变换后，本命令之后的路径会剪切 fading 盒子。此时可能有多种情况，例如，路径内部包含 fading 盒子但不能被 fading 盒子填满，或者，路径内部不包含 fading 盒子但与之有交集，或者，路径内部与 fading 盒子无交集。无论何种情况，对于路径内部的点，只要未被 fading 盒子“盖上章”，其不透明度就是 0，因而完全透明。

\pgfsetfadingforcurrentpath{⟨*name*⟩}{⟨*transformations*⟩}

这个命令类似 \pgfsetfading, 只是作用稍复杂。使用本命令后会有以下效果：

1. 如果当前路径是空的，则与 \pgfsetfading 的作用相同。
2. 如果当前路径非空，则类似 \pgfshadepath, 本命令变换 fading 盒子，使其中心与当前路径的边界盒子的中心重合，并变换 fading 盒子的宽度和高度，使之分别成为当前路径边界盒子的宽度和高度的 2 倍。
3. 执行参数 ⟨*transformations*⟩, 这是某些 PGF 变换命令，进一步变换 fading 盒子。

\pgfsetfadingforcurrentpathstroked{⟨*name*⟩}{⟨*transformations*⟩}

本命令类似 \pgfsetfadingforcurrentpath, 只是会在水平方向和竖直方向上分别扩大当前路径边界盒子的尺寸，扩大的增量是路径线宽。在计算当前路径的边界盒子时并不考虑路径线宽，如果路径线宽值较大并且需要画出路径来显示路径的 fading 效果，那么就可能需要扩大当前路径边界盒子的尺寸来容纳路径线宽。

比较下面两个图形，其中用了 fadings 程序库提供的 east 样式：

```
\begin{tikzpicture}
  \pgfsetlinewidth{4mm}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfsetfadingforcurrentpathstroked{east}{}
  \pgfusepath{stroke}
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \pgfsetlinewidth{4mm}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfsetfadingforcurrentpath{east}{}
  \pgfusepath{stroke}
\end{tikzpicture}
```

20.4 透明度组

下面的环境声明一个透明度组。

\begin{pgftransparencygroup}[⟨*options*⟩]
  ⟨*environment content*⟩
\end{pgftransparencygroup}

这个环境只能用在 {pgfpicture} 环境中。

在默认下，本环境外的透明度设置不能影响本环境内的内容；本环境的透明度设置也不会影响到环境外的内容。

环境选项 ⟨*options*⟩ 中可以使用的选项如下，其作用参考 §23.5：

knockout=⟨*true or false*⟩ (默认值 true, 初始值 false)

注意这个选项的默认值是 true, 初始值是 false, 有的渲染器不支持这个选项的作用。

isolated=⟨*true or false*⟩ (默认值 true, 初始值 true)

如果能得到驱动的支持，PGF 会猜测透明度组的内容的尺寸。

```
\pgftransparencygroup
  ⟨environment contents⟩
\endpgftransparencygroup
```

这是 Plain TeX 中的 {pgftransparencygroup} 环境。

```
\startpgftransparencygroup
  ⟨environment contents⟩
\stoppgftransparencygroup
```

这是 ConTeXt 中的 {pgftransparencygroup} 环境。

第二十一章　临时寄存器

参考文件《pgfsys.code.tex》.

如果你有全面的 TEX 编程知识，并且熟悉 PGF 的基本层，那你可以为 PGF 编写新的程序库，此时你可能用到 TEX 的临时寄存器（temporary registers），这里介绍分配给 PGF 的几个临时寄存器。

Internal dimen register \pgf@x

Internal dimen register \pgf@y

这两个 PGF 内部尺寸寄存器主要用于处理点的坐标，它们用作点的两个坐标分量。

这两个寄存器是被“全局地”设置的，PGF 的其它寄存器则是“临时地”。这两个寄存器的值是频繁变化的，所以最好不要用它们设计你的计算过程，除非你知道它们的值是怎样变化的。在设计计算过程时，推荐使用临时寄存器。

Internal dimen register \pgf@xa

Internal dimen register \pgf@xb

Internal dimen register \pgf@xc

Internal dimen register \pgf@ya

Internal dimen register \pgf@yb

Internal dimen register \pgf@yc

这几个 PGF 内部尺寸寄存器主要用作临时寄存器，你可以在一个 TEX 分组内随意修改它们的值。

注意：PGF 使用这些寄存器来执行路径操作，为了提高效率，路径命令并不监视这些寄存器，当它们的值被重定义时，PGF 并不中断处理。用户在使用内部的临时寄存器做计算时应当谨慎一些。

Internal dimen register \pgfutil@tempdima

Internal dimen register \pgfutil@tempdimb

这是两个临时的尺寸寄存器。

Internal count register \c@pgf@counta

Internal count register \c@pgf@countb

Internal count register \c@pgf@countc

Internal count register \c@pgf@countd

这是 4 个计数器，用于执行关于整数的计算，用在 TEX 组内。

Internal openout handle \w@pgf@writea

这是个 \openout 手柄。

Internal openin handle \r@pgf@reada

这是个 \openin 手柄。

Internal box \pgfutil@tempboxa

在 TEX 组内使用的临时盒子。

第二十二章　快速命令

快速命令，即“quick”commands, 是普通命令的“q”版，运行起来比普通命令要快，但也牺牲了某些功能，因此最好在适当的情况下使用，例如需要执行的命令数量太多时。

22.1　快速坐标命令

参考文件《pgfcorepoints.code.tex》.

\pgfqpoint{⟨*x*⟩}{⟨*y*⟩}

参考 \pgfqpoint→ P. 251. 类似 \pgfpoint, 不过 ⟨*x*⟩ 和 ⟨*y*⟩ 都应该是简单的尺寸，例如 1pt 或 1cm, 但不可以是 2ex 或 1cm+1pt.

```
\def\pgfqpoint#1#2{\global\pgf@x=#1\relax\global\pgf@y=#2\relax}
```

\pgfqpointxy{⟨s_x⟩}{⟨s_y⟩}

类似 \pgfpointxy, ⟨s_x⟩ 和 ⟨s_y⟩ 应当是简单的数值，如 1.234, 不可以是表达式或者长度单位。

```
\def\pgfqpointxy#1#2{%
  \global\pgf@x=#1\pgf@xx%
  \global\advance\pgf@x by #2\pgf@yx%
  \global\pgf@y=#1\pgf@xy%
  \global\advance\pgf@y by #2\pgf@yy}
```

\pgfqpointxyz{⟨s_x⟩}{⟨s_y⟩}{⟨s_z⟩}

类似 \pgfqpointxy, 用于三维坐标点，⟨s_x⟩, ⟨s_y⟩ 和 ⟨s_z⟩ 应当是简单的数值，如 1.234, 不可以是表达式或者长度单位。

\pgfqpointscale{⟨*factor*⟩}{⟨*coordinate*⟩}

类似 \pgfpointscale, ⟨*factor*⟩ 应当是简单的数值。

```
\def\pgfqpointscale#1#2{%
  \pgf@process{#2}%
  \global\pgf@x=#1\pgf@x%
  \global\pgf@y=#1\pgf@y%
}
```

\pgfqpointpolar{⟨*angle*⟩}{⟨*dimen*⟩}

此命令：

1. 尺寸 ⟨*dimen*⟩ 直接全局地赋予寄存器 \pgf@x 和 \pgf@y
2. 计算 \pgfmathcos@{⟨*angle*⟩}, 再全局地赋值

   ```
   \global\pgf@x=\pgfmathresult\pgf@x%
   ```

3. 计算 \pgfmathsin@{⟨*angle*⟩}, 再全局地赋值

   ```
   \global\pgf@y=\pgfmathresult\pgf@y\relax%
   ```

22.2 快速创建路径的命令

参考文件《pgfcorequick.code.tex》.

快速创建路径的命令有以下特点：

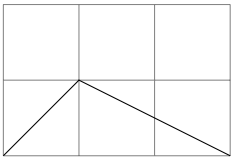
- 不跟踪边界盒子。
- 不能使用圆角。
- 不接受坐标变换。

快速创建路径的命令都使用软路径，都以 \pgfpathq 开头。

\pgfpathqmoveto{⟨*x dimension*⟩}{⟨*y dimension*⟩}

类似 \pgfpathmoveto, 可以开启一个路径或者以 move-to 方式延伸路径。此命令的定义是：

```
\def\pgfpathqmoveto#1#2{\pgfsyssoftpath@moveto{#1}{#2}}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformxshift{1cm}
  \pgfpathqmoveto{0pt}{0pt} % not transformed
  \pgfpathqlineto{1cm}{1cm} % not transformed
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfpathqlineto{⟨*x dimension*⟩}{⟨*y dimension*⟩}

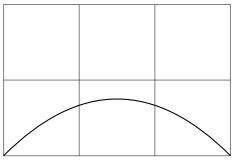
类似 \pgfpathlineto.

```
\def\pgfpathqlineto#1#2{\pgfsyssoftpath@lineto{#1}{#2}}
```

\pgfpathqcurveto{⟨s_x^1⟩}{⟨s_y^1⟩}{⟨s_x^2⟩}{⟨s_y^2⟩}{⟨t_x⟩}{⟨t_y⟩}

类似 \pgfpathcurveto.

```
\def\pgfpathqcurveto#1#2#3#4#5#6{\pgfsyssoftpath@curveto{#1}{#2}{#3}{#4}{#5}{#6}}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathqmoveto{0pt}{0pt}
  \pgfpathqcurveto{1cm}{1cm}{2cm}{1cm}{3cm}{0cm}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfpathqcircle{⟨*radius*⟩}

类似 \pgfpathcircle, 只不过构建的圆以原点为圆心，以 ⟨*radius*⟩ 为半径。

```
\def\pgfpathqcircle#1{%
  {%
    \pgf@x=#1%
    \pgf@y=0.5522847\pgf@x%
    \pgfsyssoftpath@moveto{\the\pgf@x}{0pt}%
    \pgfsyssoftpath@curveto{\the\pgf@x}{\the\pgf@y}{\the\pgf@y}{\the\pgf@x}{0pt}{
    ↪  \the\pgf@x}%
    \pgfsyssoftpath@curveto{-\the\pgf@y}{\the\pgf@x}{-\the\pgf@x}{\the\pgf@y}{-
    ↪  \the\pgf@x}{0pt}%
    \pgfsyssoftpath@curveto{-\the\pgf@x}{-\the\pgf@y}{-\the\pgf@y}{-\the\pgf@x
    ↪  }{0pt}{-\the\pgf@x}%
    \pgfsyssoftpath@curveto{\the\pgf@y}{-\the\pgf@x}{\the\pgf@x}{-\the\pgf@y}{
    ↪  \the\pgf@x}{0pt}%
    \pgfsyssoftpath@closepath%
```

```
  }%
}
```

22.3 快速使用路径的命令

快速使用路径的命令都以 \pgfusepathq 开头，它们有以下特点：

- 不能用来添加箭头。
- 不能调整路径。
- 不能截去路径的末端，对比命令 \pgfsetshortenend→ P. 299.
- 不能使用圆角。

在定义箭头时要使用这种命令。

\pgfusepathqstroke

只画出路径，没有其它动作，例如不添加箭头，不使用圆角。

```
\def\pgfusepathqstroke{%
  \pgfsyssoftpath@flushcurrentpath%
  \pgfsys@stroke%
  \pgf@resetpathsizes%
}
```

\pgfusepathqfill

只填充路径，没有其它动作。

```
\def\pgfusepathqfill{%
  \pgfsyssoftpath@flushcurrentpath%
  \pgfsys@fill%
  \pgf@resetpathsizes%
}
```

\pgfusepathqfillstroke

只填充、画出路径，没有其它动作。

```
\def\pgfusepathqfillstroke{%
  \pgfsyssoftpath@flushcurrentpath%
  \pgfsys@fillstroke%
  \pgf@resetpathsizes%
}
```

\pgfusepathqclip

用当前路径剪切本命令之后的各个路径，但是并不处理任何路径。

```
\def\pgfusepathqclip{%
  \pgfsyssoftpath@flushcurrentpath%
  \pgfsys@clipnext%
  \pgfsys@discardpath%
  \pgf@resetpathsizes%
}
```

22.4 快速文字盒子命令

参考文件《pgfcorescopes.code.tex》.

\pgfqbox{⟨*box number*⟩}

⟨*box number*⟩ 是某个 TEX 盒子的编号，这个盒子内可以包含任何允许的内容，如文字、表格环境、数学公式、{pgfpicture} 环境等。本命令用在 {pgfpicture} 环境内，将盒子 ⟨*box number*⟩ 插入到环境坐标系的原点处，盒子中心与坐标系原点重合。

```
\def\pgfqbox#1{%
  \pgfsys@hbox#1%
}
```

参考 \pgfsys@hbox→ P. 205.

\pgfqboxsynced{⟨*box number*⟩}

```
\def\pgfqboxsynced#1{%
  \pgfsys@hboxsynced#1%
}
```

参考 \pgfsys@hboxsynced→ P. 205.

第四部分

模块

第二十三章　非线性变换

非线性变换由模块 nonlineartransformations 定义。

```
\usepgfmodule{nonlineartransformations} % LaTeX and plain TeX and pure pgf
\usepgfmodule[nonlineartransformations] % ConTeXt and pure pgf
```

PGF 的构建路径的命令 (如 \pgfpathmoveto, \pgfpathlineto) 会先用当前 canvas 坐标系的矩阵对点坐标做仿射变换，再对仿射变换的结果 (即寄存器 \pgf@x 和 \pgf@y) 继续做非线性变换。

这里所谓的“非线性变换”实际是——修改仿射变换的结果，即寄存器 \pgf@x 和 \pgf@y 的值，并使所作的修改具有“非线性”。例如，假设 $t=$\pgf@x, $d=$\pgf@y, 那么 (参考 \pgfmathsincos→P. 122)：

```
\pgfmathsincos@{\pgf@sys@tonumber\pgf@x}%
\pgf@x=\pgfmathresultx\pgf@y%
\pgf@y=\pgfmathresulty\pgf@y%
```

将 ($t$pt, $d$pt) 变成 ($d\cos t$pt, $d\sin t$pt), 这是个非线性变换。

\pgf@nlt@list

这个宏在《pgfcorepathconstruct.code.tex》中定义，它的初始值等于 \pgfutil@empty. 在使用模块 nonlineartransformations 的非线性变换命令时，这个宏保存的代码就是“非线性运算”代码，这些代码最终应当具有这样的效果：为尺寸寄存器 \pgf@x 和 \pgf@y 赋值，或者修改这两个寄存器的值。对这两个寄存器值的修改应当体现出“非线性”(当然也可以是线性的)。

23.1　定义并载入一个非线性变换

\pgftransformnonlinear{⟨*transformation code*⟩}

参数 ⟨*transformation code*⟩ 是能改变寄存器 \pgf@x 和 \pgf@y 的值的代码，这些代码会被添加到宏 \pgf@nlt@list 中，也就是说，可以多次使用本命令，向 \pgf@nlt@list 中多次添加代码。这些代码规定了如何对坐标 (即寄存器 \pgf@x 和 \pgf@y) 做变换。

本命令的定义是：

```
\def\pgftransformnonlinear#1{%
  \expandafter\def\expandafter\pgf@nlt@list\expandafter{\pgf@nlt@list#1}%
  \let\pgf@nlt@moveto\pgf@nlt@moveto@nlt
  \let\pgf@nlt@lineto\pgf@nlt@lineto@nlt
  \let\pgf@nlt@curveto\pgf@nlt@curveto@nlt
  \let\pgf@nlt@closepath\pgf@nlt@closepath@nlt
}%
```

其中：

- 命令 \pgf@nlt@moveto 会被 \pgfpathmoveto 调用；
- 命令 \pgf@nlt@lineto 会被 \pgfpathlineto 调用；
- 命令 \pgf@nlt@curveto 会被 \pgfpathcurveto 调用；

- 命令 \pgf@nlt@closepath 会被 \pgfpathclose 调用，

这些命令会对构造路径的点，即寄存器 \pgf@x 和 \pgf@y 做变换，也就是用 \pgf@nlt@list 保存的代码来改变这两个寄存器的值，变换结果用于更新软路径，所以，使用 \pgftransformnonlinear 后，整个路径都会受到变换。

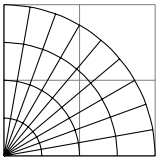
本命令的有效范围受到 TEX 组的限制。

假设 $t$pt =\pgf@x, $d$pt =\pgf@y, 下面的代码：

```
\makeatletter
\def\jizuobiaobianhuan{%
  \pgfmathsincos@{\pgf@sys@tonumber\pgf@x}%
  \pgf@x=\pgfmathresultx\pgf@y%
  \pgf@y=\pgfmathresulty\pgf@y%
}
\makeatother
```

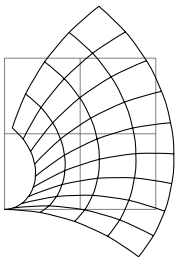
将变换 $(t\text{pt}, d\text{pt}) \to (d\cos t\text{pt}, d\sin t\text{pt})$ 保存到宏 \jizuobiaobianhuan 中。

下面利用 \jizuobiaobianhuan 将直角坐标系的网格变成极坐标系的网格。



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \pgftransformnonlinear{\jizuobiaobianhuan}
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);
\end{tikzpicture}
```

注意，在 Ti*k*Z 看来，如果路径中的坐标数据不带长度单位，那就默认它的长度单位是 cm, 然后用数学引擎的命令 \pgfmathparse 来处理坐标数据，这个命令会把各种长度单位都转换为 pt 再做计算，最后将结果中的单位 pt 去掉，仅保留结果的数值部分。这种“转换长度单位”的操作有时会带来麻烦，当对 Ti*k*Z 路径应用非线性变换时，一定要注意路径中的坐标数据的长度单位。上面例子中，作为横标的数据 90pt 带有单位 pt, 这是必要的，因为 \jizuobiaobianhuan 对横标的变换就是以 pt 为单位的 (尺寸寄存器的单位就是 pt)。



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \pgftransformrotate{20}
  \pgftransformnonlinear{\jizuobiaobianhuan}
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);
\end{tikzpicture}
```

上面例子中，网格线先被做线性变换——旋转 20 度——网格线不再是横平竖直的了，然后再做 (前面定义的) 非线性变换 \jizuobiaobianhuan, 所以结果是扭曲的。

另外定义一个非线性变换并保存在 \xpingfang 中：

```
\makeatletter%
\def\xpingfang{%
  \edef\pgfmath@temparg{\pgf@sys@tonumber\pgf@x}%
  \pgfmathsign{\pgfmath@temparg}%
  \let\fuhao\pgfmathresult%
  \pgfmathpow{\pgfmath@temparg}{2}% 即 x^2
  \pgfmathmultiply{\pgfmathresult}{0.035146}% 乘以长度单位转换因子
  \let\pingfang\pgfmathresult%
  \ifnum\fuhao>0\relax%
    \pgf@y=\pingfang pt\relax%
  \else%
```

```
    \pgf@y=-\pingfang pt\relax%
  \fi%
}
\makeatother%
```

假设 $x$pt =\pgf@x, $y$pt =\pgf@y, 上面的代码相当于变换

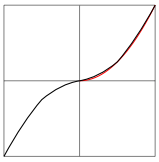
$$(x,y)\to(x,f(x)),\quad f(x)=\begin{cases}x^2, & x>0;\\ -x^2, & x\leqslant 0.\end{cases}$$

上面代码中，长度单位转换因子 0.035146 是 $\frac{1}{28.45274}$ 的近似值。在绘图时，Ti*k*Z 通常以 cm 为长度单位，但数学函数在计算时会把单位转换为 pt，而 1cm = 28.45274pt，

$$\texttt{1cm}\ \to\ \texttt{28.45274pt}\xrightarrow{\text{数值平方}}\texttt{809.5584135076pt}$$

因此，\pgfmathpow{1cm}{2} 的结果是 809.55861, 而不是 1. 所以为了让 1cm 的平方仍然对应 1cm, 这里必须乘上长度单位转换因子。

然后用 \xpingfang 作用于线段 (-1,0)--(1,0)，如下：



```
\begin{tikzpicture}
  \draw [help lines] (-1,-1) grid (1,1);
  \draw [red,line width=0.1pt] plot[domain=0:1] (\x,{\x*\x});
  \pgftransformnonlinear{\xpingfang}
  \draw (-1,0) -- (1,0);
\end{tikzpicture}
```

23.2 将非线性变换用于一个点

\pgfpointtransformednonlinear{⟨*point*⟩}

本命令先把当前的线性变换矩阵用于点 ⟨*point*⟩, 得到 P, 然后再将当前的非线性变换，即 \pgf@nlt@list 保存的代码作用于点 P, 得到点 Q. 此命令是否全局地保存点 Q (即对寄存器 \pgf@x 和 \pgf@y 的最终赋值是否为全局赋值)，决定于 \pgf@nlt@list 保存的代码。

本命令的定义是：

```
\def\pgfpointtransformednonlinear#1{%
  \pgf@process{%
    #1%
    \pgf@pos@transform@glob%
    \pgf@nlt@list%
  }
}%
```

23.3 将非线性变换用于一个路径

前文提到：

- \pgfpathmoveto 调用 \pgf@nlt@moveto=\pgf@nlt@moveto@nlt,
- \pgfpathlineto 调用 \pgf@nlt@lineto=\pgf@nlt@lineto@nlt,
- \pgfpathcurveto 调用 \pgf@nlt@curveto=\pgf@nlt@curveto@nlt,
- \pgfpathclose 调用 \pgf@nlt@closepath=\pgf@nlt@closepath@nlt,

可见非线性变换是针对 move-to, line-to, curve-to, closepath 这些构建路径的操作分别进行的。

对于 move-to 操作，就是简单地对点做变换，并做一些全局保存。

`\pgf@nlt@moveto@nlt{`⟨*dimension x*⟩`}{`⟨*dimension y*⟩`}`

参数 ⟨*dimension x*⟩ 和 ⟨*dimension y*⟩ 是刚性尺寸表达式，或者尺寸寄存器。

本命令在一个组内如下处理：

1. 将 ⟨*dimension x*⟩ 和 ⟨*dimension y*⟩ 复制到寄存器 `\pgf@x` 和 `\pgf@y` 中。
2. 用 `\pgf@nlt@list` 对 `\pgf@x` 和 `\pgf@y` 做变换，得到 ⟨*dimension x'*⟩ 和 ⟨*dimension y'*⟩, 仍然保存在 `\pgf@x` 和 `\pgf@y` 中。
3. 用 `\pgf@protocolsizes{\pgf@x}{\pgf@y}` 更新图形以及路径的边界盒子。
4. 用 `\pgfsyssoftpath@moveto{`⟨*dimension x'*⟩`}{`⟨*dimension y'*⟩`}` 更新软路径。
5. 令 `\pgf@nlt@last@moveto@orig` 全局地保存“`{`⟨*dimension x*⟩`}{`⟨*dimension y*⟩`}`”.
6. 令 `\pgf@nlt@last@coord@orig` 全局地保存“`{`⟨*dimension x*⟩`}{`⟨*dimension y*⟩`}`”.
7. 令 `\pgf@nlt@last@coord@trans` 全局地保存“`{`⟨*dimension x'*⟩`}{`⟨*dimension y'*⟩`}`”.
8. 令 `\pgf@nlt@last@coord@xaxis` 全局地保存“`{\the\pgf@xa` 的值`}{\the\pgf@ya` 的值`}`”.
9. 令 `\pgf@nlt@last@coord@yaxis` 全局地保存“`{\the\pgf@xb` 的值`}{\the\pgf@yb` 的值`}`”.

对于 line-to 操作创建的线段，变换过程依赖线段的长度。如果线段的 ∞ 范数小于 0.1pt, 则变换结果还是线段；否则，将线段看作是控制曲线，控制点是线段的端点和 2 个三等分点，对这个曲线做变换。

`\pgf@nlt@lineto@nlt{`⟨*dimension x*⟩`}{`⟨*dimension y*⟩`}`

参数 ⟨*dimension x*⟩ 和 ⟨*dimension y*⟩ 是刚性尺寸表达式，或者尺寸寄存器，或者保存尺寸的宏。

本命令在一个组内如下处理：

1. 将 ⟨*dimension x*⟩ 和 ⟨*dimension y*⟩ (被 `\edef`) 彻底展开的结果 (仍然记为 ⟨*dimension x*⟩ 和 ⟨*dimension y*⟩) 复制到寄存器 `\pgf@xc` 和 `\pgf@yc` 中。
2. 设置 `\ifpgfutil@tempswa` 的真值是 false.
3. 设 `\pgf@nlt@last@coord@orig` 保存的是点 O 的坐标，尺寸 ⟨*dimension x*⟩ 和 ⟨*dimension y*⟩ 是点 P 的坐标，如果范数 $\|P-O\|_\infty < 0.1\text{pt}$, 则设置 `\ifpgfutil@tempswa` 的真值是 true.
4. 检查 `\ifpgfutil@tempswa` 的真值：
   - 如果它的真值是 true, 说明点 O 与点 P 比较接近，则
     (a) 将 ⟨*dimension x*⟩ 和 ⟨*dimension y*⟩ 复制到寄存器 `\pgf@x` 和 `\pgf@y` 中。
     (b) 用 `\pgf@nlt@list` 对 `\pgf@x` 和 `\pgf@y` 做变换，得到 ⟨*dimension x'*⟩ 和 ⟨*dimension y'*⟩, 并全局地保存在 `\pgf@x` 和 `\pgf@y` 中。
     (c) 用 `\pgf@protocolsizes{\pgf@x}{\pgf@y}` 更新图形以及路径的边界盒子。
     (d) 用 `\pgfsyssoftpath@lineto{`⟨*dimension x'*⟩`}{`⟨*dimension y'*⟩`}` 更新软路径。也就是说，在点 O 与点 P 比较接近的情况下，非线性变换 `\pgf@nlt@list` 把线段 OP 仍然变成线段。
     (e) 令 `\pgf@nlt@last@coord@orig` 全局地保存“`{`⟨*dimension x*⟩`}{`⟨*dimension y*⟩`}`”.
     (f) 令 `\pgf@nlt@last@coord@trans` 全局地保存“`{`⟨*dimension x'*⟩`}{`⟨*dimension y'*⟩`}`”.
     (g) 令 `\pgf@nlt@last@coord@xaxis` 全局地保存“`{\the\pgf@xa` 的值`}{\the\pgf@ya` 的值`}`”.
     (h) 令 `\pgf@nlt@last@coord@yaxis` 全局地保存“`{\the\pgf@xb` 的值`}{\the\pgf@yb` 的值`}`”.

- 如果它的真值是 false, 说明点 O 与点 P 不接近，则

$$\begin{aligned}(\texttt{\textbackslash pgf@x},\texttt{\textbackslash pgf@y}) &= O,\\ (\texttt{\textbackslash pgf@xa},\texttt{\textbackslash pgf@ya}) &= \frac{2}{3}O+\frac{1}{3}P = O+\frac{1}{3}(P-O),\\ (\texttt{\textbackslash pgf@xb},\texttt{\textbackslash pgf@yb}) &= \frac{1}{3}O+\frac{2}{3}P = O+\frac{2}{3}(P-O),\\ (\texttt{\textbackslash pgf@xc},\texttt{\textbackslash pgf@yc}) &= P,\end{aligned}$$

  然后执行 `\pgf@nlt@inner@curve`. 也就是说，在点 O 与点 P 不接近的情况下，将线段 OP 看作是控制曲线，对它做非线性变换。

`\pgftransformnonlinearflatness`

这是个尺寸寄存器，它的初始值被设置为 5pt, 它被当作一个阈值使用。

`\pgfsettransformnonlinearflatness{`⟨*dimension*⟩`}`

本命令调用 `\pgfmathsetlength` 处理参数 ⟨*dimension*⟩, 处理结果赋予寄存器 `\pgftransformnonlinearflatness`.

对于 curve-to 创建的控制曲线 $C(t)$, 变换过程依赖控制点 P_1, P_2, P_3, P_4 的分布情况。如果范数 $\|P_1-P_2\|_\infty$, $\|P_2-P_3\|_\infty$, $\|P_3-P_4\|_\infty$ 中的某一个大于寄存器 `\pgftransformnonlinearflatness` 的值，那么就在 $t=0.5$ 处将 $C(t)$ 拆分为左右两部分：$C_l(t)$ 和 $C_r(t)$ ——把这种“检查范数、拆分曲线”的操作简记为“检—拆”操作。然后对 $C_l(t)$ 和 $C_r(t)$ 分别重复进行“检—拆”操作，直到拆分出来的曲线的控制点的这些范数不大于指定的阈值。然后对拆出来的这些曲线的控制点做变换，用变换后的控制点创建新曲线。

`\pgf@nlt@inner@curve`

本命令将 `(\pgf@x,\pgf@y)`, `(\pgf@xa,\pgf@ya)`, `(\pgf@xb,\pgf@yb)`, `(\pgf@xc,\pgf@yc)` 看作曲线 $C(t)$ 的控制点，对 $C(t)$ 做非线性变换。

本命令的处理是：

1. 将 `\pgf@nlt@last@coord@orig` 保存的尺寸赋予 `\pgf@x`,`\pgf@y`, 看作是点 P_1 的坐标;将 `(\pgf@xa,\pgf@ya)` 看作是点 P_2 的坐标；将 `(\pgf@xb,\pgf@yb)` 看作是点 P_3 的坐标；将 `(\pgf@xc,\pgf@yc)` 看作是点 P_4 的坐标。
2. 用 `\begingroup` 开启一个组。
3. 检查范数 $\|P_1-P_2\|_\infty$, $\|P_2-P_3\|_\infty$, $\|P_3-P_4\|_\infty$, 如果这 3 个范数中的某一个大于寄存器 `\pgftransformnonlinearflatness` 的值，则设置 `\ifpgfutil@tempswa` 的真值为 true; 否则设置其真值为 false.
4. 检查 `\ifpgfutil@tempswa` 的真值，
   - 如果它的真值是 true,
     (a) 用 `\endgroup` 结束之前开启的组。
     (b) 在一个组内执行一个迭代操作：
       i. 在 $t=0.5$ 处将 $C(t)$ 分割为左右两部分：$C_l(t)$ 和 $C_r(t)$.
       ii. 在一个组内对 $C_l(t)$ 执行 `\pgf@nlt@inner@curve`.
       iii. 在一个组内对 $C_r(t)$ 执行 `\pgf@nlt@inner@curve`.
   - 如果它的真值是 false,
     (a) 用 `\endgroup` 结束之前开启的组。
     (b) 执行 `\pgf@nlt@do@inner@curve`.

可见这个命令的处理过程会套嵌多个组，寄存器 \pgftransformnonlinearflatness 的值越小，套嵌的组越多。

\pgf@nlt@do@inner@curve

本命令处理 P_2, P_3, P_4 这 3 个控制点：

1. 令 \pgf@nlt@last@coord@orig 全局地保存 P_4 的坐标“{P_{4x}}{P_{4y}}”(两个尺寸)。
2. 用 \pgf@nlt@list 将 P_2, P_3, P_4 分别变换为：

$$P_2' = (\pgf@xa, \pgf@ya),$$
$$P_3' = (\pgf@xb, \pgf@yb),$$
$$P_4' = (\pgf@xc, \pgf@yc).$$

3. 令 \pgf@nlt@last@coord@trans 全局地保存 P_4' 的坐标“{P_{4x}'}{P_{4y}'}”(两个尺寸)。
4. 刷新图形、路径的边界盒子：

```
\pgf@protocolsizes{\pgf@xa}{\pgf@ya}%
\pgf@protocolsizes{\pgf@xb}{\pgf@yb}%
\pgf@protocolsizes{\pgf@xc}{\pgf@yc}%
```

5. 更新软路径：

```
\pgfsyssoftpath@curveto{\the\pgf@xa}{\the\pgf@ya}{\the\pgf@xb}{\the\pgf@yb}{
↪  \the\pgf@xc}{\the\pgf@yc}
```

\pgf@nlt@curveto@nlt{P_{2x}}{P_{2y}}{P_{3x}}{P_{3y}}{P_{4x}}{P_{4y}}

尺寸 P_{2x}, P_{2y} 是控制点 P_2 的坐标；尺寸 P_{3x}, P_{3y} 是控制点 P_3 的坐标；尺寸 P_{4x}, P_{4y} 是控制点 P_4 的坐标。

本命令在一个组内：

1. 赋值

$$\pgf@xa = P_{2x}, \pgf@ya = P_{2y},$$
$$\pgf@xb = P_{3x}, \pgf@yb = P_{3y},$$
$$\pgf@xc = P_{4x}, \pgf@yc = P_{4y},$$

2. 执行 \pgf@nlt@inner@curve.

对于 closepath 创建的线段，变换的过程依赖线段长度。如果线段的 ∞ 范数不大于 0.01pt, 则变换结果还是线段；否则，将线段看作是控制曲线，控制点是线段的端点和 2 个三等分点，对这个曲线做变换。

\pgf@nlt@closepath@nlt

本命令在一个组内：

1. 将 \pgf@nlt@last@moveto@orig 保存的尺寸赋予寄存器 \pgf@x 和 \pgf@y, 看作是点 M 的坐标。
2. 将 \pgf@nlt@last@coord@orig 保存的尺寸看作是点 N 的坐标。
3. 如果范数 $\|M-N\|_\infty > 0.01$pt, 则设置 \ifpgfutil@tempswa 的真值为 true, 否则设置它的真值为 false.
4. 检查 \ifpgfutil@tempswa 的真值，如果真值为 true, 则

```
\pgf@nlt@lineto@nlt{\pgf@x}{\pgf@y}
```

5. 执行 \pgfsyssoftpath@closepath, 更新软路径。

23.4 用线性变换近似非线性变换

\pgfapproximatenonlineartransformation

这个命令的作用是：创建新的 (canvas 坐标系的) 线性变换矩阵，使之在原点处能够近似“当前线性变换矩阵和非线性变换的叠加作用效果”，并且清除当前 TEX 组内的、此命令之前的线性变换、非线性变换。

本命令的处理是：如果 \pgf@nlt@list 是空的，则什么也不做；否则：

1. 用 \pgfpointtransformednonlinear 对单位点和原点做变换：

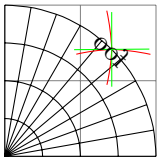
$$\begin{aligned}(1\mathrm{pt},0\mathrm{pt}) = e_1 \to e_1' = (e_{1x}'\mathrm{pt}, e_{1y}'\mathrm{pt}),\\ (0\mathrm{pt},1\mathrm{pt}) = e_2 \to e_2' = (e_{2x}'\mathrm{pt}, e_{2y}'\mathrm{pt}),\\ (0\mathrm{pt},0\mathrm{pt}) = O \to O' = (O_x'\mathrm{pt}, O_y'\mathrm{pt}),\end{aligned}$$

2. 以 $\overrightarrow{O'e_1'}$, $\overrightarrow{O'e_2'}$ 为基向量，以 O' 为原点构成一个 canvas 坐标系，这个坐标系在不变坐标系中的矩阵是

$$\begin{bmatrix} e_{1x}' - O_x' & e_{1y}' - O_y' & 0 \\ e_{2x}' - O_x' & e_{2y}' - O_y' & 0 \\ O_x'\mathrm{pt} & O_y'\mathrm{pt} & 1 \end{bmatrix}$$

3. 清空非线性变换，并还原相应的命令：

```
\let\pgf@nlt@list\pgfutil@empty%
\let\pgf@nlt@moveto\pgf@lt@moveto%
\let\pgf@nlt@lineto\pgf@lt@lineto%
\let\pgf@nlt@curveto\pgf@lt@curveto%
\let\pgf@nlt@closepath\pgf@lt@closepath%
```



macro:->{-0.70854}{0.6964}{0.7071}{0.7071}{40.2383pt}{40.2383pt}

```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \pgftransformnonlinear{\jizuobiaobianhuan}
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);
  \begin{scope}[shift={(45pt,20mm)}]
    \draw [red] (-10pt,-10pt) -- (10pt,10pt);
    \draw [red] (10pt,-10pt) -- (-10pt,10pt);
    \pgfapproximatenonlineartransformation  % 做近似
    \draw [green] (-10pt,-10pt) -- (10pt,10pt);
    \draw [green] (10pt,-10pt) -- (-10pt,10pt);
    \pgftext{foo};
    \pgfgettransform\aaaa% 获取当前的线性变换矩阵
    \xdef\aaaa{\aaaa}
  \end{scope}
\end{tikzpicture}\par\meaning\aaaa
```

\pgfapproximatenonlineartranslation

记当前的线性变换矩阵是 $\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ s\mathrm{pt} & t\mathrm{pt} & 1 \end{bmatrix}$.

命令 \pgfpointtransformednonlinear 对原点的变换是 $(0\mathrm{pt},0\mathrm{pt}) \to (O_x'\mathrm{pt}, O_y'\mathrm{pt})$, 本命令的处理是：

1. 将线性变换矩阵修改为 $\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ O'_x\mathrm{pt} & O'_y\mathrm{pt} & 1 \end{bmatrix}$.
2. 清空非线性变换，并还原相应的命令：

```
\let\pgf@nlt@list\pgfutil@empty%
\let\pgf@nlt@moveto\pgf@lt@moveto%
\let\pgf@nlt@lineto\pgf@lt@lineto%
\let\pgf@nlt@curveto\pgf@lt@curveto%
\let\pgf@nlt@closepath\pgf@lt@closepath%
```

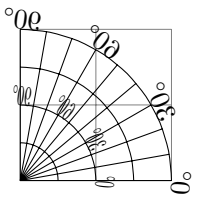
23.5 将非线性变换用于文字

非线性变换对文字无作用。不过创建文字盒子的 \pgftext, 以及创建 node 的 \pgfmultipartnode 都会调用命令 \pgfapproximatenonlineartranslation, 此命令生成一个线性变换矩阵来近似非线性变换，这个线性变换矩阵对 \pgftext 的文字盒子，以及 node 包含的文字盒子都有作用。也就是说，PGF 会先生成这个线性变换矩阵，再将盒子中的文字插入到当前位置，使文字的外观表现出这个变换的特征。

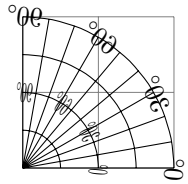
- 线性变换只能把直线变成直线，所以无法实现“文字扭曲”这种非线性变换效果。
- 命令 \pgfapproximatenonlineartranslation 只是在 canvas 坐标系的原点处获得近似非线性变换的线性变换矩阵，所以，如果文字盒子的锚定点不是原点，那么应该用平移命令将 canvas 坐标系的原点平移到其锚定点，或者用选项指定其锚定点 (这实际执行一个平移命令)。如果使用 \node 命令，还应当使用选项 transform shape nonlinear=true.

\angle=0, \dist=1→ macro:->{-0.00435}{0.49666}{1.0}{0.0}{28.45274pt}{0.0pt}
\angle=0, \dist=2→ macro:->{-0.0087}{0.99333}{1.0}{0.0}{56.9055pt}{0.0pt}
\angle=30, \dist=1→ macro:->{-0.25224}{0.42807}{0.86603}{0.5}{24.64085pt}{14.22636pt}
\angle=30, \dist=2→ macro:->{-0.50449}{0.85616}{0.86603}{0.5}{49.28174pt}{28.45274pt}
\angle=60, \dist=1→ macro:->{-0.43198}{0.24443}{0.5}{0.86603}{14.22636pt}{24.64085pt}
\angle=60, \dist=2→ macro:->{-0.86397}{0.48886}{0.5}{0.86603}{28.45274pt}{49.28174pt}
\angle=90, \dist=1→ macro:->{-0.49666}{-0.00435}{0.0}{1.0}{0.0pt}{28.45274pt}
\angle=90, \dist=2→ macro:->{-0.99333}{-0.0087}{0.0}{1.0}{0.0pt}{56.9055pt}

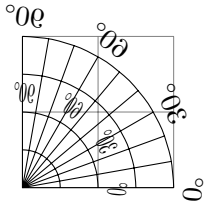
```
\pgftransformnonlinear{\jizuobiaobianhuan}
\foreach \angle in {0,30,60,90}
  \foreach \dist in {1,2}
    {
      \pgftransformshift{\pgfpoint{\angle pt}{\dist cm}}
      \pgfapproximatenonlineartransformation%
      \pgfgettransform\aaaa% 获取当前的线性变换矩阵
      \string\angle=\angle, \string\dist=\dist $\to$ \meaning\aaaa\par%
    }
```




```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \pgftransformnonlinear{\jizuobiaobianhuan}
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);
  \foreach \angle in {0,30,60,90}
    \foreach \dist in {1,2}
      {
        \pgftext[at={\pgfpoint{\angle pt}{\dist cm}},bottom]{$\angle^\circ$}
      }
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \pgftransformnonlinear{\jizuobiaobianhuan}
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);
  \foreach \angle in {0,30,60,90}
    \foreach \dist in {1,2}
      {
        \pgftransformshift{\pgfpoint{\angle pt}{\dist cm}}
        \pgftext{$\angle^\circ$}
      }
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \pgftransformnonlinear{\jizuobiaobianhuan}
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);
  \foreach \angle in {0,30,60,90}
    \foreach \dist in {1,2}
      {
        \node[at={(\angle pt,\dist cm)},above,
              transform shape nonlinear=true]{$\angle^\circ$};
      }
\end{tikzpicture}
```

第二十四章　装饰路径

介绍 decorations 模块。

```
\usepgfmodule{decorations} % LaTeX and plain TeX and pure pgf
\usepgfmodule[decorations] % ConTeXt and pure pgf
```

24.1　约定词语

对于一个完整的路径，如果它的某一部分处于 {pgfdecoration} 环境内，那么这一部分是“被装饰路径”，会被执行装饰操作。“被装饰路径”之前的那一部分是“前路径”(preexisting path)。

装饰操作是针对软路径的，“被装饰路径”的软路径会被另外保存，用于构造“装饰路径”。被装饰路径可不以 move-to 开头。“被装饰路径”的软路径会被 \pgf@decorate@parsesoftpath 解析，解析结果是一个花括号列表，每个列表项都是一个花括号组，每个组的内容是 move-to, line-to, curve-to 等的路径构造以及这个构造的长度、构造点信息。这个列表看作“输入路径”(input path)，保存在宏 \pgf@decorate@inputsegmentobjects 中，每一个列表项的内容叫做“子输入路径”。解析过程也会计算输入路径的总长度。然后依照输入路径创建装饰路径。

对于输入路径的装饰可以有不同的“风格”，每一种风格是一种“装饰类型”，每一个装饰类型由一个或者数个“状态”(state) 组成，这些状态之间可以相互切换，切换状态的规则在声明装饰类型时给出。每当执行一个状态，一般都会 (或者不会，决定于这个状态的定义) 创建一个“片段”(segment)，所有片段构成装饰路径。

模块 decorations 提供两个环境：pgfdecoration，pgfmetadecoration 来做装饰。

环境 pgfmetadecoration 要比 pgfdecoration 灵活一些。

环境 pgfdecoration 的处理过程大致是：

```
\begin{pgfpicture}
  %%%%%%%%%% 开始一个路径 %%%%%%%%%%
  % 前一段
  ⟨preexisting path⟩
  % 需要装饰的一段
  \begin{pgfdecoration}{
    {⟨decoration name 1⟩}{⟨distance 1⟩}{⟨before code 1⟩}{⟨after code 1⟩},
    {⟨decoration name 2⟩}{⟨distance 2⟩}{⟨before code 2⟩}{⟨after code 2⟩}
    ......
  }
    ⟨decorated path⟩
  \end{pgfdecoration}
  % 后一段
  ⟨postexisting path⟩
  \pgfusepath{stroke}
  %%%%%%%%%% 结束这个路径 %%%%%%%%%%
\end{pgfpicture}
```

对于上面的伪代码，被装饰路径 ⟨*decorated path*⟩ 会被解析为“输入路径”，其总长度记为 l_{total}，已装饰部分的长度记为 l_{comp}(初始为 `0pt`)，尚未被装饰部分的长度记为 l_{rem}(初始为 l_{total}). `{pgfdecoration}` 环境会执行两种套嵌的循环。

外层循环：针对环境参数列表、装饰类型的循环 对 `{pgfdecoration}` 环境的参数列表

```
{⟨decoration name 1⟩}{⟨distance 1⟩}{⟨before code 1⟩}{⟨after code 1⟩},
{⟨decoration name 2⟩}{⟨distance 2⟩}{⟨before code 2⟩}{⟨after code 2⟩}
......
```

执行一个循环：对 ⟨*decoration name i*⟩, $i = 1, 2, \ldots,$

- 若 ⟨*distance i*⟩ $< l_{rem}$, 则令 $l_{rem} = l_{rem} -$ ⟨*distance i*⟩, 继续循环
- 若 ⟨*distance i*⟩ $\geqslant l_{rem}$, 则令 ⟨*distance i*⟩ $= l_{rem}$, $l_{rem} =$ `0pt`, 结束循环

⟨*distance i*⟩ 是 ⟨*decoration name i*⟩ 的“装饰宽度”，或者说是 ⟨*decoration name i*⟩ 占据的宽度。

如果 ⟨*decoration name i*⟩ 是循环中的最后一个装饰类型，那么 ⟨*decoration name* $i+1$⟩ ... 会被忽略。这个循环的结果是：在被装饰路径上确定一系列点 $A_0, A_1, \ldots, A_n$, 其中 A_0 是路径起点，A_n 是路径终点；沿着被装饰路径，从点 A_{i-1} 到 A_i 的长度是 ⟨*distance i*⟩ 的这一段路径——记为 ⟨*path section i*⟩, 采用 ⟨*decoration name i*⟩ 这个装饰类型来装饰。

内层循环：处于单个装饰类型内的循环 在处理 ⟨*decoration name i*⟩ 这个循环时，会利用这个装饰类型的状态对被装饰路径做装饰。这个内层循环使得 ⟨*decoration name i*⟩ 的一系列状态被执行，即从一个状态切换到另一个状态。第一个状态必须是它的初始状态，当切换到它的终止状态 (名称为 `final`) 时，这个内层循环结束 (终止状态不属于这个内层循环，终止状态是内层循环结束后要执行的状态)。如果无法从当前的某个状态切换到另一个状态，那么当前状态就一直被循环。如果无法切换到终止状态，也会导致无限循环。每个状态都创建一个片段 (一个路径片段)，所有片段组成装饰路径。

循环的结果是：在 ⟨*path section i*⟩ 上决定一系列点 $B_{i,0}$，$B_{i,1}$，……，B_{i,n_i}，其中 $B_{i,0} = A_{i-1}$ 是 ⟨*path section i*⟩ 的起点，$B_{i,n_i} = A_i$ 是 ⟨*path section i*⟩ 的终点。沿着被装饰路径，$B_{i,j-1}B_{i,j}$ 这一段路径被某个状态 $\mathscr{S}_{i,j}$ 创建的片段 $S_{i,j}$ 装饰。状态 $\mathscr{S}_{i,j}$ 会在一个组中创建片段 $S_{i,j}$. 首先在一个“私有”坐标系内构造片段路径，然后片段路径被平移，使得这个“私有”坐标系的原点被平移到点 $B_{i,j-1}$; 然后再围绕点 $B_{i,j-1}$ 旋转一个角度 $\theta_{i,j}$ 得到片段 $S_{i,j}$:

- 如果点 $B_{i,j-1}$ 位于一个 lineto 子输入路径的内部，那么使用命令 `\pgfpointlineatdistance` 确定点 $B_{i,j-1}$ 的坐标，将 $B_{i,j-1}$ 作为平移向量；
  线段的方向角度使用 `\pgfmathanglebetweenpoints` 计算，保存在 `\pgfdecoratedangle` 中，作为旋转角度 $\theta_{i,j}$.
- 如果点 $B_{i,j-1}$ 位于一个 curveto 子输入路径的内部，使用命令 `\pgftransformcurveattime` 确定点 $B_{i,j-1}$ 的坐标，并作平移、旋转，旋转角度 $\theta_{i,j}$ 是曲线在点 $B_{i,j-1}$ 的切方向角度。

$B_{i,j-1}B_{i,j}$ 这一段路径长度是片段 $S_{i,j}$ (即状态 $\mathscr{S}_{i,j}$) 占据的“装饰宽度”——记为 $w_{i,j}$ ——未必等于片段 $S_{i,j}$ 自身的自然宽度 (这个片段自己的边界盒子的宽度)。片段 $S_{i,j}$ 的起点和终点也未必是 $B_{i,j-1}$ 和 $B_{i,j}$. 为了方便，点 $B_{i,j-1}$ 和 $B_{i,j}$ 分别称为状态 $\mathscr{S}_{i,j}$ (片段 $S_{i,j}$) 的“首参考点”和“末参考点”。

每当添加片段 $S_{i,j}$ 后，就会计算 $l_{i,j} =$ ⟨*distance i*⟩ $- w_{i,j}$, 这里的 $w_{i,j}$ 是宏 `\pgf@decorate@width` 保存的尺寸，这个宏被选项 `/pgf/decoration automaton/width` → P. 407 定义。当状态 $\mathscr{S}_{i,x+1}$ 的宽度 $w_{i,x+1}$ 大于 $l_{i,x}$ 时，状态 $\mathscr{S}_{i,x+1}$ 就不会被执行，而是切换到 `final` 状态——按前面的记号，`final` 状态的首参考点是 B_{i,n_i-1}, 末参考点是 B_{i,n_i}.

`final` 状态的片段形态可能会令人意外，可能的原因是，输入路径列表的最后两个列表项是

```
{\pgf@decorate@inputsegmentobject@endofinputsegments}
{\pgf@decorate@inputsegmentobject@endofinputsegments}
```

命令 \pgf@decorate@movealongpath 在读取这两个列表项时，可能导致 final 状态的参考点、路径方向出现意外偏移，所以 final 状态的片段最好不要太 fancy.

24.2 环境、命令

文件《pgfmoduledecorations.code.tex》。

```
\newdimen\pgfdecorationsegmentamplitude
\newdimen\pgfdecorationsegmentlength
\pgfdecorationsegmentamplitude2.5pt
\pgfdecorationsegmentlength10pt
\def\pgfdecorationsegmentangle{45}%
\def\pgfdecorationsegmentaspect{0.5}%
\def\pgfmetadecorationsegmentamplitude{2.5pt}%
\def\pgfmetadecorationsegmentlength{1cm}%
```

/pgf/decoration={⟨*options*⟩}

参数 ⟨*options*⟩ 是选项列表，这些选项的路径都是 /pgf/decoration. 本选项导致 ⟨*options*⟩ 被执行。

/pgf/decoration/amplitude={⟨*dimension*⟩} (initially 2.5pt)

本选项导致

```
\pgfmathsetlength\pgfdecorationsegmentamplitude{⟨dimension⟩}
```

寄存器 \pgfdecorationsegmentamplitude 的初始值是 2.5pt.

/pgf/decoration/meta-amplitude={⟨*code*⟩} (initially 2.5pt)

本选项导致

```
\def\pgfmetadecorationsegmentamplitude{⟨code⟩}
```

宏 \pgfmetadecorationsegmentamplitude 的初始值是 2.5pt.

/pgf/decoration/segment length={⟨*dimension*⟩} (initially 10pt)

本选项导致

```
\pgfmathsetlength\pgfdecorationsegmentlength{⟨dimension⟩}
```

寄存器 \pgfdecorationsegmentlength 的初始值是 2.5pt.

/pgf/decoration/meta-segment length={⟨*code*⟩} (initially 1cm)

本选项导致

```
\def\pgfmetadecorationsegmentlength{⟨code⟩}
```

宏 \pgfmetadecorationsegmentlength 的初始值是 1cm.

/pgf/decoration/angle={⟨*expression*⟩} (initially 45)

本选项导致

```
\pgfmathparse{⟨expression⟩}\let\pgfdecorationsegmentangle\pgfmathresult
```

宏 \pgfdecorationsegmentangle 的初始值是 45.

/pgf/decoration/aspect={⟨*expression*⟩} (initially 0.5)

本选项导致

```
\pgfmathparse{⟨expression⟩}\let\pgfdecorationsegmentaspect\pgfmathresult
```

宏 \pgfdecorationsegmentaspect 的初始值是 0.5.

/pgf/decoration/start radius=⟨value⟩ (initially 2.5pt)

/pgf/decoration/end radius=⟨value⟩ (initially 2.5pt)

/pgf/decoration/radius=⟨value⟩ (style)

这个样式设置 start radius=⟨value⟩, end radius=⟨value⟩.

/pgf/decoration/path has corners=true|false (initially false)

本选项设置 \ifpgfdecoratepathhascorners 的真值。

/pgf/decoration/reverse path=true|false (initially false)

本选项设置 \ifpgf@decorate@inputsegmentobjects@reverse 的真值。

24.2.1 声明一个装饰类型

\pgfdeclaredecoration{⟨decoration name⟩}{⟨initial state⟩}{⟨states⟩}

这个命令声明一种装饰类型。⟨decoration name⟩ 是装饰类型的名称。⟨initial state⟩ 是初始状态的名称。⟨states⟩ 是一个或数个 \state 命令定义的“状态”。

本命令的处理是：

1. 检查控制序列 pgf@metadecoration@@⟨decoration name⟩@initial 是否已定义 (不等于 \relax)，如果已定义则报错；如果未定义，则
2. 定义 \pgf@decorate@name 保存 ⟨decoration name⟩:

   ```
   \def\pgf@decorate@name{⟨decoration name⟩}
   ```

3. 定义控制序列 pgf@decorate@@⟨decoration name⟩@initial 保存 ⟨initial state⟩.

   ```
   \pgfutil@namedef{pgf@decorate@@⟨decoration name⟩@initial}{⟨initial state⟩}
   ```

4. 交换命令名称

   ```
   \let\pgf@orig@state\state%
   \let\state\pgf@decorate@state
   ```

5. 执行 ⟨states⟩. 这一步使用命令 \state 定义状态，应当至少定义 ⟨initial state⟩, final 两个状态。
6. \let\state\pgf@orig@state

本命令的定义有前缀 \long, 所以本命令的参数中可以有 \par.

\pgf@decorate@state{⟨state name⟩}[⟨options⟩]{⟨code⟩}

本命令定义一个名称为 ⟨state name⟩ 的状态。[⟨options⟩] 是可选的选项。⟨code⟩ 是这个状态对应的代码。

本命令定义两个控制序列：

- \csname pgf@decorate@@⟨decoration name⟩@⟨state name⟩@options\endcsname 保存 ⟨options⟩
- \csname pgf@decorate@@⟨decoration name⟩@⟨state name⟩@code\endcsname 保存 ⟨code⟩

这样使得 ⟨state name⟩ 属于 ⟨decoration name⟩. 可见这里的 ⟨code⟩ 只是被保存，只有在装饰路径的过程中，用到 ⟨state name⟩ 时才会执行 ⟨options⟩, ⟨code⟩.

⟨code⟩ 可以是某些计算命令，定义变量的代码，或者能生成软路径的命令。

24.2.1.1 状态选项

\state 命令的 ⟨*options*⟩ 选项都必须以 /pgf/decoration automaton 开头。因为执行状态的循环中要用到宏 \pgf@decorate@width(见命令 \pgf@decorate@do@code → P. 424), 所以除了 final 状态外，其他状态的 ⟨*options*⟩ 中应当使用选项 /pgf/decoration automaton/width 规定该状态占据的宽度。不过，循环中单个的状态并不被限制在组中执行，所以，只要第一个被执行的状态 (初始状态) 的 ⟨*options*⟩ 中使用了这个选项，就可以为之后的状态提供这个宏，所以初始状态必须使用这个选项，其他状态可以不用。

\pgf@decorate@switch@if{⟨*dimension*⟩}to ⟨*next state name*⟩\pgf@stop

参数 ⟨*dimension*⟩ 应当能被 \pgfmathsetlength 处理。

在 \pgf@decorate@next 等于 \relax, 并且寄存器 \pgfdecoratedremainingdistance 的值是 0pt, 或者这个寄存器的值小于 ⟨*dimension*⟩ 时，定义

```
\def\pgf@decorate@current@state{⟨next state name⟩}%
\pgf@decorate@repeatstate-1\relax%
\let\pgf@decorate@next\pgf@decorate@run%
```

本命令的定义是：

```
\def\pgf@decorate@switch@if#1to #2\pgf@stop{%
  \ifx\pgf@decorate@next\relax%
    \ifdim\pgfdecoratedremainingdistance=0pt\relax%
      \def\pgf@decorate@current@state{#2}%
      \pgf@decorate@repeatstate-1\relax%
      \let\pgf@decorate@next\pgf@decorate@run%
    \else%
      \pgfmathsetlength\pgf@x{#1}%
      \ifdim\pgfdecoratedremainingdistance<\pgf@x%
        \def\pgf@decorate@current@state{#2}%
        \pgf@decorate@repeatstate-1\relax%
        \let\pgf@decorate@next\pgf@decorate@run%
      \fi%
    \fi%
  \fi%
}%
```

\pgf@decorate@switch@ifinputsegment{⟨*dimension*⟩}to ⟨*next state name*⟩\pgf@stop

参数 ⟨*dimension*⟩ 应当能被 \pgfmathsetlength 处理。

在 \pgf@decorate@next 等于 \relax, 并且寄存器 \pgfdecoratedinputsegmentremainingdistance 的值小于 ⟨*dimension*⟩ 时，定义

```
\def\pgf@decorate@current@state{⟨next state name⟩}%
\pgf@decorate@repeatstate-1\relax%
\let\pgf@decorate@next\pgf@decorate@run%
```

本命令的定义是：

```
\def\pgf@decorate@switch@ifinputsegment#1to #2\pgf@stop{%
  \ifx\pgf@decorate@next\relax%
    \pgfmathsetlength\pgf@x{#1}%
    \ifdim\pgfdecoratedinputsegmentremainingdistance<\pgf@x%
      \def\pgf@decorate@current@state{#2}%
      \pgf@decorate@repeatstate-1\relax%
      \let\pgf@decorate@next\pgf@decorate@run%
    \fi%
  \fi%
}%
```

\ifpgf@decorate@is@closepath@

子输入路径命令 \pgf@decorate@inputsegmentobject@closepath 会设置这个 TEX-if 的真值为 true, 其他情况下它的真值是 false.

\pgf@decorate@auto@end{⟨*dimension*⟩}

参数 ⟨*dimension*⟩ 应当能被 \pgfmathsetlength 处理。

本命令在 \pgf@decorate@next 等于 \relax 时有效。

本命令的定义是：

```
\def\pgf@decorate@auto@end#1{%
  \ifx\pgf@decorate@next\relax%
    \pgfmathsetlength\pgf@x{#1}%
    \ifpgf@decorate@is@closepath@%
      \ifdim\pgfdecoratedremainingdistance>\pgf@x%
        \ifdim\pgfdecoratedinputsegmentremainingdistance>\pgf@x%
        \else%
          \pgfpathclose%
          \pgf@decorate@movealongpath{\pgfdecoratedinputsegmentremainingdistance}%
          \pgf@decorate@repeatstate-1\relax%
          \let\pgf@decorate@next\pgf@decorate@run%
        \fi%
      \else%
        \pgfpathclose%
        \def\pgf@decorate@current@state{final}%
        \def\pgf@decorate@width{\pgfdecoratedremainingdistance}
        \pgf@decorate@repeatstate-1\relax%
        \let\pgf@decorate@next\pgf@decorate@run%
      \fi%
    \else%
      \ifdim\pgfdecoratedremainingdistance>\pgf@x%
      \else%
        {%
          \pgftransformreset%
          \pgf@decorate@transformtoinputsegment%
          \pgf@decorate@additionaltransform%
          \pgfpathlineto{\pgfpoint{\pgfdecoratedremainingdistance}{0pt}}
        }%
        \def\pgf@decorate@current@state{final}%
        \def\pgf@decorate@width{\pgfdecoratedremainingdistance}
        \pgf@decorate@repeatstate-1\relax%
        \let\pgf@decorate@next\pgf@decorate@run%
      \fi%
    \fi%
  \fi%
}%
```

本命令的主要作用是，在寄存器 \pgfdecoratedremainingdistance⩽⟨*dimension*⟩ 时，添加一个 \pgfpathclose 命令 (一个闭合操作) 或者 \pgfpathlineto 命令 (向当前的子输入路径的终点画一条直线段)，然后切换到 final 状态。

\ifpgfdecoratepathhascorners

这个 TEX-if 的真值由选项 /pgf/decoration/path has corners 设置。

\pgf@decorate@auto@corner{⟨*dimension*⟩}

参数 ⟨*dimension*⟩ 应当能被 \pgfmathsetlength 处理。

本命令在 \ifpgfdecoratepathhascorners 的真值为 true, 且 \pgf@decorate@next 等于 \relax 时有效。

本命令的定义是：

```
\def\pgf@decorate@auto@corner#1{%
  \ifpgfdecoratepathhascorners%
    \ifx\pgf@decorate@next\relax%
      \pgfmathsetlength\pgf@x{#1}%
      \ifdim\pgfdecoratedinputsegmentremainingdistance>\pgf@x%
      \else%
        {%
          \pgftransformreset%
          \pgf@decorate@transformtoinputsegment%
          \pgf@decorate@additionaltransform%
          \pgfpathlineto{\pgfqpoint{\pgfdecoratedinputsegmentremainingdistance
          ↪  }{0pt}}
        }%
        \pgf@decorate@movealongpath{\pgfdecoratedinputsegmentremainingdistance}%
        \pgf@decorate@repeatstate-1\relax%
        \let\pgf@decorate@next\pgf@decorate@run%
      \fi%
    \fi%
  \fi%
}%
```

本命令的主要作用是，在寄存器 \pgfdecoratedinputsegmentremainingdistance$\leqslant\langle dimension\rangle$ 时，添加一个 \pgfpathlineto 命令 (向当前的子输入路径的终点画一条直线段)，然后将状态的末参考点设置为子输入路径的终点。

/pgf/decoration automaton/width=$\langle dimension\rangle$

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/width/.code=\def\pgf@decorate@width{#1}
  ↪  \pgf@decorate@switch@if#1 to final\pgf@stop,%
}
```

参数 $\langle dimension\rangle$ 应当能被 \pgfmathsetlength 处理。本选项的作用是：

1. 定义宏 \pgf@decorate@width 保存参数 $\langle dimension\rangle$.
2. 在 \pgf@decorate@next 等于 \relax, 并且寄存器 \pgfdecoratedremainingdistance 的值是 0pt, 或者这个寄存器的值小于 $\langle dimension\rangle$ 时，定义

```
\def\pgf@decorate@current@state{final}%
\pgf@decorate@repeatstate-1\relax%
\let\pgf@decorate@next\pgf@decorate@run%
```

/pgf/decoration automaton/switch if less than=$\langle dimension\rangle$to $\langle next\ state\ name\rangle$

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/switch if less than/.code=\pgf@decorate@switch@if#1
  ↪  \pgf@stop,%
}
```

参数 $\langle dimension\rangle$ 应当能被 \pgfmathsetlength 处理。本选项的作用是：在 \pgf@decorate@next 等于 \relax, 并且寄存器 \pgfdecoratedremainingdistance 的值是 0pt, 或者这个寄存器的值小于 $\langle dimension\rangle$ 时，定义

```
\def\pgf@decorate@current@state{⟨next state name⟩}%
\pgf@decorate@repeatstate-1\relax%
\let\pgf@decorate@next\pgf@decorate@run%
```

/pgf/decoration automaton/switch if input segment less than=⟨*dimension*⟩to ⟨*next state name*⟩

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/switch if input segment less
  ↪  than/.code=\pgf@decorate@switch@ifinputsegment#1\pgf@stop,%
}
```

参数 ⟨*dimension*⟩ 应当能被 \pgfmathsetlength 处理。本选项的作用是：在 \pgf@decorate@next 等于 \relax, 并且寄存器 \pgfdecoratedinputsegmentremainingdistance 的值小于 ⟨*dimension*⟩ 时，定义

```
\def\pgf@decorate@current@state{⟨next state name⟩}%
\pgf@decorate@repeatstate-1\relax%
\let\pgf@decorate@next\pgf@decorate@run%
```

/pgf/decoration automaton/next state=⟨*next state name*⟩

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/next state/.store in=\pgf@decorate@next@state,%
}
```

本选项导致定义：

```
\def\pgf@decorate@next@state{⟨next state name⟩}
```

/pgf/decoration automaton/persistent precomputation={⟨*pre code*⟩}

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/persistent precomputation/.store
  ↪  in=\pgf@decorate@persistent@pre,%
}
```

本选项导致定义：

```
\def\pgf@decorate@persistent@pre{⟨pre code⟩}
```

/pgf/decoration automaton/persistent postcomputation={⟨*post code*⟩}

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/persistent postcomputation/.store
  ↪  in=\pgf@decorate@persistent@post,%
}
```

本选项导致定义：

```
\def\pgf@decorate@persistent@post{⟨post code⟩}
```

/pgf/decoration automaton/repeat state={⟨*number*⟩}

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/repeat state/.code={%
    \ifnum\pgf@decorate@repeatstate=-1\relax%
```

```
      \pgfmathsetcount\pgf@decorate@repeatstate{#1}%
    \fi%
  },%
}
```

/pgf/decoration automaton/if input segment is closepath={⟨*options*⟩}

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/if input segment is closepath/.code={%
    \ifpgf@decorate@is@closepath@%
      \pgfkeysalso{#1}%
    \fi%
  },
}
```

/pgf/decoration automaton/auto end on length={⟨*dimension*⟩}

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/auto end on length/.code=\pgf@decorate@auto@end{#1},
}
```

/pgf/decoration automaton/auto corner on length={⟨*dimension*⟩}

本选项的定义是：

```
\pgfkeys{
  /pgf/decoration automaton/auto corner on length/.code=\pgf@decorate@auto@corner
  ↪  {#1},
}
```

24.2.1.2 状态代码中可用的命令

\pgfdecorateexistingpath

这个宏保存 {pgfdecoration} 环境之前的前路径 (是软路径)。

\pgfdecoratedpath

这个宏保存 {pgfdecoration} 环境之内的被装饰路径的软路径。

\pgfpointdecoratedpathfirst

这个宏保存当前输入路径的起点。

\pgfpointdecoratedpathlast

这个宏保存当前的输入路径的终点。

\pgfdecoratedpathlength

这个宏保存的是，当前状态所属的装饰类型 ⟨*decoration name*⟩ 所占据的“装饰宽度”⟨*distance*⟩.

\pgfdecoratedremainingdistance

这个寄存器的值是，当前状态所属的装饰类型 ⟨*decoration name*⟩ 所占据的“装饰宽度”⟨*distance*⟩ 中，尚未被装饰的距离。

\pgfdecoratedcompleteddistance

这个寄存器的值是，当前状态所属的装饰类型 ⟨*decoration name*⟩ 所占据的“装饰宽度”⟨*distance*⟩ 中，已被装饰的距离。

\pgfdecoratedinputsegmentlength

这个宏保存当前的子输入路径的长度。

\pgfpointdecoratedinputsegmentfirst

这个宏保存当前的子输入路径的起点。

\pgfpointdecoratedinputsegmentlast

这个宏保存当前的子输入路径的终点。

\pgfdecoratedinputsegmentremainingdistance

这个寄存器的值是，当下的，子输入路径的未装饰部分的长度。

\pgfdecoratedinputsegmentcompleteddistance

这个寄存器的值是，当下的，子输入路径的已装饰部分的长度。

\pgfdecoratedangle

前面提到，当前状态会在一个 TEX 组中创建片段，并对片段路径做平移、旋转，这个宏保存旋转角度。

24.2.2 pgfdecoration 环境

\begin{pgfdecoration}{⟨*a comma separated list of decoration specifications*⟩}
⟨*environment content*⟩
\end{pgfdecoration}

本环境的参数是一个用逗号分隔的列表，列表项的格式是：

{⟨*decoration name*⟩}{⟨*distance*⟩}{⟨*before code*⟩}{⟨*after code*⟩}

上述格式中，可以把 {⟨*before code*⟩} 或者 {⟨*after code*⟩} 省略 (如果省略，就相当于 \pgfutil@empty)，但不能省略 {⟨*decoration name*⟩} 或者 {⟨*distance*⟩}. 例如：

```
\pgfdecoration
  {
    {lineto}{\pgfdecoratedpathlength/3},
    {zigzag}{\pgfdecoratedpathlength/3},
    {lineto}{\pgfdecoratedpathlength/3}
  }
```

本环境的内容是能够生成软路径的命令，且不能包含 \pgfusepath 命令。本环境所做的装饰，就是针对环境内容生成的软路径的。

参数 ⟨***distance***⟩ 从当前的装饰起点开始，长度为 ⟨*distance*⟩ 的一段输入路径采用 ⟨*decoration name*⟩ 这种装饰类型来做装饰，是 ⟨*decoration name*⟩ 占据的“装饰宽度”。

⟨*distance*⟩ 会被 \pgfmathsetlength 处理。

可以在 ⟨*distance*⟩ 中使用：

- 宏 \pgfdecoratedpathlength, 整个输入路径 (被装饰路径) 的总长度。
- 寄存器 \pgfdecoratedremainingdistance, 整个输入路径 (被装饰路径) 的尚未被装饰部分的长度。
- 宏 \pgfpointdecoratedpathfirst, 这个宏保存输入路径的起点。
- 宏 \pgfpointdecoratedpathlast, 这个宏保存输入路径的终点。

参数 ⟨***before code***⟩ 先执行 ⟨*before code*⟩, 再开始 ⟨*decoration name*⟩ 的装饰。

可以在 ⟨*before code*⟩ 中使用：

- 宏 \pgfdecoratedpathlength, 它此时保存 ⟨*distance*⟩

- 寄存器 \pgfdecoratedremainingdistance, 它此时的值是 ⟨*distance*⟩

参数 ⟨***after code***⟩ ⟨*decoration name*⟩ 的装饰完毕后，执行 ⟨*after code*⟩.

⟨*decoration name*⟩ 的装饰必定以它的初始状态为第一个状态，以 final 状态为最后一个状态。

{pgfdecoration} 环境结束后，下面的命令可用：

\pgfdecorateexistingpath

这个宏保存 {pgfdecoration} 环境之前的前路径的软路径。

\pgfdecoratedpath

这个宏保存 {pgfdecoration} 环境之内的被装饰路径的软路径。

\pgfdecorationpath

这个宏保存 {pgfdecoration} 环境产生的输出路径 (装饰路径) 的软路径。如果输出的某一段装饰路径在 {pgfdecoration} 环境中被使用，即在状态规定的片段（装饰路径）中或者在 ⟨*after code*⟩ 中使用了 \pgfusepath 命令，那么这一段装饰路径不会保存到这个宏中。这个宏只保存最后一段未被使用的装饰路径。

\pgfpoint@decorated@pathlast

这个宏保存 {pgfdecoration} 环境内原来的输入路径的终点。

24.2.2.1 环境的处理过程

{pgfdecoration} 环境的处理过程：

```
% 以下 \pgfdecoration%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\begingroup%
  \def\pgf@decorate@decorationlist{#1}%
% 以下 \pgf@decoration@env%%%%%%%%%%%%%%%%%%%%%%%%%%
  \pgfgetpath\pgfdecorateexistingpath%
  \pgfsetpath\pgfutil@empty%
  \let\pgfdecorationpath\pgfutil@empty%
  \let\pgfdecoratedpath\pgfutil@empty%
  \let\pgfpoint@decorated@pathlast\pgfpointorigin%
  \edef\pgfpoint@decorate@existingpathlast{\pgf@x\the\pgf@path@lastx\pgf@y\the
  ↪  \pgf@path@lasty}%
  % Begin a group so transformations don't mess things up.
  \bgroup%
% 以上 \pgf@decoration@env%%%%%%%%%%%%%%%%%%%%%%%%%%
% 以上 \pgfdecoration%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      ⟨ 环境内容⟩
% 以下 \endpgfdecoration%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 以下 \pgf@decoration@endenv%%%%%%%%%%%%%%%%%%%%%%%
  \egroup%
  \pgftransformreset%
  % Save the existing soft path and restore the existing path.
  \pgfgetpath\pgfdecoratedpath%
  \pgfsetpath\pgfdecorateexistingpath%
  \ifx\pgfdecoratedpath\pgfutil@empty%
    \pgferror{I cannot decorate an empty path}%
  \else%
    % If the path consists of a single moveto token, make it
    % a very small horizontal line.
    \pgf@decorate@path@check@moveto\pgfdecoratedpath{%
      \advance\pgf@x by0.0001pt\relax%
```

```
30       \edef\pgfdecoratedpath{%
31         \expandafter\noexpand\pgfdecoratedpath%
32         \noexpand\pgfsyssoftpath@linetotoken{\the\pgf@x}{\the\pgf@y}%
33       }%
34     }%
35     {}%
36     % Remove special round tokens and get points.
37     \pgfprocessround{\pgfdecoratedpath}{\pgfdecoratedpath}%
38     % Parse the soft path into a series of decorated input segment objects.
39     \pgf@decorate@parsesoftpath{\pgfdecoratedpath}{\pgf@decorate@inputsegmentobjects}%
40     % Setup further options
41     \pgfkeys{/pgf/every decoration/.try}%
42     % Reverse objects if necessary.
43     \ifpgf@decorate@inputsegmentobjects@reverse%
44       \pgf@decorate@inputsegmentobjects@reverse{\pgf@decorate@inputsegmentobjects}{
       ↪   \pgf@decorate@inputsegmentobjects}%
45     \fi%
46     \let\pgf@decorated@remainingdistance\pgf@decorate@totalpathlength%
47     \let\pgfpoint@decorated@totalpathfirst\pgfpoint@decorated@firstparsed%
48     \let\pgfpoint@decorated@totalpathlast\pgfpoint@decorate@lastnonmovetoparsed%
49     \let\pgfpoint@decorated@pathfirst\pgfpoint@decorated@totalpathfirst
50     \let\pgfpoint@decorated@pathlast\pgfpoint@decorated@totalpathlast%
51     % Set up the first input segment.
52     \let\pgf@decorate@currentinputsegmentobjects\pgf@decorate@inputsegmentobjects%
53     \let\pgf@decorate@transformtoinputsegment\pgfutil@empty%
54     \pgf@decorate@getnextinputsegmentobject\pgf@decorate@nextinputsegmentobject%
55     \pgf@decorate@processnextinputsegmentobject%
56     \pgf@decorate@distancetomove0pt\relax%
57   \fi%
58 % 以上 \pgf@decoration@endenv%%%%%%%%%%%%%%%%%%%%%
59   \ifx\pgfdecoratedpath\pgfutil@empty%
60   \else%
61     % Perform the decoration(s).
62     \pgf@decorate@for\pgf@temp:=\pgf@decorate@decorationlist\do{%
63       \ifx\pgf@temp\pgfutil@empty%
64       \else%
65         \expandafter\pgf@decorate@invoke\expandafter{\pgf@temp}%
66       \fi%
67     }%
68   \fi%
69   \pgfgetpath\pgfdecorationpath%
70     % Take stuff outside the group.
71   \global\let\pgf@decorate@decorationpathtemp\pgfdecorationpath%
72   \global\let\pgf@decorate@decoratedpathtemp\pgfdecoratedpath%
73   \global\let\pgf@decorate@existingpathtemp\pgfdecorateexistingpath%
74   \global\let\pgfpoint@decorated@pathlasttemp\pgfpoint@decorated@pathlast%
75 \endgroup%
76 % Are we in LaTeX?
77 \pgfutil@ifnextchar\@checkend{\aftergroup\pgf@decorate@installmacrosatend}%
78 {\pgf@decorate@installmacrosatend}%
79 % 以上 \endpgfdecoration%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

注释：

5, 6, 21, 22 本环境之前的软路径保存到宏 \pgfdecorateexistingpath 中；本环境之内的软路径保存到宏 \pgfdecoratedpath 中，是需要装饰的路径。本环境只对 \pgfdecoratedpath 进行操作。本环境得到 \pgfdecoratedpath 后，就将 \pgfdecorateexistingpath 用作当前的软路径。

23-25 如果 \pgfdecoratedpath 是空的，会导致错误。

28-35 检查 \pgfdecoratedpath 保存的内容是否如下

```
\pgfsoftpath@***token{⟨dimension x⟩}{⟨dimension y⟩}
```

这样，只有这 (简单的) 一组记号。如果是，就重定义 \pgfdecoratedpath, 使得它保存如下内容：

```
\pgfsoftpath@***token{⟨dimension x⟩}{⟨dimension y⟩}
\pgfsoftpath@linetotoken{⟨dimension x + 0.0001pt⟩}{⟨dimension y⟩}
```

37 处理 \pgfdecoratedpath 中的圆角参数，得到“纯粹的软路径”，即只包含花括号、尺寸、以及下面记号：

- \pgfsyssoftpath@movetotoken
- \pgfsyssoftpath@linetotoken
- \pgfsyssoftpath@curvetosupportatoken
- \pgfsyssoftpath@curvetosupportbtoken
- \pgfsyssoftpath@curvetotoken
- \pgfsyssoftpath@rectcornertoken
- \pgfsyssoftpath@rectsizetoken
- \pgfsyssoftpath@closepath

的软路径，仍然保存到 \pgfdecoratedpath 中。

39 解析 \pgfdecoratedpath 保存的软路径：

- 将路径的总长度保存在宏 \pgf@decorate@totalpathlength 中；
- 将软路径转换为“输入路径列表”，并保存在宏 \pgf@decorate@inputsegmentobjects 中，其内容例如

```
{\pgf@decorate@inputsegmentobject@moveto{\pgf@x 0.0pt\pgf@y0.0pt}}
{\pgf@decorate@inputsegmentobject@curveto{81.06355pt}
  {\pgf@x 0.0pt\pgf@y 0.0pt}
  {\pgf@x -27.21713pt\pgf@y 7.85689pt}
  {\pgf@x-62.02069pt\pgf@y3.19408pt}
  {\pgf@x -77.73447pt\pgf@y -10.41449pt}
}
```

注意其中用花括号包裹子输入路径。

40 执行样式 /pgf/every decoration 保存的选项。

43 如果 \ifpgf@decorate@inputsegmentobjects@reverse 的真值是 true, 则将路径信息列表反转，结果仍保存在宏 \pgf@decorate@inputsegmentobjects 中。

54-55 拆分列表 \pgf@decorate@currentinputsegmentobjects 并做计算。

59-68 用一个循环执行装饰操作，循环变量 \pgf@temp 的取值列表是环境参数。循环体的主要部分是执行装饰操作的命令：

```
\pgf@decorate@invoke{⟨decoration name⟩}{⟨distance⟩}{⟨before code⟩}{⟨after code⟩}
```

这个命令的参数就是环境参数中的列表项。

71-74 全局保存某些变量。

77-78 将上一步全局保存的变量转为“公共”版本。

24.2.2.2 环境相关的命令

\pgf@decorate@parsesoftpath\⟨softpath⟩\⟨a macro name⟩

本命令的参数 \⟨softpath⟩ 是保存软路径的宏，其中的软路径必须是“单纯的”软路径 (不包含特殊记

号，如圆角记号，否则会导致错误)；参数 \⟨*a macro name*⟩ 是一个宏形式，不需要提前定义。
本命令对软路径 \⟨*softpath*⟩ 做分析，将软路径中的记号、花括号逐个吃掉，获取软路径的信息 (构造片段、构造点、片段的长度)，将这些信息做成一个花括号列表，保存在 \⟨*a macro name*⟩ 中。

- 对 \⟨*softpath*⟩ 中的记号、花括号包裹的尺寸按次序分析，例如，如果读取了

```
\pgfsyssoftpath@movetotoken{⟨dimension x⟩}{⟨dimension y⟩}
```

就执行 \pgf@decorate@parsemoveto{⟨*dimension x*⟩}{⟨*dimension y*⟩}，诸如此类。

– 如果执行 \pgf@decorate@parsemoveto{⟨*dimension x*⟩}{⟨*dimension y*⟩}，导致
  * 将子输入路径

```
{\pgf@decorate@inputsegmentobject@moveto{\pgf@x ⟨dimension x⟩ \pgf@y
↪  ⟨dimension y⟩}}
```

  添加到宏 \pgf@decorate@inputsegmentobjects 中，注意其中使用花括号包裹所有记号。

– 如果执行 \pgf@decorate@parselineto{⟨*dimension x*⟩}{⟨*dimension y*⟩}，导致
  * 计算这个 line-to 线段的长度，
  * 将这个线段长度加到被解析的软路径的总长度中，
  * 将如下的子输入路径

```
{\pgf@decorate@inputsegmentobject@lineto
  {⟨ 线段长度，单位 pt⟩}
  {⟨ 代表线段起点的 \pgf@x, \pgf@y 的赋值代码⟩}
  {⟨ 代表线段终点的 \pgf@x, \pgf@y 的赋值代码⟩}
}
```

  添加到宏 \pgf@decorate@inputsegmentobjects 中，注意其中使用花括号包裹所有记号。

– 如果执行 \pgf@decorate@parsecurveto...，导致
  * 计算这个 curve-to 曲线的长度，
  * 将这个曲线长度加到被解析的软路径的总长度中，
  * 将如下的子输入路径

```
{\pgf@decorate@inputsegmentobject@curveto
  {⟨ 曲线长度，单位 pt⟩}
  {⟨ 代表曲线起点的 \pgf@x, \pgf@y 的赋值代码⟩}
  {⟨ 代表曲线第一控制点的 \pgf@x, \pgf@y 的赋值代码⟩}
  {⟨ 代表曲线第二控制点的 \pgf@x, \pgf@y 的赋值代码⟩}
  {⟨ 代表曲线终点的 \pgf@x, \pgf@y 的赋值代码⟩}
}
```

  添加到宏 \pgf@decorate@inputsegmentobjects 中，注意其中使用花括号包裹所有记号。

– 如果执行 \pgf@decorate@parseclosepath{⟨*dimension x*⟩}{⟨*dimension y*⟩}，导致
  * 计算这个 line-to 线段的长度，
  * 将这个线段长度加到被解析的软路径的总长度中，
  * 将如下的子输入路径

```
{\pgf@decorate@inputsegmentobject@closepath
  {⟨ 线段长度，单位 pt⟩}
  {⟨ 代表线段起点的 \pgf@x, \pgf@y 的赋值代码⟩}
  {⟨ 代表线段终点的 \pgf@x, \pgf@y 的赋值代码⟩}
}
```

添加到宏 \pgf@decorate@inputsegmentobjects 中，注意其中使用花括号包裹所有记号。

- 如果执行 \pgf@decorate@parserect..., 这转换为前述几种情况。
- 把 \⟨*softpath*⟩ 全部解析完毕后，将

```
{\pgf@decorate@inputsegmentobject@endofinputsegments}
{\pgf@decorate@inputsegmentobject@endofinputsegments}
```

添加到宏 \pgf@decorate@inputsegmentobjects 中，注意其中使用花括号包裹所有记号。

- 在创建第一个子输入路径后，第一个子输入路径会被执行一次，能定义
  - \pgfdecoratedinputsegmentlength, 子输入路径的长度
  - 构造点的坐标
    - \pgf@decorate@inputsegment@first, 子输入路径的起点坐标
    - \pgf@decorate@inputsegment@supporta, 子输入路径的第 1 支持点坐标，
    - \pgf@decorate@inputsegment@supportb, 子输入路径的第 2 支持点坐标，
    - \pgf@decorate@inputsegment@last, 子输入路径的终点坐标

    moveto 类型的子输入路径的这 4 个构造点重合；lineto 类型的子输入路径的这 4 个构造点是线段的起点、三等分点、终点。
  - \pgf@decorate@movealonginputsegment, 此命令用于计算子输入路径上的一个点坐标
    - \pgf@decorate@inputsegmentobject@lineto 会定义

```
\let\pgf@decorate@movealonginputsegment
↪  \pgf@decorate@movealonginputsegment@line
```

    - \pgf@decorate@inputsegmentobject@curveto 会定义

```
\let\pgf@decorate@movealonginputsegment
↪  \pgf@decorate@movealonginputsegment@curve
```

  - \pgf@decorate@transformtoinputsegment, 这个命令用于对片段做变换
    - \pgf@decorate@inputsegmentobject@lineto 会定义

```
\let\pgf@decorate@transformtoinputsegment
↪  \pgf@decorate@transformtoinputsegment@line
```

    - \pgf@decorate@inputsegmentobject@curveto 会定义

```
\let\pgf@decorate@transformtoinputsegment
↪  \pgf@decorate@transformtoinputsegment@curve
```

  - \pgfdecorationcurrentinputsegment, 子输入路径的类型,可能是 moveto, lineto, curveto, closepath, last.
  - 真值 \pgf@decorate@is@closepath@true, 仅当子输入路径是 closepath 类型时，会有这个真值。
- \⟨*softpath*⟩ 中的第一个坐标被保存到宏 \pgfpoint@decorated@firstparsed 中 (为 \pgf@x 和 \pgf@y 赋值的代码)。
- \⟨*softpath*⟩ 中的最后一个坐标被保存到宏 \pgfpoint@decorate@lastparsed 中 (为 \pgf@x 和 \pgf@y 赋值的代码)。
- \pgfpoint@decorate@lastnonmovetoparsed 等于 \pgfpoint@decorate@lastparsed.
- 宏 \pgf@decorate@inputsegmentobjects 保存一个列表 (列表项是花括号包裹的记号)，其内容例如

```
{\pgf@decorate@inputsegmentobject@moveto{\pgf@x 0.0pt\pgf@y0.0pt}}
{\pgf@decorate@inputsegmentobject@curveto{81.06355pt}
  {\pgf@x 0.0pt\pgf@y 0.0pt}
  {\pgf@x -27.21713pt\pgf@y 7.85689pt}
  {\pgf@x-62.02069pt\pgf@y3.19408pt}
  {\pgf@x -77.73447pt\pgf@y -10.41449pt}
}
```

- 令宏 \⟨*a macro name*⟩ 等于 \pgf@decorate@inputsegmentobjects.
- 将输入路径的总长度保存在宏 \pgf@decorate@totalpathlength 中。
- 计算线段长度的命令是 \pgf@decorate@linelength;
- 计算控制曲线长度的命令是 \pgf@decorate@curvelength.

```
软路径: macro:->\pgfsyssoftpath@movetotoken {0.0pt}{0.0pt}\pgfsyssoftpath@linetotoken
{28.45274pt}{28.45274pt}\pgfsyssoftpath@curvetosupportatoken
{28.45274pt}{44.16696pt}\pgfsyssoftpath@curvetosupportbtoken
{15.71422pt}{56.90549pt}\pgfsyssoftpath@curvetotoken
{0.0pt}{56.90549pt}\pgfsyssoftpath@closepathtoken {0.0pt}{0.0pt}
输入路径列表: macro:->{\pgf@decorate@inputsegmentobject@moveto
{\pgf@x 0.0pt\pgf@y 0.0pt}}{\pgf@decorate@inputsegmentobject@lineto
{40.20877pt}{\pgf@x 0.0pt\pgf@y 0.0pt}{\pgf@x 28.45274pt\pgf@y
28.45274pt}}{\pgf@decorate@inputsegmentobject@curveto {44.69781pt}{\pgf@x
28.45274pt\pgf@y 28.45274pt}{\pgf@x 28.45274pt\pgf@y 44.16696pt}{\pgf@x 15.71422pt\pgf@y
56.90549pt}{\pgf@x 0.0pt\pgf@y 56.90549pt}}{\pgf@decorate@inputsegmentobject@closepath
{56.90549pt}{\pgf@x 0.0pt\pgf@y 56.90549pt}{\pgf@x 0.0pt\pgf@y
0.0pt}}{\pgf@decorate@inputsegmentobject@endofinputsegments
}{\pgf@decorate@inputsegmentobject@endofinputsegments }
```

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpatharc{0}{90}{1cm}
  \pgfpathclose
  \pgfresetboundingbox
  \pgfgetpath{\aaaa}
  \expandafter\gdef\expandafter\aaaa\expandafter{\aaaa}
\end{pgfpicture}
{\ttfamily
软路径: \meaning\aaaa\par
\makeatletter%
\pgf@decorate@parsesoftpath\aaaa\bbbb%
\makeatother%
输入路径列表: \meaning\bbbb
}
```

\pgf@decorate@linelength{⟨*point A*⟩}{⟨*point B*⟩}

本命令的参数是 PGF 点，或能为 \pgf@x, \pgf@y 赋值的代码。

本命令利用 \pgfmathveclen@ 计算两点间的线段长度，结果保存在 \pgfmathresult.

\pgf@decorate@curvelength{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}

这里只需要参数 ⟨P_1⟩, ⟨P_2⟩, ⟨P_3⟩, ⟨P_4⟩ 是能为 \pgf@x, \pgf@y 赋值的代码即可。

假设点 P_1, P_2, P_3, P_4 是曲线 $C(t)(t \in [0,1])$ 的 4 个控制点，本命令近似计算 $C(t)$ 的长度，结果保存在 \pgfmathresult.

本命令执行一个迭代。本命令的计算用到一个阈值：

\pgf@decorate@curvelength@tolerance

这个宏的初始值是 1pt

```
\def\pgf@decorate@curvelength@tolerance{1pt}%
```

\pgf@decorate@curvelength 的处理是：

1. 将 $C(t)$ 在 $t=0.5$ 处分成左右两部分，左侧部分——记为 $CL(t)(t\in[0,1])$——的控制点是：

$$L_1=P_1, L_2=0.5(P_1+P_2), L_3=0.25(P_1+2P_2+P_3), L_4=0.125(P_1+3P_2+3P_3+P_4),$$

右侧部分——记为 $CR(t)(t\in[0,1])$——的控制点是：

$$R_1=0.125(P_1+3P_2+3P_3+P_4), R_2=0.25(P_2+2P_3+P_4), R_3=0.5(P_3+P_4), R_4=P_4,$$

2. 针对 $CL(t)$ 计算弦 $L_4-L_1=(l_x,l_y)$，
   - 如果 $\max(|l_x|,|l_y|)<$ \pgf@decorate@curvelength@tolerance，就把 $|L_4-L_1|$ 作为 $CL(t)$ 的长度；
   - 如果 $\max(|l_x|,|l_y|)\geqslant$ \pgf@decorate@curvelength@tolerance，就令 $CL(t)=C(t)$，从第 1 步开始对 $CL(t)$ 做分割计算；
3. 针对 $CR(t)$ 计算弦 $R_4-R_1=(r_x,r_y)$，
   - 如果 $\max(|r_x|,|r_y|)<$ \pgf@decorate@curvelength@tolerance，就把 $|R_4-R_1|$ 作为 $CR(t)$ 的长度；
   - 如果 $\max(|r_x|,|r_y|)\geqslant$ \pgf@decorate@curvelength@tolerance，就令 $CR(t)=C(t)$，从第 1 步开始对 $CR(t)$ 做分割计算；
4. 把那些符合阈值限制的 $CL(t)$ 长度和 $CR(t)$ 长度累加起来，作为 $C(t)$ 的长度。

在本命令的迭代过程中，利用 \expandafter 和 \pgfmath@returnone 避免大量的“\if—\fi”，“\begingroup—\endgroup”相套嵌。

\ifpgf@decorate@inputsegmentobjects@reverse

这个 TEX-if 的真值由选项 /pgf/decoration/reverse path 决定。

\pgf@decorate@inputsegmentobjects@reverse\⟨*input decorated path*⟩\⟨*reversed decorated path*⟩

参数 \⟨*input decorated path*⟩ 是保存“输入路径列表”的宏，本命令将这个路径列表做反向处理，结果保存到宏 \⟨*reversed decorated path*⟩ 中。

\pgf@decorate@getnextinputsegmentobject\⟨*a macro name*⟩

参数 \⟨*a macro name*⟩ 是宏形式的记号。

本命令拆分当前的“输入路径列表”\pgf@decorate@currentinputsegmentobjects:

- 当前列表的第 1 项的内容 (花括号里的内容) 保存到 \⟨*a macro name*⟩;
- 列表的其他剩余的项仍然保存到宏 \pgf@decorate@currentinputsegmentobjects; 如果只是剩余 1 项，则将这一项的内容 (花括号里的内容) 保存到这个宏。

\pgf@decorate@processnextinputsegmentobject

这个命令配合 \pgf@decorate@getnextinputsegmentobject 使用。

本命令会保存当前、之后的子输入路径，并对当前、之后的子输入路径做分析。

1. 保存之前的子输入路径的类型

```
\let\pgfdecorationpreviousinputsegment\pgfdecoratecurrentinputsegment%
```

2. 保存当前、之后的子输入路径：

```
\let\pgf@decorate@currentinputsegmentobject
↪  \pgf@decorate@nextinputsegmentobject%
\pgf@decorate@getnextinputsegmentobject\pgf@decorate@nextinputsegmentobject%
```

3. 设置真值 \pgf@decorate@is@closepath@false
4. 执行 \pgf@decorate@currentinputsegmentobject, 能定义
   - \pgfdecoratedinputsegmentlength, 子输入路径的长度
   - 构造点的坐标
     - \pgf@decorate@inputsegment@first, 子输入路径的起点坐标
     - \pgf@decorate@inputsegment@supporta, 子输入路径的第 1 支持点坐标，
     - \pgf@decorate@inputsegment@supportb, 子输入路径的第 2 支持点坐标，
     - \pgf@decorate@inputsegment@last, 子输入路径的终点坐标

     moveto 类型的子输入路径的这 4 个构造点重合；lineto 类型的子输入路径的这 4 个构造点是线段的起点、三等分点、终点。
   - \pgf@decorate@movealonginputsegment, 此命令用于计算子输入路径上的一个点坐标
     - \pgf@decorate@inputsegmentobject@lineto 会定义

       ```
       \let\pgf@decorate@movealonginputsegment
       ↪  \pgf@decorate@movealonginputsegment@line
       ```

     - \pgf@decorate@inputsegmentobject@curveto 会定义

       ```
       \let\pgf@decorate@movealonginputsegment
       ↪  \pgf@decorate@movealonginputsegment@curve
       ```

   - \pgf@decorate@transformtoinputsegment, 这个命令用于对片段做变换
     - \pgf@decorate@inputsegmentobject@lineto 会定义

       ```
       \let\pgf@decorate@transformtoinputsegment
       ↪  \pgf@decorate@transformtoinputsegment@line
       ```

     - \pgf@decorate@inputsegmentobject@curveto 会定义

       ```
       \let\pgf@decorate@transformtoinputsegment
       ↪  \pgf@decorate@transformtoinputsegment@curve
       ```

   - \pgfdecorationcurrentinputsegment, 子输入路径的类型,可能是 moveto, lineto, curveto, closepath, last.
   - 真值 \pgf@decorate@is@closepath@true, 仅当子输入路径是 closepath 类型时，会有这个真值。
5. 检查 \pgfdecorationcurrentinputsegment 的当前值 (当前子输入路径的类型) 是否 move-to:

```
\ifx\pgfdecorationcurrentinputsegment\pgfdecorationinputsegmentmoveto%
  \pgfpathmoveto{\pgf@decorate@inputsegment@first}%
  \let\pgfdecorationpreviousinputsegment\pgfdecorationcurrentinputsegment%
  \let\pgf@decorate@currentinputsegmentobject
  ↪  \pgf@decorate@nextinputsegmentobject%
  \pgf@decorate@getnextinputsegmentobject\pgf@decorate@nextinputsegmentobject%
  \pgf@decorate@is@closepath@false%
  \pgf@decorate@currentinputsegmentobject%
\fi%
```

6. 检查当前的子输入路径的长度：

```
\ifdim\pgfdecoratedinputsegmentlength=0pt\relax%
  \def\pgfdecoratedinputsegmentlength{0.05pt}%
\fi%
```

7. 准备寄存器 \pgfdecoratedinputsegmentremainingdistance, 代表当前子输入路径的未装饰部分的长度

```
\pgfdecoratedinputsegmentremainingdistance\pgfdecoratedinputsegmentlength
↪  \relax%
```

8. 准备寄存器 \pgfdecoratedinputsegmentcompleteddistance, 代表当前子输入路径的已装饰部分的长度

```
\pgfdecoratedinputsegmentcompleteddistance0pt\relax%
\def\pgf@decorate@inputsegmenttime{0}%
```

9. 准备宏 \pgf@decorate@inputsegmenttime, 代表沿着当前子输入路径行进的某个时刻

```
\def\pgf@decorate@inputsegmenttime{0}%
```

10. 检查当前的子输入路径的类型是否 curve-to 或者其他类型 (lineto, moveto, closepath, last)，计算
    - \pgfdecoratedangle, 保存曲线 (线段，或其他) 在起点处的切方向角度
    - \pgfdecoratedinputsegmentstartangle, 保存曲线 (线段，或其他) 在起点处的切方向角度
    - \pgfdecoratedinputsegmentendangle, 保存曲线 (线段，或其他) 在终点处的切方向角度
11. 在一个组内：
    (a) 执行子输入路径 \pgf@decorate@nextinputsegmentobject, 这会定义一些变量、命令。
    (b) 全局保存当前子输入路径的类型

    ```
    \global\let\pgf@decorate@temp\pgfdecorationcurrentinputsegment
    ```

    (c) 检查 \pgfdecorationcurrentinputsegment 的当前值 (当前子输入路径的类型)，如果是 moveto, 则重新执行 \pgf@decorate@currentinputsegmentobject.
    (d) 再次检查 \pgfdecorationcurrentinputsegment 的当前值，计算曲线 (线段，或其他) 在起点处的切方向角度，保存到 \pgfmathresult.
    然后定义宏 \pgfdecoratedangletonextinputsegment 的值是
    \pgfmathresult − \pgfdecoratedangle.
    然后将宏 \pgfdecoratedangletonextinputsegment 的定义推到组外。
12. 定义宏 \pgfdecorationnextinputsegmentobject, 保存之后的子输入路径的类型

```
\let\pgfdecorationnextinputsegmentobject\pgf@decorate@temp
```

\pgf@decorate@movealonginputsegment{⟨*a dimension*⟩}

这个命令被 let 为其他命令：

- \pgf@decorate@inputsegmentobject@lineto 会定义

```
\let\pgf@decorate@movealonginputsegment
↪  \pgf@decorate@movealonginputsegment@line
```

- \pgf@decorate@inputsegmentobject@curveto 会定义

```
\let\pgf@decorate@movealonginputsegment
↪  \pgf@decorate@movealonginputsegment@curve
```

- \pgf@decorate@inputsegmentobject@closepath 会定义

```
\let\pgf@decorate@movealonginputsegment
↪  \pgf@decorate@movealonginputsegment@line
```

\pgf@decorate@movealonginputsegment@line{⟨*a distance*⟩}

本命令的定义是：

```
\def\pgf@decorate@movealonginputsegment@line#1{}%
```

\pgf@decorate@movealonginputsegment@curve{⟨*a distance*⟩}

假设当前的子输入路径是曲线，本命令计算：从曲线起点开始，沿着曲线行进距离 ⟨*a distance*⟩ 所到达的点，以及曲线在该点处的时刻、切向量。

本命令先用 \ifdim 检查 ⟨*a distance*⟩ 是否 0pt，如果是就什么也不做；如果不是，则执行以下步骤：记 \pgf@decorate@inputsegmenttime 的值 (代表一个时刻) 为 t_0；当前子输入路径是曲线 $C(t)$，它的长度是 l_c，

1. 计算曲线在 t_0 时刻的点

```
\pgfpointcurveattime{\pgf@decorate@inputsegmenttime}%
  {\pgf@decorate@inputsegment@first}{\pgf@decorate@inputsegment@supporta}%
  {\pgf@decorate@inputsegment@supportb}{\pgf@decorate@inputsegment@last}%
```

这会计算点 $C(t_0)$ 的坐标，以及曲线在该点的切方向。

有公式：

$$\begin{aligned} C(t) &= (1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3, \\ A(t) &= (1-t)^2P_1 + 2t(1-t)P_2 + t^2P_3, \\ B(t) &= (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2, \\ \frac{1}{3}C'(t) &= A(t) - B(t). \end{aligned}$$

本命令保存计算结果：

- 宏 \pgf@time@s 保存 t_0
- 宏 \pgf@time@t 保存 $1 - t_0$
- 寄存器 (\pgf@x, \pgf@y) 全局地保存 $C(t_0)$
- 寄存器 (\pgf@xa, \pgf@ya) 保存 $A(t_0)$
- 寄存器 (\pgf@xb, \pgf@yb) 保存 $B(t_0)$
- 寄存器 (\pgf@xc, \pgf@yc) 保存 P_3

2. 设置一个变量

$$t_v = \begin{cases} 0.03125, \text{ if } 0\text{pt} < l_c < 128\text{pt}; \\ 0.015625, \text{ if } 128\text{pt} \leqslant l_c < 512\text{pt}; \\ 0.00390625, \text{ if } 512\text{pt} \leqslant l_c < 2048\text{pt}; \\ 0.0009765625, \text{ if } 2048\text{pt} \leqslant l_c; \end{cases}$$

3. 设置 4 个初值 $d_a = 0$, $c_a = 1$, $t_a = t_0$, $C_a = C(t_0)$, 执行一个循环
   (a) $t_a = t_a + c_a \cdot t_v$
   (b) 计算曲线在 t_a 时刻的点 $C(t_a)$

```
\pgfpoint@decorate@curveattime{\pgfmath@tonumber{\pgf@xb}}%
  {\pgf@decorate@inputsegment@first}{\pgf@decorate@inputsegment@supporta}%
  {\pgf@decorate@inputsegment@supportb}{\pgf@decorate@inputsegment@last}%
```

   (c) 记 $l_a = \|C(t_a) - C_a\|$, $d_a = d_a + c_a \cdot l_{c_a}$, $C_a = C(t_a)$;

(d) 检查 c_a 的值，
- 如果 $c_a > 0$, 并且 $d_a\text{pt} > \langle a\ distance\rangle$, 则令 $c_a = -c_a$, $t_v = 0.5t_v$;
- 如果 $c_a \leqslant 0$, 并且 $d_a\text{pt} < \langle a\ distance\rangle$, 则令 $c_a = -c_a$, $t_v = 0.5t_v$;

(e) 检查 t_v 的值，如果 $t_v = 0$ 则结束循环；否则从头循环。

4. 保存时刻 t_a

```
\edef\pgf@decorate@inputsegmenttime{\pgfmath@tonumber{\pgf@xb}}
```

5. 计算曲线在 t_a 时刻的点以及切向量

```
\pgfpointcurveattime{\pgf@decorate@inputsegmenttime}%
  {\pgf@decorate@inputsegment@first}{\pgf@decorate@inputsegment@supporta}%
  {\pgf@decorate@inputsegment@supportb}{\pgf@decorate@inputsegment@last}%
```

并把切向量保存到 \pgfdecoratedangle.

\pgf@decorate@movealongpath{⟨*a distance*⟩}

参数 ⟨*a dimension*⟩ 会被 \pgfmathsetlength 处理。

本命令的处理是：

1. 用 \pgfmathsetlength 处理 ⟨*a dimension*⟩

```
\pgfmathsetlength\pgf@decorate@distancetomove{⟨a dimension⟩}
```

2. 修改寄存器

```
\advance\pgfdecoratedcompleteddistance\pgf@decorate@distancetomove%
\advance\pgfdecoratedremainingdistance-\pgf@decorate@distancetomove%
```

3. 执行 \pgf@decorate@@movealongpath

\pgf@decorate@@movealongpath

本命令的处理是：

1. 修改寄存器

```
\advance\pgfdecoratedinputsegmentcompleteddistance
↪  \pgf@decorate@distancetomove%
\advance\pgfdecoratedinputsegmentremainingdistance-
↪  \pgf@decorate@distancetomove%
```

2. 检查 \pgfdecoratedinputsegmentremainingdistance 的值
   - 如果它的值 > 0pt, 则执行 \pgf@decorate@@@movealongpath
   - 如果它的值 $\leqslant 0$pt, 则
     (a) 重设寄存器

```
\pgf@decorate@distancetomove-
↪  \pgfdecoratedinputsegmentremainingdistance%
```

     (b) 处理当前、之后的子输入路径

```
\pgf@decorate@processnextinputsegmentobject%
```

     注意，\pgf@decorate@currentinputsegmentobjects 中的最后两个列表项是

```
{\pgf@decorate@inputsegmentobject@endofinputsegments}
{\pgf@decorate@inputsegmentobject@endofinputsegments}
```

     (c) 再检查 \pgf@decorate@currentinputsegmentobjects 的值是否为空。
     – 如果是空的，设置寄存器值 \pgfdecoratedremainingdistance

```
\pgfdecoratedremainingdistance0pt\relax
```

然后结束本命令

– 如果不是空的，就执行 \pgf@decorate@@movealongpath

\pgf@decorate@@@movealongpath

本命令的定义是：

```
\def\pgf@decorate@@@movealongpath{%
  \pgf@decorate@movealonginputsegment{\the\pgf@decorate@distancetomove}%
  \pgf@decorate@distancetomove0pt\relax%
  %
  % Grrr. Hacking to control some inaccuracies.
  %
  \ifdim\pgf@decorate@inputsegmenttime pt>1pt\relax%
    \let\pgf@decorate@inputsegmenttimetemp\pgf@decorate@inputsegmenttime%
    \pgf@decorate@processnextinputsegmentobject%
    \pgf@x\pgf@decorate@inputsegmenttimetemp pt\relax%
    \advance\pgf@x-1pt\relax%
    \edef\pgf@decorate@inputsegmenttime{\pgfmath@tonumber{\pgf@x}}%
    \ifx\pgf@decorate@currentinputsegmentobjects\pgfutil@empty%
      \pgfdecoratedremainingdistance0pt\relax%
    \fi%
  \fi%
}%
```

\pgf@decorate@transformtoinputsegment

这个命令用于对片段做变换

- \pgf@decorate@inputsegmentobject@lineto 会定义

```
\let\pgf@decorate@transformtoinputsegment
↪  \pgf@decorate@transformtoinputsegment@line
```

\pgf@decorate@transformtoinputsegment@line

本命令做平移、旋转，其定义是：

```
\def\pgf@decorate@transformtoinputsegment@line{%
  \pgftransformshift{%
    \pgfpointlineatdistance{\pgfdecoratedinputsegmentcompleteddistance}{
    ↪  \pgf@decorate@inputsegment@first}{\pgf@decorate@inputsegment@last}
  }%
  \pgftransformrotate{\pgfdecoratedangle}%
}%
```

可见使用本命令前需要定义：

– 距离 \pgfdecoratedinputsegmentcompleteddistance,

* \pgf@decorate@processnextinputsegmentobject 规定这个寄存器的初值为 0pt;
* \pgf@decorate@@movealongpath 修改这个寄存器值。

在一个状态中，先执行 \pgf@decorate@transformtoinputsegment, 再执行 \pgf@decorate@@movealongpath 计算这个寄存器值，所以这是为下一个状态做准备的计算，而不是为当前状态准备的。

– 角度 \pgfdecoratedangle, 计算这个角度的是

* \pgf@decorate@processnextinputsegmentobject
* \pgf@decorate@movealonginputsegment@curve

- \pgf@decorate@inputsegmentobject@curveto 会定义

```
\let\pgf@decorate@transformtoinputsegment
↪  \pgf@decorate@transformtoinputsegment@curve
```

\pgf@decorate@transformtoinputsegment@curve

本命令做平移、旋转，其定义是：

```
\def\pgf@decorate@transformtoinputsegment@curve{%
  \pgfslopedattimetrue%
  \pgfallowupsidedownattimetrue%
  \pgftransformcurveattime{\pgf@decorate@inputsegmenttime}%
    {\pgf@decorate@inputsegment@first}{\pgf@decorate@inputsegment@supporta}
    ↪  %
    {\pgf@decorate@inputsegment@supportb}{\pgf@decorate@inputsegment@last}
    ↪  %
}%
```

可见使用本命令前需要定义时刻 \pgf@decorate@inputsegmenttime.

- \pgf@decorate@processnextinputsegmentobject 定义这个时刻的初值为 0;
- 对于子输入路径是曲线的情况，\pgf@decorate@@@movealongpath 调用

  \pgf@decorate@movealonginputsegment@curve

  计算这个时刻，并确保这个时刻 ⩽ 1.

\pgf@decorate@for\⟨*temp var*⟩=⟨*comma list*⟩\do{⟨*code*⟩}

\⟨*temp var*⟩ 是宏形式的记号，代表循环变量。⟨*comma list*⟩ 是用逗号分隔的列表，是循环变量的取值列表。⟨*code*⟩ 是循环代码，应当能处理 \⟨*temp var*⟩ 的展开值，即 ⟨*comma list*⟩ 中的列表项。

a,b. Hello,you.

```
\def\showlistitem#1#2{#1,#2.\space}
\makeatletter
\pgf@decorate@for\tempforvar:=ab,{Hello}{you}\do{
  \expandafter\showlistitem\tempforvar
}
\makeatother
```

\pgf@decorate@invoke{⟨*decoration name*⟩}{⟨*distance*⟩}{⟨*before code*⟩}{⟨*after code*⟩}

参数 ⟨*decoration name*⟩ 是装饰类型的名称。

装饰类型 ⟨*decoration name*⟩ 占用的输入路径的长度是 ⟨*distance*⟩.

⟨*distance*⟩ 会被 \pgfmathsetlength 处理。

在开始 ⟨*decoration name*⟩ 的装饰前执行 ⟨*before code*⟩.

在结束 ⟨*decoration name*⟩ 的装饰后执行 ⟨*after code*⟩.

参数 ⟨*before code*⟩ 或者 ⟨*after code*⟩ 可以省略 (如果省略就当成是空的)，例如

```
\pgf@decorate@invoke{⟨decoration name⟩}{⟨distance⟩}{⟨ 空的⟩}{⟨after code⟩}
\pgf@decorate@invoke{⟨decoration name⟩}{⟨distance⟩}{⟨before code⟩}
\pgf@decorate@invoke{⟨decoration name⟩}{⟨distance⟩}
```

参数 ⟨*decoration name*⟩, ⟨*distance*⟩ 不能省略。

本命令的处理是：

1. 检查控制序列 pgf@decorate@@⟨*decoration name*⟩@initial 是否已定义 (不等于 \relax)，即检查 ⟨*decoration name*⟩ 是否已定义。如果未定义就报错，如果已定义就继续下面的步骤。
2. 检查 ⟨*before code*⟩ 或者 ⟨*after code*⟩ 是否为空，如果是空的，就看作是 \pgfutil@empty.
3. 准备一个寄存器，一个宏：

```
\pgfdecoratedremainingdistance\pgf@decorated@remainingdistance\relax%
\let\pgfdecoratedpathlength\pgf@decorate@totalpathlength%
```

如果在 ⟨*distance*⟩ 中使用了这两个变量，那么这就是它们的值。

4. 处理 ⟨*distance*⟩, 用 l_r 记宏 \pgf@decorated@remainingdistance 的值 (当前尚未被装饰的路径长度)：

```
\pgfmathsetlength\pgf@xa{#2}%
\ifdim\pgf@xa>\pgf@decorated@remainingdistance\relax%
  \pgf@xa\pgf@decorated@remainingdistance\relax%
\fi%
```

如果 ⟨*distance*⟩ 大于当前的尚未被装饰路径长度 l_r，就令 ⟨*distance*⟩ 等于 l_r.

5. 定义宏 \pgf@decorate@currentpathlength 保存此时的 ⟨*distance*⟩.
6. 重定义 2 个宏、2 个寄存器：
   - 宏 \pgf@decorated@remainingdistance: $l_r = l_r - \langle distance \rangle$.
   - 宏 \pgfdecoratedpathlength 保存尺寸 ⟨*distance*⟩.
   - 寄存器 \pgfdecoratedremainingdistance=⟨*distance*⟩.
   - 寄存器 \pgfdecoratedcompleteddistance= 0pt.

   如果在 ⟨*before code*⟩ 中使用了这 2 个宏、2 个寄存器，那么这就是它们的值。
7. 执行 ⟨*before code*⟩
8. 令 \pgf@decorate@current@state 保存 ⟨*decoration name*⟩ 的初始状态的名称，即控制序列 pgf@decorate@@⟨*decoration name*⟩@initial 保存的内容。
9. 执行 \pgf@decorate@run.

\pgf@decorate@run

本命令的处理是：如果 \pgf@decorate@current@state 保存的是 final (终止状态的名称)，那么什么也不做，否则执行 \pgf@decorate@do@state.

\pgf@decorate@do@state

本命令会执行 ⟨*decoration name*⟩ 的、除了 final 的状态来做装饰。

本命令的处理如下：

(a) 准备：

```
\let\pgf@decorate@next\relax%
\let\pgf@decorate@next@state\pgf@decorate@current@state%
\let\pgf@decorate@persistent@pre=\relax%
\let\pgf@decorate@persistent@post=\relax%
```

(b) 执行当前状态 \pgf@decorate@current@state 的选项

```
\pgfqkeys{/pgf/decoration automaton}{⟨当前状态的选项⟩}
```

这些选项的路径都是 /pgf/decoration automaton. 以上选项可能重定义以下宏：

- \pgf@decorate@next, 此时这个宏可能等于 \relax, \pgf@decorate@do@code, \pgf@decorate@run.
- \pgf@decorate@persistent@pre
- \pgf@decorate@persistent@post
- \pgf@decorate@next@state

(c) 检查以上选项是否重定义了 \pgf@decorate@next, 即检查它是否等于 \relax,

- 如果不等于 \relax, 那就执行 \pgf@decorate@next.
- 如果等于 \relax, 那就执行 \pgf@decorate@do@code.

\pgf@decorate@do@code

本命令的处理是：

(a) 执行 \pgf@decorate@persistent@pre

(b) 设置一个花括号组，在这个组内执行绘制片段的代码。

i. 用 { 开启一个组，

ii. 设置变换矩阵并执行当前状态的绘图代码

```
\pgftransformreset%
\pgf@decorate@transformtoinputsegment%
\pgf@decorate@additionaltransform%
\csname pgf@decorate@@\pgf@decorate@name @
↪  \pgf@decorate@current@state @code\endcsname%
```

iii. 用 } 结束组

(c) 执行 \pgf@decorate@persistent@post

(d) 执行 \pgf@decorate@movealongpath→P. 421{\pgf@decorate@width}

(e) 检查是否需要重复执行当前状态，即检查计数器 \pgf@decorate@repeatstate 的值 (它的初始值是 −1)：

```
\ifnum\pgf@decorate@repeatstate>0\relax%
  \advance\pgf@decorate@repeatstate-1\relax%
\else%
  \pgf@decorate@repeatstate-1\relax%
  \let\pgf@decorate@current@state\pgf@decorate@next@state%
\fi%
```

(f) 循环重复 \pgf@decorate@run

10. 检查宏 \pgf@decorated@remainingdistance 的当前值 l_r,
    - 如果此时 $l_r < 1\text{pt}$, 就认为到达了输入路径的终点，转存终点坐标：

      ```
      \let\pgfpoint@decorated@pathlast\pgfpoint@decorated@totalpathlast
      ```

    - 如果此时 $l_r \geqslant 1\text{pt}$, 计算终点坐标 \pgfpoint@decorated@pathlast:

      ```
      {%
        \pgf@decorate@movealonginputsegment{\the\pgfdecoratedremainingdistance}%
        \pgf@decorate@transformtoinputsegment%
        \pgfpointorigin%
        \pgf@pos@transform@glob%
        \global\pgf@x\pgf@x%
        \global\pgf@y\pgf@y%
      }%
      \edef\pgfpoint@decorated@pathlast{\pgf@x\the\pgf@x\pgf@y\the\pgf@y}%
      ```

11. 在一个组内执行 final 状态的代码

    ```
    {%
      \pgftransformreset%
      \pgf@decorate@transformtoinputsegment%
      \pgf@decorate@additionaltransform%
      \csname pgf@decorate@@\pgf@decorate@name @final@code\endcsname%
    }%
    ```

12. ```
    \pgf@decorate@movealonginputsegment{\the\pgfdecoratedremainingdistance}
    ```

13. 执行 ⟨*after code*⟩

14. 准备下一个循环
    ```

```
\let\pgf@decorate@additionaltransform\pgfutil@empty%
\let\pgf@decorate@inputsegmentobjects
↪  \pgf@decorate@inputsegmentobjects@aftersplit%
\let\pgfpoint@decorated@pathfirst\pgfpoint@decorated@pathlast%
```

\pgf@decorate@installmacrosatend

本命令的定义是：

```
\def\pgf@decorate@installmacrosatend{%
  \let\pgfdecorationpath\pgf@decorate@decorationpathtemp%
  \let\pgfdecoratedpath\pgf@decorate@decoratedpathtemp%
  \let\pgfdecorateexistingpath\pgf@decorate@existingpathtemp%
  \let\pgfpoint@decorated@pathlast\pgfpoint@decorated@pathlasttemp%
}%
```

{pgfdecoration} 环境结束后，本命令定义的这几个宏可用。

/pgf/every decoration={⟨*options*⟩} (style)

可以把这个键定义成样式：

```
\pgfkeys{
  /pgf/every decoration/.style={⟨ 某某⟩}
}
注意，⟨options⟩ 中应当使用完整的键
```

也可以把这个键定义成 code:

```
\pgfkeys{
  /pgf/every decoration/.code={⟨code⟩}
}
```

在装饰过程中会执行：

```
\pgfkeys{/pgf/every decoration/.try}
```

\pgfsetdecorationsegmenttransformation{⟨*transformation command*⟩}

本命令的定义是：

```
\def\pgfsetdecorationsegmenttransformation#1{\def\pgf@decorate@additionaltransform
↪  {#1}}%
\let\pgf@decorate@additionaltransform\pgfutil@empty%
```

本命令保存的变换作为“额外”的变换，作用于状态创建的片段路径。

24.2.3 声明一个 meta 装饰类型

\pgfdeclaremetadecoration{⟨*meta decoration name*⟩}{⟨*initial state*⟩}{⟨*states*⟩}

这个命令声明一种 meta 装饰类型。⟨*meta decoration name*⟩ 是装饰类型的名称。⟨*initial state*⟩ 是初始状态的名称。⟨*states*⟩ 是一个或数个 \state 命令定义的“状态”。

本命令的处理是：

1. 检查控制序列 pgf@decorate@@⟨*meta decoration name*⟩@initial 是否已定义 (不等于 \relax)，如果已定义则报错；如果未定义，则
2. 定义 \pgf@metadecoration@name 保存 ⟨*meta decoration name*⟩:
3. 定义控制序列 pgf@metadecoration@@⟨*decoration name*⟩@initial 保存 ⟨*initial state*⟩.
4. 交换命令名称

```
\let\pgf@orig@state\state%
\let\state\pgf@metadecoration@state
```

5. 执行 ⟨*states*⟩. 这一步使用命令 `\state` 定义状态，应当至少定义 ⟨*initial state*⟩, `final` 两个状态。
6. `\let\state\pgf@orig@state`

本命令的定义有前缀 `\long`, 所以本命令的参数中可以有 `\par`.

`\pgf@metadecoration@state`{⟨*state name*⟩}[⟨*options*⟩]{⟨*code*⟩}

本命令定义一个名称为 ⟨*state name*⟩ 的状态。[⟨*options*⟩] 是可选的选项。⟨*code*⟩ 是这个状态对应的代码。

本命令定义两个控制序列：

- `\csname pgf@metadecoration@@`⟨*meta decoration name*⟩`@`⟨*state name*⟩`@options\endcsname` 保存 ⟨*options*⟩
- `\csname pgf@metadecoration@@`⟨*meta decoration name*⟩`@`⟨*state name*⟩`@code\endcsname` 保存 ⟨*code*⟩

这样使得 ⟨*state name*⟩ 属于 ⟨*meta decoration name*⟩. 可见这里的 ⟨*code*⟩ 只是被保存，只有在装饰路径的过程中，用到 ⟨*state name*⟩ 时才会执行 ⟨*options*⟩, ⟨*code*⟩.

⟨*code*⟩ 可以是某些计算命令，定义变量的代码，或者能生成软路径的命令。也可以在 ⟨*code*⟩ 中使用：

- `\decoration{`⟨*decoration name*⟩`}` 来指定一个装饰类型
- `\beforedecoration{`⟨*before code*⟩`}` 来指定代码
- `\afterdecoration{`⟨*after code*⟩`}` 来指定代码

在 ⟨*options*⟩ 中应使用选项 `width` 来指定状态 ⟨*state name*⟩ 占据的宽度 (宏 `\pgf@metadecoration@width`)，这个状态会采用装饰类型 ⟨*decoration name*⟩ 来装饰选项 `width` 指定的宽度，实际上会执行

```
\pgf@decorate@invoke{%
    {⟨decoration name⟩}{\pgf@metadecoration@width}%
    {⟨before code⟩}{⟨after code⟩}%
}%
```

24.2.3.1 状态选项

在 ⟨*options*⟩ 中一定要指定切换状态的方式，并且最后能切换到 `final` 状态。因为执行状态的循环过程用到宏 `\pgf@metadecoration@width`, 所以除了 `final` 状态外，其他状态应当使用 `/pgf/meta-decoration automaton/width` 来规定该状态占据的宽度，不过，单次的状态循环不被放在组中，只要第一个状态 (初始状态) 使用了这个选项，就可以为其他状态提供这个宏。所以初始状态必须使用这个选项，其他状态则不必。

`\pgf@metadecoration@switch@if`{⟨*dimension*⟩}`to` ⟨*next state name*⟩`\pgf@stop`

参数 ⟨*dimension*⟩ 应当能被 `\pgfmathsetlength` 处理。

本命令的定义是：

```
\def\pgf@metadecoration@switch@if#1to #2\pgf@stop{%
  \ifx\pgf@metadecoration@next\relax%
    \pgfmathsetlength\pgf@x{#1}%
    \ifdim\pgf@decorated@remainingdistance<\pgf@x%
      \def\pgf@metadecoration@current@state{#2}%
      \let\pgf@metadecoration@next\pgf@metadecoration@run%
    \fi%
  \fi%
}%
```

\pgf@metadecoration@switch@ifinputsegment{⟨*dimension*⟩}to ⟨*next state name*⟩\pgf@stop

参数 ⟨*dimension*⟩ 应当能被 \pgfmathsetlength 处理。

本命令的定义是：

```
\def\pgf@metadecoration@switch@ifinputsegment#1to #2\pgf@stop{%
  \ifx\pgf@metadecoration@next\relax%
    \pgfmathsetlength\pgf@x{#1}%
    \ifdim\pgfdecoratedinputsegmentremainingdistance<\pgf@x%
      \def\pgf@metadecoration@current@state{#2}%
      \let\pgf@metadecoration@next\pgf@metadecoration@run%
    \fi%
  \fi%
}%
```

/pgf/meta-decoration automaton/width=⟨*dimension*⟩

本选项的定义是：

```
\pgfkeys{
  /pgf/meta-decoration automaton/width/.code=\def\pgf@metadecoration@width{#1}
  ↪  \pgf@metadecoration@switch@if#1 to final\pgf@stop,%
}
```

参数 ⟨*dimension*⟩ 应当能被 \pgfmathsetlength 处理。本选项的作用是：

1. 定义宏 \pgf@metadecoration@width 保存参数 ⟨*dimension*⟩.
2. 在 \pgf@metadecoration@next 等于 \relax, 并且寄存器 \pgfdecoratedremainingdistance 的值小于 ⟨*dimension*⟩ 时，定义

```
\def\pgf@metadecoration@current@state{final}%
\let\pgf@metadecoration@next\pgf@metadecoration@run%
```

/pgf/meta-decoration automaton/switch if less than=⟨*dimension*⟩to ⟨*next state name*⟩

本选项的定义是：

```
\pgfkeys{
  /pgf/meta-decoration automaton/switch if less
  ↪  than/.code=\pgf@metadecoration@switch@if#1\pgf@stop,%
}
```

参数 ⟨*dimension*⟩ 应当能被 \pgfmathsetlength 处理。本选项的作用是：在 \pgf@metadecoration@next 等于 \relax, 并且寄存器 \pgfdecoratedremainingdistance 的值小于 ⟨*dimension*⟩ 时，定义

```
\def\pgf@metadecoration@current@state{⟨next state name⟩}%
\let\pgf@metadecoration@next\pgf@metadecoration@run%
```

/pgf/meta-decoration automaton/switch if input segment less than=⟨*dimension*⟩to ⟨*next state name*⟩

本选项的定义是：

```
\pgfkeys{
  /pgf/meta-decoration automaton/switch if input segment less
  ↪  than/.code=\pgf@metadecoration@switch@ifinputsegment#1\pgf@stop,%
}
```

参数 ⟨*dimension*⟩ 应当能被 \pgfmathsetlength 处理。本选项的作用是：在 \pgf@metadecoration@next 等于 \relax, 并且寄存器 \pgfdecoratedinputsegmentremainingdistance 的值小于 ⟨*dimension*⟩ 时，定义

```
\def\pgf@metadecoration@current@state{⟨next state name⟩}%
\let\pgf@metadecoration@next\pgf@metadecoration@run%
```

/pgf/meta-decoration automaton/next state=⟨next state name⟩

本选项的定义是：

```
\pgfkeys{
  /pgf/meta-decoration automaton/next state/.store
  ↪  in=\pgf@metadecoration@next@state,%
}
```

本选项导致定义：

```
\def\pgf@metadecoration@next@state{⟨next state name⟩}
```

24.2.3.2 状态代码中可用的命令

在状态的 ⟨code⟩ 中可以使用下面的命令。

\decoration{⟨decoration name⟩}

本命令在 {pgfmetadecoration} 环境的开头被定义是：

```
\let\decoration\pgf@metadecoration@decoration
%...
\def\pgf@metadecoration@decoration#1{%
  \edef\pgf@decorate@tempname{#1}%
}%
```

\beforedecoration{⟨before code⟩}

本命令在 {pgfmetadecoration} 环境的开头被定义是：

```
\let\beforedecoration\pgf@metadecoration@beforedecoration
%...
\def\pgf@metadecoration@beforedecoration#1{%
  \def\pgf@decorate@tempbefore{#1}%
}%
```

\afterdecoration{⟨after code⟩}

本命令在 {pgfmetadecoration} 环境的开头被定义是：

```
\let\afterdecoration\pgf@metadecoration@afterdecoration
%...
\def\pgf@metadecoration@afterdecoration#1{%
  \def\pgf@decorate@tempafter{#1}%
}%
```

\pgfmetadecoratedpathlength

这个宏在装饰操作开始前 (获取输入路径列表后、执行各个状态前) 被定义：

```
\let\pgfmetadecoratedpathlength\pgf@decorate@totalpathlength
```

这个宏保存输入路径的长度。

\pgfmetadecoratedinputsegmentcompleteddistance

这个宏在装饰操作开始前 (获取输入路径列表后、执行各个状态前) 被定义：

```
\def\pgfmetadecoratedinputsegmentcompleteddistance{\pgfdecoratedcompleteddistance}
```

这个宏保存当下的子输入路径的“已装饰部分”的长度。

\pgfmetadecoratedinputsegmentremainingdistance

这个宏在装饰操作开始前 (获取输入路径列表后、执行各个状态前) 被定义：

```
\def\pgfmetadecoratedinputsegmentremainingdistance{\pgfdecoratedremainingdistance}
```

这个宏保存当下的子输入路径的“未装饰部分”的长度。

\pgfmetadecoratedcompleteddistance

本命令在执行状态选项前 (已经开始执行状态，但尚未执行状态选项) 被定义：

```
\let\pgfmetadecoratedremainingdistance\pgf@decorated@remainingdistance%
  \pgf@x-\pgfmetadecoratedremainingdistance\relax%
  \advance\pgf@x\pgf@decorate@totalpathlength\relax%
  \edef\pgfmetadecoratedcompleteddistance{\the\pgf@x}%
```

这个宏保存当下的输入路径的“已装饰部分”的长度。

\pgfmetadecoratedremainingdistance

本命令在执行状态选项前 (已经开始执行状态，但尚未执行状态选项) 被定义：

```
\let\pgfmetadecoratedremainingdistance\pgf@decorated@remainingdistance
```

这个宏保存当下的输入路径的“未装饰部分”的长度。

\pgfpointmetadecoratedpathfirst

这个宏保存当前子输入路径的起点，这个宏可以用在 ⟨*before code*⟩ 中。

\pgfpointmetadecoratedpathlast

这个宏保存当前子输入路径的终点，这个宏可以用在 ⟨*after code*⟩ 中。

24.2.4 {pgfmetadecoration} 环境

\begin{**pgfmetadecoration**}{⟨*meta decoration name*⟩}
  ⟨*environment content*⟩
\end{**pgfmetadecoration**}

这个环境采用 ⟨*meta decoration name*⟩ 这种 meta 装饰类型来对环境内容生成的软路径做装饰。

24.2.4.1 环境的处理过程

{pgfmetadecoration} 环境的处理过程：

```
% 以下 \pgfmetadecoration%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\begingroup%
  \let\decoration\pgf@metadecoration@decoration%
  \let\beforedecoration\pgf@metadecoration@beforedecoration%
  \let\afterdecoration\pgf@metadecoration@afterdecoration%
  \def\pgf@metadecoration@name{#1}%
% 以下 \pgf@decoration@env%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  \pgfgetpath\pgfdecorateexistingpath%
  \pgfsetpath\pgfutil@empty%
  \let\pgfdecorationpath\pgfutil@empty%
  \let\pgfdecoratedpath\pgfutil@empty%
  \let\pgfpoint@decorated@pathlast\pgfpointorigin%
  \edef\pgfpoint@decorate@existingpathlast{\pgf@x\the\pgf@path@lastx\pgf@y\the
  ↪   \pgf@path@lasty}%
  % Begin a group so transformations don't mess things up.
  \bgroup%
```

```
% 以上 \pgf@decoration@env%%%%%%%%%%%%%%%%%%%%%%%%
% 以上 \pgfmetadecoration%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      〈 环境内容〉
% 以下 \endpgfmetadecoration%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 以下 \pgf@decoration@endenv%%%%%%%%%%%%%%%%%%%%%
  \egroup%
  \pgftransformreset%
  % Save the existing soft path and restore the existing path.
  \pgfgetpath\pgfdecoratedpath%
  \pgfsetpath\pgfdecorateexistingpath%
  \ifx\pgfdecoratedpath\pgfutil@empty%
    \pgferror{I cannot decorate an empty path}%
  \else%
    % If the path consists of a single moveto token, make it
    % a very small horizontal line.
    \pgf@decorate@path@check@moveto\pgfdecoratedpath{%
      \advance\pgf@x by0.0001pt\relax%
      \edef\pgfdecoratedpath{%
        \expandafter\noexpand\pgfdecoratedpath%
        \noexpand\pgfsyssoftpath@linetotoken{\the\pgf@x}{\the\pgf@y}%
      }%
    }%
    {}%
    % Remove special round tokens and get points.
    \pgfprocessround{\pgfdecoratedpath}{\pgfdecoratedpath}%
    % Parse the soft path into a series of decorated input segment objects.
    \pgf@decorate@parsesoftpath{\pgfdecoratedpath}{\pgf@decorate@inputsegmentobjects}%
    % Setup further options
    \pgfkeys{/pgf/every decoration/.try}%
    % Reverse objects if necessary.
    \ifpgf@decorate@inputsegmentobjects@reverse%
      \pgf@decorate@inputsegmentobjects@reverse{\pgf@decorate@inputsegmentobjects}{\pgf@decorate@inputsegmentobjects}%
    \fi%
    \let\pgf@decorated@remainingdistance\pgf@decorate@totalpathlength%
    \let\pgfpoint@decorated@totalpathfirst\pgfpoint@decorated@firstparsed%
    \let\pgfpoint@decorated@totalpathlast\pgfpoint@decorate@lastnonmovetoparsed%
    \let\pgfpoint@decorated@pathfirst\pgfpoint@decorated@totalpathfirst
    \let\pgfpoint@decorated@pathlast\pgfpoint@decorated@totalpathlast%
    % Set up the first input segment.
    \let\pgf@decorate@currentinputsegmentobjects\pgf@decorate@inputsegmentobjects%
    \let\pgf@decorate@transformtoinputsegment\pgfutil@empty%
    \pgf@decorate@getnextinputsegmentobject\pgf@decorate@nextinputsegmentobject%
    \pgf@decorate@processnextinputsegmentobject%
    \pgf@decorate@distancetomove0pt\relax%
  \fi%
% 以上 \pgf@decoration@endenv%%%%%%%%%%%%%%%%%%%%%
  \ifx\pgfdecoratedpath\pgfutil@empty%
  \else%
    \let\pgfmetadecoratedpathlength\pgf@decorate@totalpathlength%
    \def\pgfmetadecoratedinputsegmentremainingdistance{\pgfdecoratedremainingdistance}%
    \def\pgfmetadecoratedinputsegmentcompleteddistance{\pgfdecoratedcompleteddistance}%
    % Perform the meta decoration...
    \expandafter\let\expandafter\pgf@metadecoration@current@state%
```

```
      \csname pgf@metadecoration@@\pgf@metadecoration@name @initial\endcsname%
    \pgf@metadecoration@run%
    % ..until the final state.
    \let\pgf@decorate@tempname\pgfutil@empty%
    \let\pgf@decorate@tempbefore\pgfutil@empty%
    \let\pgf@decorate@tempafter\pgfutil@empty%
    \csname pgf@metadecoration@@\pgf@metadecoration@name @
    ↪  \pgf@metadecoration@current@state @code\endcsname%
    \ifx\pgf@decorate@tempname\pgfutil@empty%
      \def\pgf@decorate@tempname{moveto}
    \fi%
    \pgf@decorate@invoke{%
      {\pgf@decorate@tempname}{\pgfdecoratedremainingdistance}%
      {\pgf@decorate@tempbefore}{\pgf@decorate@tempafter}%
    }%
  \fi%
  \pgfgetpath\pgfdecorationpath%
  % Take stuff outside the group.
  \global\let\pgf@decorate@decorationpathtemp\pgfdecorationpath%
  \global\let\pgf@decorate@decoratedpathtemp\pgfdecoratedpath%
  \global\let\pgf@decorate@existingpathtemp\pgfdecorateexistingpath%
  \global\let\pgfpoint@decorated@pathlasttemp\pgfpoint@decorated@pathlast%
\endgroup%
% Are we in LaTeX?
\pgfutil@ifnextchar\@checkend{\aftergroup\pgf@decorate@installmacrosatend}%
    {\pgf@decorate@installmacrosatend}%
% 以上 \endpgfmetadecoration%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

24.2.4.2 环境相关的命令

\pgf@metadecoration@run

本命令的处理是：如果 \pgf@metadecoration@current@state 保存的是 final (终止状态的名称)，那么什么也不做，否则执行 \pgf@metadecoration@do@state.

\pgf@metadecoration@do@state

本命令会执行 ⟨*meta decoration name*⟩ 的、除了 final 的状态来做装饰。

本命令的处理如下：

1. 准备：

```
\let\pgf@metadecoration@next\relax%
\let\pgf@metadecoration@next@state\pgf@metadecoration@current@state%
% Set up some macros.
\let\pgfmetadecoratedremainingdistance\pgf@decorated@remainingdistance%
\pgf@x-\pgfmetadecoratedremainingdistance\relax%
\advance\pgf@x\pgf@decorate@totalpathlength\relax%
\edef\pgfmetadecoratedcompleteddistance{\the\pgf@x}%
```

2. 执行当前状态 \pgf@metadecoration@current@state 的选项

```
\pgfqkeys{/pgf/meta-decoration automaton}{⟨当前状态的选项⟩}
```

这些选项的路径都是 /pgf/meta-decoration automaton.

3. 检查以上选项是否重定义了 \pgf@metadecoration@next, 即检查它是否等于 \relax,
   - 如果不等于 \relax, 那就执行 \pgf@metadecoration@next.
   - 如果等于 \relax, 那就执行 \pgf@metadecoration@do@code.

\pgf@metadecoration@do@code

本命令的处理是：

1. 准备

```
\let\pgf@decorate@tempname\pgfutil@empty%
\let\pgf@decorate@tempbefore\pgfutil@empty%
\let\pgf@decorate@tempafter\pgfutil@empty%
\let\pgfpointmetadecoratedpathfirst\pgfpointdecoratedpathfirst%
\let\pgfpointmetadecoratedpathlast\pgfpointdecoratedpathlast%
```

2. 执行当前状态 \pgf@metadecoration@current@state 的 ⟨*code*⟩.
3. 检查 \pgf@decorate@tempname 的值

```
\ifx\pgf@decorate@tempname\pgfutil@empty%
  \def\pgf@decorate@tempname{moveto}
\fi%
```

4. 执行 \pgf@decorate@invoke

```
\pgf@decorate@invoke{%
  {\pgf@decorate@tempname}{\pgf@metadecoration@width}%
  {\pgf@decorate@tempbefore}{\pgf@decorate@tempafter}%
}%
```

5. 切换当前状态

```
\let\pgf@metadecoration@current@state\pgf@metadecoration@next@state
```

6. 从头执行 \pgf@metadecoration@run

24.2.5 使用一个装饰类型的命令

\pgfdecoratepath{⟨*decoration name*⟩}{⟨*path*⟩}

本命令的定义是：

```
\long\def\pgfdecoratepath#1#2{%
  \pgfdecoration{{#1}{\pgfdecoratedpathlength}{\pgfdecoratebeforecode}{
  ↪  \pgfdecorateaftercode}}%
  #2%
  \endpgfdecoration}%
```

本命令的定义有前缀 \long.

参数 ⟨*path*⟩ 是能生成软路径的命令，但不能含有 \pgfusepath, 不过在 ⟨*decoration name*⟩ 的状态的 ⟨*code*⟩ 中可以含有 \pgfusepath. 本命令对 ⟨*path*⟩ 生成的软路径做装饰。

本命令需要 \pgfdecoratebeforecode, \pgfdecorateaftercode 的配合。这两个宏的初始值是 \pgfutil@empty.

\pgfdecoratecurrentpath{⟨*decoration name*⟩}

本命令的定义是：

```
\def\pgfdecoratecurrentpath#1{%
  \pgfgetpath\pgf@decorate@currentpath%
  \pgfsetpath\pgfutil@empty%
  \pgfdecoration{{#1}{\pgfdecoratedpathlength}{\pgfdecoratebeforecode}{
  ↪  \pgfdecorateaftercode}}%
    \pgfsetpath\pgf@decorate@currentpath%
  \endpgfdecoration}%
```

本命令使用装饰类型 ⟨*decoration name*⟩ 对当前的软路径做装饰。在创建装饰路径后，还会恢复这个

被装饰路径。

本命令需要 \pgfdecoratebeforecode, \pgfdecorateaftercode 的配合。这两个宏的初始值是 \pgfutil@empty.

24.3 预定义的装饰类型

预定义的装饰类型有 lineto, moveto, curveto.

第二十五章　Nodes 与 Shape

文件《pgfmoduleshapes.code.tex》。

```
\usepgfmodule{shapes} % LaTeX and plain TeX and pure pgf
\usepgfmodule[shapes] % ConTeXt and pure pgf
```

25.1　声明一个 shape

预定义的 shape 只有三个，即 `rectangle`, `circle`, `coordinate`. 另外，程序库 shapes.symbols, shapes.geometric, shapes.callouts, shapes.misc, shapes.arrows, shapes.multipart 等还定义了大量的 shape.

PGF 应当有能力处理包含数百个 node 的图形，也应当有能力处理包含数千个 node 的文档。但是，不能让 PGF 计算并记住所有 node 的各种锚位置。

将 shape 的锚位置 (anchors) 分为 Normal Anchors 与 Saved Anchors 两类。例如，对于形状 `rectangle` 来说，其右上角位置是 `north east`, 这个“名称”是 normal anchor, 通常用在绘图命令中；而在 `rectangle` 的定义中有宏 `\northeast`, 这是个 saved anchor.

每当使用一个形状时，无论是否用到该形状的 saved anchors, 其 saved anchors 都会被计算并保存；而对于 normal anchors 来说，只有在用到它们时才会按照其定义代码将其计算出来，这样会节省存储空间。在计算 normal anchors 时，可以利用已保存的 saved anchors, 例如，形状 `rectangle` 只有两个 saved anchors, 即右上角 `\northeast` 和左下角 `\southwest`, 利用这两个点可以计算出矩形上的其它点。坐标点 `\southwest` 的 x 分量与 `\northeast` 的 y 分量可以组合成锚位置 `north west`, 这是个 normal anchor. 目前在 `rectangle` 上定义了 13 个 normal anchor, 都是利用两个 saved anchors 定义的。

所有的锚位置 (包括 saved anchors 和 normal anchors) 都在“局部形状坐标空间”(local shape coordinate space) 中指定，命令 `\pgfnode` 会自动把坐标变换应用于这个空间中的坐标点。

如果希望能在 shape 中放置文字盒子 (做成 node)，应当指定一个或数个 node part.

定义 shape 时可能要给出以下要素：

- 名称。
- 计算 saved anchors 和 saved dimensions 的代码。
- 用 saved anchors 计算锚位置的代码。
- 画出 background path 和 foreground path 的代码，可以没有这个内容。
- 在画出 background path 和 foreground path 之前 (之后) 需要执行的代码，可以没有这个内容。
- 规定 node parts, 可以没有这个内容。

命令 `\pgfdeclareshape` 用于声明新的 shape, 这个命令由许多子命令构成，例如

◊ `\savedanchor`→P. 438`{⟨command⟩}{⟨code⟩}`

◊ `\saveddimen`→P. 441`{⟨command⟩}{⟨code⟩}`

◊ `\savedmacro`→P. 441`{⟨command⟩}{⟨code⟩}`

◊ `\anchor`→P. 443`{⟨anchor name⟩}{⟨code⟩}`

◊ `\anchorborder`→P. 445`{⟨code⟩}`

◊ \behindbackgroundpath→P. 445{⟨code⟩}
◊ \backgroundpath→P. 446{⟨code⟩}
◊ \beforebackgroundpath→P. 446{⟨code⟩}
◊ \behindforegroundpath→P. 446{⟨code⟩}
◊ \foregroundpath→P. 446{⟨code⟩}
◊ \beforeforegroundpath→P. 446{⟨code⟩}

当执行命令

\pgfnode{⟨shape⟩}{⟨anchor⟩}{⟨label text⟩}{⟨name⟩}{⟨path usage command⟩}

或

\pgfmultipartnode{⟨shape⟩}{⟨anchor⟩}{⟨name⟩}{⟨path usage command⟩}

时，有以下次序的执行步骤：

1. 执行各个 saved anchor, saved dimen 的定义代码 (由命令 \savedanchor, \saveddimen 保存的定义代码)
2. 执行各个 saved macro 的定义代码 (由命令 \savedmacro 保存的定义代码)
3. 在一个花括号组内执行 ⟨anchor⟩ (这是个 normal anchor) 对应的 ⟨code⟩(即由 \anchor⟨anchor⟩⟨code⟩ 保存的代码)，然后全局地令 \pgf@x, \pgf@y 的值变成原值的负值
4. 在一个花括号组内执行 \behindbackgroundpath 保存的 ⟨code⟩(即 bbg)
5. 执行 \backgroundpath 保存的 ⟨code⟩(即 bg)
6. 执行参数 ⟨path usage command⟩(这个参数可以是，例如 \pgfusepath{stroke})
7. 在一个花括号组内执行 \beforebackgroundpath 保存的 ⟨code⟩(即 fbg)
8. 在一个花括号组内安插文字盒子
9. 在一个花括号组内执行 \behindforegroundpath 保存的 ⟨code⟩(即 bfg)
10. 执行 \foregroundpath 保存的 ⟨code⟩(即 fg)
11. 执行参数 ⟨path usage command⟩(这个参数可以是，例如 \pgfusepath{stroke})
12. 在一个花括号组内执行 \beforeforegroundpath 保存的 ⟨code⟩(即 ffg)

按以上可知：

- 在 ⟨anchor⟩ 对应的 ⟨code⟩ 中可以使用 saved anchor, saved dimen, saved macro 提供的命令、变量，但不能反过来。
- 在 \backgroundpath 的参数 ⟨code⟩ 中可以使用 saved anchor, saved dimen, saved macro.
- 第一次执行 ⟨path usage command⟩ 可能会把 bbg 和 bg 都利用起来。
- 第二次执行 ⟨path usage command⟩ 可能会把 fbg, bfg, fg 都利用起来。
- ffg 不被 ⟨path usage command⟩ 利用。

25.1.1 命令 \pgfdeclareshape

\pgfdeclareshape{⟨shape name⟩}{⟨specification⟩}

本命令的定义有前缀 \long; 本命令设置一个 TEX 组，所有操作都在这个组内进行。

```
\long\def\pgfdeclareshape#1#2{%
  {%
    \def\pgf@sm@shape@name{#1}%
    \let\savedanchor=\pgf@sh@savedanchor
    \let\saveddimen=\pgf@sh@saveddimen
    \let\savedmacro=\pgf@sh@savedmacro% MW
    \let\deferredanchor=\pgf@sh@deferredanchor% CJ
    \let\anchor=\pgf@sh@anchor
```

```
    \let\anchorborder=\pgf@sh@anchorborder
    \let\behindbackgroundpath=\pgf@sh@behindbgpath
    \let\backgroundpath=\pgf@sh@bgpath
    \let\beforebackgroundpath=\pgf@sh@beforebgpath
    \let\behindforegroundpath=\pgf@sh@behindfgpath
    \let\foregroundpath=\pgf@sh@fgpath
    \let\beforeforegroundpath=\pgf@sh@beforefgpath
    \let\nodeparts=\pgf@sh@boxes
    \let\inheritsavedanchors=\pgf@sh@inheritsavedanchors
    \let\inheritanchor=\pgf@sh@inheritanchor
    \let\inheritanchorborder=\pgf@sh@inheritanchorborder
    \let\inheritbehindbackgroundpath=\pgf@sh@inheritbehindbgpath
    \let\inheritbackgroundpath=\pgf@sh@inheritbgpath
    \let\inheritbeforebackgroundpath=\pgf@sh@inheritbeforebgpath
    \let\inheritbehindforegroundpath=\pgf@sh@inheritbehindfgpath
    \let\inheritforegroundpath=\pgf@sh@inheritfgpath
    \let\inheritbeforeforegroundpath=\pgf@sh@inheritbeforefgpath
    \let\inheritnodeparts=\pgf@sh@inheritboxes
    \anchorborder{\csname pgf@anchor@#1@center\endcsname}%
    \anchor{text}{\pgfpointorigin}%
    \nodeparts{text}%
    \expandafter\global\expandafter\let\csname pgf@sh@s@\pgf@sm@shape@name
    ↪  \endcsname=\pgfutil@empty%
    #2%
  }%
}%
```

本命令的操作分为两个部分，第一部分定义一些命令，第二部分利用这些命令来规定 ⟨*shape name*⟩ 的各种特点。

在以上定义中：

- 执行

  ```
  \anchorborder{\csname pgf@anchor@⟨shape name⟩@center\endcsname}%
  ```

  导致

  ```
  \expandafter\gdef\csname pgf@anchor@⟨shape name⟩@border\endcsname#1{
  ↪  \pgf@process{#1}\csname pgf@anchor@⟨shape name⟩@center\endcsname}
  ```

  这就是控制序列 `\csname pgf@anchor@`⟨*shape name*⟩`@border\endcsname` 的最初定义，其定义内容指向 ⟨*shape name*⟩ 的 `center` 位置。

- 执行

  ```
  \anchor{text}{\pgfpointorigin}%
  ```

  导致

  ```
  \expandafter\gdef\csname pgf@anchor@⟨shape name⟩@text\endcsname{\pgfpointorigin
  ↪  }
  ```

  即定义 ⟨*shape name*⟩ 的锚位置 `text` 处于原点。

- 执行

  ```
  \nodeparts{text}%
  ```

  导致

  ```
  \expandafter\gdef\csname pgf@sh@boxes@⟨shape name⟩\endcsname{text}
  ```

  即只规定一个名称为 `text` 的文字盒子。

- 执行

```
\expandafter\global\expandafter\let\csname pgf@sh@s@⟨shape name⟩\endcsname=
↪  \pgfutil@empty%
```

全局地规定这个控制序列的初始值，见参考。

\nodeparts{⟨*list of node parts*⟩}

\let\nodeparts=\pgf@sh@boxes

```
\def\pgf@sh@boxes#1{\expandafter\gdef\csname pgf@sh@boxes@\pgf@sm@shape@name
↪  \endcsname{#1}}%
```

本命令将逗号分隔的列表 ⟨*list of node parts*⟩ 保存到控制序列

\csname pgf@sh@boxes@\pgf@sm@shape@name\endcsname

命令 \pgfmultipartnode 会用到这个列表。

⟨*list of node parts*⟩ 的列表项是各个 node part 的名称。按 \pgfdeclareshape 的定义，在默认下，一个 shape 只有一个 node part，其名称为 text. 有的形状，如 circle split 有两个 node parts，其名称分别是 text 和 lower，因此在 circle split 的定义中应该有 (见《pgflibraryshapes.multipart.code.tex》):

```
\nodeparts{text,lower}
```

\savedanchor⟨*command*⟩{⟨*code*⟩}

\let\savedanchor=\pgf@sh@savedanchor

关于本命令：

- 在执行 \pgfmultipartnode 时，⟨*command*⟩ 是 (在一个组内) 局部定义、局部计算、局部使用地，因此即使名称 ⟨*command*⟩ 由很简短的字符构成 (如 \center 或 \a)，也不太可能引起名称冲突。
- 本命令的参数 ⟨*code*⟩ 应该是能为尺寸尺寸寄存器 \pgf@x, \pgf@y 赋值的代码。本命令不会对 ⟨*code*⟩ 做任何展开，只是保存它。
- 可以在 ⟨*code*⟩ 中使用文字盒子的宽度、高度、深度来计算 saved anchor(即 ⟨*command*⟩)，也可以使用 \pgfshapeminwidth 或 \pgfshapeinnerxsep 等宏，也可以包含由命令 \pgfnode 引入的关于形状的其它变量。这样就可以把形状的锚位置 ⟨*command*⟩ 与文字盒子的尺寸联系起来，使得 ⟨*command*⟩ 的位置能随文字盒子尺寸的变化而变化。如果要在 ⟨*code*⟩ 中使用宏 \pgfshapeminwidth 或者 \pgfshapeinnerxsep, 需要注意这两个宏是“纯文本宏”，而不是尺寸宏。表面上宏 \pgfshapeminwidth 的值可能是“2cm”，但是这个“2cm”是纯文本，不是尺寸，所以“0.5\pgfshapeminwidth”是“0.52cm”，而不是“1cm”，所以需要使用以下句式：

```
\savedanchor{\upperrightcorner}{
  \pgf@y=.5\ht\pgfnodeparttextbox % 文字盒子的高度
  \pgf@x=.5\wd\pgfnodeparttextbox % 文字盒子的宽度
  \setlength{\pgf@xa}{\pgfshapeminwidth}
  \ifdim\pgf@x<.5\pgf@xa
    \pgf@x=.5\pgf@xa
  \fi
}
```

\pgf@sh@savedanchor⟨*command*⟩{⟨*code*⟩}

本命令的定义是：

```
\def\pgf@sh@savedanchor#1#2{%
  \expandafter\pgfutil@g@addto@macro\csname pgf@sh@s@\pgf@sm@shape@name\endcsname{
  ↪  \pgf@sh@resavedanchor{#1}{#2}}}%
```

命令 \pgfutil@g@addto@macro→P. 29 见文件《pgfutil-common.tex》。
本命令的作用是全局地重定义控制序列

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname

也就是全局地把

\pgf@sh@resavedanchor{⟨*command*⟩}{⟨*code*⟩}

添加到这个控制序列的内容中。
例如，如果执行：

```
\makeatletter
\tikz\node[shape=rectangle]{};
\expandafter\meaning\csname pgf@sh@s@rectangle\endcsname
\makeatother
```

就会得到很长的一串符号，其中就有：

```
\pgf@sh@resavedanchor{\northeast}{%
  \pgf@x =\the \wd \pgfnodeparttextbox
  \pgfmathsetlength \pgf@xc {\pgfkeysvalueof{/pgf/inner xsep}}
  \advance\pgf@x by 2\pgf@xc
  \pgfmathsetlength \pgf@xb {\pgfkeysvalueof{/pgf/minimum width}}
  \ifdim \pgf@x <\pgf@xb
    \pgf@x =\pgf@xb
  \fi
  \pgf@x =.5\pgf@x
  \advance\pgf@x by.5\wd \pgfnodeparttextbox
  \pgfmathsetlength \pgf@xa {\pgfkeysvalueof{/pgf/outer xsep}}
  \advance \pgf@x by\pgf@xa
  \pgf@y =\ht \pgfnodeparttextbox
  \advance \pgf@y by\dp \pgfnodeparttextbox
  \pgfmathsetlength\pgf@yc {\pgfkeysvalueof{/pgf/inner ysep}}
  \advance \pgf@y by 2\pgf@yc
  \pgfmathsetlength\pgf@yb {\pgfkeysvalueof{/pgf/minimum height}}
  \ifdim \pgf@y<\pgf@yb
    \pgf@y =\pgf@yb
  \fi
  \pgf@y =.5\pgf@y
  \advance \pgf@y by-.5\dp \pgfnodeparttextbox
  \advance \pgf@y by.5\ht \pgfnodeparttextbox
  \pgfmathsetlength\pgf@ya {\pgfkeysvalueof{/pgf/outer ysep}}
  \advance \pgf@y by\pgf@ya
}
```

上面的符号是关于 rectangle 这个 shape 的 \northeast 这个 saved anchor 位置的。

\pgf@sh@resavedanchor⟨*command*⟩{⟨*code*⟩}

本命令的定义是：

```
\def\pgf@sh@resavedanchor#1#2{%
  \pgf@process{#2}%
  \edef\pgf@sh@marshal{%
    \noexpand\pgfutil@g@addto@macro\noexpand\pgf@sh@savedpoints{%
      \noexpand\def\noexpand#1{\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}%
    }}%
  \pgf@sh@marshal%
}%
```

命令 \pgf@process→P. 250 见文件《pgfcorepoints.code.tex》。

本命令的参数 ⟨*code*⟩ 应该是能为尺寸尺寸寄存器 \pgf@x, \pgf@y 赋值的代码。

本命令的作用是全局地重定义 \pgf@sh@savedpoints, 也就是把

\def⟨*command*⟩{\pgfqpoint{⟨*x* 尺寸代码⟩}{⟨*y* 尺寸代码⟩}}

添加到 \pgf@sh@savedpoints 的定义中。保存在 \pgf@sh@savedpoints 中的内容就是一个或者数个这样的定义，例如：

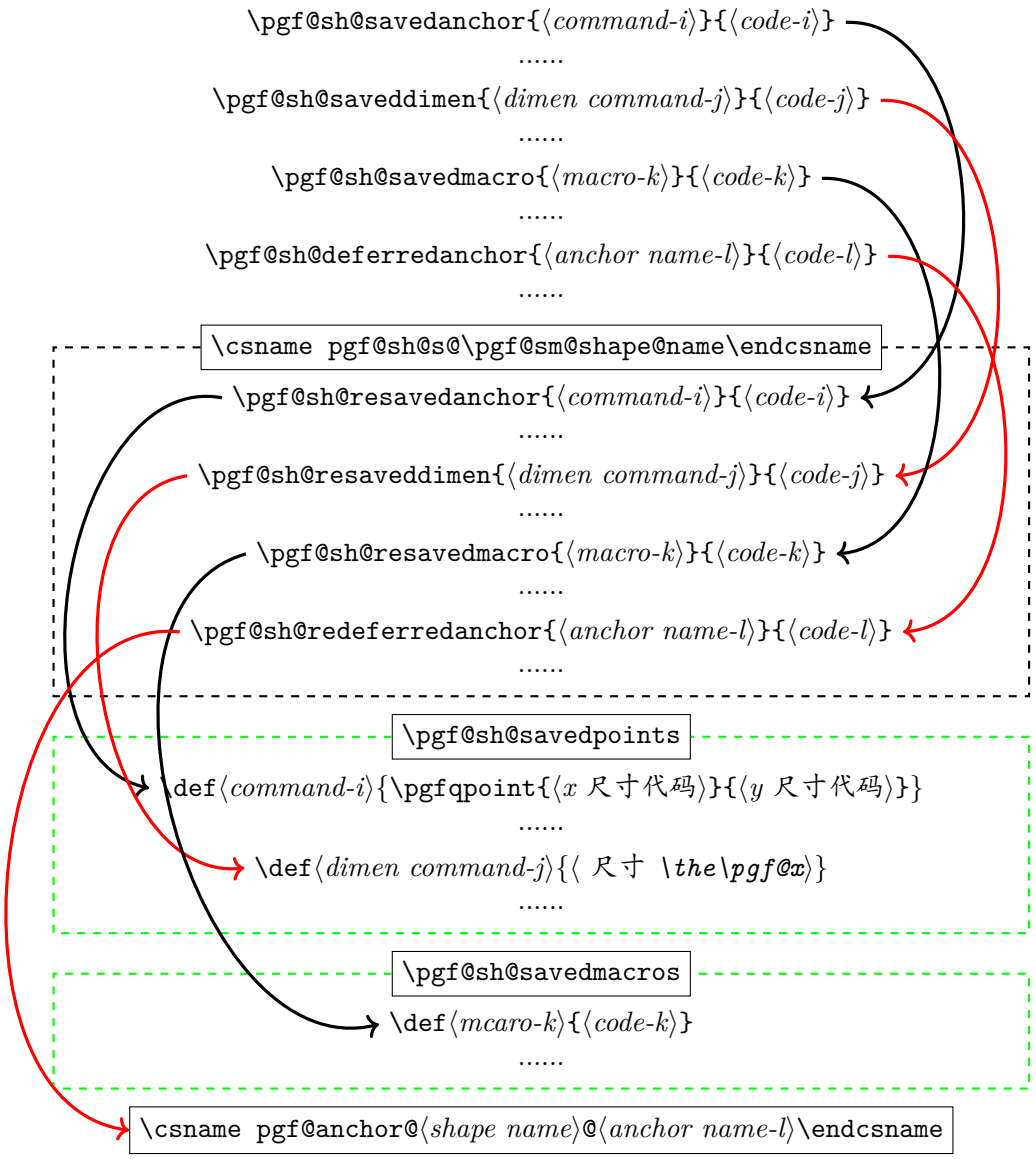
```
macro:->\def \northeast {\pgfqpoint {3.71274pt}{3.71274pt}}\def \southwest
{\pgfqpoint {-3.71274pt}{-3.71274pt}}
```

```
\makeatletter
\tikz \node{};
{\ttfamily\meaning\pgf@sh@savedpoints}
\makeatother
```

\pgf@sh@savedpoints

这个宏保存 2 类定义：

- 定义 saved anchor 的代码
- 定义 saved dimen 的代码



如上图，命令 \pgf@sh@savedanchor 向 \csname pgf@sh@s@\pgf@sm@shape@name\endcsname 中添加 \pgf@sh@resavedanchor; 命令 \pgf@sh@resavedanchor 向 \pgf@sh@savedpoints 中添加 saved

anchor 点的定义。

在命令 \pgfmultipartnode 的处理中会执行

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname, \pgf@sh@savedpoints

这两个命令。

命令 \csname pgf@sh@s@\pgf@sm@shape@name\endcsname 的初始值是\pgfutil@empty, 这是全局地初值。

在 \pgfmultipartnode 的定义中，\pgf@sh@savedpoints 被 let 为 \pgfutil@empty.

命令 \pgfnode 会调用 \pgfmultipartnode.

\saveddimen⟨*command*⟩{⟨*code*⟩}

\let\saveddimen=\pgf@sh@saveddimen

\pgf@sh@saveddimen⟨*command*⟩{⟨*code*⟩}

本命令的定义是：

```
\def\pgf@sh@saveddimen#1#2{%
  \expandafter\pgfutil@g@addto@macro\csname pgf@sh@s@\pgf@sm@shape@name\endcsname{
  ↪  \pgf@sh@resaveddimen{#1}{#2}}}%
```

本命令的参数 ⟨*code*⟩ 应该是能为尺寸尺寸寄存器 \pgf@x 赋值的代码。

本命令的作用是全局地重定义控制序列

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname

也就是把

\pgf@sh@resaveddimen{⟨*command*⟩}{⟨*code*⟩}

添加到这个控制序列的定义内容中。

\pgf@sh@resaveddimen⟨*command*⟩{⟨*code*⟩}

命令 \pgf@sh@resaveddimen 的定义是

```
\def\pgf@sh@resaveddimen#1#2{%
  {#2\global\pgf@x=\pgf@x}%
  \edef\pgf@sh@marshal{%
    \noexpand\pgfutil@g@addto@macro\noexpand\pgf@sh@savedpoints{%
      \noexpand\def\noexpand#1{\the\pgf@x}%
    }}%
  \pgf@sh@marshal%
}%
```

本命令的参数 ⟨*code*⟩ 应该是能为尺寸尺寸寄存器 \pgf@x 赋值的代码。

可见本命令的作用是把

\def⟨*command*⟩{⟨ 尺寸 *\the\pgf@x*⟩}

添加到 \pgf@sh@savedpoints →P. 440 保存的内容中 (对它作全局地重定义)，其中的 \pgf@x 是 ⟨*code*⟩ 计算出来的尺寸。

注意在预定义的形状 rectangle 的定义中没有使用命令 \saveddimen.

\savedmacro⟨*macro*⟩{⟨*code*⟩}

\let\savedmacro=\pgf@sh@savedmacro

\pgf@sh@savedmacro⟨*macro*⟩{⟨*code*⟩}

本命令的定义是：

```
\def\pgf@sh@savedmacro#1#2{% MW
  \expandafter\pgfutil@g@addto@macro\csname pgf@sh@s@\pgf@sm@shape@name\endcsname{
  ↪  \pgf@sh@resavedmacro{#1}{#2}}}% MW
```

执行 \pgf@sh@savedmacro{⟨*macro*⟩}{⟨*code*⟩} 的作用是全局地重定义控制序列

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname

也就是把

\pgf@sh@resavedmacro{⟨*command*⟩}{⟨*code*⟩}

添加到这个控制序列保存的内容中。

在 \pgfmultipartnode 的定义中，\pgf@sh@savedmacro 被 let 为 \pgfutil@empty.

\pgf@sh@resavedmacro⟨*macro*⟩{⟨*code*⟩}

本命令的定义是

```
\def\pgf@sh@resavedmacro#1#2{%
    \let#1\pgfutil@empty%
    \def\addtosavedmacro##1{%
        \expandafter\def\expandafter\pgf@sh@addtomacro@temp\expandafter{#1
        ↪  \noexpand\def\noexpand##1{##1}}%
        {\expandafter\pgfutil@toks@\expandafter{\pgf@sh@addtomacro@temp}
        ↪  \expandafter}%
        \expandafter\def\expandafter#1\expandafter{\the\pgfutil@toks@}%
    }%
  #2\relax%
  \edef\pgf@sh@marshal{%
    \noexpand\pgfutil@g@addto@macro\noexpand\pgf@sh@savedmacros{%
      \noexpand\def\noexpand#1{#1}%
    }}%
  \pgf@sh@marshal%
}%
```

可见本命令的作用是把定义 ⟨*macro*⟩ 的句子:

\def⟨*mcaro*⟩{定义内容}

添加到 \pgf@sh@savedmacros 保存的内容中 (对此命令作全局地重定义)。

参数 ⟨*code*⟩ 应当能实现对 ⟨*mcaro*⟩ 作定义，如可以包含

\def⟨*mcaro*⟩{...}, \let⟨*mcaro*⟩⟨*something*⟩

这样的代码。

从上面定义看，本命令的展开过程中会临时定义一个命令 \addtosavedmacro, 在 ⟨*code*⟩ 中可以使用这个命令。

\pgf@sh@savedmacros

如上。

注意在形状 rectangle 的定义中没有使用命令 \savedmacro.

如果执行下面的代码 (事先调用 shapes.geometric 库):

```
\makeatletter
\expandafter\meaning\csname pgf@sh@s@regular polygon\endcsname
\makeatother
```

就会得到很长的一串:

```
macro:->\pgf@sh@resavedmacro {\sides }{\pgfmathtruncatemacro \sides {\pgfkeysvalueof
↪  {/pgf/regular polygon sides}}}\pgf@sh@resavedmacro {\anglestep }{...}...
```

\deferredanchor{⟨*anchor name*⟩}{⟨*code*⟩}

\let\deferredanchor=\pgf@sh@deferredanchor
本命令类似 \anchor, 区别是，如果 ⟨anchor name⟩ 中含有可展开的命令，本命令不会将其展开，只是保存。

\pgf@sh@deferredanchor{⟨anchor name⟩}{⟨code⟩}
本命令的定义是：

```
\def\pgf@sh@deferredanchor#1#2{% CJ
  \expandafter\pgfutil@g@addto@macro
    \csname pgf@sh@s@\pgf@sm@shape@name\endcsname{\pgf@sh@redeferredanchor{#1}{#2
    ↪  }}}%
```

本命令的参数 ⟨code⟩ 应该是能为尺寸寄存器 \pgf@x, \pgf@y 赋值的代码，也就是计算锚位置 ⟨anchor name⟩ 的代码。
本命令的作用是全局地重定义控制序列

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname

也就是把

\pgf@sh@redeferredanchor{⟨anchor name⟩}{⟨code⟩}

添加到这个控制序列的定义中。

\pgf@sh@redeferredanchor{⟨anchor name⟩}{⟨code⟩}
本命令的定义是：

```
\def\pgf@sh@redeferredanchor#1#2{% CJ
  \expandafter\gdef\csname pgf@anchor@\pgf@sm@shape@name @#1\endcsname{#2}}
  ↪  % CJ
```

本命令的作用是全局地定义控制序列

\csname pgf@anchor@\pgf@sm@shape@name @⟨anchor name⟩\endcsname

这个控制序列保存 ⟨code⟩, 只是 (全局地) 保存，不做任何展开。

命令 \pgf@sh@deferredanchor 不展开 ⟨anchor name⟩ 中的可展开命令，因为 ⟨anchor name⟩ 处于“定义内容”代码中。

注意，命令 \savedanchor, \saveddimen, \savedmacro, \deferredanchor 都会全局地重定义控制序列 \csname pgf@sh@s@\pgf@sm@shape@name\endcsname.

\csname pgf@sh@s@⟨shape name⟩\endcsname
这个控制序列中保存了关于 ⟨shape name⟩ 的 saved anchor, saved dimen, saved macro, deferred anchor 的计算方法，如前。这个控制序保存了一系列的

- \pgf@sh@resavedanchor, 能全局地重定义 \pgf@sh@savedpoints
- \pgf@sh@resaveddimen, 能全局地重定义 \pgf@sh@savedpoints
- \pgf@sh@resavedmacro, 能全局地重定义 \pgf@sh@savedmacros
- \pgf@sh@redeferredanchor, 能全局地定义 \csname pgf@anchor@⟨shape name⟩@⟨anchor name⟩\endcsname

\anchor{⟨anchor name⟩}{⟨code⟩}
\let\anchor=\pgf@sh@anchor
关于本命令：

- 这个命令定义一个名称为 ⟨anchor name⟩ 的 normal anchor(名称不以反斜线开头)。由于名称 ⟨anchor name⟩ 不会被传递到系统层 (system level)，所以其构成可以比较随意，例如可以包含字母，数字，冒号。

- 本命令的参数 ⟨*code*⟩ 应该是能为尺寸寄存器 `\pgf@x`, `\pgf@y` 赋值的代码，也就是计算锚位置 ⟨*anchor name*⟩ 的代码。
- 命令 `\pgfnode` 或 `\pgfmultipartnode` 总是先执行各个 saved anchor 的定义代码，然后再执行定义 normal anchor 的代码，因此在 ⟨*code*⟩ 中可以使用 saved anchor(宏的形式)。
- 执行命令 `\anchor{`⟨*anchor name*⟩`}{`⟨*code*⟩`}` 时，并不对 ⟨*code*⟩ 做任何展开。⟨*code*⟩ 会被 (全局地) 保存在下面的控制序列中：

  `\csname pgf@anchor@`⟨*shape name*⟩`@`⟨*anchor name*⟩`\endcsname`

  当在命令 `\pgfnode` 或 `\pgfmultipartnode` 中使用参数 ⟨*anchor name*⟩ 时，此控制序列会“在一个花括号组内”被展开 (即执行 ⟨*code*⟩), 得到相应的 `\pgf@x` 和 `\pgf@y` 的值，然后全局地令 `\pgf@x=-\pgf@x`, `\pgf@y=-\pgf@y` ——这就是执行 ⟨*code*⟩ 的结果——没有其他结果 (严格限制执行 ⟨*code*⟩ 的副作用)。

  当多次使用命令 `\anchor` 时，例如：

```
\anchor{⟨anchor name 1⟩}{⟨code 1⟩}
\anchor{⟨anchor name 2⟩}{⟨code 2⟩}
```

  在 ⟨*code 2*⟩ 中可以使用 `\csname pgf@anchor@`⟨*shape name*⟩`@`⟨*anchor name 1*⟩`\endcsname`, 这会导致 ⟨*code 1*⟩ 被执行，从而引入 ⟨*anchor name 1*⟩ 对应的 `\pgf@x` 和 `\pgf@y`.
- 在绘图命令中使用的是 normal anchor，而不是 saved anchor. 一个 saved anchor 不能自动转换为一个 normal anchor，如果要转换，可以使用本命令。如：

```
\anchor{north east}{\upperrightcorner}
↪  % 假设 \upperrightcorner 是个 saved anchor
```

  也就是用 saved anchor 来定义相应的 normal anchor.

  锚位置 `north west` 可以如下定义：

```
\anchor{north west}{
  \upperrightcorner  % 这里 \upperrightcorner 是个 saved anchor
  \pgf@x=-\pgf@x  % 变成原值的相反数
}
```

- 在默认下，一个 shape 只有一个 node part，这个 node part 的名称默认为 `text`; 与它对应的锚位置 `text` (名称与 node part 一样) 被默认在 shape 坐标系的原点 (`\pgfpointorigin`) 处。放置文字盒子时，文字盒子的左下角 (基点) 会处于锚位置 `text` 上。当然你也可以把锚位置 `text` 指定到其它位置，例如：

```
\savedanchor{\upperrightcorner}{
  \pgf@y=.5\ht\pgfnodeparttextbox
  \pgf@x=.5\wd\pgfnodeparttextbox
}
\anchor{text}{
  \upperrightcorner
  \pgf@x=-\pgf@x
  \pgf@y=-\pgf@y
}
```

  按上面代码规定锚位置 `text` 后，文字盒子 `\pgfnodeparttextbox` 的中心 (而不是左下角) 就位于 shape 坐标系的原点了 (盒子的基点位于锚位置 `text` 上)。
- 当使用命令 `\nodeparts`⟨*part name 1,part name 2,...*⟩ 声明多个 node parts 时，你需要用

```
\anchor{⟨part name i⟩}{⟨code⟩}
```

  为每个 node part 定义放置文字盒子的锚位置，当然这个锚位置的名称与所在的 node part 的名

称要保持一致。

- 如果不使用 \nodeparts 自定义 node parts, 那么命令 \pgfmultipartnode 会自动插入名称为 \pgfnodeparttextbox 的文字盒子，并且在插入这个盒子前自动设置针对这个盒子的变换，将这个盒子变换到预期的位置、状态，而设置变换时要用到名称为 text 的 normal anchor. 类似地，对其他 node part 上的文字盒子也会做类似的操作。设置变换、插入文字盒子的操作是在一个花括号组内完成的，所设置的变换的有效范围就限制在这个组内。
- 本命令与前面的 \deferredanchor 类似，区别是，如果 ⟨*anchor name*⟩ 中含有可展开的命令，本命令会将其展开，以展开的结果作为一个 normal anchor name.

\pgf@sh@anchor{⟨*anchor name*⟩}{⟨*code*⟩}

本命令的定义是：

```
\def\pgf@sh@anchor#1#2{\expandafter\gdef\csname pgf@anchor@\pgf@sm@shape@name @#1
↪  \endcsname{#2}}%
```

本命令的作用是全局地定义命令

\csname pgf@anchor@\pgf@sm@shape@name @⟨*anchor name*⟩\endcsname

也就是把 ⟨*code*⟩ 全局地保存在这个命令中。

\csname pgf@anchor@\pgf@sm@shape@name @⟨*anchor name*⟩\endcsname

如前，命令 \pgf@sh@deferredanchor, \pgf@sh@anchor 都会重定义这个控制序例，其中保存的是计算 ⟨*anchor name*⟩ 这个锚位置的代码，只有在用到 ⟨*anchor name*⟩ 这个锚位置为 shape 或 node 定位时 (让锚位置位于锚定点上)，其中保存的代码才会被执行，做一些计算，为寄存器 \pgf@x, \pgf@y 赋值，作为锚位置 ⟨*anchor name*⟩ 的坐标。

\anchorborder{⟨*code*⟩}

```
\let\anchorborder=\pgf@sh@anchorborder
```

\pgf@sh@anchorborder{⟨*code*⟩}

本命令的定义是：

```
\def\pgf@sh@anchorborder#1{\expandafter\gdef\csname pgf@anchor@\pgf@sm@shape@name
↪  @border\endcsname##1{\pgf@process{##1}#1}}%
```

本命令会导致全局地定义控制序列

\csname pgf@anchor@\pgf@sm@shape@name @border\endcsname

```
\expandafter\gdef\csname pgf@anchor@\pgf@sm@shape@name @border\endcsname#1{
↪  \pgf@process{#1}⟨code⟩}
```

可见这个控制序列是个函数，执行这个控制序列时需要给一个参数。

\csname pgf@anchor@\pgf@sm@shape@name @border\endcsname{⟨*arg*⟩}

如前，这个命令能处理一个参数。本命令的目的是计算 shape 的边界上与 ⟨*arg*⟩ 对应的一个点。本命令先用 \pgf@process 处理参数 ⟨*arg*⟩, 为尺寸寄存器 \pgf@x, \pgf@y 赋值；然后再执行 ⟨*code*⟩, 而 ⟨*code*⟩ 应当能做一些计算，再次为寄存器 \pgf@x, \pgf@y 赋值，从而规定一个点，这个点就当作是 shape 的边界 (border) 上的一个点。

所以参数 ⟨*arg*⟩ 是能为 \pgf@process 接受的代码。

\behindbackgroundpath{⟨*code*⟩}

```
\let\behindbackgroundpath=\pgf@sh@behindbgpath
```

\pgf@sh@behindbgpath{⟨*code*⟩}

本命令的定义是：

```
\long\def\pgf@sh@behindbgpath#1{\expandafter\gdef\csname pgf@sh@bbg@
↪  \pgf@sm@shape@name\endcsname{#1}}%
```

执行 \pgf@sh@behindbgpath{⟨*code*⟩} 导致全局地定义

\csname pgf@sh@bbg@\pgf@sm@shape@name\endcsname

```
\expandafter\gdef\csname pgf@sh@bbg@\pgf@sm@shape@name\endcsname{⟨code⟩}
```

⟨*code*⟩ 中的绘图命令就作为 shape 的 behind background path.

\backgroundpath{⟨*code*⟩}

\let\backgroundpath=\pgf@sh@bgpath

```
\long\def\pgf@sh@bgpath#1{\expandafter\gdef\csname pgf@sh@bg@\pgf@sm@shape@name
↪  \endcsname{#1}}%
```

\beforebackgroundpath{⟨*code*⟩}

\let\beforebackgroundpath=\pgf@sh@beforebgpath

```
\long\def\pgf@sh@beforebgpath#1{\expandafter\gdef\csname pgf@sh@fbg@
↪  \pgf@sm@shape@name\endcsname{#1}}%
```

\behindforegroundpath{⟨*code*⟩}

\let\behindforegroundpath=\pgf@sh@behindfgpath

```
\long\def\pgf@sh@behindfgpath#1{\expandafter\gdef\csname pgf@sh@bfg@
↪  \pgf@sm@shape@name\endcsname{#1}}%
```

\foregroundpath{⟨*code*⟩}

\let\foregroundpath=\pgf@sh@fgpath

```
\long\def\pgf@sh@fgpath#1{\expandafter\gdef\csname pgf@sh@fg@\pgf@sm@shape@name
↪  \endcsname{#1}}%
```

\beforeforegroundpath{⟨*code*⟩}

\let\beforeforegroundpath=\pgf@sh@beforefgpath

```
\long\def\pgf@sh@beforefgpath#1{\expandafter\gdef\csname pgf@sh@ffg@
↪  \pgf@sm@shape@name\endcsname{#1}}%
```

\inheritsavedanchors[from=⟨*another shape name*⟩]

\let\inheritsavedanchors=\pgf@sh@inheritsavedanchors

\pgf@sh@inheritsavedanchors[from=⟨*another shape name*⟩]

本命令的定义是：

```
\def\pgf@sh@inheritsavedanchors[from=#1]{%
  \expandafter\pgfutil@g@addto@macro\csname pgf@sh@s@\pgf@sm@shape@name\endcsname{
  ↪  \csname pgf@sh@s@#1\endcsname}}%
```

本命令会导致全局地重定义控制序列

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname

也就是把命令

\csname pgf@sh@s@⟨*another shape name*⟩\endcsname

(不展开这个命令) 添加到这个控制序列的定义内容中。

\inheritanchor[from=⟨*another shape name*⟩]{⟨*anchor name*⟩}

\let\inheritanchor=\pgf@sh@inheritanchor

\pgf@sh@inheritanchor[from=⟨another shape name⟩]{⟨anchor name⟩}

本命令的定义是：

```
\def\pgf@sh@inheritanchor[from=#1]#2{%
  \edef\pgf@marshal{\global\let\expandafter\noexpand\csname
    pgf@anchor@\pgf@sm@shape@name @#2\endcsname=\expandafter\noexpand\csname
    pgf@anchor@#1@#2\endcsname}%
  \pgf@marshal%
}%
```

本命令会全局地重定义控制序列

\csname pgf@anchor@\pgf@sm@shape@name @⟨anchor name⟩\endcsname

使之全局地等于

\csname pgf@anchor@⟨another shape name⟩@⟨anchor name⟩\endcsname

\inheritanchorborder[from=⟨another shape name⟩]

\let\inheritanchorborder=\pgf@sh@inheritanchorborder

```
\def\pgf@sh@inheritanchorborder[from=#1]{%
  \edef\pgf@marshal{\global\let\expandafter\noexpand\csname
    pgf@anchor@\pgf@sm@shape@name @border\endcsname=\expandafter\noexpand\csname
    pgf@anchor@#1@border\endcsname}%
  \pgf@marshal%
}%
```

\inheritbehindbackgroundpath[from=⟨another shape name⟩]

\let\inheritbehindbackgroundpath=\pgf@sh@inheritbehindbgpath

```
\def\pgf@sh@inheritbehindbgpath[from=#1]{\pgf@sh@inheritor{bbg}{#1}}%
```

命令 \pgf@sh@inheritor 的定义是：

```
\def\pgf@sh@inheritor#1#2{%
  \edef\pgf@marshal{\global\let\expandafter\noexpand\csname
    pgf@sh@#1@\pgf@sm@shape@name\endcsname=\expandafter\noexpand\csname
    pgf@sh@#1@#2\endcsname}%
  \pgf@marshal%
}%
```

命令 \pgf@sh@inheritor{⟨something⟩}{⟨another shape name⟩} 会全局地重定义控制序列

\csname pgf@sh@⟨something⟩@\pgf@sm@shape@name\endcsname

使之全局地等于

\csname pgf@sh@⟨something⟩@⟨another shape name⟩\endcsname

\inheritbackgroundpath[from=⟨another shape name⟩]

\let\inheritbackgroundpath=\pgf@sh@inheritbgpath

```
\def\pgf@sh@inheritbgpath[from=#1]{\pgf@sh@inheritor{bg}{#1}}%
```

\inheritbeforebackgroundpath[from=⟨another shape name⟩]

\let\inheritbeforebackgroundpath=\pgf@sh@inheritbeforebgpath

```
\def\pgf@sh@inheritbeforebgpath[from=#1]{\pgf@sh@inheritor{fbg}{#1}}%
```

\inheritbehindforegroundpath[from=⟨another shape name⟩]

\let\inheritbehindforegroundpath=\pgf@sh@inheritbehindfgpath

```
\def\pgf@sh@inheritbehindfgpath[from=#1]{\pgf@sh@inheritor{bfg}{#1}}%
```

\inheritforegroundpath[from=⟨another shape name⟩]

\let\inheritforegroundpath=\pgf@sh@inheritfgpath

```
\def\pgf@sh@inheritfgpath[from=#1]{\pgf@sh@inheritor{fg}{#1}}%
```

\inheritbeforeforegroundpath[from=⟨another shape name⟩]

\let\inheritbeforeforegroundpath=\pgf@sh@inheritbeforefgpath

```
\def\pgf@sh@inheritbeforefgpath[from=#1]{\pgf@sh@inheritor{ffg}{#1}}%
```

\inheritnodeparts[from=⟨another shape name⟩]

\let\inheritnodeparts=\pgf@sh@inheritboxes

```
\def\pgf@sh@inheritboxes[from=#1]{\pgf@sh@inheritor{boxes}{#1}}%
```

25.1.2 形状 rectangle 的定义

在文件《pgfmoduleshapes.code.tex》中有形状 rectangle 的声明，代码如下。首先注意，下面代码中，如果一行代码没有结束就换行，那么在换行处都有注释符号“%”.

```
% Value keys for shapes:
%
% /pgf/inner xsep     : recommended inner x separation
% /pgf/inner ysep     : recommended inner y separation
% /pgf/outer xsep     : recommended outer x separation
% /pgf/outer ysep     : recommended outer y separation
% /pgf/minimum width  : recommended minimum width
% /pgf/minimum height : recommended minimum height

\pgfset{
  inner xsep/.initial     =.3333em,
  inner ysep/.initial     =.3333em,
  inner sep/.style        ={/pgf/inner xsep=#1,/pgf/inner ysep=#1},
  outer xsep/.initial     =.5\pgflinewidth,
  outer ysep/.initial     =.5\pgflinewidth,
  outer sep/.code         =\pgf@handle@outer@sep{#1},
  minimum width/.initial  =1pt,
  minimum height/.initial =1pt,
  minimum size/.style     ={/pgf/minimum width=#1,/pgf/minimum height=#1},
}
```

上面代码声明了几个选项 (key)，其中 outer sep 的值 \pgf@handle@outer@sep{#1} 在下面的代码中定义。

```
\def\pgf@handle@outer@sep#1{%
  \def\pgf@temp{#1}%
  \ifx\pgf@temp\pgf@auto@text%
    \def\pgf@outer@adjust@hook{%
      \pgftransformationadjustments%
      \pgfkeyssetvalue{/pgf/outer xsep}{.5
      ↪  \pgflinewidth*\pgfhorizontaltransformationadjustment}%
      \pgfkeyssetvalue{/pgf/outer ysep}{.5
      ↪  \pgflinewidth*\pgfverticaltransformationadjustment}%
      \pgf@outer@auto@adjust@hook%
    }%
  \else%
    \pgfkeyssetvalue{/pgf/outer xsep}{#1}%
```

```
    \pgfkeyssetvalue{/pgf/outer ysep}{#1}%
  \fi%
}
\def\pgf@auto@text{auto}

\let\pgf@outer@auto@adjust@hook\relax
```

以上代码定义了命令 \pgf@handle@outer@sep{#1}，其中命令 \pgftransformationadjustments→P. 270 与另两个相关的变换的作用是：

- 如果用户给出的选项 outer sep 的值 (即参数 #1 的值) 是 auto，那么就把选项 /pgf/outer xsep 和 /pgf/outer ysep 的值都设为线宽的一半 (即.5\pgflinewidth)；
- 否则就把选项 /pgf/outer xsep 和 /pgf/outer ysep 的值都设为参数 #1 的值。

```
% Keys for rotating the shape border.
% (may not be supported by all shapes)
%
% /pgf/shape border uses incircle : Calculate the shape border using the incircle
%                                   around the node contents (+inner sep).
%
% /pgf/shape border rotate        : Angle of independent border rotation.

\newif\ifpgfshapeborderusesincircle
\pgfkeys{/pgf/shape border uses incircle/.is if=pgfshapeborderusesincircle}
\pgfkeys{/pgf/shape border rotate/.initial=0}
```

上面定义的选项 /pgf/shape border uses incircle 的值是布尔值，会对相应的 if 操作有影响。

```
%
% Rectangle
%

\pgfdeclareshape{rectangle}
{
  \savedanchor\northeast{%
    % Calculate x
    %
    % First, is width < minimum width?
    \pgf@x=\the\wd\pgfnodeparttextbox% 将文字盒子的宽度赋予\pgf@x
    \pgfmathsetlength\pgf@xc{\pgfkeysvalueof{/pgf/inner xsep}}
    ↪  % 将选项/pgf/inner xsep 保存的尺寸赋予\pgf@xc
    \advance\pgf@x by 2\pgf@xc% 将\pgf@x 的值增加 2\pgf@xc
    \pgfmathsetlength\pgf@xb{\pgfkeysvalueof{/pgf/minimum width}}
    ↪  % 将选项/pgf/minimum width 保存的尺寸赋予\pgf@xb
    \ifdim\pgf@x<\pgf@xb% 使用一个 if 语句，如果\pgf@x<\pgf@xb
      % yes, too small. Enlarge...
      \pgf@x=\pgf@xb% 将\pgf@xb 的值赋予\pgf@x，现在\pgf@x 的值等于矩形的总宽度
    \fi% 结束 if 语句
    % Now, calculate right border: .5\wd\pgfnodeparttextbox + .5 \pgf@x + outer sep,
% 计算右侧边界，这里的前提是文字盒子\pgfnodeparttextbox 的左下角位于 shape 坐标系的原点，
% 所以右边界的横标等于“半个矩形宽度 + 半个文字盒子宽度 + outer sep”
    \pgf@x=.5\pgf@x%
    \advance\pgf@x by.5\wd\pgfnodeparttextbox%
    \pgfmathsetlength\pgf@xa{\pgfkeysvalueof{/pgf/outer xsep}}%
    \advance\pgf@x by\pgf@xa% 现在\pgf@x 的值等于矩形右边界的横标
    % Calculate y
    %
    % First, is height+depth < minimum height?
```

```
    \pgf@y=\ht\pgfnodeparttextbox%
    \advance\pgf@y by\dp\pgfnodeparttextbox%
    \pgfmathsetlength\pgf@yc{\pgfkeysvalueof{/pgf/inner ysep}}%
    \advance\pgf@y by 2\pgf@yc%
    \pgfmathsetlength\pgf@yb{\pgfkeysvalueof{/pgf/minimum height}}%
    \ifdim\pgf@y<\pgf@yb%
      % yes, too small. Enlarge...
      \pgf@y=\pgf@yb%
    \fi%
    % Now, calculate upper border: .5\ht-.5\dp + .5 \pgf@y + outer sep
    \pgf@y=.5\pgf@y%
    \advance\pgf@y by-.5\dp\pgfnodeparttextbox%
    \advance\pgf@y by.5\ht\pgfnodeparttextbox%
    \pgfmathsetlength\pgf@ya{\pgfkeysvalueof{/pgf/outer ysep}}%
    \advance\pgf@y by\pgf@ya%
  }
```

上面代码使用命令 `\pgfdeclareshape{rectangle}` 声明形状 `rectangle`, 定义一个 saved anchor, 其位置保存在宏 `\northeast` 中。

```
\savedanchor\southwest{%
    % Calculate x
    %
    % First, is width < minimum width?
    \pgf@x=\wd\pgfnodeparttextbox%
    \pgfmathsetlength\pgf@xc{\pgfkeysvalueof{/pgf/inner xsep}}%
    \advance\pgf@x by 2\pgf@xc%
    \pgfmathsetlength\pgf@xb{\pgfkeysvalueof{/pgf/minimum width}}%
    \ifdim\pgf@x<\pgf@xb%
      % yes, too small. Enlarge...
      \pgf@x=\pgf@xb%
    \fi%
    % Now, calculate left border: .5\wd\pgfnodeparttextbox - .5 \pgf@x - outer sep
    \pgf@x=-.5\pgf@x%
    \advance\pgf@x by.5\wd\pgfnodeparttextbox%
    \pgfmathsetlength\pgf@xa{\pgfkeysvalueof{/pgf/outer xsep}}%
    \advance\pgf@x by-\pgf@xa%
    % Calculate y
    %
    % First, is height+depth < minimum height?
    \pgf@y=\ht\pgfnodeparttextbox%
    \advance\pgf@y by\dp\pgfnodeparttextbox%
    \pgfmathsetlength\pgf@yc{\pgfkeysvalueof{/pgf/inner ysep}}%
    \advance\pgf@y by 2\pgf@yc%
    \pgfmathsetlength\pgf@yb{\pgfkeysvalueof{/pgf/minimum height}}%
    \ifdim\pgf@y<\pgf@yb%
      % yes, too small. Enlarge...
      \pgf@y=\pgf@yb%
    \fi%
     % Now, calculate upper border: .5\ht-.5\dp - .5 \pgf@y - outer sep
    \pgf@y=-.5\pgf@y%
    \advance\pgf@y by-.5\dp\pgfnodeparttextbox%
    \advance\pgf@y by.5\ht\pgfnodeparttextbox%
    \pgfmathsetlength\pgf@ya{\pgfkeysvalueof{/pgf/outer ysep}}%
    \advance\pgf@y by-\pgf@ya%
}
```

上面代码定义一个 saved anchor, 其位置保存在宏 \southwest 中。

```
%
% Anchors
%
\anchor{center}{
  \pgf@process{\northeast}% 引入保存在\northeast 中的\pgf@x 和\pgf@y，并使之全局化
  \pgf@xa=.5\pgf@x%
  \pgf@ya=.5\pgf@y%
  \pgf@process{\southwest}% 引入保存在\southwest 中的\pgf@x 和\pgf@y，并使之全局化
  \pgf@x=.5\pgf@x%
  \pgf@y=.5\pgf@y%
  \advance\pgf@x by \pgf@xa%
  \advance\pgf@y by \pgf@ya%
}
```

上面代码定义一个 normal anchor, 其名称是 center, 实际上是全局定义命令：

```
\gdef\pgf@anchor@rectangle@center{为 \pgf@x, \pgf@y 赋值}
```

然后

```
\anchor{mid}{\pgf@anchor@rectangle@center\pgfmathsetlength\pgf@y{.5ex}}
```

上面代码定义一个名称是 mid 的 normal anchor, 实际上是全局定义命令：

```
\gdef\pgf@anchor@rectangle@mid{\pgf@anchor@rectangle@center\pgfmathsetlength\pgf@y
↪  {.5ex}}
```

由以上代码可见，锚位置 mid 与 center 上下对齐，而 mid 位于文字盒子的底边之上 0.5ex 处。实际上，文字盒子的基线通过锚位置 text, 而锚位置 text 是 shape 坐标系的原点，所以 mid 位于文字盒子的基线 (shape 坐标系的横轴) 之上 0.5ex 处。

```
\anchor{base}{\pgf@anchor@rectangle@center\pgf@y=0pt}
```

上面定义的锚位置 base 位于文字盒子的基线 (shape 坐标系的横轴) 上，与锚位置 center 上下对齐。

```
\anchor{north}{
  \pgf@process{\southwest}%
  \pgf@xa=.5\pgf@x%
  \pgf@process{\northeast}%
  \pgf@x=.5\pgf@x%
  \advance\pgf@x by \pgf@xa%
}
\anchor{south}{
  \pgf@process{\northeast}%
  \pgf@xa=.5\pgf@x%
  \pgf@process{\southwest}%
  \pgf@x=.5\pgf@x%
  \advance\pgf@x by \pgf@xa%
}
\anchor{west}{
  \pgf@process{\northeast}%
  \pgf@ya=.5\pgf@y%
  \pgf@process{\southwest}%
  \pgf@y=.5\pgf@y%
  \advance\pgf@y by \pgf@ya%
}
\anchor{mid west}{\southwest\pgfmathsetlength\pgf@y{.5ex}}
```

```
\anchor{base west}{\southwest\pgf@y=0pt}
\anchor{north west}{
  \southwest
  \pgf@xa=\pgf@x
  \northeast
  \pgf@x=\pgf@xa}
\anchor{south west}{\southwest}
\anchor{east}{%
  \pgf@process{\southwest}%
  \pgf@ya=.5\pgf@y%
  \pgf@process{\northeast}%
  \pgf@y=.5\pgf@y%
  \advance\pgf@y by \pgf@ya%
}
\anchor{mid east}{\northeast\pgfmathsetlength\pgf@y{.5ex}}
\anchor{base east}{\northeast\pgf@y=0pt}
\anchor{north east}{\northeast}
\anchor{south east}{
  \northeast
  \pgf@xa=\pgf@x
  \southwest
  \pgf@x=\pgf@xa
}
```

上面代码定义锚位置 north, south, west, mid west, base west, north west, south west, east, mid east, base east, north east, south east.

```
  \anchorborder{%
    \pgf@xb=\pgf@x% xb/yb is target
% 这里\pgf@x 和\pgf@y 保存的是
% 命令\pgfpointshapeborder{<node>}{<point>} 的参数 <point> 的坐标数据，
% 将坐标数据转存到\pgf@xb 和\pgf@yb 中
    \pgf@yb=\pgf@y%
    \southwest% 引入锚位置\southwest 的坐标数据
    \pgf@xa=\pgf@x% xa/ya is se 将坐标数据转存到\pgf@xa 和\pgf@ya 中
    \pgf@ya=\pgf@y%
    \northeast%
    \advance\pgf@x by-\pgf@xa%
    \advance\pgf@y by-\pgf@ya%
    \pgf@xc=.5\pgf@x% x/y is half width/height 这里\pgf@xc 和\pgf@yc 保存的坐标点是:
% \center - \southwest, 这个点位于中心点的右上方
    \pgf@yc=.5\pgf@y%
    \advance\pgf@xa by\pgf@xc% xa/ya becomes center
    \advance\pgf@ya by\pgf@yc%
    \edef\pgf@marshal{%
      \noexpand\pgfpointborderrectangle
      {\noexpand\pgfqpoint{\the\pgf@xb}{\the\pgf@yb}}
      {\noexpand\pgfqpoint{\the\pgf@xc}{\the\pgf@yc}}%
    }%
    \pgf@process{\pgf@marshal}%
    \advance\pgf@x by\pgf@xa%
    \advance\pgf@y by\pgf@ya%
  }
```

上面代码定义边界上的锚位置。

```
  %
  % Background path
  %
  \backgroundpath{
    \pgfpathrectanglecorners
    {\pgfpointadd{\southwest}{\pgfpoint{\pgfkeysvalueof{/pgf/outer xsep}}{
    ↪  \pgfkeysvalueof{/pgf/outer ysep}}}}
    {\pgfpointadd{\northeast}{\pgfpointscale{-1}{\pgfpoint{\pgfkeysvalueof{/pgf/outer
    ↪  xsep}}{\pgfkeysvalueof{/pgf/outer ysep}}}}}
  }
}
```

上面代码定义背景路径，结束 \pgfdeclareshape{rectangle}{...} 命令。

形状 rectangle 的各个锚位置详见手册的 §71.2(Predefined Shapes). 从以上代码中看出，对形状 rectangle 的锚位置的定义主要用到了赋值运算，加减运算，乘法运算，运算句法主要使用 TEX 的句法，频繁使用命令 \advance...by... 来做加减运算。实际上也可以使用 PGF 的数学引擎来做计算，但是要注意，数学引擎中的函数、命令的定义中很可能使用了尺寸宏 \pgf@x 或 \pgf@y，或 \pgf@xa 等，如果不注意可能会导致奇怪的计算结果。

25.2 创建 node

创建只有一个 node part 的 node 时，直接使用命令 \pgfnode.

创建有多个 node part 的 node 的一般步骤是：

1. 首先挑选一个已定义的 ⟨*shape*⟩, 这个 shape 有 n 个 node part, 也就是说，在声明 ⟨*shape*⟩ 时，使用了

   ```
   \nodeparts{⟨part 1⟩,⟨part 2⟩,...,⟨part n⟩}
   ```

   并且也定义了相应的 anchor 位置

   ```
   \anchor{⟨part 1⟩}{...}
   ...
   \anchor{⟨part n⟩}{...}
   ```

2. 声明盒子

   ```
   \newbox\pgfnodepart⟨part 1⟩box
   ......
   \newbox\pgfnodepart⟨part n⟩box
   ```

   盒子名称中的 ⟨*part i*⟩ 就是 node part 名称。

3. 把文字放到盒子里

   ```
   \setbox\pgfnodepart⟨part 1⟩box=...
   ......
   \setbox\pgfnodepart⟨part n⟩box=...
   ```

4. 然后使用命令 \pgfmultipartnode 即可创建一个 node. 如果有

   ```
   \anchor{⟨part i⟩}{⟨point⟩}
   ```

   这个命令会把盒子 \pgfnodepart⟨*part i*⟩box 的基点放到 node 坐标系中的 ⟨*point*⟩ 点上。

25.2.1 命令 \pgfmultipartnode

命令 \pgfmultipartnode 的定义是：

```
\def\pgfmultipartnode#1#2#3#4{%
  \pgfutil@ifundefined{pgf@sh@s@#1}%
  {\pgferror{Unknown shape ``#1''}}%
  {%
    {%
      \ifpgflatenodepositioning%
        \pgfsys@beginscope%
      \fi%
      \pgf@outer@adjust@hook%
      \let\pgf@sh@savedmacros=\pgfutil@empty% MW
      \let\pgf@sh@savedpoints=\pgfutil@empty%
      \def\pgf@sm@shape@name{#1}% CJ % TT added prefix!
      \csname pgf@sh@s@#1\endcsname%
      \pgf@sh@savedpoints%
      \pgf@sh@savedmacros% MW
      \pgftransformshift{%
        \pgf@sh@reanchor{#1}{#2}%
        \pgf@x=-\pgf@x%
        \pgf@y=-\pgf@y%
      }%
      \expandafter\pgfsavepgf@process\csname pgf@sh@sa@#3\endcsname{%
        \pgf@sh@reanchor{#1}{#2}% FIXME : this is double work!
      }%
      % Save the saved points and the transformation matrix
      \edef\pgf@node@name{#3}%
      \ifpgflatenodepositioning%
        \pgf@shapes@late@pos@begin%
      \fi%
      \ifx\pgf@node@name\pgfutil@empty%
      \else%
        \expandafter\xdef\csname pgf@sh@ns@\pgf@node@name\endcsname{#1}%
        \edef\pgf@sh@@temp{\noexpand\gdef\expandafter\noexpand\csname pgf@sh@np@
        ↪  \pgf@node@name\endcsname}%
        \expandafter\pgf@sh@@temp\expandafter{\pgf@sh@savedpoints}%
        \edef\pgf@sh@@temp{\noexpand\gdef\expandafter\noexpand\csname pgf@sh@ma@
        ↪  \pgf@node@name\endcsname}% MW
        \expandafter\pgf@sh@@temp\expandafter{\pgf@sh@savedmacros}% MW
        \pgfgettransform\pgf@temp%
        \expandafter\xdef\csname pgf@sh@nt@\pgf@node@name\endcsname{\pgf@temp}%
        \expandafter\xdef\csname pgf@sh@pi@\pgf@node@name\endcsname{\pgfpictureid}%
      \fi%
      \pgfutil@ifundefined{pgf@sh@bbg@#1}{}
      {{\pgfusetype{.behind background}\pgfidscope\pgfscope\csname pgf@sh@bbg@#1
      ↪  \endcsname\endpgfscope\endpgfidscope}}%
      \pgfutil@ifundefined{pgf@sh@bg@#1}{%
        \global\let\pgfpositionnodelaterpath\pgfutil@empty%
      }%
      {\pgfpushtype%
       \pgfusetype{.background}\csname pgf@sh@bg@#1\endcsname%
        \ifpgflatenodepositioning%
          \pgfsyssoftpath@getcurrentpath\pgfpositionnodelaterpath%
          \pgfprocessround{\pgfpositionnodelaterpath}{\pgfpositionnodelaterpath}%
          \global\let\pgfpositionnodelaterpath\pgfpositionnodelaterpath%
        \fi%
        #4\pgfpoptype}%
      \pgfutil@ifundefined{pgf@sh@fbg@#1}{}
```

```
      {{\pgfusetype{.before background}\pgfidscope\pgfscope\csname pgf@sh@fbg@#1
      ↪  \endcsname\endpgfscope\endpgfidscope}}%
      {%
        \expandafter\pgfutil@for\expandafter\pgf@shape@com\expandafter:\expandafter=
        ↪  \csname pgf@sh@boxes@#1\endcsname\do{%
          {%
            \pgfusetype{.\pgf@shape@com}%
            \pgftransformshift{\pgf@sh@reanchor{#1}{\pgf@shape@com}}%
            \pgfapproximatenonlineartransformation%
            \expandafter\pgfqboxsynced\expandafter{\csname pgfnodepart\pgf@shape@com
            ↪  box\endcsname}%
          }%
        }%
      }%
      \pgfutil@ifundefined{pgf@sh@bfg@#1}{}%
      {{\pgfusetype{.behind foreground}\pgfidscope\pgfscope\csname pgf@sh@bfg@#1
      ↪  \endcsname\endpgfscope\endpgfidscope}}%
      \pgfutil@ifundefined{pgf@sh@fg@#1}{}%
      {\pgfpushtype\pgfusetype{.foreground}\csname pgf@sh@fg@#1\endcsname#4\pgfpoptype
      ↪  }%
      \pgfutil@ifundefined{pgf@sh@ffg@#1}{}%
      {{\pgfusetype{.before foreground}\pgfidscope\pgfscope\csname pgf@sh@ffg@#1
      ↪  \endcsname\endpgfscope\endpgfidscope}}%
      \ifpgflatenodepositioning%
        \pgf@shapes@late@pos@end%
        \pgfsys@endscope%
      \else%
        \expandafter\pgf@nodecallback\expandafter{\pgf@node@name}%
      \fi%
    }%
  }%
}%
\let\pgf@outer@adjust@hook\relax
```

命令 `\pgfmultipartnode{`⟨*shape*⟩`}{`⟨*anchor*⟩`}{`⟨*name*⟩`}{`⟨*path usage command*⟩`}` 的处理过程如下。首先执行命令

```
\pgfutil@ifundefined{pgf@sh@s@⟨shape⟩}%
```

检查名称为 `pgf@sh@s@`⟨*shape*⟩ 的命令是否已经定义 (见前文)。如果未定义，则执行 `\pgferror{Unknown shape ```⟨*shape*⟩`''}`, 给出错误信息；如果已定义，则继续下面的步骤：

1. 用 `{` 开启一个 TEX 组。
2. 检查 `\ifpgflatenodepositioning` 的真值：
   - 如果它的真值是 true, 则执行 `\pgfsys@beginscope` 开启一个 scope, 这个 scope 将在 21b 那里结束。
   - 如果它的真值是 false, 则什么也不做。
3. 执行 `\pgf@outer@adjust@hook`, 这个命令是命令 `\pgf@handle@outer@sep` 的子命令，也就是说，在执行命令 `\pgf@handle@outer@sep` 的过程中会定义命令 `\pgf@outer@adjust@hook`, 否则命令 `\pgf@outer@adjust@hook` 等于 `\relax`.

`\pgf@handle@outer@sep=`⟨*dimension*⟩`|auto`

命令 `\pgf@handle@outer@sep` 的定义是：

```
\def\pgf@handle@outer@sep#1{%
  \def\pgf@temp{#1}%
  \ifx\pgf@temp\pgf@auto@text%
    \def\pgf@outer@adjust@hook{%
      \pgftransformationadjustments%
      \pgfkeyssetvalue{/pgf/outer xsep}{.5
      ↪  \pgflinewidth*\pgfhorizontaltransformationadjustment}%
      \pgfkeyssetvalue{/pgf/outer ysep}{.5
      ↪  \pgflinewidth*\pgfverticaltransformationadjustment}%
      \pgf@outer@auto@adjust@hook%
    }%
  \else%
    \pgfkeyssetvalue{/pgf/outer xsep}{#1}%
    \pgfkeyssetvalue{/pgf/outer ysep}{#1}%
  \fi%
}%
\def\pgf@auto@text{auto}%

\let\pgf@outer@auto@adjust@hook\relax
```

可见仅当执行 \pgf@handle@outer@sep{auto} 时，命令 \pgf@outer@adjust@hook 才是命令 \pgf@handle@outer@sep 的子命令，此时执行 \pgf@outer@adjust@hook 的效果是把选项 (键) /pgf/outer xsep 和 /pgf/outer ysep 的值设为线宽的一半 (并且排除伸缩变换对尺寸的影响)。

如果执行的是 \pgf@handle@outer@sep{⟨尺寸⟩}, 那么 \pgf@outer@adjust@hook 就等于 \relax. 此时直接把选项 (键) /pgf/outer xsep 和 /pgf/outer ysep 的值设为 ⟨尺寸⟩.

4. 规定

```
\let\pgf@sh@savedmacros=\pgfutil@empty%
\let\pgf@sh@savedpoints=\pgfutil@empty%
```

这是定义命令 \pgf@sh@savedmacros 和 \pgf@sh@savedpoints 的初始状态，因为在后面步骤中需要这两个命令是“已定义的”。

5. 定义 \def\pgf@sm@shape@name{⟨*shape*⟩}

6. 执行命令 \csname pgf@sh@s@⟨*shape*⟩\endcsname, 这个命令是被全局定义的。

   参考 \savedanchor→P. 438, \pgf@sh@savedanchor→P. 438.

7. 执行 \pgf@sh@savedpoints→P. 440, 定义各个 saved anchor.

8. 执行命令 \pgf@sh@savedmacros→P. 442, 定义各个 saved macro.

9. 执行平移命令

```
\pgftransformshift{%
  \pgf@sh@reanchor{⟨shape⟩}{⟨anchor⟩}%
  \pgf@x=-\pgf@x%
  \pgf@y=-\pgf@y%
}%
```

\pgf@sh@reanchor{⟨*shape*⟩}{⟨*anchor*⟩}

命令 \pgf@sh@reanchor 的定义是：

```
\def\pgf@sh@reanchor#1#2{%
  \pgfutil@ifundefined{pgf@anchor@#1@#2}%
  {%
    \pgfutil@ifundefined{pgf@anchor@generic@#2}{%
        \pgfmathsetcounter{pgf@counta}{#2}%
        \csname pgf@anchor@#1@border\endcsname{\pgfqpointpolar{
        ↪  \the\c@pgf@counta}{1pt}}%
    }{%
        \csname pgf@anchor@generic@#2\endcsname{#1}%
    }%
  }%
  {\csname pgf@anchor@#1@#2\endcsname}%
}%
```

执行 \pgf@sh@reanchor{⟨*shape*⟩}{⟨*anchor*⟩} 导致的处理是：

(a) 如果名称为 pgf@anchor@⟨*shape*⟩@⟨*anchor*⟩ 的命令已定义，则执行之。

(b) 否则，如果名称为 pgf@anchor@generic@⟨*anchor*⟩ 的命令已定义，则执行之，此命令在执行 \pgfdeclaregenericanchor 时被定义：

```
\def\pgfdeclaregenericanchor#1#2{%
  \expandafter\def\csname pgf@anchor@generic@#1\endcsname##1{#2}%
}%
```

(c) 否则，

i. 规定计数器值 \pgfmathsetcounter{pgf@counta}{⟨*anchor*⟩}.

ii. 执行

```
\csname pgf@anchor@⟨shape⟩@border\endcsname{\pgfqpointpolar{\the
↪  \c@pgf@counta}{1pt}}
```

也就是计算，例如 a.60 这种与角度对应的 shape 边界上的点，注意角度是计数器的值是整数。

最后的执行结果应该是给寄存器 \pgf@x 与 \pgf@y 赋值 (未必是全局的赋值) 的句子，这两个寄存器值对应 ⟨*anchor*⟩ 位置，换句话说，执行 \pgf@sh@reanchor{⟨*shape*⟩}{⟨*anchor*⟩} 就是把保存在

\csname pgf@anchor@⟨*shape*⟩@⟨*anchor*⟩\endcsname 或

\csname pgf@anchor@generic@⟨*anchor*⟩\endcsname 或

\csname pgf@anchor@⟨*shape*⟩@border\endcsname

中的代码执行了，从而得到锚位置 ⟨*anchor*⟩ 的坐标 (保存在 \pgf@x, \pgf@y 中)。

在这一步执行 \pgftransformshift 得到两个结果：(1) 改变变换矩阵，把 ⟨*anchor name*⟩ 对应的向量的负向量用作平移向量 (将用于平移 shape, 使它的 ⟨*anchor name*⟩ 位置位于锚定点上)；(2) 还把这个负向量做了全局的声明，即全局地把 \pgf@x, \pgf@y 的值变成这个负向量的分量。这两个结果就是参数 ⟨*anchor*⟩ 的全部意义：因为保存在

\csname pgf@anchor@⟨*shape*⟩@⟨*anchor*⟩\endcsname

中的代码是在一个花括号组内被执行的 (参照 \pgftransformshift 的定义)，除了这两个结果外，没有其他副作用。

10. 执行

```
\expandafter\pgfsavepgf@process\csname pgf@sh@sa@⟨name⟩\endcsname{%
  \pgf@sh@reanchor{⟨shape⟩}{⟨anchor⟩}% FIXME : this is double work!
}%
```

在文件《pgfcorepoints.code.tex》中有 \pgfsavepgf@process 的定义：

```
\def\pgfextract@process#1#2{%
    \pgf@process{#2}%
    \edef#1{\noexpand\global\pgf@x=\the\pgf@x
    ↪  \noexpand\relax\noexpand\global\pgf@y=\the\pgf@y\noexpand\relax}%
}
% This needed until old shapes code changed.
\let\pgfsavepgf@process\pgfextract@process%
```

执行此命令的结果是：将 \pgf@sh@reanchor{⟨*shape*⟩}{⟨*anchor*⟩} 得到的 \pgf@x 与 \pgf@y 的值 (两个尺寸) 全局化, 然后定义命令

\csname pgf@sh@sa@⟨*name*⟩\endcsname

此命令的内容是：“将这两个尺寸全局化地保存在 \pgf@x 与 \pgf@y 中”。

11. 将 node 名称 ⟨*name*⟩ 保存在 \pgf@node@name 中。

12. 检查 \ifpgflatenodepositioning 的真值

    - 如果有 \pgflatenodepositioningtrue, 则执行 \pgf@shapes@late@pos@begin, 与此命令对应的 \pgf@shapes@late@pos@end 在 21a 那里。
    - 如果有 \pgflatenodepositioningfalse, 则什么也不做。

13. 执行 \ifx, 检查 \pgf@node@name 与 \pgfutil@empty 的定义是否相同，即检查是否给出了 node 名称。

    - 如果相同，则什么也不做。
    - 如果不同，则

      (a) 全局地定义

      \csname pgf@sh@ns@\pgf@node@name\endcsname

      为 ⟨*xdef* 展开的 ⟨*shape*⟩⟩, 将 node name 与 shape 名称联系起来。

      (b) 全局地定义

      \csname pgf@sh@np@\pgf@node@name\endcsname

      为 ⟨*exp* 展开的 *\pgf@sh@savedpoints* → P. 440⟩.

      (c) 全局地定义

      \csname pgf@sh@ma@\pgf@node@name\endcsname

      为 ⟨*exp* 展开的 *\pgf@sh@savedmacros* → P. 442⟩.

      (d) 将当前的变换矩阵保存在 \pgf@temp 中。

      (e) 全局地定义

      \csname pgf@sh@nt@\pgf@node@name\endcsname

      为 ⟨*xdef* 展开的 *\pgf@temp*⟩，即保存当前的变换矩阵。

      (f) 全局地定义

      \csname pgf@sh@pi@\pgf@node@name\endcsname

      为 ⟨*xdef* 展开的 *\pgfpictureid*⟩.

      命令 \pgfpictureid 是命令 \pgfpicture 的子命令，见《pgfcorescopes.code.tex》.

14. 执行 \pgfutil@ifundefined, 检查名称为 pgf@sh@bbg@⟨*shape*⟩ 的命令是否已定义。

    参考 \behindbackgroundpath → P. 445, \pgf@sh@behindbgpath → P. 445.

- 如果未定义则什么也不做。
- 如果已定义，则设置一个花括号组，在这个组内执行

```
\pgfusetype{.behind background}\pgfidscope\pgfscope\csname pgf@sh@bbg@⟨shape⟩
↪  \endcsname\endpgfscope\endpgfidscope
```

命令 \pgfusetype 注册并启用类型“.behind background”；命令 \pgfidscope 开启一个 id scope，命令 \pgfscope 开启一个 graph scope，这个 graph scope 的 type 是“.behind background”，这个 graph scope 的内容就是命令

\csname pgf@sh@bbg@⟨*shape*⟩\endcsname

中保存的绘图命令 (见前文)。

命令 \pgfusetype, \pgfidscope 参考《pgfcorescopes.code.tex》.

15. 执行 \pgfutil@ifundefined, 检查名称为 pgf@sh@bg@⟨*shape*⟩ 的命令是否已定义。

- 如果未定义，则

```
\global\let\pgfpositionnodelaterpath\pgfutil@empty%
```

- 如果已定义，则

(a) 执行 \pgfpushtype.

(b) 执行 \pgfusetype{.background}\csname pgf@sh@bg@⟨*shape*⟩\endcsname

(c) 检查 \ifpgflatenodepositioning 的真值：

– 如果有 \pgflatenodepositioningtrue, 则

i. 执行

```
\pgfsyssoftpath@getcurrentpath\pgfpositionnodelaterpath
```

定义命令 \pgfpositionnodelaterpath, 把当前的软路径保存到这个命令中。

ii. 执行

```
\pgfprocessround{\pgfpositionnodelaterpath}{\pgfpositionnodelaterpath
↪  }
\global\let\pgfpositionnodelaterpath\pgfpositionnodelaterpath
```

命令 \pgfprocessround 的定义见文件《pgfcorepathprocessing.code.tex》：

– 如果有 \pgflatenodepositioningfalse, 则什么也不做。

(d) 执行 ⟨*path usage command*⟩.

(e) 执行 \pgfpoptype.

16. 执行 \pgfutil@ifundefined, 检查名称为 pgf@sh@fbg@⟨*shape*⟩ 的命令是否已定义。

- 如果未定义，则无动作。
- 如果已定义，则设置一个花括号组，在组内执行

```
\pgfusetype{.before background}\pgfidscope\pgfscope\csname pgf@sh@fbg@#1
↪  \endcsname\endpgfscope\endpgfidscope
```

17. 设置一个花括号组，在组内执行一个 for 循环，即 \pgfutil@for...\do{...}, 这个循环把各个 (已经装入文字的) 文字盒子放到相应的位置上。各个单次的循环也是一个组：

```
{%
  \pgfusetype{.\pgf@shape@com}%
  \pgftransformshift{\pgf@sh@reanchor→P. 456{⟨shape⟩}{⟨node part name⟩}}%
```

```
  \pgfapproximatenonlineartransformation%
  \expandafter\pgfqboxsynced\expandafter{\csname pgfnodepart\pgf@shape@com box
  ↪  \endcsname}%
}%
```

在这个组内，先设置平移变换、非线性变换的线性近似，再插入名称为 `\pgfnodepart`⟨*node part name*⟩`box` 的盒子。这里的 ⟨*node part name*⟩ 必须是已经用命令 `\anchor{`⟨*node part name*⟩`}` 规定的锚位置。

18. 执行 `\pgfutil@ifundefined`, 检查名称为 `pgf@sh@bfg@`⟨*shape*⟩ 的命令是否已定义……
19. 执行 `\pgfutil@ifundefined`, 检查名称为 `pgf@sh@fg@`⟨*shape*⟩ 的命令是否已定义……
20. 执行 `\pgfutil@ifundefined`, 检查名称为 `pgf@sh@ffg@`⟨*shape*⟩ 的命令是否已定义……
21. 检查 `\ifpgflatenodepositioning` 的真值：
    - 如果 `\pgflatenodepositioningtrue`, 则
      (a) 执行 `\pgf@shapes@late@pos@end`
      (b) 执行 `\pgfsys@endscope`, 此命令对应前面步骤 2 中的 `\pgfsys@beginscope`.
    - 如果 `\pgflatenodepositioningfalse` 则执行

      ```
      \expandafter\pgf@nodecallback\expandafter{\pgf@node@name}
      ```

22. 用 `}` 结束花括号分组。

注意由步骤 1 和步骤 22 设置了一个花括号组，命令 `\pgfmultipartnode` 的处理过程基本上都处于这个组内。

25.2.1.1 真值 `\pgflatenodepositioningtrue` 对命令 `\pgfmultipartnode` 的影响

如果在 `\pgflatenodepositioningtrue` 之下执行命令

`\pgfmultipartnode{`⟨*shape*⟩`}{`⟨*anchor*⟩`}{`⟨*name*⟩`}{`⟨*path usage command*⟩`}`

那么：

- 从步骤 3 开始，直到步骤 21a 都会被放入 scope 环境。
- 从步骤 13 开始，直到步骤 20 都会被“另存”。

步骤 12 执行 `\pgf@shapes@late@pos@begin`.

`\pgf@shapes@late@pos@begin`

此命令的定义是：

```
\def\pgf@shapes@late@pos@begin{%
  % Rename node
  \edef\pgf@node@name{not yet positionedPGFINTERNAL\pgf@node@name}%
  % Interrupt bounding box!
  \pgfinterruptboundingbox%
    % Put everything in our box:
    \pgf@relevantforpicturesizetrue%
    \setbox\pgfpositionnodelaterbox=\hbox%
      \bgroup%
        \pgfsys@beginscope%
}%
```

它的处理是：

1. 重定义宏 \pgf@node@name, 也就是把 node 名称修改为 not yet positionedPGFINTERNAL⟨name⟩
2. 执行 \pgfinterruptboundingbox 中断环境。
3. 设置真值 \pgf@relevantforpicturesizetrue, 开始计算边界盒子。
4. 使用命令 \setbox 定义盒子 \pgfpositionnodelaterbox, 因为 PGF 自己定义

```
\def\pgfpositionnodelaterbox{0}%
```

   所以这个盒子的编号是 0.
5. 执行 \bgroup 开启一个组，此后直到步骤 21a 都是盒子 \pgfpositionnodelaterbox 的内容。
6. 执行 \pgfsys@beginscope 开启一个 scope 环境。此后直到步骤 20 的内容都处于这个 scope 环境中。

步骤 21a 执行 \pgf@shapes@late@pos@end.

\pgf@shapes@late@pos@end

此命令的定义是：

```
\def\pgf@shapes@late@pos@end{%
      \pgfsys@endscope%
    \egroup% Close box
  \ifdim\pgf@picminx>\pgf@picmaxx\relax% happens for empty nodes
    \def\pgfpositionnodelaterminx{0.0pt}%
    \let\pgfpositionnodelaterminy\pgfpositionnodelaterminx%
    \let\pgfpositionnodelatermaxx\pgfpositionnodelaterminx%
    \let\pgfpositionnodelatermaxy\pgfpositionnodelaterminx%
  \else%
    \edef\pgfpositionnodelaterminx{\the\pgf@picminx}%
    \edef\pgfpositionnodelaterminy{\the\pgf@picminy}%
    \edef\pgfpositionnodelatermaxx{\the\pgf@picmaxx}%
    \edef\pgfpositionnodelatermaxy{\the\pgf@picmaxy}%
  \fi%
  \let\pgfpositionnodelatername=\pgf@node@name%
  \pgf@positionnodelater@macro%
 \endpgfinterruptboundingbox%
}%
```

这个命令的处理是：

1. 执行 \pgfsys@endscope, 结束由 \pgf@shapes@late@pos@begin 开启的 scope 环境。
2. 执行 \egroup, 结束盒子 \pgfpositionnodelaterbox 的内容。也就是说，这个盒子的内容是个 scope 环境。
3. 定义宏
   - \pgfpositionnodelaterminx
   - \pgfpositionnodelaterminy
   - \pgfpositionnodelatermaxx
   - \pgfpositionnodelatermaxy

   分别保存盒子的上下左右界限 (用尺寸表示的坐标)，盒子的锚定点是原点。
4. 执行 \let\pgfpositionnodelatername=\pgf@node@name.
5. 执行 \pgf@positionnodelater@macro, 在命令 \pgfpositionnodelater 的定义中有：

```
\def\pgfpositionnodelater#1{%
  \let\pgf@positionnodelater@macro=#1%
  \ifx\pgf@positionnodelater@macro\relax%
    \pgflatenodepositioningfalse%
```

```
  \else%
    \pgflatenodepositioningtrue%
  \fi%
}%
\newif\ifpgflatenodepositioning
\pgfpositionnodelater{\relax}%
```

可见 \pgf@positionnodelater@macro 等于命令 \pgfpositionnodelater{⟨*macro name*⟩} 的参数，如果不事先自定义 ⟨*macro name*⟩, 那么就等于 \relax.

6. 执行 \endpgfinterruptboundingbox, 恢复由 \pgf@shapes@late@pos@begin 中断的环境。

注意从步骤 3 到步骤 11 的内容不被放入盒子 \pgfpositionnodelaterbox 中，此盒子保存的是 node 的 saved anchor, saved dimen, saved macro 的定义，以及各种背景路径，文字内容等。

由步骤 2 和步骤 21b 设置了一个 scope 环境，以上处理都限制在这个 scope 环境内，所以：

- 宏\pgfpositionnodelaterminx, \pgfpositionnodelaterminy, \pgfpositionnodelatermaxx,\pgfpositionnodelatermaxy 的定义是受 scope 环境限制的，为了能在 \pgfmultipartnode 之后利用这 4 个宏的值，命令 \pgf@positionnodelater@macro 应该能全局地转存这 4 个宏的值。
- 盒子 \pgfpositionnodelaterbox 的定义是受 scope 环境限制的，为了能在 \pgfmultipartnode 之后利用这个盒子，命令 \pgf@positionnodelater@macro 应该能全局地转存这个盒子。
- 宏 \pgfpositionnodelatername 的定义是受 scope 环境限制的，它保存 node 名称 not yet positionedPGFINTERNAL⟨*name*⟩, 为了能在 \pgfmultipartnode 之后利用这个名称，命令 \pgf@positionnodelater@macro 应该能全局地转存这个名称。

另外在步骤 15c 中，还全局地定义了宏 \pgfpositionnodelaterpath, 其中保存的是 node 的背景路径 (bg) 的软路径形式。

\pgf@shift@node{⟨*node name*⟩}{⟨*point*⟩}

参数 ⟨*node name*⟩ 是某个已创建的 node 名称。⟨*point*⟩ 是 PGF 点，或者是为 \pgf@x, \pgf@y 赋值的代码。

本命令的定义是：

```
\def\pgf@shift@node#1#2{%
  % This internal command shifts the recorded coordinates for node #1
  % by the vector #2. It is used to
  % correct the position of the node if the recorded coordinate
  % happens to be wrong
  {%
    \pgfsettransform{\csname pgf@sh@nt@#1\endcsname}%
    \pgf@process{#2}%
    \advance\pgf@pt@x by\pgf@x%
    \advance\pgf@pt@y by\pgf@y%
    \pgfgettransform{\pgf@temp}%
    \expandafter\xdef\csname pgf@sh@nt@#1\endcsname{\pgf@temp}%
  }%
}%
```

在创建 ⟨*node name*⟩ 时 (前) 的变换矩阵会被保存到控制序列

\csname pgf@sh@nt@⟨*node name*⟩\endcsname

中。本命令修改这个控制序列保存的矩阵中的平移元素，将 ⟨*point*⟩ 作为一个平移向量加到原来的平移元素上。

25.2.1.2 命令 \pgfnode

\pgfnode{⟨*shape*⟩}{⟨*anchor*⟩}{⟨*label text*⟩}{⟨*name*⟩}{⟨*path usage command*⟩}

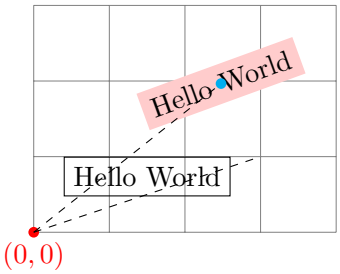
这个命令创建只有一个 node part (名称为默认名称 text) 的 node.

其中 ⟨*shape*⟩ 是某个已经用命令 \pgfdeclareshape 定义的 shape 的名称，⟨*anchor*⟩ 是 ⟨*shape*⟩ 的一个锚位置。本命令会把锚位置 ⟨*anchor*⟩ 放在原点上，如果你想把锚位置 ⟨*anchor*⟩ 放在其它点上，就需要在本命令之前使用坐标变换命令。

⟨*label text*⟩ 是 node 的文字，文字会被放入名称为 \pgfnodeparttextbox 的 TEX 盒子中.

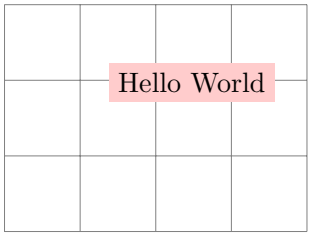
⟨*name*⟩ 是所创建的 node 的名称，用于之后索引该 node. 如果没有 ⟨*name*⟩, 那么在画出该 node 之后，程序就会“忘记”这个 node.

⟨*path usage command*⟩ 是使用路径的命令，对 background path 或者 foreground path 进行操作，例如，画出路径、填充路径等。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,3);
  \fill [red] circle (2pt) node [below] {$(0,0)$};
  {
    \pgftransformshift{\pgfpoint{1.5cm}{1cm}}
    \pgfnode{rectangle}{north}{Hello World}{hellonode}{\pgfusepath{stroke}}
  }
  {
    \color{red!20}
    \pgftransformrotate{20}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgfnode{rectangle}{center}
      {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
  }
  \fill [cyan] (hellonode.center) circle (2pt);
  \draw [dashed](0,0)--(3,1);
  \draw [dashed,rotate=20] (0,0)--(3,1);
\end{tikzpicture}
```

从上面的例子看出，坐标变换对 shape 和文字都有效。默认 node 的锚定点是原点，当用坐标变换改变 node 的锚定点的位置时，node 的位置也会随之变化。如果还有旋转变换，node 在整体上也会被旋转。如果不希望旋转变换对 node 起作用，就需要在命令 \pgfnode 之前使用命令 \pgftransformresetnontranslations.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,3);
  {
    \color{red!20}
    \pgftransformrotate{20}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgftransformresetnontranslations
    \pgfnode{rectangle}{center}
      {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
  }
\end{tikzpicture}
```

在《pgfmoduleshapes.code.tex》中有定义：

```
\newbox\pgfnodeparttextbox
......
\def\pgfnode#1#2#3#4#5{%
  {%
    \setbox\pgfnodeparttextbox=\hbox%
    {%
      \pgfinterruptpicture%
        {#3}%
      \endpgfinterruptpicture%
    }%
    \pgfmultipartnode{#1}{#2}{#4}{#5}%
  }
}%
```

可见本命令的所有操作都放在一个花括号组内。本命令先把 ⟨*label text*⟩ 放入名称为 \pgfnodeparttextbox 的左右盒子中，这样一来，\wd\pgfnodeparttextbox, \ht\pgfnodeparttextbox, \dp\pgfnodeparttextbox 都是实际可用的尺寸。

25.2.1.3 其他命令

\pgfcoordinate{⟨*name*⟩}{⟨*coordinate*⟩}

这个命令在坐标点 ⟨*coordinate*⟩ 处，创建一个形状为 coordinate 的、名称为 ⟨*name*⟩ 的 node.

\pgfnodealias{⟨*new name*⟩}{⟨*existing node*⟩}

⟨*existing node*⟩ 是某个已创建的 node 的名称，本命令为该 node 再设置一个名称 ⟨*new name*⟩, 也就是说，该 node 有两个名称，任何一个都可以用来索引该 node. 别名是被全局地定义的。

\pgfnoderename{⟨*new name*⟩}{⟨*existing node*⟩}

⟨*existing node*⟩ 是某个已创建的 node 的名称，本命令为该 node 重命名，即修改其名称为 ⟨*new name*⟩, 废弃原来的名称 ⟨*existing node*⟩(不再有效)。这个名称修改是全局地。

25.2.2 关于预定义 node 的键

```
\pgfset{
  inner xsep/.initial     =.3333em,
  inner ysep/.initial     =.3333em,
  inner sep/.style        ={/pgf/inner xsep=#1,/pgf/inner ysep=#1},
  outer xsep/.initial     =.5\pgflinewidth,
  outer ysep/.initial     =.5\pgflinewidth,
  outer sep/.code         =\pgf@handle@outer@sep{#1},
  minimum width/.initial  =1pt,
  minimum height/.initial =1pt,
  minimum size/.style     ={/pgf/minimum width=#1,/pgf/minimum height=#1},
}%
```

/pgf/minimum width=⟨*dimension*⟩ (no default, initially 1pt)

/tikz/minimum width=⟨*dimension*⟩

这个选项设置 node 的最小宽度，即 node 的实际宽度可以大于但不能小于 ⟨*dimension*⟩.

注意这个选项的初始值是 1pt，并且只是个“推荐值”，在某些情况下这个选项值可能会被忽略。所谓“推荐值”实际指的是“初始值”，这个值是用手柄 /.initial 规定的。


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,1);
  \pgfset{minimum width=3cm}
  \pgfnode{rectangle}{south west}{\color{cyan} Hello World}{}
    {\pgfsetstrokecolor{red} \pgfusepath{stroke}}
\end{tikzpicture}
```

/pgf/minimum height=⟨*dimension*⟩ (no default, initially `1pt`)

/tikz/minimum height=⟨*dimension*⟩

这个选项设置 node 的最小高度。这个选项值只是个“推荐值”。

/pgf/minimum size=⟨*dimension*⟩ (no default)

/tikz/minimum size=⟨*dimension*⟩

本选项同时设置 `/pgf/minimum width` 和 `/pgf/minimum height` 的值为 ⟨*dimension*⟩.

/pgf/inner xsep=⟨*dimension*⟩ (no default, initially `0.3333em`)

/tikz/inner xsep=⟨*dimension*⟩

这个选项在水平方向上，设置 node 的背景形状路径与文字的间距为 ⟨*dimension*⟩, 这个选项值只是个“推荐值”，在某些情况下这个选项可能会被忽略。

注意，这里的间距指的是路径线条的中心与文字的间距，而不是路径线条的外缘与文字的间距。

/pgf/inner ysep=⟨*dimension*⟩ (no default, initially `0.3333em`)

/tikz/inner ysep=⟨*dimension*⟩

这个选项在竖直方向上，设置 node 的边界形状路径与文字的间距为 ⟨*dimension*⟩, 这个选项值只是个“推荐值”。

/pgf/inner sep=⟨*dimension*⟩ (no default)

/tikz/inner sep=⟨*dimension*⟩

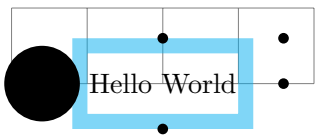
本选项同时设置 `/pgf/inner xsep` 和 `/pgf/inner ysep` 的值为 ⟨*dimension*⟩.

/pgf/outer xsep=⟨*dimension*⟩ (no default, initially `.5\pgflinewidth`)

/tikz/outer xsep=⟨*dimension*⟩

这个选项在水平方向上，设置 node 的背景边界形状路径与“外部锚位置”的间距。例如，如果 ⟨*dimension*⟩ 是 1cm，那么锚位置 east 与背景形状路径的边界的距离就是 1cm. 这个选项值只是个“推荐值”。

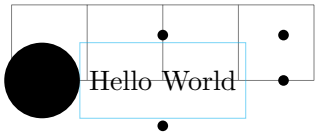
注意，这里的间距指的是路径线条的中心与“外部锚位置”的间距，而不是路径线条的外缘与“外部锚位置”的间距。本选项的初始值是 `0.5\pgflinewidth`，这恰好使得“外部锚位置”处于路径线条的外缘上。




```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);
  \tikzset{line width=2mm}  % 这个选项是 tikz 的选项，修改 \pgflinewidth 的值，
```

```
  \pgfset{minimum height=1cm, outer xsep=.5cm}
  \pgfnode{rectangle}{center}{Hello World}{x}
    {
      \pgfsetlinewidth{2mm}  % 规定 node 形状路径线条的线宽为 2mm，否则其线宽为默认值 0.4pt
      \pgfsetstrokecolor{cyan}
      \pgfsetstrokeopacity{0.5}
      \pgfusepath{stroke}
    }
  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{.5cm}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

将上面例子中的 \pgfsetlinewidth{2mm} 注释掉，得到下面的图形，注意比较 node 的形状路径线条的线宽：




```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);
  \tikzset{line width=2mm}  % 这个选项是 tikz 的选项，修改 \pgflinewidth 的值，
  \pgfset{minimum height=1cm, outer xsep=.5cm}
  \pgfnode{rectangle}{center}{Hello World}{x}
    {
%     \pgfsetlinewidth{2mm}  % 注释掉这一命令，此时 node 形状路径线条的线宽为默认值 0.4pt
      \pgfsetstrokecolor{cyan}
      \pgfsetstrokeopacity{0.5}
      \pgfusepath{stroke}
    }
  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{.5cm}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

在上面两个例子中，只是设置 outer xsep 的值，而 outer ysep 的值仍然是初始值 0.5\pgflinewidth，通过对比可知，在确定“外部锚位置”时使用的线宽是 \pgflinewidth=2mm. 也就是说，设置命令 \tikzset{line width=2mm}, 对后面画圆点的命令 \pgfpathcircle 中的锚位置有效。但在画出 node 的形状路径线条时，用到的线宽是由 node 本身的设置决定的，与 node 之外的设置无关。

/pgf/outer ysep=⟨*dimension*⟩ (no default, initially .5\pgflinewidth)

/tikz/outer ysep=⟨*dimension*⟩

这个选项在竖直方向上，设置 node 的背景边界形状路径与“外部锚位置”的间距。这个选项值只是个“推荐值”。

注意，这里的间距指的是路径线条的中心与“外部锚位置”的间距，而不是路径线条的外缘与“外部锚位置”的间距。

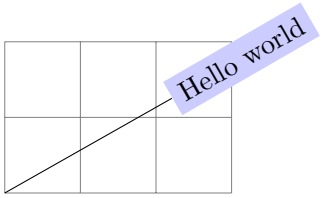
/pgf/outer sep=⟨*dimension*⟩ (no default)

/tikz/outer sep=⟨*dimension*⟩

本选项同时设置 /pgf/outer xsep 和 /pgf/outer ysep 的值为 ⟨*dimension*⟩.

25.2.3 late node

参考前文，真值 \pgflatenodepositioningtrue 使得命令 \pgfmultipartnode 创建 late node, 意思是，先创建 node, 但不直接插入到图形中，而是保存起来，稍后再插入到图形中。




```
\newbox\mybox  % 指定一个盒子寄存器
\def\mysaver{  % 定义宏 \mysaver
  \global\setbox\mybox=\box\pgfpositionnodelaterbox  % 定义盒子 \mybox 为全局盒子
  \global\let\myname=\pgfpositionnodelatername  % 定义全局命令 \myname
  \global\let\myminx=\pgfpositionnodelaterminx
  \global\let\myminy=\pgfpositionnodelaterminy
  \global\let\mymaxx=\pgfpositionnodelatermaxx
  \global\let\mymaxy=\pgfpositionnodelatermaxy
}

\begin{tikzpicture}
  {  % 开启一个分组
    \pgfpositionnodelater{\mysaver}  % 使用命令 \pgfpositionnodelater
    \node [fill=blue!20,below,rotate=30] (hi) at (1,0) {Hello world};  % 定义一个 node
  }  % 关闭分组
  \draw [help lines] (0,0) grid (3,2);
  \let\pgfpositionnodelatername=\myname
  \let\pgfpositionnodelaterminx=\myminx
  \let\pgfpositionnodelaterminy=\myminy
  \let\pgfpositionnodelatermaxx=\mymaxx
  \let\pgfpositionnodelatermaxy=\mymaxy
  \setbox\pgfpositionnodelaterbox=\box\mybox
  \pgfpositionnodenow{\pgfqpoint{2cm}{2cm}}
  \draw (hi) -- (0,0);
\end{tikzpicture}
```

这个添加 node 的方法的一般步骤如上面的例子所示：

1. 自己声明一个盒子，如 \mybox
2. 自己定义一个宏 \def\⟨*macro name*⟩{⟨*definition*⟩}, 在 ⟨*definition*⟩ 中至少包含下面的代码：

```
\global\setbox\mybox=\box\pgfpositionnodelaterbox
\global\let\myname=\pgfpositionnodelatername
\global\let\myminx=\pgfpositionnodelaterminx
\global\let\myminy=\pgfpositionnodelaterminy
\global\let\mymaxx=\pgfpositionnodelatermaxx
\global\let\mymaxy=\pgfpositionnodelatermaxy
```

   其中的命令 \global 必不可少。
3. 执行 \pgfpositionnodelater{⟨*macro name*⟩}
4. 创建一个 node.
5. 执行下面代码

```
\let\pgfpositionnodelatername=\myname
\let\pgfpositionnodelaterminx=\myminx
\let\pgfpositionnodelaterminy=\myminy
\let\pgfpositionnodelatermaxx=\mymaxx
\let\pgfpositionnodelatermaxy=\mymaxy
\setbox\pgfpositionnodelaterbox=\box\mybox
```

把保存在 \myname 等宏中的值转到 \pgfpositionnodelatername 等宏中，把盒子 \mybox 的内容转存到盒子 \pgfpositionnodelaterbox 中。

6. 执行 \pgfpositionnodenow{⟨*coordinate*⟩} 添加 node.

如果定义 ⟨*macro name*⟩ 为 \relax, 就会取消这个方法。

\pgfpositionnodelater{⟨*macro name*⟩}

本命令的作用是：

- 如果 ⟨*macro name*⟩ 等于 \relax(未定义)，则设置真值 \pgflatenodepositioningfalse
- 如果 ⟨*macro name*⟩ 不等于 \relax(有定义)，则设置真值 \pgflatenodepositioningtrue

因为 PGF 自己执行 \pgfpositionnodelater{\relax}, 所以默认 \pgflatenodepositioningfalse. 这个命令的有效范围受到“域”(scope) 的限制。假如事先定义了宏 ⟨*macro name*⟩, 那么使用本命令之后有真值 \pgflatenodepositioningtrue, 这会改变命令 \pgfnode, \pgfmultipartnode 的工作方式：不管所定义的 node 的锚定点是哪个位置，都被看作是原点；node 并不被立即添加到图形中，而是被保存在盒子 \pgfpositionnodelaterbox 中；node 也不直接与图形的边界盒子相关联，但 PGF 仍然计算盛放 node 的边界盒子，这个盒子的上、下、左、右边界值 (用尺寸表达的坐标) 会被保存在宏 \pgfpositionnodelaterminx, \pgfpositionnodelaterminy, \pgfpositionnodelatermaxx, \pgfpositionnodelatermaxy 之内。

\pgfpositionnodelaterbox

目前，这个盒子寄存器的编号是 0，用来保存所定义的 node. 在定义宏 ⟨*macro name*⟩ 时，你需要把盒子 \pgfpositionnodelaterbox 的内容全局地转移到另一个你自定义的盒子中。

\pgfpositionnodelatername

在真值 \pgflatenodepositioningtrue 之下创建的 node 的名称会被修改。如果你给出的 node 名称是 ⟨*node name*⟩, 那么此名称会被改为 not yet positionedPGFINTERNAL⟨*node name*⟩(加上前缀)，保存在这个宏中。执行命令 \pgfpositionnodenow 后，node 的名称会被改回原来的名称 ⟨*node name*⟩.

\pgfpositionnodelaterminx

node 的边界盒子的左侧边界坐标保存在这个宏中。这个宏的值是以 pt 为单位的尺寸。注意这个计算的前提是设定 node 的锚定点为原点。

\pgfpositionnodelaterminy

\pgfpositionnodelatermaxx

\pgfpositionnodelatermaxy

\pgfpositionnodelaterpath

这个宏保存的是 node 的背景路径 (bg) 的软路径形式，它是被全局地定义的。

\pgfpositionnodenow{⟨*coordinate*⟩}

这里的参数 ⟨*coordinate*⟩ 会被命令 \pgf@process 处理。本命令的作用是：

1. 用 \pgfinterruptpath 中断当前路径。
2. 设置一个花括号组，在组内：
   (a) 设置平移变换

```
\pgfpointtransformed{#1}%
\edef\pgf@temp@shift{\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}
\pgftransformreset%
```

```
\pgftransformshift{\pgf@temp@shift}%
```

(b) 释放盒子 \pgfpositionnodelaterbox 保存的 node, 将它添加到图形中，

```
\pgfsys@pictureboxsynced\pgfpositionnodelaterbox%
```

(c) 修改矩阵 \csname pgf@sh@nt@\pgfpositionnodelatername\endcsname

```
\pgf@shift@node{\pgfpositionnodelatername}{\pgf@temp@shift}%
```

(d) 刷新边界盒子

```
\pgfpointtransformed{\pgfqpoint{\pgfpositionnodelaterminx}{
↪  \pgfpositionnodelaterminy}}%
\pgf@protocolsizes{\pgf@x}{\pgf@y}
\pgfpointtransformed{\pgfqpoint{\pgfpositionnodelatermaxx}{
↪  \pgfpositionnodelatermaxy}}%
\pgf@protocolsizes{\pgf@x}{\pgf@y}
```

(e) 改回普通名称 ⟨*node name*⟩, 还是保存在 \pgfpositionnodelatername 中

```
\expandafter\pgfpositionnodenow@rename\pgfpositionnodelatername\relax
```

3. 设置一个花括号组，在组内执行由命令 \pgfnodepostsetupcode 定义的控制序列

   \csname pgf@lns@not yet positionedPGFINTERNAL⟨*node name*⟩\endcsname

   这个控制序列保存一些代码。如果之前没有使用过 \pgfnodepostsetupcode, 那这个控制序列就等于 \relax. 然后全局地令这个控制序列等于 \relax.
4. 用 \endpgfinterruptpath 恢复中断的路径。

\pgfnodepostsetupcode{⟨*node name*⟩}{⟨*code*⟩}

在真值 \pgflatenodepositioningtrue 之下本命令才有效，否则本命令什么也不做。

在命令 \pgfpositionnodelater 的有效范围内，本命令将 ⟨*code*⟩ 全局地保存到内部变量

\csname pgf@lns@not yet positionedPGFINTERNAL⟨*node name*⟩\endcsname

中。在使用命令 \pgfpositionnodenow 处理名称为 ⟨*node name*⟩ 的 late node 的过程中，这个内部变量被执行，执行完毕后，就令这个变量全局地等于 \relax.

如果多次使用这个命令,那么各次的 ⟨*code*⟩ 会被依次保存到这个内部变量中。直到使用命令 \pgfpositionnodenow, 将这个变量 relax.

25.2.4 引用与 node 相关的点

当一个 node 的形状、位置、内容确定后，就可以引用与它相关的点，例如它的 anchor 位置，它的边界上的点。这主要用到命令 \pgfpointanchor, \pgfpointshapeborder, 这两个命令都会全局地规定寄存器 \pgf@x, \pgf@y 的值。

\pgfpointanchor{⟨*node name*⟩}{⟨*anchor*⟩}

这个命令指定一个坐标点 (全局地规定寄存器 \pgf@x, \pgf@y 的值)，即名称为 ⟨*node name*⟩ 的 node 的锚位置 ⟨*anchor*⟩. 此命令可以用在构建路径的命令，如 \pgfpathmoveto 中。

本命令的处理是，首先检查命令 \csname pgf@sh@ns@⟨*node name*⟩\endcsname 是否未定义：

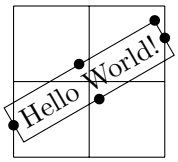
- 如果未定义，则检查命令 pgf@sh@ns@not yet positionedPGFINTERNAL⟨*node name*⟩ 是否未定义，
  - 如果未定义，则执行 \pgferror 报错
  - 如果已定义，则返回原点 \pgfpointorigin
- 如果已定义，则执行 \pgf@process{.....}, 其处理是：

1. 设置一个开花括号 {
2. 定义 \edef\pgfreferencednodename{⟨node name⟩}
3. 执行 \csname pgf@sh@ma@⟨node name⟩\endcsname, 得到各个 saved macros 的定义。
4. 执行 \csname pgf@sh@np@⟨node name⟩\endcsname, 得到各个 saved anchor, saved dimension 的定义。
5. 执行 \pgf@sh@reanchor{\csname pgf@sh@ns@⟨node name⟩\endcsname}{⟨anchor⟩}, 计算 ⟨anchor⟩ 位置点的坐标
6. 用开花括号 { 设置一个组
7. 执行 \pgfsettransform\csname pgf@sh@nt@⟨node name⟩\endcsname, 将这个控制序列保存的矩阵用作当前的变换矩阵，注意这个矩阵是执行创建 node 的命令时 (前) 面临的变换矩阵，而不是执行 \pgfpointanchor 时的变换矩阵
8. 执行 \pgf@pos@transform@glob, 此命令的处理是：
   - 如果当前的矩阵是单位矩阵，则什么也不做
   - 否则，用当前矩阵来变换当前点，即全局地改变 \pgf@x, \pgf@y 的值

$$(x,y)\begin{bmatrix} a_a & a_b \\ b_a & b_b \end{bmatrix} + (s,t)$$

9. 用闭花括号 } 结束一个组，使得变换矩阵恢复到组之前的状态
10. 执行 \pgf@shape@interpictureshift{⟨node name⟩}, 只有在跨图引用 ⟨node name⟩ 时，这个命令才有意义
11. 执行 \pgftransforminvert, 引入当前变换矩阵 (不是创建 node 时的变换矩阵) 的逆矩阵 (成为当前变换矩阵)，
12. 执行 \pgf@pos@transform@glob, 用逆矩阵全局地改变 \pgf@x, \pgf@y 的值
13. 用闭花括号 } 结束一个组，使得变换矩阵恢复到组之前的状态

注意，命令 \pgfpointanchor 在计算 ⟨anchor⟩ 位置点的坐标后，先用创建 node 时的变换矩阵来变换这个坐标，然后再用当前变换矩阵的逆矩阵改变 \pgf@x, \pgf@y 的值。

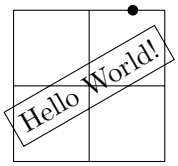



```
\begin{pgfpicture}
  \pgfpathgrid{\pgfpoint{-1cm}{-1cm}}{\pgfpoint{1cm}{1cm}}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}

  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

对于上面例子，似乎旋转变换应当对命令 \pgfnode 和 \pgfpathcircle 都有效，首先 \pgfnode 接受旋转矩阵，使得 node 被旋转，从而其锚位置被旋转；之后 \pgfpathcircle 接受旋转矩阵，使得锚位置再次被旋转，因此那些锚位置应当脱离 node 的背景路径。但实际上那些锚位置并没有脱离 node 的背景路径，这是因为命令 \pgfpointanchor 会引入旋转矩阵的逆矩阵，将作用于 \pgfpathcircle 中心点的旋转取消了。

下面的例子中，使用命令 \pgftransformreset 将命令 \pgfpointanchor 引入的旋转矩阵的逆矩阵变成单位矩阵，从而使得锚位置被旋转两次，脱离 node 的背景路径：




```
\begin{pgfpicture}
  \pgfpathgrid{\pgfpoint{-1cm}{-1cm}}{\pgfpoint{1cm}{1cm}}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}
  {
    \pgftransformreset
    \pgfpointanchor{x}{east}
    \makeatletter
    \xdef\mycoordinate{\noexpand\pgfpoint{\the\pgf@x}{\the\pgf@y}}
    \makeatother
  }
  \pgfpathcircle{\mycoordinate}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

\pgfpointshapeborder{⟨*node name*⟩}{⟨*point*⟩}

这个命令确定一个坐标点。以名称为 ⟨*node name*⟩ 的 node 的锚位置 center 为始点，做经过坐标点 ⟨*point*⟩ 的射线，射线与 ⟨*node name*⟩ 的边界形状路径相交，交点就是本命令确定的点。如果 ⟨*node name*⟩ 的边界形状路径很复杂，本命令会把这个复杂路径退化为一个相对简单的路径来计算，此时本命令确定的点可能偏离 ⟨*node name*⟩ 的边界形状路径。

本命令的处理是，首先检查命令 \csname pgf@sh@ns@⟨*node name*⟩\endcsname 是否未定义：

- 如果未定义，则执行 \pgferror 报错，返回原点 \pgfpointorigin.
- 如果已定义，则执行 \pgf@process{.....}, 其处理是：
  1. 设置一个开花括号 {
  2. 定义 \edef\pgfreferencednodename{⟨*node name*⟩}
  3. 执行 \csname pgf@sh@ma@⟨*node name*⟩\endcsname, 获取各个 saved macros 的定义。
  4. 执行 \csname pgf@sh@np@⟨*node name*⟩\endcsname, 获取各个 saved anchor, saved dimen 的定义。
  5. 设置一个开花括号 {
  6. 执行 \pgf@process{\pgfpointtransformed{⟨*point*⟩}}, 效果是用当前的线性变换矩阵用于点 ⟨*point*⟩, 得到点 ⟨*Point*⟩, 然后将得到的点的坐标全局地保存在寄存器 \pgf@x 和 \pgf@y 中。
  7. 执行 \pgfsettransform\csname pgf@sh@nt@⟨*node name*⟩\endcsname, 将这个控制序列保存的矩阵用作当前的变换矩阵。
  8. 执行 \pgftransforminvert, 将当前的变换矩阵换成其逆矩阵。
  9. 执行 \pgf@pos@transform@glob, 效果如前述。
  10. 保存尺寸 \pgf@xa=\pgf@x, \pgf@ya=\pgf@y
  11. 执行

      ```
      \pgf@process{\csname pgf@anchor@\csname pgf@sh@ns@⟨node name⟩\endcsname
      ↪  @center\endcsname}%
      ```

      得到 ⟨*node name*⟩ 的 center 位置的坐标 (全局地保存在寄存器 \pfd@x 和 \pgf@y 中)。

      参考 \csname pgf@sh@ns@⟨*node name*⟩\endcsname,

      参考 \csname pgf@anchor@⟨*shape*⟩@border\endcsname.
  12. 执行

      ```
      \pgf@process{\pgf@shape@interpictureshift{⟨node name⟩}}
      ```

      只有在跨图引用 ⟨*node name*⟩ 时，这个命令才有效果。
  13. 计算

```
\advance\pgf@xa by-\pgf@x%
\advance\pgf@ya by-\pgf@y%
```

这计算的是从 ⟨*node name*⟩ 的 center 位置到点 ⟨*Point*⟩ 的向量，保存在 \pgf@xa 和 \pgf@ya 中。

14. 设置 \pgf@xb 的值：
    - 如果 -1pt<\pgf@xa<1pt, 则令 \pgf@xb=10*\pgf@xa;
    - 否则令 \pgf@xb=10pt;
15. 设置 \pgf@yb 的值：
    - 如果 -1pt<\pgf@ya<1pt, 则令 \pgf@yb=10*\pgf@ya;
    - 否则令 \pgf@yb=10pt;
16. 执行

```
\ifdim\dimexpr\expandafter\Pgf@geT\the\pgf@xb\pgf@xb
  +\expandafter\Pgf@geT\the\pgf@yb\pgf@yb\relax<0.04pt
  \expandafter\pgfutil@firstoftwo
\else
  \expandafter\pgfutil@secondoftwo
\fi
```

即检查是否满足条件 xb^2 + yb^2 < 0.04pt, 只有 -1pt < \pgf@xa < 1pt 且 -1pt < \pgf@ya < 1pt 时才可能有这个条件，所以这个条件蕴含 sqrt(xa^2 + ya^2) < 0.02pt.

- 如果满足这个条件，则执行 \pgfutil@firstoftwo, 导致

```
\pgfwarning
  {Returning node center instead of a point on node border. Did you
  specify a point identical to the center of node
  ``\pgfreferencednodename''?}%
\pgf@sh@reanchor{\csname pgf@sh@ns@#1\endcsname}{center}%
```

即发出警告，并返回 center 位置点。

- 如果不满足这个条件，则执行

```
\csname pgf@anchor@\csname pgf@sh@ns@⟨node name⟩\endcsname @border
↪  \endcsname{\pgfqpoint{\pgf@xa}{\pgf@ya}}
```

计算边界点。

参考 \csname pgf@sh@ns@⟨*node name*⟩\endcsname,

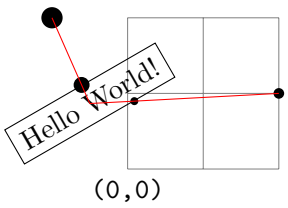
参考 \csname pgf@anchor@⟨*shape*⟩@border\endcsname.

17. 执行

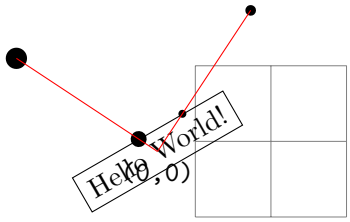
```
\pgfsettransform{\csname pgf@sh@nt@⟨node name⟩\endcsname}
```

将这个控制序列保存的矩阵用作当前的变换矩阵。

18. 执行 \pgf@pos@transform@glob, 效果如前述。
19. 全局赋值 \global\pgf@x=\pgf@x, \global\pgf@y=\pgf@y.
20. 用闭花括号 } 结束一个组
21. 执行 \pgf@shape@interpictureshift{⟨*node name*⟩}, 只有在跨图引用 ⟨*node name*⟩ 时，这个命令才有意义
22. 执行 \pgftransforminvert, 将当前的变换矩阵换成其逆矩阵。
23. 执行 \pgf@pos@transform@glob, 效果如前述。
24. 用闭花括号 } 结束一个组




```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \begin{pgfscope}  % 用 pgfscope 环境限制旋转变换
    \pgftransformrotate{30}
    \pgftransformshift{\pgfpoint{0cm}{1cm}}
    \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}
  \end{pgfscope}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{2cm}{1cm}}}{1.5pt}
  \pgfpathcircle{\pgfpoint{2cm}{1cm}}{2pt}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{-1cm}{2cm}}}{3pt}
  \pgfpathcircle{\pgfpoint{-1cm}{2cm}}{4pt}
  \pgfusepath{fill}
  \draw [red](x.center)--(2,1);
  \draw [red](x.center)--(-1,2);
  \node [below] {\tt(0,0)};
\end{tikzpicture}
```



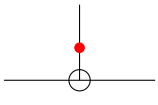

```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \pgftransformrotate{30}
  \pgftransformshift{\pgfpoint{0cm}{1cm}}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{2cm}{1cm}}}{1.5pt}
  \pgfpathcircle{\pgfpoint{2cm}{1cm}}{2pt}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{-1cm}{2cm}}}{3pt}
  \pgfpathcircle{\pgfpoint{-1cm}{2cm}}{4pt}
  \pgfusepath{fill}
  \draw [red](x.center)--(2,1);
  \draw [red](x.center)--(-1,2);
  \node [below] {\tt(0,0)};
\end{tikzpicture}
```

25.3 预定义的特殊 node

下面是文件《pgfmoduleshapes.code.tex》预定义的 node.

25.3.1 current bounding box

这个 node 的形状是 `rectangle`, 它是当前绘图环境创建的图形的边界盒子。在当前环境内，每添加一个路径，这个盒子就会被计算一次，因此它的上、下、左、右界限可能不断改变。



```
\tikz {
  \draw(0,0)--(2,0);
  \draw(current bounding box.center) circle (4pt)--(1,1);
  \fill [red] (current bounding box.center) circle (2pt);}
```

这个预定义 node 的定义是 (见文件《pgfmoduleshapes.code.tex》)：

```
1 \expandafter\def\csname pgf@sh@ns@current bounding box\endcsname{rectangle}%
2 \expandafter\def\csname pgf@sh@np@current bounding box\endcsname{%
3   \def\southwest{\pgfqpoint{\pgf@picminx}{\pgf@picminy}}%
4   \def\northeast{\pgfqpoint{\pgf@picmaxx}{\pgf@picmaxy}}%
5 }%
6 \expandafter\def\csname pgf@sh@nt@current bounding box\endcsname{{1}{0}{0}{1}{0pt}{0pt
  ↪  }}%
7 \expandafter\def\csname pgf@sh@pi@current bounding box\endcsname{\pgfpictureid}%
```

这个定义：

1. 第 1 行，参考 \csname pgf@sh@ns@⟨*node name*⟩\endcsname, 将 node 名称 current bounding box 与形状 rectangle 联系起来。
2. 第 2 行，参考 \csname pgf@sh@np@⟨*node name*⟩\endcsname.
3. 第 6 行，参考 \csname pgf@sh@nt@⟨*node name*⟩\endcsname.
4. 第 7 行，参考 \csname pgf@sh@pi@⟨*node name*⟩\endcsname.

25.3.2 current path bounding box

这个 node 的形状是 rectangle, 它是当前路径的边界盒子。关于这个预定义 node 要注意两点：

1. 计算当前路径的边界盒子时不考虑线宽，只是把路径当作无宽度的曲线来计算其边界盒子。线宽是“图形状态”参数，它表现出来的线条粗细只是对路径的一种“修饰”或“标示”，并不属于路径本身。
2. 计算当前路径的边界盒子时不考虑添加到路径上的 node.

x x x

```
x\begin{tikzpicture}
  \draw [line width=10mm,draw opacity=0.3](0,0)--(1,0)
    node[at=(current path bounding box.north)]{\color{red} x};
\end{tikzpicture}x
```

x x x

```
x\begin{tikzpicture}
\draw (0,0)--(1,0)node[above=1cm]{\color{green}x}--(2,0)
  node[at=(current path bounding box.north)]{\color{red}y};
\end{tikzpicture}x
```

```
\expandafter\def\csname pgf@sh@ns@current path bounding box\endcsname{rectangle}%
\expandafter\def\csname pgf@sh@np@current path bounding box\endcsname{%
  \def\southwest{\pgfqpoint{\pgf@pathminx}{\pgf@pathminy}}%
  \def\northeast{\pgfqpoint{\pgf@pathmaxx}{\pgf@pathmaxy}}%
}%
\expandafter\def\csname pgf@sh@nt@current path bounding box\endcsname{{1}{0}{0}{1}{0pt
↪  }{0pt}}%
\expandafter\def\csname pgf@sh@pi@current path bounding box\endcsname{\pgfpictureid}%
```

25.3.3 current subpath start

这个 node 的形状是 coordinate, 它是之前的 move-to 操作的“落脚点”。

```
\expandafter\def\csname pgf@sh@ns@current subpath start\endcsname{coordinate}%
\expandafter\def\csname pgf@sh@np@current subpath start\endcsname{%
  \def\centerpoint{\expandafter\pgfqpoint\pgfsyssoftpath@lastmoveto}%
}%
\expandafter\def\csname pgf@sh@nt@current subpath start\endcsname{{1}{0}{0}{1}{0pt
↪  }{0pt}}%
\expandafter\def\csname pgf@sh@pi@current subpath start\endcsname{\pgfpictureid}%
```

25.3.4 current page

将当前页面假想为一个图形——一个 node，就是 current page, 并且这个图形是“被记住”的，因此你可以在任何绘图环境中引用这个 node，只要这个绘图环境也是“被记住”的。

下面的例子中，将一段文字放在当前页面的左下角，注意要使用适当的驱动来支持“记住图形的功能”，否则文字会被插入到当前位置。

```
\begin{pgfpicture}
  \pgfusepath{use as bounding box}
  \pgftransformshift{\pgfpointanchor{current page}{south west}}
  \pgftransformshift{\pgfpoint{1cm}{1cm}}
  \pgftext[left,base]{
    \textcolor{red}{
      Text absolutely positioned in
      the lower left corner.}
  }
\end{pgfpicture}
```

Text absolutely positioned in the lower left corner.

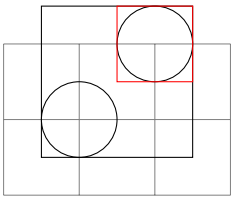
```
\expandafter\def\csname pgf@sh@ns@current page\endcsname{rectangle}%
\expandafter\def\csname pgf@sh@np@current page\endcsname{%
  \def\southwest{\pgfpointorigin}%
  \def\northeast{\pgfpoint{\pgfsys@thepagewidth}{\pgfsys@thepageheight}}%
}%
\expandafter\def\csname pgf@sh@nt@current page\endcsname{{1}{0}{0}{1}{0pt}{0pt}}%
\expandafter\def\csname pgf@sh@pi@current page\endcsname{pgfpageorigin}%
```

25.3.5 当前 scope 的边界盒子

/pgf/local bounding box=⟨*node name*⟩ (no default)

/tikz/local bounding box=⟨*node name*⟩

这个选项用在当前的 scope (组) 内有效，本选项创建一个名称为 ⟨*node name*⟩ 的 node, 这个 node 可以作为该 scope (组) 内的、本选项之后的图形的边界盒子。多次使用这个选项可能会让 TEX 的处理速度变慢。



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  { [local bounding box=outer box]  % 这个组开启一个 scope 环境
  \draw (1,1) circle (.5) [local bounding box=inner box] (2,2) circle
  ↪  (.5);
  }
  \draw (outer box.south west) rectangle (outer box.north east);
  \draw[red] (inner box.south west) rectangle (inner box.north east);
\end{tikzpicture}
```

本选项的定义是：

```
\pgfkeys{
  /pgf/local bounding box/.code={%
    \expandafter\gdef\csname pgf@sh@ns@#1\endcsname{rectangle}
    \expandafter\gdef\csname pgf@sh@np@#1\endcsname{%
      \def\southwest{\pgfqpoint{\csname pgf@lbb@minx@#1\endcsname}{\csname
      ↪  pgf@lbb@miny@#1\endcsname}}%
      \def\northeast{\pgfqpoint{\csname pgf@lbb@maxx@#1\endcsname}{\csname
      ↪  pgf@lbb@maxy@#1\endcsname}}%
    }
```

```
    \expandafter\gdef\csname pgf@sh@nt@#1\endcsname{{1}{0}{0}{1}{0pt}{0pt}}
    \expandafter\gdef\csname pgf@sh@pi@#1\endcsname{\pgfpictureid}
    \expandafter\gdef\csname pgf@lbb@maxx@#1\endcsname{-16000pt}%
    \expandafter\gdef\csname pgf@lbb@minx@#1\endcsname{16000pt}%
    \expandafter\gdef\csname pgf@lbb@maxy@#1\endcsname{-16000pt}%
    \expandafter\gdef\csname pgf@lbb@miny@#1\endcsname{16000pt}%
    \pgf@size@hookedtrue%
    \expandafter\def\expandafter\pgf@path@size@hook\expandafter{
    ↪  \pgf@path@size@hook\pgf@lbb@do{#1}}
  },
}
```

命令 `\pgf@lbb@do{`⟨*node name*⟩`}` 的定义是：

```
\def\pgf@lbb@do#1{%
  \ifdim\pgf@size@hook@x<\csname pgf@lbb@minx@#1\endcsname\expandafter\xdef\csname
  ↪  pgf@lbb@minx@#1\endcsname{\the\pgf@size@hook@x}\fi%
  \ifdim\pgf@size@hook@x>\csname pgf@lbb@maxx@#1\endcsname\expandafter\xdef\csname
  ↪  pgf@lbb@maxx@#1\endcsname{\the\pgf@size@hook@x}\fi%
  \ifdim\pgf@size@hook@y<\csname pgf@lbb@miny@#1\endcsname\expandafter\xdef\csname
  ↪  pgf@lbb@miny@#1\endcsname{\the\pgf@size@hook@y}\fi%
  \ifdim\pgf@size@hook@y>\csname pgf@lbb@maxy@#1\endcsname\expandafter\xdef\csname
  ↪  pgf@lbb@maxy@#1\endcsname{\the\pgf@size@hook@y}\fi%
}%
```

在文件《pgfcorepathconstruct.code.tex》中有：

```
\def\pgf@protocolsizes#1#2{%
  \ifpgf@relevantforpicturesize%
    \ifdim#1<\pgf@picminx\global\pgf@picminx#1\fi%
    \ifdim#1>\pgf@picmaxx\global\pgf@picmaxx#1\fi%
    \ifdim#2<\pgf@picminy\global\pgf@picminy#2\fi%
    \ifdim#2>\pgf@picmaxy\global\pgf@picmaxy#2\fi%
  \fi%
  \ifpgf@size@hooked%
    \let\pgf@size@hook@x#1\let\pgf@size@hook@y#2\pgf@path@size@hook%
  \fi%
  \ifdim#1<\pgf@pathminx\global\pgf@pathminx#1\fi%
  \ifdim#1>\pgf@pathmaxx\global\pgf@pathmaxx#1\fi%
  \ifdim#2<\pgf@pathminy\global\pgf@pathminy#2\fi%
  \ifdim#2>\pgf@pathmaxy\global\pgf@pathmaxy#2\fi%
}
\newif\ifpgf@size@hooked
\let\pgf@path@size@hook=\pgfutil@empty%
```

每当构造路径的命令 (moveto, lineto, curveto) 处理一个构造点时，都会利用 `\pgf@protocolsizes` 来刷新图形、当前路径的边界盒子。在 `\ifpgf@size@hooked` 的真值为 true 的情况下，还会定义 `\pgf@size@hook@x`, `\pgf@size@hook@y`, 并执行 `\pgf@path@size@hook`.

使用选项 `/pgf/local bounding box=`⟨*node name*⟩ 后，⟨*node name*⟩ 这个 node 就被创建，不过，它的边界是变化的。由于本选项设置真值 `\pgf@size@hookedtrue`, 所以每当 `\pgf@protocolsizes` 被执行，它的边界就变化一次。因为 `\ifpgf@size@hooked` 的真值受到 TEX 组的限制，所以本选项的创建的 node 的边界也受到组的限制。当然，node 都是全局的，在组之后也可以使用这个 node.

`/pgf/freeze local bounding box=`⟨*node name*⟩

这个选项重定义 `\csname pgf@sh@np@`⟨*node name*⟩`\endcsname`, 使得它保存的两个 saved anchor: `\southwest`, `\northeast` 的定义被“固定”，不会再变化到其他位置。

```
\pgfkeys{
  /pgf/freeze local bounding box/.code={%
    {%
      \csname pgf@sh@np@#1\endcsname%
      \southwest%
      \pgf@xa=\pgf@x%
      \pgf@ya=\pgf@y%
      \northeast%
      \expandafter\xdef\csname pgf@sh@np@#1\endcsname{%
        \noexpand\def\noexpand\southwest{\noexpand\pgfqpoint{\the\pgf@xa}{
        ↪  \the\pgf@ya}}%
        \noexpand\def\noexpand\northeast{\noexpand\pgfqpoint{\the\pgf@x}{
        ↪  \the\pgf@y}}%
      }%
    }%
  },
}
```

很多预定义 node 都采用上面的办法来定义。文件《pgfmoduleshapes.code.tex》中对项 /tikz/local bounding box[→P. 475]=⟨*name*⟩ 的定义;文件《tikz.code.tex》中,命令 \tikz@finish 的展开中对 pgf@sh@ns@path picture bounding box 的定义,都是这样的方式定义的。这样定义后,就可以使用命令 \pgfpointanchor[→P. 469], \pgfpointshapeborder[→P. 471] 来计算相关的 anchor 位置，边界上的点。

第二十六章　matrix 模块

首先调用模块 matrix.

```
\usepgfmodule{matrix} % LaTeX and plain TeX and pure pgf
\usepgfmodule[matrix] % ConTeXt and pure pgf
```

26.1　Overview

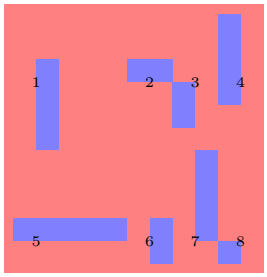
命令 \pgfmatrix 利用 TEX 的命令 \halign 创建一个表格，即矩阵。矩阵被创建为一个 node, 矩阵的元素称为“cell”. 矩阵元素被处于花括号组中的 pgfsys@beginpicture 环境包围，所以矩阵元素应当是 PGF 的绘图命令。矩阵的元素可以被留空。

26.2　矩阵元素的对齐方式

每个 cell 图形都在自己的“私有”坐标系内被构造。在默认下，一行之内的各 cell 图形的原点处于同一水平线上，一列之内的各 cell 图形的原点处于同一竖直线上。在一行之内，各个 cell 的最大高度和最大深度就是这一行的高度和深度。一列之内，各个 cell 的最大宽度就是这一列的宽度。元素图形没有“层”(layers) 的概念，但有边界盒子（用于对齐）。

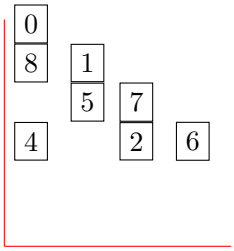
在一行之内，相邻两个元素之间用命令 \pgfmatrixnextcell 隔开。在每一行的末尾（包括最后一行）使用命令 \pgfmatrixendrow 结束，也可以使用两个反斜线符号“\\”结束一行。

\pgfmatrixnextcell, \pgfmatrixendrow 以及 \\ 都可以带有选项。




```
\begin{tikzpicture}[x=3mm,y=3mm,fill=blue!50]
↪  % 选项 fill=blue!50 对元素图形的 \fill 命令无效,
  \def\atorig#1{\node[black] at (0,0) {\tiny #1};}
  \def\pgfmatrixbegincode{\pgfsetfillcolor{blue!50}}
  ↪  % 这个设置对元素图形的 \fill 命令有效
  \pgfmatrix{rectangle}{center}{mymatrix}{\pgfsetfillcolor{red!50}
  ↪  \pgfusepath{fill}}
    {\pgfpointorigin}{}
    {
      \fill (0,-3) rectangle (1,1); \atorig1 \pgfmatrixnextcell
      \fill (-1,0) rectangle (1,1); \atorig2 \pgfmatrixnextcell
      \fill (-1,-2) rectangle (0,0); \atorig3 \pgfmatrixnextcell
      \fill (-1,-1) rectangle (0,3); \atorig4 \\
      \fill (-1,0) rectangle (4,1); \atorig5 \pgfmatrixnextcell
      \fill (0,-1) rectangle (1,1); \atorig6 \pgfmatrixnextcell
      \fill (0,0) rectangle (1,4); \atorig7 \pgfmatrixnextcell
      \fill (-1,-1) rectangle (0,0); \atorig8 \\
    }
\end{tikzpicture}
```

矩阵元素的行列排布类似表格。矩阵某两行的元素个数可以不相等，某两列的元素个数也可以不相等，观察下面的例子：




```
\begin{tikzpicture}[every node/.style=draw]
  \draw [red](0,0) -- (3,0);
  \draw [red](0,0) -- (0,3);
  \pgfsetmatrixcolumnsep{1mm}
  \pgfmatrix{rectangle}{south west}{mymatrix}{\pgfusepath{}}{\pgfpoint{0
  ↪  }{-1cm}}{\let\&=\pgfmatrixnextcell}
  {
    \node {0}; \\
    \node {8}; \&[2mm] \node {1}; \\
                  \&      \node {5}; \&[1mm] \node {7}; \\
    \node {4}; \&                      \& \node {2}; \&[2mm] \node {6}; \\
  }
\end{tikzpicture}
```

26.3 矩阵命令

\pgfmatrix{⟨*shape*⟩}{⟨*anchor*⟩}{⟨*name*⟩}{⟨*usage*⟩}{⟨*shift*⟩}{⟨*pre-code*⟩}{⟨*matrix cells*⟩}

这个命令创建一个矩阵，矩阵实为一个 node, 名称为 ⟨*name*⟩, 形状为 ⟨*shape*⟩. 在默认下，矩阵的锚位置 ⟨*anchor*⟩ 处于（绘图环境坐标系的）原点上。向量 ⟨*shift*⟩ 会使得矩阵被平移，不过平移向量是 ⟨*shift*⟩ 的负向量。可以这样理解：先平移矩阵使得锚位置 ⟨*anchor*⟩ 处于原点上，再按 ⟨*shift*⟩ 的负向量平移矩阵，确定矩阵的位置。

⟨*matrix cells*⟩ 规定矩阵的各个元素图形。元素图形的代码是直接的绘图命令，PGF 会跟踪元素图形的边界盒子，并将元素按规则对齐。

⟨*usage*⟩ 是关于使用路径的命令，使用的是矩阵 ⟨*name*⟩ 的背景路径 (background path)，不是元素图形中的路径。

关于本命令：

1. 首先检查 \ifpgfmatrix 的真值，如果是 true, 就报错，实际意思是，不能在命令 \pgfmatrix 内部套嵌使用 \pgfmatrix.
2. 开启一个组，这个组将在后面的 \pgf@end@matrix 那里结束。本命令的所有操作，除了创建矩阵 node 的操作，都限制在这个组内。
3. 清空 \everycr{}
4. 执行 ⟨*pre-code*⟩.
5. 如果 ⟨*name*⟩ 是空的，那就为矩阵定义一个内部名称：

```
\def\pgf@matrix@par@name{pgf@matrix@internal}%
```

6. 将 \\ 定义为分行符号：

```
\let\\=\pgfmatrixendrow%
```

7. 设置关于对齐列表的间距：

```
\tabskip=0pt%
\offinterlineskip%
```

8. 设置真值 \pgfmatrixtrue
9. 然后设置盒子：

```
\setbox\pgf@matrix@box=\hbox\bgroup\vbox\bgroup%
  \halign\bgroup%
    \pgf@matrix@init@row%
    \pgf@matrix@step@column%
    {%
      \pgf@matrix@startcell%
```

```
      ##%
      \pgf@matrix@endcell%
    }%
    &%
    ##\pgf@matrix@padding&&%
    \pgf@matrix@step@column%
    {%
      \pgf@matrix@startcell%
      ##%
      \pgf@matrix@endcell%
    }&%
    ##\pgf@matrix@padding%
    \cr%
```

盒子 \pgf@matrix@box 等于 PGF 为 node 预先声明的盒子 \pgfnodeparttextbox. 盒子 \pgf@matrix@box 被设置为一个水平盒子，它的内容是一个垂直盒子，垂直盒子的内容是 \halign 创建的对齐列表。

参考《TEX By Topic》的 25 章。

10. 然后处理 ⟨*matrix cells*⟩. 对元素图形的处理主要依赖 \halign 的模板，即

```
{%
  \pgf@matrix@startcell%
  ⟨code of a cell picture⟩%
  \pgf@matrix@endcell%
}%
```

可见每个 cell 图形都被放入一个花括号组中,并且在这个花括号组中还被 \pgf@matrix@startcell 和 \pgf@matrix@endcell 包裹。注意，按 TEX 的处理流程，先将 ⟨*code of a cell picture*⟩ 的第一个记号展开，然后再执行 \pgf@matrix@startcell.

处理 ⟨*code of a cell picture*⟩ 的大致流程是：

- \pgfinterruptboundingbox
- \pgftransformreset
- 将 ⟨*code of a cell picture*⟩ 放入盒子 \pgf@matrix@cell 中：

```
\setbox\pgf@matrix@cell=\hbox\bgroup\bgroup%
    % make sure that cell pictures are not affected if matrices have
    % 'overlay' option on:
    \pgf@relevantforpicturesizetrue
    \pgfsys@beginpicture%
    \normalbaselines%
    % Find out whether the cell is empty:
    \pgfutil@ifnextchar\let%
    {% ok, candidate, check following symbol
      \afterassignment\pgf@matrix@empty@check\let\pgf@matrix@temp=
      ↪  % get rid of \let
    }%
    {% no, not empty
      \pgf@matrix@empty@cell@false%
      \pgfmatrixbegincode%
    }%
    ⟨code of a cell picture⟩
    \ifpgf@matrix@empty@cell@%
    \else%
      \expandafter\expandafter\expandafter\pgfmatrixendcode%
```

```
      \expandafter\ifpgf@matrix@last@cell@in@row\expandafter
      ↪  \pgf@matrix@last@cell@in@rowtrue\expandafter\fi%
    \fi%
    \pgfsys@endpicture%
    ⟨ 一些关于行号、列号，cell 图形尺寸、位置的计算⟩
  \egroup\egroup
```

注意上面代码中，在 (非空的) ⟨*code of a cell picture*⟩ 之前执行 \pgfmatrixbegincode→P. 484，之后执行 \pgfmatrixendcode→P. 484.

- 一些关于行高、行深、列宽度的计算。
- 释放盒子 \box\pgf@matrix@cell
- \endpgfinterruptboundingbox

11. 在处理完所有的 cell 图形后，最后一个 \pgfmatrixendrow 命令会导向 \pgf@end@matrix, 这个命令的主要作用是：
    - 结束之前开启的组
    - 取消变换矩阵中的非平移成分，参照参数 ⟨*shift*⟩ 重新计算平移矩阵
    - 用 \pgfmultipartnode 创建矩阵 node. 前面用 \halign 创建的对齐列表就放在这个 node 的文字盒子里。
    - 如果元素图形是 node, 那么调整这些 node 的位置。

26.3.1 分行、分列符号

前文已经提到 \pgfmatrixnextcell, \pgfmatrixendrow 以及 \\. 这些命令实际使用 \halign 来实现其作用。

注意，在 PGF 中不能使用 & 作为分隔左右两个元素的符号，除非让 & 等于 \pgfmatrixnextcell. 在 Ti*k*Z 中可以使用 & 作为分列符，Ti*k*Z 对此有规定：

```
\catcode`\&=13%
\let&=\pgfmatrixnextcell%
```

即先把 & 做成活动符，然后让 & 等于 \pgfmatrixnextcell.

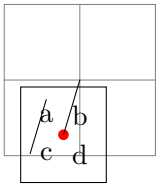
因为 {minipage} 环境会重定义 \\, 所以在 {minipage} 环境中使用矩阵时，最好不用 \\ 换行。

26.3.2 矩阵的锚位置与锚定点

矩阵是只有一个 node part 的 node, 这个 node part 的名称默认为 text, 其中盛放文字的盒子用来放置矩阵的元素。盒子的左下角 (基点) 位于矩阵的锚位置 text 上。

假设矩阵的元素都是 node, 每个元素 node 都有自己的各种锚位置。假设有个元素 node 的名称是 inner node, 它有锚位置 inner node.north. 前面提到，向量 ⟨*shift*⟩ 会使得平移矩阵按 ⟨*shift*⟩ 的负向量平移。当读取 (处理) ⟨*shift*⟩ 时，各个元素 node 的各种锚位置都是可以引用的 (已定义的)。如果 ⟨*shift*⟩ 是 \pgfpointanchor{inner node}{north}，矩阵又会怎样平移呢？

先看一下 \pgfpointanchor{inner node}{north} 是怎样确定的。以矩阵的锚位置 text 为原点建立一个坐标系，从这个原点到点 inner node.north 的向量就是 \pgfpointanchor{inner node}{north}. 因此，如果在矩阵命令中，让 ⟨*anchor*⟩ 是 text, 让 ⟨*shift*⟩ 是 \pgfpointanchor{inner node}{north}, 那么点 inner node.north 就会处于图形的原点——绘图环境坐标系的原点。




```
\begin{tikzpicture}
  \draw [help lines] (-1,-1) grid (1,1);
  \pgfmatrix{rectangle}{center}{mymatrix}{\pgfusepath{draw}}{\pgfpointanchor
  ↪  {a}{north}}{}
  {
    \node (a){a}; \pgfmatrixnextcell \node {b}; \pgfmatrixendrow
    \node {c}; \pgfmatrixnextcell \node {d}; \pgfmatrixendrow
  }
  \fill [red] (mymatrix.center)circle(2pt);
  \draw (mymatrix.center)--(0,0);
  \draw (mymatrix.text)--(a.north);  % 两条线段平行且长度相等
\end{tikzpicture}
```

26.3.3 旋转与放缩

如果矩阵命令之前有旋转、放缩、平移命令，那么这些命令对矩阵无效，将来不打算改变这一点。如果要对矩阵做变换，你只能自己规定画布变换。

在矩阵元素图形的命令中使用变换选项，可以变换元素图形。

26.3.4 调用命令

宏 `\pgfmatrixbegincode`, `\pgfmatrixendcode`, `\pgfmatrixemptycode` 分别保存一组代码。

当执行任何一个元素图形代码时，会先执行宏 `\pgfmatrixbegincode` 保存的代码。

在执行任何一个元素图形代码之后，会执行 `\pgfmatrixendcode` 保存的代码。

如果元素图形代码是空的，就在这个元素的位置执行宏 `\pgfmatrixemptycode` 保存的代码。

26.3.5 pre-code

矩阵的代码会被放入一个 TEX 组内执行，矩阵命令中的 ⟨*pre-code*⟩（一组代码）也会被放入这个 TEX 组内，但在矩阵内容之前被执行。

利用 ⟨*pre-code*⟩ 可以做某些事情，例如：

1. 你可以在 ⟨*pre-code*⟩ 中定义

   ```
   \let\&=\pgfmatrixnextcell
   ```

   将 `&` 作为分列符。

2. 也可以在 ⟨*pre-code*⟩ 中使用命令 `\aftergroup`, 待 TEX 分组结束后执行某些操作。

26.3.6 元素对齐过程中的宏展开

矩阵元素的对齐过程（行方向与列方向）实际是个构造列表的过程，即把元素图形排布成一个矩形列表，其内部操作使用命令 `\halign`, 选项 `/tikz/node halign header` 也利用了这个命令。如前述，这个命令可能会做一些奇怪的事情，先将 ⟨*code of a cell picture*⟩ 的第一个记号展开，然后再执行 `\pgf@matrix@startcell`. 因此你需要注意：

- 如果某个元素的代码的第一个记号是一个宏,它展开后包含 `\pgfmatrixnextcell` 或 `\pgfmatrixendrow`, 你需要注意不要再重复使用这两个命令。
- 你可以定义一个宏，它展开后包含 `\pgfmatrixnextcell` 或 `\pgfmatrixendrow`, 这样就可添加矩阵的列或行。

26.4 行间距与列间距

\pgfsetmatrixcolumnsep{⟨*sep list*⟩}

本命令将 ⟨*sep list*⟩ 保存到宏 \pgfmatrixcolumnsep 中，这个宏代表“列间距”，它的初始值是 0pt. ⟨*sep list*⟩ 是一个列表，这个列表的列表项可以是尺寸，字符串 between borders 或 between origins, 例如

```
\pgfsetmatrixcolumnsep{1mm,2mm,3mm}
或者
\pgfsetmatrixcolumnsep{1mm,between borders,3mm,between origins}
```

这个列表会被 \pgf@matrix@addtolength 处理：

- 其中的尺寸会被累加起来，累加的和就作为列间距，这个列间距针对所有相邻的两个列；
- 字符串 between borders 会导致真值 \pgf@matrix@fixedfalse, 这会导致，列间距以元素图形的边界为参照来计算
- 字符串 between origins 会导致真值 \pgf@matrix@fixedtrue, 这会导致，列间距以元素图形的原点为参照来计算

默认的是 \pgf@matrix@fixedfalse, 即使用 between borders.

\pgfmatrixnextcell[⟨*additional sep list*⟩]

这个命令有两个作用：

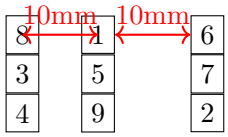
- 全局地保存可选项 ⟨*additional sep list*⟩, 这里 ⟨*additional sep list*⟩ 是一个列表，这个列表的列表项可以是尺寸，字符串 between borders 或 between origins, 如前述。
- 使用记号序列 &\pgf@matrix@correct@calltrue& 分隔左右两个元素图形，注意在这个记号序列中：
  - 左侧的制表符 & 标志着左侧元素图形代码的结束位置，命令 \pgf@matrix@endcell 会在这个制表符前面被执行，这个命令会引入 \pgfmatrixendcode →P. 484;
  - 右侧的制表符 & 标志着右侧元素图形代码的开始位置，命令 \pgf@matrix@startcell 会在这个制表符后面被执行 (不过会先把元素图形代码的第一个记号展开)，这个命令会引入 \pgfmatrixbegincode →P. 484.

这个 ⟨*additional sep list*⟩ 是可选的，如果不给出它，那么列间距就是由宏 \pgfsetmatrixcolumnsep{⟨*sep list*⟩} 指定的默认尺寸；如果给出它，那么列间距就是默认尺寸加上这个可选尺寸。也就是说，本命令指定的是个“附加间距”。

这个命令可以用在任意一行。

如果此命令与 \pgfsetmatrixcolumnsep 有冲突 (关于模式 between borders 或者 between origins)，那么此命令优先。

注意，如果 ⟨*additional sep list*⟩ 中使用字符串 between borders 或 between origins, 那么本命令只能用在第一行中。




```
\begin{tikzpicture}[every node/.style=draw]
  \pgfsetmatrixcolumnsep{1cm,between origins}
  \pgfmatrix{rectangle}{center}{mymatrix}{\pgfusepath{}}{\pgfpointorigin}
  {\let\&=\pgfmatrixnextcell}
  {
    \node (a) {8}; \& \node (b) {1}; \&[between borders] \node (c) {6}; \\
```

```
    \node       {3}; \& \node       {5}; \&                    \node       {7}; \\
    \node       {4}; \& \node       {9}; \&                    \node       {2}; \\
  }
  \begin{scope}[every node/.style=]
    \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {10mm};
    \draw [<->,red,thick] (b.east) -- (c.west) node [above,midway]{10mm};
    \end{scope}
\end{tikzpicture}
```

以上关于列间距的机制对行间距也是一样的。

\pgfsetmatrixrowsep{⟨*sep list*⟩}

类似宏 \pgfsetmatrixcolumnsep，只是针对任意相邻两行的间距。本命令将 ⟨*sep list*⟩ 保存在宏 \pgfmatrixrowsep 中，这个宏代表行间距，它的初始值是 0pt.

\pgfmatrixendrow[⟨*additional sep list*⟩]

类似命令 \pgfmatrixnextcell，只是针对行间距。

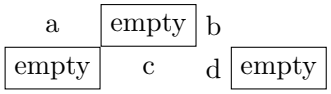
注意符号 \\ 与本命令的作用是一样的。

在最后一行的末尾也要使用本命令，但此时它的选项无效。

26.5 调用命令

\pgfmatrixemptycode

PGF 会在空的元素图形 (empty cells) 的位置上执行本命令，可以用 \def 定义此命令的展开内容。例如：




```
\begin{tikzpicture}
  \def\pgfmatrixemptycode{\node[draw]{empty};}
  \pgfmatrix{rectangle}{center}{mymatrix}{\pgfusepath{}}{\pgfpointorigin}
  {\let\&=\pgfmatrixnextcell}
  {
    \node {a}; \& \& \node {b}; \\
                  \& \node{c}; \& \node {d}; \& \\
  }
\end{tikzpicture}
```

上面例子说明，如果一个位置之后没有 & 或命令 \pgfmatrixnextcell，那么这个位置就不是空元素图形 (empty cells) 的位置。

\pgfmatrixbegincode

当定义

```
\def\pgfmatrixbegincode{⟨code⟩}
```

之后，此命令会在所有“非空元素图形的代码”的前面被执行。

\pgfmatrixendcode

当定义

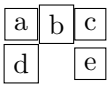
```
\def\pgfmatrixendcode{⟨code⟩}
```

之后，此命令会在所有“非空元素图形的代码”的后面被执行。

在宏 \pgfmatrixbegincode 与 \pgfmatrixendcode 之间，并非只有非空元素的绘图命令，其间 PGF 还会插入某些不可见的命令，包括 \let 或者 \gdef. 如果 \pgfmatrixbegincode 的定义代码以 \csname

结束，并且 \pgfmatrixendcode 的定义代码以 \endcsname 开头，那么可能会导致错误——这不是个好主意。

下面例子中，所列出的矩阵元素都是字母，不是绘图命令，但是通过对宏 \pgfmatrixbegincode 和 \pgfmatrixendcode 的定义，将 \node [draw] \bgroup 放在字母开头，将 \egroup; 放在字母结尾，从而做成一个 \node 命令。




```
\begin{tikzpicture}
  \def\pgfmatrixbegincode{\node[draw]\bgroup}
  \def\pgfmatrixendcode{\egroup;}  % 会把元素转成 node
  \pgfmatrix{rectangle}{center}{mymatrix}{\pgfusepath{}}{\pgfpointorigin}
  {\let\&=\pgfmatrixnextcell}
  {
    a \& b \& c \\
    d \&    \& e \\
  }
\end{tikzpicture}
```

\pgfmatrixcurrentrow

这个宏是个计数器，它的值是当前行的行号，不要随意改动它的值。

\pgfmatrixcurrentcolumn

这个宏是个计数器，它的值是当前列的列号，不要随意改动它的值。

第二十七章　创建 Plots

本节介绍 plot 模块。

```
\usepgfmodule{plot} % LaTeX and plain TeX and pure pgf
\usepgfmodule[plot] % ConTeXt and pure pgf
```

这个模块定义了一些命令，能实现 plot 功能。PGF 会自动加载这个模块。如果只是在内核 pgfcore 下使用本模块，那就需要手工调用这个模块。

27.1　Overview

大体上讲，PGF 按两个步骤创建 plot：首先生成一个图流（plot stream），这个流由一些坐标点构成；然后将某个图柄（plot handler）作用于图流。PGF 预定义了多个图柄，例如 `\pgfplothandlerlineto`，库 plothandlers 也定义了多个图柄。图柄对图流的作用是，例如图柄 `\pgfplothandlerlineto` 对图流的作用是，将 line-to 操作用于图流中的坐标点创建折线段。

对于一个已创建的图流来说，其中有已确定的要素 (例如点的坐标数据可能是已确定的)，也有未确定的要素 (例如用何种方式构造路径)。当一个图柄处理图流时，会把其中未确定的要素确定下来，然后顺次执行图流中的代码，这就是用图柄处理图流的实际意思。

27.2　创建图流

“创建一个图流”指的是用命令 `\pgfplothandlerrecord`⟨*a macro*⟩ 将一个图流保存在宏 ⟨*a macro*⟩ 中。注意创建的图流是“全局地”，不受 TEX 组的限制，你可以在绘图环境之外创建图流。例如可以在绘图环境外写出下面的代码：

```
\pgfplothandlerrecord{\mystream}
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
\pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{3cm}{1cm}}
\pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
\pgfplotstreamend
```

上面代码中命令 `\pgfplothandlerrecord` 将图流保存在宏 `\mystream` 中，然后在绘图环境内可以用图柄处理宏 `\mystream`, 将图流变成路径。在宏 `\mystream` 保存的图流中，点的坐标数据是已确定的；而 `\pgfplotstreamstart`, `\pgfplotstreampoint` 这两个命令则可能是未确定的，也可能是已确定的。先使用图柄重定义这两个命令，然后将图流展开，得到需要的路径。

27.2.1　图流的基本结构

用以下命令构建图流：

- `\pgfplotstreamstart`

- \pgfplotstreampoint
- \pgfplotstreampointoutlier
- \pgfplotstreampointundefined
- \pgfplotstreamnewdataset
- \pgfplotstreamspecial
- \pgfplotstreamend

上面命令的名称大致表明了它们各自的用处。在以上任意两个命令之间可以插入任何代码，也可以修改以上命令的定义。下面是个创建图流的例子：

```
\pgfplotstreamstart  % 开启一个图流
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}  % 将点 (1cm,1cm) 添加到图流中
\newdimen\mydim  % 定义一个新尺寸
\mydim=2cm  % 为新尺寸赋值
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}  % 将点 (2cm,2cm) 添加到图流中
\advance \mydim by 3cm  % 将尺寸 \mydim 变成 5cm
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}  % 将点 (5cm,2cm) 添加到图流中
\pgfplotstreamend  % 结束图流
```

有以下初始定义：

```
\def\pgfplot@ignorer#1#2#3{#1}%
\def\pgfplot@jumper#1#2#3{#2}%
\def\pgfplot@plotter#1#2#3{#3}%

\let\pgfplot@outliers\pgfplot@jumper
\let\pgfplot@undefined\pgfplot@jumper
\let\pgfplot@newdata\pgfplot@jumper
```

\pgfplotstreamstart

本命令的定义是：

```
\def\pgfplotstreamstart{\pgf@plotstreamstart}%
```

本命令开启一个图流，它会调用内部命令 \pgf@plotstreamstart 以确定 plot 开端的动作。它也会修改某些内部命令，如 \pgf@plotstreampoint.

\pgfplotstreampoint{⟨*point*⟩}

本命令的定义是：

```
\def\pgfplotstreampoint#1{\gdef\pgfplotlastpoint{#1}\pgf@plotstreampoint{#1}}%
```

本命令将点 ⟨*point*⟩ 添加到当前的图流中。本命令调用内部命令 \pgf@plotstreampoint.

当一个图柄起作用时，图柄会以某种方式定义 (重定义) 内部命令 \pgf@plotstreampoint，在图流中修改命令 \pgf@plotstreampoint 的意义是被允许的。例如，图柄可以设置 \pgf@plotstreampoint 使之对第一个点执行某种操作，然后重定义 \pgf@plotstreampoint，使之对其它点执行别的操作。\pgfplotstreamstart 总会重置命令 \pgf@plotstreampoint，使得命令 \pgf@plotstreampoint 的作用符合图柄的要求。

\pgfplotstreampointoutlier{⟨*point*⟩}

本命令的定义是：

```
\def\pgfplotstreampointoutlier#1{\pgfplot@outliers{}{\pgf@plotstreamjump}{
↪  \pgfplotstreampoint{#1}}}%
```

本命令将点 ⟨*point*⟩ 设置为 outlier，即“考虑范围之外的点”，例如，一个 outlier 点可能代表无穷远点，或者曲线的间断点。

对于 outlier 点的处理方式由下面的选项决定：

/pgf/handle outlier points in plots=⟨*how*⟩ (no default, initially jump)

这个选项的定义是：

```
\pgfset{
  handle outlier points in plots/.is choice,
  handle outlier points in plots/ignore/.code=\let\pgfplot@outliers
  ↪  \pgfplot@ignorer,
  handle outlier points in plots/jump/.code=\let\pgfplot@outliers
  ↪  \pgfplot@jumper,
  handle outlier points in plots/plot/.code=\let\pgfplot@outliers
  ↪  \pgfplot@plotter,
}
```

/tikz/handle outlier points in plots=⟨*how*⟩ (no default, initially jump)

其中 ⟨*how*⟩ 可以取的值是：plot，ignore，jump.

\pgfplotstreampointoutlier{⟨*point*⟩} 展开为

```
\pgfplot@outliers{}{\pgf@plotstreamjump}{\pgfplotstreampoint{⟨point⟩}}
```

这个键的 3 个可用值分别对 \pgfplot@outliers 做不同定义，导致不同情况：

- plot 导致 \pgfplotstreampoint{⟨*point*⟩}, 把 outlier 点作为普通点对待，不过有的图柄可能会临时修改 \pgfplotstreampoint 的定义，使得它“不再普通”
- jump 导致 \pgf@plotstreamjump, 实际效果要看命令 \pgf@plotstreamjump 是如何定义的，定义这个命令的是键
  - /pgf/plots/@handler options/start →P. 497, 这个键全局地规定 \pgf@plotstreamjump 等于 \pgf@plotstreamjump@init
  - /pgf/plots/@handler options/jump →P. 499 或 /pgf/plots/@handler options/jump macro →P. 500, 这两个键定义 \pgf@plotstreamjump@init

  不同图柄会做出不同定义
- ignore 导致 {}, 导致“忽略 outlier 点”⟨*point*⟩——什么都不做，连 \relax 都不执行

举例来说，如果下面线段：

```
(0,0)--(1,0)--(1.2,0)--(1.4,0)--(2,0)
```

中的点 (1.2,0) 是被“jump”的点，那么多数图柄会把这个线段分成两段：

```
(0,0)--(1,0) (1.4,0)--(2,0)
```

如果点 (1.2,0) 是被“ignore”的点，那么这个线段就可能是：

```
(0,0)--(1,0)--(1.4,0)--(2,0)
```

\pgfplotstreampointundefined

本命令的定义是：

```
\def\pgfplotstreampointundefined{\pgfplot@outliers{}{\pgf@plotstreamjump}{}}%
```

这个命令不带参数，它生成一个“未定义点”(undefined)，这个点是无法画出的，因此本命令不需要参数。“未定义点”的作用是引起某种动作，这个动作由以下选项规定：

/pgf/handle undefined points in plots=⟨*how*⟩ (no default, initially jump)

这个选项的定义是：

```
\pgfset{
  handle undefined points in plots/.is choice,
```

```
  handle undefined points in plots/ignore/.code=\let\pgfplot@undefined
  ↪  \pgfplot@ignorer,
  handle undefined points in plots/jump/.code=\let\pgfplot@undefined
  ↪  \pgfplot@jumper,
}
```

/tikz/handle undefined points in plots=⟨*how*⟩ (no default, initially jump)

这个选项规定对 undefined 点的处理方式，⟨*how*⟩ 可以取以下值：

- ignore，这个值导致“忽略 undefined 点”。
- jump，这个值导致内部宏 \pgf@plotstreamjump 被调用，会在图流中制造一个“缺口”。

\pgfplotstreamnewdataset

本命令的定义是：

```
\def\pgfplotstreamnewdataset{\pgfplot@outliers{}{\pgf@plotstreamjump}{}}%
```

图流中的点是可以被“分组”的，本命令不会终止图流，而是“在逻辑上”中断图流，并把之后的点看作是“属于同一组”的点。例如，当从某个外部文件读取数据列表时，列表中的空行往往代表着“旧组的结束、新组的开始”。命令 \pgfplotstreamnewdataset 是新组与旧组的分界线，这个分界线所引起的动作决定于下一选项：

/pgf/handle new data sets in plots=⟨*how*⟩ (no default, initially jump)

这个选项的定义是：

```
\pgfset{
  handle new data sets in plots/.is choice,
  handle new data sets in plots/ignore/.code=\let\pgfplot@newdata
  ↪  \pgfplot@ignorer,
  handle new data sets in plots/jump/.code=\let\pgfplot@newdata\pgfplot@jumper,
}
```

/tikz/handle new data sets in plots=⟨*how*⟩ (no default, initially jump)

这个选项规定命令 \pgfplotstreamnewdataset 所引起的动作，⟨*how*⟩ 可以取以下值：

- ignore，忽略这个命令。
- jump，这个值导致内部宏 \pgf@plotstreamjump 被调用，会在图流中制造一个“缺口”。

\pgfplotstreamspecial={⟨*text*⟩}

本命令的定义是：

```
\def\pgfplotstreamspecial{\pgf@plotstreamspecial}%
```

这个命令导致内部宏 \pgf@plotstreamspecial 被调用，⟨*text*⟩ 是这个内部宏的参数，这个宏可以向图柄传递某些信息。所有“正常的”图柄都会忽略这个命令。

\pgfplotstreamend

本命令的定义是：

```
\def\pgfplotstreamend{\pgf@plotstreamend}%
```

本命令结束一个图流。这个命令调用内部宏 \pgf@plotstreamend，执行某些必要的收尾清理工作。

注意，图流不会被缓冲 (buffered)，图流中的坐标点会在读取时被立即处理。

27.2.2 生成图流的命令

可以手工创建一个图流，也可以利用外部的数据表文件创建图流。

\pgfplotxyfile{⟨file name⟩}

创建二维绘图流。⟨file name⟩ 是某个外部文件的名称，通常这个文件是个包含数据的文本文件。本命令的处理是：

1. 开启一个组 \begingroup
2. 执行

```
\def\b@pgfplotsxyfile@scanning@for@first{1}%
\pgfplotstreamstart%
```

3. 检查能否 (以读取方式) 打开文件 ⟨file name⟩, 如果不能打开就报错；如果能打开，就继续以下步骤：
4. 设置类代码

```
\catcode`\#=14% 注释符号
\catcode`\^^M=5% 回车符号
```

5. 执行 \pgf@readxyfile, 逐行读取文件 ⟨file name⟩ 的内容，每读取一行，处理一行。\pgf@readxyfile 的处理是：

   (a) 将读取的一行内容保存到宏 \pgf@temp, 然后彻底展开

```
\edef\pgf@temp{\pgf@temp}
```

   这说明在文件 ⟨file name⟩ 中可以使用宏，只要这些宏在此时有定义。然后检查：

   - 如果 \pgf@temp 是空的，或它保存的是 \par, 则

```
\if1\b@pgfplotsxyfile@scanning@for@first
\else
  \ifeof\r@pgf@reada\else\pgfplotstreamnewdataset\fi%
\fi
```

     也就是说，如果读取的是一个注释行，那么什么也不做；如果到达文件的结束处，那么什么也不做；否则就认为读取了一个空行，此时插入 \pgfplotstreamnewdataset.
   - 如果 \pgf@temp 不是以上情况，就执行

```
\expandafter\pgf@parsexyline\pgf@temp\pgf@stop%
```

     即用 \pgf@parsexyline 处理 \pgf@temp 保存的内容。

\pgf@parsexyline⟨arg1⟩ ⟨arg2⟩ ⟨arg3⟩\pgf@stop

这个命令的定义是：

```
\def\pgf@parsexyline#1 #2 #3\pgf@stop{%
  \def\b@pgfplotsxyfile@scanning@for@first{0}%
  \edef\pgf@xyline@flag@val{#3}%
  \ifx\pgf@xyline@flag@val\pgf@xyline@flag@undef%
    \pgfplotstreampointundefined%
  \else\ifx\pgf@xyline@flag@val\pgf@xyline@flag@out%
    \pgfplotstreampointoutlier{\pgfpointxy{#1}{#2}}%
  \else%
    \pgfplotstreampoint{\pgfpointxy{#1}{#2}}%
  \fi\fi%
}%

\edef\pgf@xyline@flag@out{o\space}%
\edef\pgf@xyline@flag@undef{u\space}%
```

这个命令的定义决定了文件 ⟨file name⟩ 一行内容的格式，或者彻底展开这一行内容后

的格式。

一行内容应当是使用空格作为分隔符号的列表：

- 列表项的个数可以是 0, 1, 2, 3, 4 等等，但至多利用前三项
- 如果第 3 个列表项是小写字母 u, 那么这一行就被转为 `\pgfplotstreampointundefined`
- 如果第 3 个列表项是小写字母 o, 那么这一行就被转为 `\pgfplotstreampointoutlier{\pgfpointxy{`⟨列表项 *1*⟩`}{`⟨列表项 *2*⟩`}}`
- 如果第 3 个列表项不是以上两个情况，那么这一行就被转为 `\pgfpointxy{`⟨列表项 *1*⟩`}{`⟨列表项 *2*⟩`}`

(b) 再检查是否到达文件的结束处，如果是就什么也不做；否则，就再次执行 `\pgf@readxyfile`, 再读取一行文件内容。

6. 在读完文件 ⟨*file name*⟩ 的内容后，插入 `\pgfplotstreamend`
7. 用 `\endgroup` 结束之前开启的组。

文件 ⟨*file name*⟩ 的格式及其转换如下所示：

```
% Some comments          \pgfplotstreamstart
0 Nan u                  \pgfplotstreamnewdataset
1 1 some text            \pgfplotstreampointundefined
3 9                      \pgfplotstreampoint{\pgfpointxy{1}{1}}
                         \pgfplotstreampoint{\pgfpointxy{3}{9}}
4 16 o                   \pgfplotstreamnewdataset
5 25 oo                  \pgfplotstreampointoutlier{\pgfpointxy{4}{16}}
                         \pgfplotstreampoint{\pgfpointxy{5}{25}}
                         \pgfplotstreamend
```

```
# Some comments          \pgfplotstreamstart
2 -5 2 first entry       \pgfplotstreamnewdataset
2 -.2 2 o                \pgfplotstreampoint{\pgfpointxy{2}{-5}}
2 -5 2 third entry       \pgfplotstreampointoutlier{\pgfpointxy{2}{-.2}}
                         \pgfplotstreampoint{\pgfpointxy{2}{-5}}
                         \pgfplotstreamend
```

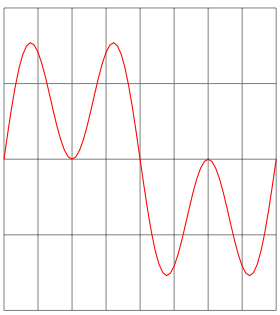
其中以 % 或 # 开头的行是注释行,注释行的内容会被忽略,注释行和空行都被转为命令 `\pgfplotstreamnewdataset`

`\pgfplotxyzfile{`⟨*file name*⟩`}`

用于创建三维图流。文件 ⟨*file name*⟩ 中的数据点由 3 个数字构成，数字会被解释到 xyz 坐标系统中。本命令各方面的特点类似 `\pgfplotxyfile`.

`\pgfplotfunction{`⟨*variable*⟩`}{`⟨*sample list*⟩`}{`⟨*function point*⟩`}`

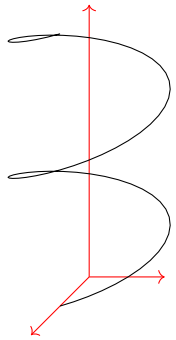
本命令利用函数创建图流，先看一个例子：



```
\begin{tikzpicture}[x=3.6cm/360]
  \draw[xstep=4.5mm,help lines](0,-2) grid (360,2);
  \pgfplothandlerlineto
  \pgfplotfunction{\t}{0,5,...,360}
    {\pgfpointxy{\t}{sin(\t)+sin(3*\t)}}
  \pgfsetstrokecolor{red}
  \pgfusepath{stroke}
\end{tikzpicture}
```

本命令中的 ⟨*variable*⟩ 和 ⟨*sample list*⟩ 就是 `\foreach` ⟨*variable*⟩ `in` ⟨*sample list*⟩`...` 中的变量和变

量取值列表，其格式要符合 \foreach 语句的规则。⟨*function point*⟩ 是 PGF 点，其中可以用数学表达式，数学表达式中的变量必须是 ⟨*variable*⟩ 中列出的变量。实际上 ⟨*function point*⟩ 必须是这样的命令或者代码：它能作为 \pgf@process 的参数，并且能设置尺寸寄存器 \pgf@x, \pgf@y 的值，这两个值会用作图流中的点的坐标。为了便于计算，在 ⟨*variable*⟩ 中可以列出数个变量，在 ⟨*function point*⟩ 中使用这些变量做计算。具体细节可参考此命令的定义。



```
\begin{tikzpicture}[y=3.6cm/360]
  \foreach \m in {(1,0,0),(0,360,0),(0,0,2)}
  \draw [->,red](0,0,0)--\m;
  \pgfplothandlerlineto
  \pgfplotfunction{\y}{0,5,...,360}
    {\pgfpointxyz{sin(2*\y)}{\y}{cos(2*\y)}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

若 ⟨*function point*⟩ 中的数学表达式很复杂，则可能导致处理过程变慢。这个命令的定义是：

```
\def\pgfplotfunction#1#2#3{%
  \pgfplotstreamstart%
  \foreach#1in{#2}%
  {%
    \pgf@process{#3}%
    \edef\pgf@marshal{\noexpand\pgfplotstreampoint{\noexpand\pgfqpoint{\the\pgf@x
    ↪  }{\the\pgf@y}}}%
    \pgf@marshal%
  }
  \pgfplotstreamend%
}%
```

可见本命令使用 \foreach 语句来得到图流中的点。

\pgfplotgnuplot[⟨*prefix*⟩]{⟨*function*⟩}

本命令调用外部程序 gnuplot 来创建函数 ⟨*function*⟩ 的图流，这里 ⟨*function*⟩ 是 gnuplot 的命令。如果不指定 ⟨*prefix*⟩，则它是 \jobname，即当前 tex 文件的名称。

在第一次处理本命令时，需要在命令行使用 -shell-escape 来编译。如果本命令成功运行，则会生成文件 ⟨*prefix*⟩.gnuplot 和 ⟨*prefix*⟩.table.

本命令的处理是：

1. 设置真值 \pgf@resample@plottrue
2. 尝试 (以读取方式) 打开文件 ⟨*prefix*⟩.gnuplot,
   - 如果不能打开 (文件不存在)，就什么也不做；
   - 如果能打开,就读取这个文件的第一行——保存到 \pgf@temp, 以及第二行——保存到 \pgf@plot@line. 然后关闭这个文件。

     然后检查 \pgf@plot@line 的内容是否参数 ⟨*function*⟩:
     - 如果是，就再尝试 (以读取方式) 打开文件 ⟨*prefix*⟩.table. 如果不能打开 (文件不存在)，就什么也不做；如果能打开，就关闭这个文件，并设置真值 \pgf@resample@plotfalse.
     - 如果不是，就什么也不做。
3. 检查 \ifpgf@resample@plot 的真值，
   - 如果真值是 false, 则什么也不做。
   - 如果真值是 true, 则

     (a) 立即以写入方式打开文件 ⟨*prefix*⟩.gnuplot, 如果这个文件不存在，就创建这个文件 (通

常是在当前文件夹中)。

(b) 立即向文件 ⟨*prefix*⟩.gnuplot 中写入

```
set table "⟨prefix⟩.table"; set format "%.5f"
⟨function⟩
```

这会导致原来的文件内容被清除。

(c) 立即关闭文件 ⟨*prefix*⟩.gnuplot

(d) 执行系统命令

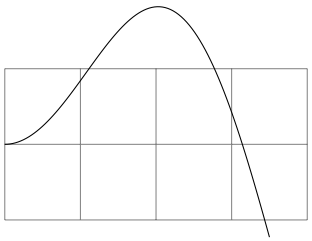
```
\pgfutil@shellescape{%
  \pgfkeysvalueof{/pgf/plot/gnuplot call} \pgf@plotgnuplotfile}%
在默认下这就是执行
\immediate\write18{gnuplot ⟨prefix⟩.gnuplot}%
```

如果这一步能成功，那么 gnuplot 会 (通常是在当前文件夹中) 创建文件 ⟨*prefix*⟩.table, 这个文件的内容是需要的数据信息。

4. 执行 \pgfplotxyfile{⟨*prefix*⟩.table}, 生成一个图流。

按以上处理过程，如果文件 ⟨*prefix*⟩.gnuplot 和 ⟨*prefix*⟩.table 都已经存在，并且 ⟨*prefix*⟩.gnuplot 的第二行就是参数 ⟨*function*⟩, 那么就用真值 \pgf@resample@plotfalse, 此时本命令就不会调用 gunplot, 也就不会再次创建 ⟨*prefix*⟩.table 文件 (这个情况下用户最好不要自行修改 .table 文件的数据)。

例如，调用 gnuplot 创建下面的图形：



```
\begin{tikzpicture}
  \draw[help lines] (0,-1) grid (4,1);
  \pgfplothandlerlineto
  \pgfplotgnuplot[pgfplotgnuplot-example]{plot [x=0:3.5] x*sin(x)}
  \pgfusepath{stroke}
\end{tikzpicture}
```

得到的 .gnuplot 文件内容如下：

```
set table "pgfplotgnuplot-example.table"; set format "%.5f"
plot [x=0:3.5] x*sin(x)
```

得到的 .table 文件内容如下：

```
# Curve 0 of 1, 100 points
# Curve title: "x*sin(x)"
# x y type
0.00000 0.00000  i
0.03535 0.00125  i
0.07071 0.00500  i
% 省略若干
```

/pgf/plot/gnuplot call=⟨*gnuplot invocation*⟩ (no default, initially gnuplot)

这个选项可以改变 gnuplot 的调用方式。对于有的 MikTeX 发行版需要如下设置：

```
\pgfkeys{/pgf/plot/gnuplot call="/Programs/gnuplot/binary/gnuplot"}
```

27.3 图柄

图柄（plot handler）决定用什么方式将图流中的点联系起来。你必须在执行图流的代码之前写出所用的图柄。图柄会定义或重定义 \pgf@plotstreamstart，\pgf@plotstreampoint，\pgf@plotstreamend

等命令，这些命令一起组成了“图柄”。引入一个图柄 (即定义这些命令) 后，再展开一个图流，就得到一个路径。

注意，构成图柄的那些命令是被全局定义的，要想修改那些命令，可以引入另一个图柄。

下面是 PGF 预定义的几个图柄，另外库 pgflibraryplothandlers 也定义了大量图柄。

\pgf@plot@first@action{⟨*point*⟩}

多数预定义的图柄都会利用这个命令处理图流中的第一个点坐标。

本命令的初始值是 \pgfpathmoveto:

```
\let\pgf@plot@first@action=\pgfpathmoveto
```

\pgfsetmovetofirstplotpoint

本命令的定义是：

```
\def\pgfsetmovetofirstplotpoint{\let\pgf@plot@first@action=\pgfpathmoveto}%
```

本命令的作用是令 \pgf@plot@first@actio 等于 \pgfpathmoveto, 它能处理 1 个 PGF 坐标点。

宏 \pgf@plot@first@actio 会被某些图柄调用,将 move-to 操作作用于图流的第一个点。例如 \pgfplothandlerli 会调用本命令。

\pgfsetlinetofirstplotpoint

本命令的定义是：

```
\def\pgfsetlinetofirstplotpoint{\let\pgf@plot@first@action=\pgfpathlineto}%
```

本命令的作用是令 \pgf@plot@first@actio 等于 \pgfpathlineto, 它能处理 1 个 PGF 坐标点。

宏 \pgf@plot@first@actio 会被某些图柄调用，将 line-to 操作作用于图流的第一个点。

\pgfplothandlerlineto

这个图柄的定义是：

```
\pgfdeclareplothandler{\pgfplothandlerlineto}{}{
  point macro=\pgf@plot@line@handler,
  jump=\global\let\pgf@plotstreampoint\pgf@plot@line@handler@move%
}%

\def\pgf@plot@line@handler#1{%
  \pgf@plot@first@action{#1}%
  \global\let\pgf@plotstreampoint=\pgfpathlineto%
}%

\def\pgf@plot@line@handler@move#1{%
  \pgfpathmoveto{#1}%
  \global\let\pgf@plotstreampoint=\pgfpathlineto%
}%
```

使用这个图柄后：

- \pgfplotstreamstart 导致执行选项

  ```
  point macro=\pgf@plot@line@handler,
  jump=\global\let\pgf@plotstreampoint\pgf@plot@line@handler@move%
  ```

- 第一个图流命令 \pgfplotstreampoint{⟨*point*⟩} 导致

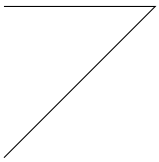
  ```
  \pgf@plot@first@action{⟨point⟩}%
  \global\let\pgf@plotstreampoint=\pgfpathlineto%
  ```

  即用 \pgf@plot@first@action 处理图流中的第一个点，然后再令 \pgf@plotstreampoint 等于 \pgfpathlineto, 处理之后的图流点。

- 命令 \pgf@plotstreamjump 会导致 \pgf@plotstreampoint 被重定义。
  \pgf@plotstreamjump\pgf@plotstreampoint{⟨*point*⟩} 导致

```
\pgfpathmoveto{⟨point⟩}%
\global\let\pgf@plotstreampoint=\pgfpathlineto%
```

这个图柄将命令 \pgfpathlineto 用于图流中的点，但本身不决定如何处理图流的第一个点。



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfplothandlerlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfplothandlerpolygon

这个图柄类似 \pgfplothandlerlineto,只是本命令会在图流的最后一个点之后加上命令 \pgfpathclose，将图流作成一个封闭的多边形。

\pgfplothandlerdiscard

这个图柄直接把图流“扔掉”，这个命令用在图流开始前才有效。

这个图柄的定义是：

```
\pgfdeclareplothandler{\pgfplothandlerdiscard}{}{}%
```

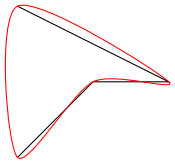
这个声明导致：

```
\def\pgfplothandlerdiscard{%
  \pgfkeys{%
    /pgf/plots/@handler options/.cd,
    start=\relax,
    end macro=\relax,
    point macro=\pgfutil@gobble,
    jump macro=\relax,
    special macro=\pgfutil@gobble,%
  }%
}%
```

使用这个图柄会导致图流中的所有数据都被吃掉，什么也不留下。

\pgfplothandlerrecord{⟨*macro*⟩}

⟨*macro*⟩ 是个宏，不需要提前定义。写出本命令后，本命令之后的一个图流会被保存在宏 ⟨*macro*⟩ 中，然后可以用宏 ⟨*macro*⟩ 调用这个图流。注意本命令只能保存一个图流，如果本命令之后有数个图流，那么本命令只保存最后一个图流。本命令对宏 ⟨*macro*⟩ 的定义是全局地 (全局地保存一个图流)。



```
\begin{pgfpicture}
  \pgfplothandlerrecord{\mystream}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfplothandlerlineto
  \mystream
  \pgfusepath{stroke}
  \pgfplothandlerclosedcurve
  \mystream
  \pgfsetstrokecolor{red}
  \pgfusepath{stroke}
\end{pgfpicture}
```

其定义是：

```
\pgfdeclareplothandler{\pgfplothandlerrecord}{#1}{%
  start=\gdef#1{\pgfplotstreamstart},
  point=\expandafter\gdef\expandafter#1\expandafter{#1\pgfplotstreampoint{##1}},
  jump=\expandafter\gdef\expandafter#1\expandafter{#1\pgf@plotstreamjump},
  special=\expandafter\gdef\expandafter#1\expandafter{#1\pgfplotstreamspecial{##1
  ↪  }},
  end=\expandafter\gdef\expandafter#1\expandafter{#1\pgfplotstreamend},
}%
```

可见 \pgfplothandlerrecord 被声明为一个图柄，这个声明这导致：

```
\def\pgfplothandlerrecord#1{%
  \pgfkeys{%
    /pgf/plots/@handler options/.cd,
    start=\relax,
    end macro=\relax,
    point macro=\pgfutil@gobble,
    jump macro=\relax,
    special macro=\pgfutil@gobble,%
    start=\gdef#1{\pgfplotstreamstart},
    point=\expandafter\gdef\expandafter#1\expandafter{#1\pgfplotstreampoint{##1}},
    jump=\expandafter\gdef\expandafter#1\expandafter{#1\pgf@plotstreamjump},
    special=\expandafter\gdef\expandafter#1\expandafter{#1\pgfplotstreamspecial
    ↪  {##1}},
    end=\expandafter\gdef\expandafter#1\expandafter{#1\pgfplotstreamend},
  }%
}%
```

执行 \pgfplothandlerrecord{\⟨*macro*⟩} 导致：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  start=\gdef\⟨macro⟩{\pgfplotstreamstart},
  point=\expandafter\gdef\expandafter\⟨macro⟩\expandafter{\⟨macro⟩
  ↪  \pgfplotstreampoint{#1}},
  jump=\expandafter\gdef\expandafter\⟨macro⟩\expandafter{\⟨macro⟩
  ↪  \pgf@plotstreamjump},
  special=\expandafter\gdef\expandafter\⟨macro⟩\expandafter{\⟨macro⟩
  ↪  \pgfplotstreamspecial{#1}},
  end=\expandafter\gdef\expandafter\⟨macro⟩\expandafter{\⟨macro⟩\pgfplotstreamend},
}%
```

这导致重定义命令

- \pgf@plotstreamstart
- \pgf@plotstreampoint
- \pgf@plotstreamjump
- \pgf@plotstreamspecial
- \pgf@plotstreamend

当执行 \pgfplotstreamstart, \pgfplotstreampoint, \pgfplotstreamend 等命令时，实际是在对 \⟨*macro*⟩ 进行重定义。在执行 \pgfplotstreamend 后，\⟨*macro*⟩ 就保存一个图流。

27.4 定义新图柄

定义图柄的过程中用到 3 个层次的命令：(1) 底层的命令，命令名称都以 @init 结尾；(2) 中间层的命令，命令名称不以 @init 结尾，但有符号 @; (3) 顶层的命令，即用户直接使用的命令。底层的命令可以用“选项”修改。

\pgfdeclareplothandler{⟨*macro*⟩}{⟨*arguments*⟩}{⟨*configuration*⟩}

在《pgfmoduleplot.code.tex》中这个命令的定义是：

```
\def\pgfdeclareplothandler#1#2#3{%
  \def#1#2{%
    \pgfkeys{%
      /pgf/plots/@handler options/.cd,
      start=\relax,
      end macro=\relax,
      point macro=\pgfutil@gobble,
      jump macro=\relax,
      special macro=\pgfutil@gobble,%
      #3%
    }%
  }%
}%
```

可见执行 \pgfdeclareplothandler{⟨*macro*⟩}{⟨*arguments*⟩}{⟨*configuration*⟩} 的结果是给出宏 ⟨*macro*⟩ 的定义，⟨*arguments*⟩ 是定义中的变量列举格式，如 #1，#1#2#3 这种形式；而 ⟨*configuration*⟩ 是能被命令 \pgfkeys 处理的键值列表，这些键的前缀路径都是 /pgf/plots/@handler options.

例如定义一个图柄 \myhandler:

```
\pgfdeclareplothandler{\myhandler}{#1}{...}
```

下文介绍省略号处可以使用的选项，顺便以图柄 \myhandler 为例介绍这些选项。

/pgf/plots/@handler options/start=⟨*code*⟩

这个键的定义是：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  start/.code=%
    \gdef\pgf@plotstreamstart{%
      \global\pgf@plot@startedfalse%
      \global\let\pgf@plotstreamend\pgf@plotstreamend@init%
      \global\let\pgf@plotstreampoint\pgf@plotstreampoint@init%
      \global\let\pgf@plotstreamjump\pgf@plotstreamjump@init%
      \global\let\pgf@plotstreamspecial\pgf@plotstreamspecial@init%
```

```
      #1%
    },%
}
```

执行这个键导致全局地定义命令 \pgf@plotstreamstart.

因为命令 \pgfplotstreamstart→P. 487 就是 \pgf@plotstreamstart, 所以在执行这个键之后，展开命令 \pgfplotstreamstart 有两方面作用：

1. 全局地定义
   - \pgf@plotstreamend 等于 \pgf@plotstreamend@init,
   - \pgf@plotstreampoint 等于 \pgf@plotstreampoint@init,
   - \pgf@plotstreamjump 等于 \pgf@plotstreamjump@init,
   - \pgf@plotstreamspecial 等于 \pgf@plotstreamspecial@init
2. 执行 ⟨*code*⟩，⟨*code*⟩ 中可以使用 ⟨*arguments*⟩ 列出的变量参数。

Hi 某.Bye 某.

```
\pgfdeclareplothandler{\myhandler}{#1}{
  start = Hi #1.,
  end = Bye #1.,
}
\myhandler{某}
\pgfplotstreamstart
\pgfplotstreamend
```

/pgf/plots/@handler options/end=⟨*code*⟩

这个键的定义是：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  end/.code=\gdef\pgf@plotstreamend@init{#1},
}
```

执行这个键导致全局地定义命令 \pgf@plotstreamend@init.

如前述，展开命令 \pgfplotstreamstart 时需要用到 \pgf@plotstreamend@init.

⟨*code*⟩ 中可以使用 ⟨*arguments*⟩ 列出的变量参数。

/pgf/plots/@handler options/point=⟨*code*⟩

这个键的定义是：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  point/.code=\gdef\pgf@plotstreampoint@init##1{#1},
}
```

执行这个键导致全局地定义命令 \pgf@plotstreampoint@init:

```
\gdef\pgf@plotstreampoint@init#1{⟨code⟩}
```

注意 \pgf@plotstreampoint@init 需要处理 1 个参数。

⟨*code*⟩ 中可以使用 ⟨*arguments*⟩ 列出的变量参数。

命令 \pgfplotstreampoint→P. 487{⟨*point*⟩} 就是 \pgf@plotstreampoint@init{⟨*point*⟩}, 如果希望 ⟨*code*⟩ 处理 ⟨*point*⟩, 那么在 ⟨*code*⟩ 中应使用处理 ##1 的代码。

Hi 某甲.
nice 天气, 嗯
Bye 某乙.
nice 天气, 啊

```
\pgfdeclareplothandler{\myhandler}{#1#2#3}{
  start = Hi #1.,
  end = Bye #2.,
  point = {nice #3, ##1}
}
\myhandler{某甲}{某乙}{天气}
\color{red}
\pgfplotstreamstart \\
\pgfplotstreampoint{嗯} \\
\color{cyan}
\pgfplotstreamend \\
\pgfplotstreampoint{啊}
```

/pgf/plots/@handler options/jump=⟨*code*⟩

这个键的定义是：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  jump/.code=\gdef\pgf@plotstreamjump@init{#1},
}
```

执行这个键导致全局地定义命令 \pgf@plotstreamjump@init.

命令

- \pgfplotstreampointoutlier → P. 487
- \pgfplotstreampointundefined → P. 488
- \pgfplotstreamnewdataset → P. 489

都可能导致 \pgf@plotstreamjump@init 被执行。

在 ⟨*code*⟩ 中可以使用 ⟨*arguments*⟩ 列出的变量。

/pgf/plots/@handler options/special=⟨*code*⟩

这个键的定义是：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  special/.code=\gdef\pgf@plotstreamspecial@init##1{#1},
}
```

执行这个键导致全局地定义命令 \pgf@plotstreamspecial@init:

```
\gdef\pgf@plotstreamspecial@init#1{⟨code⟩}
```

注意 \pgf@plotstreampoint@init 需要处理 1 个参数。

在 ⟨*code*⟩ 中可以使用 ⟨*arguments*⟩ 列出的变量。

命令 \pgfplotstreamspecial → P. 489 等于这个命令。当执行 \pgfplotstreamspecial{⟨*something*⟩} 时，如果希望 ⟨*code*⟩ 处理 ⟨*something*⟩, 那么在 ⟨*code*⟩ 中应使用处理 ##1 的代码。

/pgf/plots/@handler options/point macro=⟨*a macro*⟩

这个键的定义是：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  point macro/.code=\global\let\pgf@plotstreampoint@init#1,
}
```

这个键全局地使命令 \pgf@plotstreampoint@init 等于 ⟨*a macro*⟩.

⟨*a macro*⟩ 是个宏，使用这个选项后，图流命令 \pgfplotstreampoint 等同于宏 ⟨*a macro*⟩. 宏 ⟨*a macro*⟩ 应当能处理 1 个参数。

/pgf/plots/@handler options/special macro=⟨*a macro*⟩

这个键的定义是：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  special macro/.code=\global\let\pgf@plotstreamspecial@init#1,
}
```

这个键全局地使命令 \pgf@plotstreamspecial@init 等于 ⟨*a macro*⟩.

⟨*a macro*⟩ 是个宏，使用这个选项后，图流命令 \pgfplotstreamspecial 等于宏 ⟨*a macro*⟩. 宏 ⟨*a macro*⟩ 应当能处理 1 个参数。

/pgf/plots/@handler options/start macro=⟨*code*⟩

这个键等效于 /pgf/plots/@handler options/start, 它们的定义是一样的。

/pgf/plots/@handler options/end macro=⟨*a macro*⟩

这个键的定义是：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  end macro/.code=\global\let\pgf@plotstreamend@init#1,
}
```

这个键全局地使命令 \pgf@plotstreamend@init 等于 ⟨*a macro*⟩. 执行这个选项后,图流命令 \pgfplotstreamend 等于宏 ⟨*a macro*⟩.

/pgf/plots/@handler options/jump macro=⟨*a macro*⟩

这个键的定义是：

```
\pgfkeys{%
  /pgf/plots/@handler options/.cd,
  jump macro/.code=\global\let\pgf@plotstreamjump@init#1,
}
```

这个键全局地使命令 \pgf@plotstreamjump@init 等于 ⟨*a macro*⟩.

27.5 \pgfplothandlercurveto

在文件《pgflibraryplothandlers.code.tex》中定义了图柄 \pgfplothandlercurveto：

```
\pgfdeclareplothandler{\pgfplothandlercurveto}{}{%
  point macro=\pgf@plot@curveto@handler@initial,
  jump macro=\pgf@plot@smooth@next@moveto,
  end macro=\pgf@plot@curveto@handler@finish
}%

\def\pgf@plot@smooth@next@moveto{%
  \pgf@plot@curveto@handler@finish%
  \global\pgf@plot@startedfalse%
  \global\let\pgf@plotstreampoint\pgf@plot@curveto@handler@initial%
}%

\def\pgf@plot@curveto@handler@initial#1{%
```

```
  \pgf@process{#1}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@plot@first@action{\pgfqpoint{\pgf@xa}{\pgf@ya}}%
  \xdef\pgf@plot@curveto@first{\noexpand\pgfqpoint{\the\pgf@xa}{\the\pgf@ya}}%
  \global\let\pgf@plot@curveto@first@support=\pgf@plot@curveto@first%
  \global\let\pgf@plotstreampoint=\pgf@plot@curveto@handler@second%
}%

\def\pgf@plot@curveto@handler@second#1{%
  \pgf@process{#1}%
  \xdef\pgf@plot@curveto@second{\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}%
  \global\let\pgf@plotstreampoint=\pgf@plot@curveto@handler@third%
  \global\pgf@plot@startedtrue%
}%

\def\pgf@plot@curveto@handler@third#1{%
  \pgf@process{#1}%
  \xdef\pgf@plot@curveto@current{\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}%
  % compute difference vector:
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{\pgf@plot@curveto@first}
  \advance\pgf@xa by-\pgf@x%
  \advance\pgf@ya by-\pgf@y%
  % compute support directions:
  \pgf@xa=\pgf@plottension\pgf@xa%
  \pgf@ya=\pgf@plottension\pgf@ya%
  % first marshal:
  \pgf@process{\pgf@plot@curveto@second}%
  \pgf@xb=\pgf@x%
  \pgf@yb=\pgf@y%
  \pgf@xc=\pgf@x%
  \pgf@yc=\pgf@y%
  \advance\pgf@xb by-\pgf@xa%
  \advance\pgf@yb by-\pgf@ya%
  \advance\pgf@xc by\pgf@xa%
  \advance\pgf@yc by\pgf@ya%
  \edef\pgf@marshal{\noexpand\pgfpathcurveto{\noexpand\pgf@plot@curveto@first@support}
  ↪  %
    {\noexpand\pgfqpoint{\the\pgf@xb}{\the\pgf@yb}}{\noexpand\pgf@plot@curveto@second
    ↪  }}%
  {\pgf@marshal}%
```

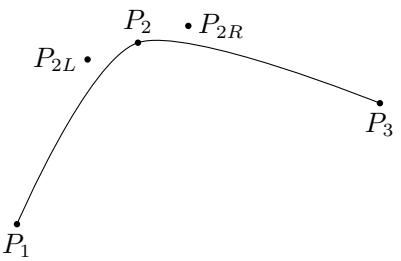
```
55    % Prepare next:
56    \global\let\pgf@plot@curveto@first=\pgf@plot@curveto@second%
57    \global\let\pgf@plot@curveto@second=\pgf@plot@curveto@current%
58    \xdef\pgf@plot@curveto@first@support{\noexpand\pgfqpoint{\the\pgf@xc}{\the\pgf@yc}}%
59  }%
60
61  \def\pgf@plot@curveto@handler@finish{%
62    \ifpgf@plot@started%
63      \pgfpathcurveto{\pgf@plot@curveto@first@support}{\pgf@plot@curveto@second}{
      ↪  \pgf@plot@curveto@second}%
64    \fi%
65  }%
66
67
68  % This commands sets the tension for smoothing of plots.
69  %
70  % #1 = tension of curves. A value of 1 will yield a circle when the
71  %      control points are at quarters of a circle. A smaller value
72  %      will result in a tighter curve. Default is 0.5.
73  %
74  % Example:
75  %
76  % \pgfsetplottension{0.2}
77
78  \def\pgfsetplottension#1{%
79    \pgf@x=#1pt\relax%
80    \pgf@x=0.2775\pgf@x\relax%
81    \edef\pgf@plottension{\pgf@sys@tonumber\pgf@x}}%
82  \pgfsetplottension{0.5}%
```

上面第 78 至 82 行表明：若 \pgfsetplottension 的参数是 t，则 \pgf@plottension 的值是 $\bar{t} = 0.2775 \times t$，默认的是 $\bar{t} = 0.2775 \times 0.5 = 0.13875$，这个乘积用在了第 40, 41 行。

图柄 \pgfplothandlercurveto 的作用是，例如：

- 记宏 \pgf@plottension 的值是 $\bar{t}$，假设图柄 \pgfplothandlercurveto 辖制的图流中有 3 个点 P_1, P_2, P_3，则：

  1. 执行 \pgf@plot@first@action{P_1}，通常这就是 \pgfpathmoveto{P_1}，点 P_1 成为当前点；
  2. 计算 $P_{2L} = P_2 - \bar{t} \cdot (P_3 - P_1)$，$P_{2R} = P_2 + \bar{t} \cdot (P_3 - P_1)$；
  3. 以当前点 P_1 为起点，执行 \pgfpathcurveto{P_1}{P_{2L}}{P_2}，此时 P_2 成为当前点；
  4. 以当前点 P_2 为起点，执行 \pgfpathcurveto{P_{2R}}{P_3}{P_3}.

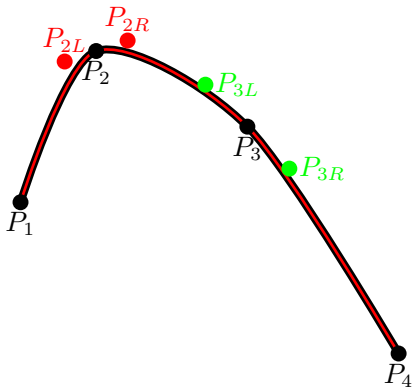



```
\begin{tikzpicture}[scale=0.8]
  \pgfplothandlercurveto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{3cm}}
  \pgfplotstreampoint{\pgfpoint{6cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
  \fill (0,0) circle [radius=1.5pt] coordinate (p1) node[below]{$P_1$};
  \fill (2,3) circle [radius=1.5pt] coordinate (p2) node[above]{$P_2$};
  \fill (6,2) circle [radius=1.5pt] coordinate (p3) node[below]{$P_3$};
  \coordinate (p13) at ($(p3)-(p1)$);
  \fill ($(p2)-0.13875*(p13)$) circle [radius=1.5pt] coordinate (q12) node[left]{$P_{2L}$};
  \fill ($(p2)+0.13875*(p13)$) circle [radius=1.5pt] coordinate (q23) node[right]{$P_{2R}$};
\end{tikzpicture}
```

- 记宏 \pgf@plottension 的值是 $\bar{t}$，假设图柄 \pgfplothandlercurveto 辖制的图流中有 4 个点 P_1, P_2, P_3，P_4，则：
  1. 执行 \pgf@plot@first@action{P_1}，通常这就是 \pgfpathmoveto{P_1}，点 P_1 成为当前点；
  2. 计算 $P_{2L} = P_2 - \bar{t}\cdot(P_3 - P_1)$，$P_{2R} = P_2 + \bar{t}\cdot(P_3 - P_1)$；
  3. 以当前点 P_1 为起点，执行 \pgfpathcurveto{P_1}{P_{2L}}{P_2}，此时 P_2 成为当前点；
  4. 计算 $P_{3L} = P_3 - \bar{t}\cdot(P_4 - P_2)$，$P_{3R} = P_3 + \bar{t}\cdot(P_4 - P_2)$；
  5. 以当前点 P_2 为起点，执行 \pgfpathcurveto{P_{2R}}{P_{3L}}{P_3}，此时 P_3 成为当前点；
  6. 以当前点 P_3 为起点，执行 \pgfpathcurveto{P_{3R}}{P_4}{P_4}.

可见图柄 \pgfplothandlercurveto 是一种插值方法。图柄 \pgfplothandlercurveto 构造数段首尾相接的 3 次控制曲线，其中的第一段和最后一段曲线有点特别——有控制点重合。

下面例子中先用 \pgfplothandlercurveto 画线，再用 Ti*k*Z 的 curve-to 操作画线，两条线重合：




```
\begin{tikzpicture}
    \begin{pgfscope}
        \pgfsetlinewidth{3pt}
        \pgfplothandlercurveto
        \pgfplotstreamstart
        \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
        \pgfplotstreampoint{\pgfpoint{2cm}{2cm}}
```

```
        \pgfplotstreampoint{\pgfpoint{4cm}{1cm}}
        \pgfplotstreampoint{\pgfpoint{6cm}{-2cm}}
        \pgfplotstreamend
        \pgfusepath{stroke}
    \end{pgfscope}

  \tikzmath{
    \T=0.13875; % =0.2775*0.5, 因为默认 \pgfsetplottension{0.5}
    coordinate \p,\q,\r;
    \p1=(1,0);
    \p2=(2,2);
    \p3=(4,1);
    \p4=(6,-2);
    \q1=($(\p2)-\T*($(\p3)-(\p1)$)$);
    \q2=($(\p2)+\T*($(\p3)-(\p1)$)$);
    \r1=($(\p3)-\T*($(\p4)-(\p2)$)$);
    \r2=($(\p3)+\T*($(\p4)-(\p2)$)$);
  }
  \draw [red,line width=1pt](\p1)..controls(\p1)and(\q1)..(\p2)..controls(\q2)and(\r1)..(
  ↪  \p3)..controls(\r2)and(\p4)..(\p4);

  \foreach \i/\j in{(\p1)/{P_1},(\p2)/{P_2},(\p3)/{P_3},(\p4)/{P_4}}
      \fill \i circle (3pt) node [below] {$\j$};
  \foreach \i/\j in{(\q1)/{P_{2L}},(\q2)/{P_{2R}}}
      \fill [red] \i circle (3pt) node [above] {$\j$};
  \foreach \i/\j in{(\r1)/{P_{3L}},(\r2)/{P_{3R}}}
      \fill [green] \i circle (3pt) node [right] {$\j$};
\end{tikzpicture}
```

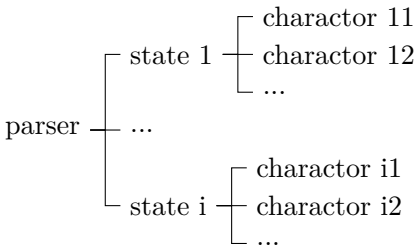
第二十八章　parser 模块

```
\usepgfmodule{parser} % LaTeX and plain TeX and pure pgf
\usepgfmodule[parser] % ConTeXt and pure pgf
```

这个模块提供一些命令，用于创建简单的“逐字解析的”(letter-by-letter) 解析器，实际上是“逐个记号”的解析器。

对于给定的一串字符——其中可以有特殊符号，例如 {，}，#，␣ 等，并且其中的括号不必平衡匹配——解析器逐个扫描 (吃掉) 其中的字符 (实际上是记号)：首先解析器处于初始状态，在这个状态下扫描第一个字母 (记号)，根据扫描结果来执行相应的代码，并且解析器可能会切换到另一个状态 (也可能不切换到其它状态)，然后扫描第二个字母 (记号)；同样根据扫描结果来执行相应的代码，且解析器可能会切换到另一个状态，再扫描第三个字母 (记号)，如此继续，直到解析器达到终结状态 (名称为 final, 这是个关键词)，结束解析过程。

一个解析器可以包含多个状态，一个状态可以处理多个字符：



如上图所示，在本模块的 \pgfparserdef 命令的处理下，一个“⟨*parser-state-char*⟩”组合对应两个命令，一个 type 命令：

```
\expandafter\def\csname pgfparser ⟨parser⟩ ⟨state⟩ ⟨by meaning⟩ type\endcsname{...}
```

一个 code 命令：

```
\expandafter\def\csname pgfparser ⟨parser⟩ ⟨state⟩ ⟨by meaning⟩ code\endcsname...{...}
```

其中的 ⟨*by meaning*⟩ 是与 \meaning⟨字符⟩ 有关的符号。

下面是个例子。

ccc There are 9 a’ s.

```
\newcount\mycount
\pgfparserdef{myparser}{initial}{the letter a}{\advance\mycount by 1\relax}
\pgfparserdef{myparser}{initial}{the letter b}{}
\pgfparserdef{myparser}{initial}{the letter c}{\pgfparserswitch{final}}
\pgfparserparse{myparser}aabaababababbbbbabaabcccc
There are \the\mycount\ a’ s.
```

上面例子中，使用 3 个 \pgfparserdef 命令定义解析器 myparser, 为这个解析器定义了 1 种状态 (名称为 initial) 的 3 个字符命令 (分别针对字母 a, b, c)。上面例子中的字符串的结束处有 4 个字母 c, 第一个 c 被解析器扫描 (吃掉) 并导致解析器切换到 final 状态，结束解析过程，剩余 3 个字母 c 被打印到屏幕上。

28.1 基本命令

\pgfparserdef{⟨parser name⟩}{⟨state⟩}⟨symbol meaning⟩[⟨arguments⟩]{⟨action⟩}

\pgfparserdef{⟨parser name⟩}{⟨state⟩}{⟨symbol meaning⟩}[⟨arguments⟩]{⟨action⟩}

关于本命令：

1. 参数 ⟨parser name⟩ 是解析器名称，⟨state⟩ 是状态名称。
2. 必须定义的状态：按 \pgfparserparse→P. 511 的处理过程，必须定义的状态是 initial 或者 all. 在命令 \pgfparserparse 的开始处，把初始状态名称定义为 initial, 然后再开始解析操作，读取字符。在解析过程中，如果读取了某个字符，但在当前状态下没有找到能够解析这个字符的命令，就自动地去 all 状态下查找解析这个字符的命令。
   解析过程的最后一个状态是名称为 final 的状态，用户可以不定义这个状态，但必须指明在什么情况下能切换到、并且要确保能切换到这个终结状态。在默认下，这个终结状态只是结束解析过程，其他的什么也不做。
   如果要定义 final 状态，应当使用命令 \pgfparserdeffinal→P. 514 来定义。
3. 本命令会先调用 \pgfparser@initiate@parser→P. 510, 定义与空格符号 (blank space) 有关的 all 状态的命令：
   - 把 \csname ifpgfparserconditional ⟨parser name⟩ silent\endcsname 声明为一个 TEX-if.
   - 定义键：

```
\pgfparserset{%
  ⟨parser name⟩/silent/.is if=\pgfparser@ifname{#1 silent}%
}% 这个 TeX-if 就是
%\csname ifpgfparserconditional ⟨parser name⟩ silent\endcsname
```

   - 定义关于 ⟨parser name⟩-all-⟨空格⟩ 组合的 type 命令和 code 命令：
     – 定义 ⟨parser name⟩-all-⟨空格⟩-type 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ all blank space \space
↪  \space type\endcsname{noarg}
```

     – 定义 ⟨parser name⟩-all-⟨空格⟩-code 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ all blank space \space
↪  \space code\endcsname{\pgfparser@getnexttoken}
```

   然后再定义与字符 ⟨symbol meaning⟩ 有关的状态命令。
4. 关于参数 ⟨symbol meaning⟩
   - 如果参数 ⟨symbol meaning⟩ 是空格，或者宏 \pgfutil@sptoken, 并且不被花括号包裹，那么会被忽略；如果被花括号包裹，可能比较难理解。最好不要这么做。
     宏 \pgfutil@sptoken 是被全局定义的：

```
{\def\:{\global\let\pgfutil@sptoken= } \: }
```

   - 如果参数 ⟨symbol meaning⟩ 被花括号包裹，即写出 {⟨symbol meaning⟩}, 则 ⟨symbol meaning⟩ 不会被 \meaning 处理；如果参数 ⟨symbol meaning⟩ 不被花括号包裹，则 ⟨symbol meaning⟩ 会被 \meaning 处理。
   - 参数 ⟨symbol meaning⟩ 的形式可以是：
     – 能被命令 \meaning 处理的 ⟨单个记号⟩，注意此时不能用花括号包裹 ⟨单个记号⟩，否则 ⟨单个记号⟩ 就不会被命令 \meaning 处理。

```
\pgfparserdef{myparser}{initial}a{foo}
% 不可以 \pgfparserdef{myparser}{initial}{a}{foo}
```

在这个情况下 \pgfparserdef 会定义 type 命令和 code 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ \meaning⟨ 单个记号⟩
↪  type\endcsname{...}
以及
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ \meaning⟨ 单个记号⟩
↪  code\endcsname...{...}
```

– 可以是命令“\meaning⟨单个记号⟩”，或者这种命令的处理结果，或者是保存这个处理结果的宏——暂且都记为 ⟨*meaning*⟩, 注意此时必须用花括号包裹 ⟨*meaning*⟩, 否则 ⟨*meaning*⟩ 会再次被 \meaning 处理。

在这个情况下 \pgfparserdef 会定义 type 命令和 code 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ ⟨meaning⟩ type
↪  \endcsname{...}
以及
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ ⟨meaning⟩ code
↪  \endcsname...{...}
```

例如，

* 可以写：

```
\pgfparserdef{myparser}{initial}{\meaning\bgroup}{⟨action⟩}
```

此时 \pgfparserdef 会定义 type 命令和 code 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ \meaning\bgroup
↪  type\endcsname{...}
以及
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ \meaning\bgroup
↪  code\endcsname...{...}
```

* 因为

the letter A `\meaning A`

所以可以写：

```
\edef\aaaa{\meaning{}
\pgfparserdef{myparser}{initial}{the letter A}{⟨action⟩}
```

此时 \pgfparserdef 会定义 type 命令和 code 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ the letter A type
↪  \endcsname{...}
以及
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ the letter A code
↪  \endcsname...{...}
```

* 因为

begin-group character { `\meaning{`

所以可以写：

```
\edef\aaaa{\meaning{}
\pgfparserdef{myparser}{initial}{\aaaa}{⟨action⟩}
```

此时 \pgfparserdef 会定义 type 命令和 code 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ \aaaa type
↪  \endcsname{...}
以及
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ \aaaa code
↪  \endcsname...{...}
```

* 因为

  begin-group character { `\meaning\bgroup`

  所以可以写：

```
\def\aaaa{\meaning\bgroup}
\pgfparserdef{myparser}{initial}{\aaaa}{⟨action⟩}
```

  此时 \pgfparserdef 会定义 type 命令和 code 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ \aaaa type
↪  \endcsname{...}
以及
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ \aaaa code
↪  \endcsname...{...}
```

– 可以是字符串“blank space”，这个字符串必须用花括号包裹起来。
在这个情况下 \pgfparserdef 会定义 type 命令和 code 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ blank space \space
↪  \space type\endcsname{...}
以及
\expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ blank space \space
↪  \space code\endcsname...{...}
```

实际上

```
\pgfparserdef{myparser}{initial}{the letter a}{foo}
等效于
\pgfparserdef{myparser}{initial}a{foo}
```

5. 同一 ⟨*parser name*⟩, ⟨*state*⟩ 可以对应不同的 ⟨*symbol meaning*⟩, 例如

```
\pgfparserdef{myparser}{initial}A{foo}
\pgfparserdef{myparser}{initial}B{foo}
```

   是有效的。

6. 参数 ⟨*action*⟩ 是某些代码，是与“⟨*parser name*⟩⟨*state*⟩⟨*symbol meaning*⟩”对应的动作。在 ⟨*action*⟩ 中几乎可以使用任何代码，这些代码也不会被限制在一个域 (scope) 中，因此在解析过程结束后代码的效果仍然存在。每当 ⟨*action*⟩ 被执行完毕后，控制权就还给解析器。在 ⟨*action*⟩ 中最好不要使用解析器 (使用命令 \pgfparserparse 引入解析器)，除非将该解析器限制在一个域 (scope) 中。在 ⟨*action*⟩ 中可以使用 \pgfparserdef.

7. 参数 ⟨*arguments*⟩ 是参数格式，类似 xparse 宏包所规定的 argument specification, 其中列举的是参数类型标示符号（不用逗号分隔）。在 ⟨*arguments*⟩ 中至多使用 9 个参数类型标示符号，可用的参数类型标示符号如下：

   m 代表强制参数，对应必须给出的参数。当这种参数由多个符号组成时，要用花括号把该参数括起来。例如：

   $f(x)$

```
\pgfparserdef{myparser}{initial}a[m]{$#1$}
\pgfparserdef{myparser}{initial}{the letter c}{\pgfparserswitch{final}}
\pgfparserparse{myparser}a{f(x)}c
```

   上面例子中，变量符号 #1 对应 [m] 中的 m.

r⟨*delim*⟩ 代表强制参数，⟨*delim*⟩ 作为定界符，当遇到 ⟨*delim*⟩ 时就认为参数列举完毕。例如：

f(x)y(x)

```
\pgfparserdef{myparser}{initial}a[mr;]{$#1$#2}
\pgfparserdef{myparser}{initial}{the letter c}{\pgfparserswitch{final}}
\pgfparserparse{myparser}a f(x)y(x);c
```

上面例子中，变量符号 #1 对应 [mr;] 中的 m; 变量符号 #2 对应 [mr;] 中的 r, 而符号 ; 是参数 #2 的结束标志。解析字符串时，字母“f”对应 #1, 被放入数学模式中；而“(x)y(x)”对应 #2, 符号“;”用于界定第二个参数 (被吃掉)；字母“c”引起 final 状态 (也被吃掉)。

o 代表可选参数，它对应以“[⟨*something*⟩]”这种形式给出的可选参数。在默认下，这里的 ⟨*something*⟩ 等于某个“特殊 mark”, 也就是说，如果不给出参数“[⟨*something*⟩]”, 那么就把“特殊 mark”作为参数。例如：

必选可选

默认的 mark 是：-PGFparserXmark-

```
\pgfparserdef{myparser}{initial}a[mo]{#1#2}
\pgfparserdef{myparser}{initial}c{\pgfparserswitch{final}}
\pgfparserparse{myparser}a{必选}[\texttt{可选}]c\par
\pgfparserparse{myparser}a{默认的 mark 是：}c
```

上面例子中，变量符号 #1 对应 [mo] 中的 m; 变量符号 #2 对应 [mo] 中的 o.

O{⟨*default*⟩} 代表可选参数，它对应以“[⟨*something*⟩]”这种形式给出的可选项，其中 ⟨*something*⟩ 的默认值是 ⟨*default*⟩. 例如：

e^t
e^2

```
\pgfparserdef{myparser}{initial}a[mO{2}]{$#1^{#2}$}
\pgfparserdef{myparser}{initial}{the letter c}{\pgfparserswitch{final}}
\pgfparserparse{myparser}a{\mathrm e}[t]c\par
\pgfparserparse{myparser}a{\mathrm e}c
```

上面例子中，变量符号 #1 对应 [mO{2}] 中的 m; 变量符号 #2 对应 [mO{2}] 中的 O.

d⟨*delim1*⟩⟨*delim2*⟩ 代表可选参数，⟨*delim1*⟩ 与 ⟨*delim2*⟩ 用作定界符，它对应以“⟨*delim1*⟩⟨*options*⟩⟨*delim2*⟩”这种形式给出的可选项，其中 ⟨*options*⟩ 的默认值是某个特殊 mark. 例如：

必选可选

默认的 mark 是：-PGFparserXmark-

```
\pgfparserdef{myparser}{initial}a[md()]{#1#2}
\pgfparserdef{myparser}{initial}{the letter c}{\pgfparserswitch{final}}
\pgfparserparse{myparser}a{必选}(\texttt{可选})c\par
\pgfparserparse{myparser}a{默认的 mark 是：}c
```

上面例子中，变量符号 #1 对应 [md()] 中的 m; 变量符号 #2 对应 [md()] 中的 d.

D⟨*delim1*⟩⟨*delim2*⟩{⟨*default*⟩} 代表可选参数，⟨*delim1*⟩ 与 ⟨*delim2*⟩ 用作定界符，它对应以“⟨*delim1*⟩⟨*code*⟩⟨*delim2*⟩”这种形式给出的可选项，其中 ⟨*code*⟩ 的默认值是 ⟨*default*⟩. 例如：

可选可选
默认：哈

```
\pgfparserdef{myparser}{initial}a[O{默认：}D(){哈}]{#1#2}
\pgfparserdef{myparser}{initial}{the letter c}{\pgfparserswitch{final
↪ }}
\pgfparserparse{myparser}a[可选](\texttt{可选})c\par
\pgfparserparse{myparser}ac
```

上面例子中，变量符号 #1 对应 O; 变量符号 #2 对应 D.

t⟨*token*⟩ 用于测试下一个 letter 是不是 ⟨*token*⟩, 如果是，则把它吃掉，并且参数被设置为某个特殊 mark. 例如：

默认的 mark 是：$-PGFparserXmark-$

```
\pgfparserdef{myparser}{initial}a[O{}t*]{#1$#2$}
\pgfparserdef{myparser}{initial}{the letter c}{\pgfparserswitch{final}}
\pgfparserparse{myparser}a[默认的 mark 是：]*c
```

8. 关于 [⟨*arguments*⟩]:
   - 如果写出

     ```
     \pgfparserdef{myparser}{initial}a    [⟨something⟩]{foo}
     ```

     那么字母 a 之后、方括号之前的空格会被忽略。
   - 如果在 [⟨*arguments*⟩] 中套嵌使用方括号，例如 [a[bc]d], 那么类似 LaTeX 的做法，需要用花括号标识出套嵌的层次、范围：[{a[bc]d}].
   - ⟨*arguments*⟩ 对“⟨*parser-state-char*⟩”组合对应的 type 命令，code 命令的影响是：
     - 如果没有 ⟨*arguments*⟩, 命令 \pgfparserdef 会定义 type 命令和 code 命令：

       ```
       \expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ blank space \space
       ↪  type\endcsname{noarg}
       以及
       \expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ blank space \space
       ↪  code\endcsname{⟨action⟩\pgfparser@getnexttoken}
       ```

     - 如果有 ⟨*arguments*⟩, 假设 ⟨*arguments*⟩ 中使用 3 个 (不能多于 9 个) 参数类型标示符号，那么命令 \pgfparserdef 会定义 type 命令和 code 命令：

       ```
       \expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ blank space \space
       ↪  type\endcsname{⟨arguments⟩}
       以及
       \expandafter\def\csname pgfparser ⟨parser name⟩ ⟨state⟩ blank space \space
       ↪  code\endcsname#1#2#3{⟨action⟩\pgfparser@getnexttoken}
       ```

\pgfparser@initiate@parser{⟨*parser name*⟩}

本命令检查 \csname ifpgfparserconditional ⟨*parser name*⟩ silent\endcsname 这个 TeX-if 是否存在，如果存在，就什么也不做；如果不存在：

1. 定义关于 ⟨*parser name*⟩-all-⟨空格⟩ 组合的 type 命令和 code 命令：
   - 定义 ⟨*parser name*⟩-all-⟨空格⟩-type 命令：

     ```
     \expandafter\def\csname pgfparser ⟨parser name⟩ all blank space \space\space
     ↪  type\endcsname{noarg}
     ```

   - 定义 ⟨*parser name*⟩-all-⟨空格⟩-code 命令：

     ```
     \expandafter\def\csname pgfparser ⟨parser name⟩ all blank space \space\space
     ↪  code\endcsname{\pgfparser@getnexttoken}
     ```

2. 把 \csname ifpgfparserconditional ⟨*parser name*⟩ silent\endcsname 声明为一个 TeX-if.
3. 定义键：

   ```
   \pgfparserset{%
     ⟨parser name⟩/silent/.is if=\pgfparser@ifname{#1 silent}%
   }% 这个 TeX-if 就是
   %\csname ifpgfparserconditional ⟨parser name⟩ silent\endcsname
   ```

从 \pgfparserparse 的处理过程看，\pgfparser@initiate@parser 的作用可以认为是“声明 ⟨*parser name*⟩”这个解析器 (声明那个 TeX-if)。

\pgfparser@if{⟨*if name body*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

本命令检查 \csname ifpgfparserconditional ⟨*if name body*⟩\endcsname 这个 TeX-if 是否已被声明，

- 如果未声明，报错；
- 如果已声明，再检查它的真值，如果它的真值是 true, 就执行 ⟨*true code*⟩; 如果它的真值是 false,

就执行 ⟨*false code*⟩.

\pgfparser@msg{⟨*message*⟩}

本命令立即向 0 号文件写入信息。

```
\pgfutil@protected\def\pgfparser@msg#1%
  {%
    \immediate\write0{(pgfparser) #1}%
  }
```

\pgfparserparse{⟨*parser name*⟩}⟨*text*⟩

这里的 ⟨*parser name*⟩ 是已经定义的解析器名称，⟨*text*⟩ 是一串字母，本命令使用 ⟨*parser name*⟩ 来解析 ⟨*text*⟩, 注意 ⟨*text*⟩ 并不处于花括号中。

⟨*text*⟩ 中可以含有字母、标点符号、括号 (不必配对)。⟨*text*⟩ 中的空格会被忽略。如果 ⟨*text*⟩ 中含有宏 \pgfutil@sptoken, 那么这个宏会被解析。如果希望正常解析 \pgfutil@sptoken, 那么需要提前定义：

```
\pgfparserdef{⟨parser name⟩}{⟨state⟩}{blank space}{⟨action⟩}
```

本命令的处理是：

首先检查 \csname ifpgfparserconditional ⟨*parser name*⟩ silent\endcsname 这个 TEX-if 是否存在，如果不存在就报错，如果存在就执行下面的步骤：

1. 保存解析器的内部名称 \pgfparser@current

   ```
   \edef\pgfparser@current{pgfparser ⟨parser name⟩}%
   ```

2. 保存用户写出的解析器名称 \pgfparser@usersname

   ```
   \edef\pgfparser@usersname{⟨parser name⟩}%
   ```

3. 执行

   ```
   \pgfparser@if{pgfparser status}{\pgfparser@msg{parser '⟨parser name⟩' started}}{
   ↪  }%
   ```

4. 执行 \pgfparserswitch{initial}, 其作用是：

   (a) 执行

   ```
   \pgfparser@if{pgfparser status}
         {\pgfparser@msg{|= switched to state 'initial'}}{}%
   ```

   (b) 定义当前状态名称 \def\pgfparserstate{initial}, 这就使得初始状态为 initial.

5. 执行 \pgfparser@getnexttoken

\pgfparser@getnexttoken

本命令会被 ⟨*parser name*⟩-⟨*state*⟩-⟨字符⟩-code 命令调用。

这个命令的处理是：

首先检查当前状态名称 \pgfparserstate 是否 final; 如果是，就执行

\csname pgfparser ⟨*parser name*⟩ final action\endcsname

如果不是，就执行以下步骤：

1. 利用

   ```
   \futurelet\pgfparsertoken\pgfparser@getnexttoken@a⟨ 记号序列⟨text⟩⟩
   ```

   读取 ⟨*text*⟩ 中的第 1 个记号并保存到 \pgfparsertoken, 因为使用 \futurelet 读取，所以不吃掉这个记号。然后

2. 检查 \pgfparsertoken 是否 4 个特殊记号：\bgroup, \egroup, \pgfutil@sptoken(空格)，# 之一，
   - 如果是就定义 \pgfparserletter

     ```
     \def\pgfparserletter{\bgroup} 或者
     \def\pgfparserletter{\egroup} 或者
     \def\pgfparserletter{ } 或者
     \def\pgfparserletter{#}
     ```

     并吃掉 ⟨*text*⟩ 中的这个记号 (就是 \pgfparsertoken 对应的那个记号)；
   - 如果不是，就吃掉 ⟨*text*⟩ 中的这个记号 (就是 \pgfparsertoken 对应的那个记号)，并定义为 \pgfparserletter 的替换文本。

     ```
     \def\pgfparserletter{⟨first token⟩}
     ```

   使用这种方式定义 \pgfparserletter 的原因是，例如 \def\pgfparserletter{{} 会导致错误。
3. 检查当前需要的 ⟨*parser name*⟩-⟨*state*⟩-\pgfparsertoken-type 命令，即控制序列

   ```
   \csname pgfparser ⟨parser name⟩ ⟨state⟩ \meaning\pgfparsertoken type\endcsname
   ```

   是否已定义，
   - 如果已定义，
     (a) 执行

     ```
     \pgfparser@if{pgfparser status}
       {\pgfparser@msg{| \space|= rule '\meaning\pgfparsertoken' executed}}
       ↪  {}%
     ```

     (b) 执行当前需要的 ⟨*parser name*⟩-⟨*state*⟩-\pgfparsertoken-code 命令。
   - 如果未定义，再检查 all 状态下的 ⟨*parser name*⟩-all-\pgfparsertoken-type 命令，即控制序列

     ```
     \csname pgfparser ⟨parser name⟩ all \meaning\pgfparsertoken type
     ↪  \endcsname
     ```

     是否已定义，
     – 如果已定义，
       (a) 执行

       ```
       \pgfparser@if{pgfparser status}
         {\pgfparser@msg{| \space|= (all) rule '\meaning\pgfparsertoken'
         ↪  executed}}{}%
       ```

       (b) 执行当前需要的 ⟨*parser name*⟩-all-\pgfparsertoken-code 命令。
     – 如果未定义，
       (a) 检查 \csname ifpgfparserconditional every parser silent\endcsname 的真值，
       ◇⟨*iftrue*⟩ 如果是 true, 不报错，继续后面的步骤。
       ◇⟨*iffalse*⟩ 如果是 false, 就再检查
       \csname ifpgfparserconditional ⟨*parser name*⟩ silent\endcsname 的真值，
       ◇⟨*iffalse*⟩.⟨*iftrue*⟩ 如果是 true, 不报错，继续后面的步骤；
       ◇⟨*iffalse*⟩.⟨*iffalse*⟩ 如果是 false, 报错。
       (b) 检查控制序列 \csname pgfparser ⟨*parser name*⟩ ⟨*state*⟩ unknown\endcsname 是

否有定义，
◇⟨*iftrue*⟩ 如果已定义，
◇⟨*iftrue*⟩ – 1 执行

```
\pgfparser@if{pgfparser status}
  {\pgfparser@msg{| \space|= unknown for '\meaning\pgfparsertoken'
  ↪  executed}}{}%
```

◇⟨*iftrue*⟩ – 2 执行控制序列
`\csname pgfparser` ⟨*parser name*⟩ ⟨*state*⟩ `unknown\endcsname`
◇⟨*iffalse*⟩ 如果未定义，就再检查控制序列
`\csname pgfparser` ⟨*parser name*⟩ `all unknown\endcsname` 是否有定义，
◇⟨*iffalse*⟩.⟨*iftrue*⟩ 如果已定义，
◇⟨*iffalse*⟩.⟨*iftrue*⟩ – 1 执行

```
\pgfparser@if{pgfparser status}
  {\pgfparser@msg{| \space|= (all) unknown for '\meaning
  ↪  \pgfparsertoken' executed}}{}%
```

◇⟨*iffalse*⟩.⟨*iftrue*⟩ – 2 执行控制序列
`\csname pgfparser` ⟨*parser name*⟩ `all unknown\endcsname`
◇⟨*iffalse*⟩.⟨*iffalse*⟩ 如果未定义，则什么也不做。

(c) 执行 `\pgfparser@getnexttoken`

`\pgfparserlet{`⟨*parser name 1*⟩`}{`⟨*state 1*⟩`}`⟨*symbol meaning 1*⟩`[`⟨*opt 1*⟩`][`⟨*opt 2*⟩`]`⟨*symbol meaning 2*⟩

本命令中的 ⟨*parser name 1*⟩ 是某个解析器的名称；⟨*state 1*⟩ 是属于解析器 ⟨*parser name 1*⟩ 的某个状态；⟨*symbol meaning 1*⟩ 是状态 ⟨*state 1*⟩ 所对应的 letter.
⟨*symbol meaning 2*⟩ 是某个解析器的某个状态所对应的 letter; `[`⟨*opt 1*⟩`][`⟨*opt 2*⟩`]` 是可选项。
本命令可以涉及两个解析器：⟨*parser name 1*⟩ 和 ⟨*parser name 2*⟩; 两个状态：⟨*state 1*⟩ 和 ⟨*state 1*⟩; 两个 letter: ⟨*symbol meaning 1*⟩ 和 ⟨*symbol meaning 2*⟩.
本命令假设 ⟨*symbol meaning 2*⟩ 所引起的操作是已定义的，并利用 ⟨*symbol meaning 2*⟩ 来规定 ⟨*symbol meaning 1*⟩ 所引起的操作。
在使用本命令时，有以下三种情况：

- 第一种，在一个解析器的一个状态之内，借用字符对应的命令。假设解析器 ⟨*parser name 1*⟩ 的状态 ⟨*state 1*⟩ 针对 ⟨*symbol meaning 2*⟩ 的操作 (记此操作为 ⟨*deal*⟩) 是已定义的，那么

  ```
  \pgfparserlet{⟨parser name 1⟩}{⟨state 1⟩}{⟨symbol meaning 1⟩}{⟨symbol meaning 2⟩}
  ```

  就使得状态 ⟨*state 1*⟩ 针对 ⟨*symbol meaning 1*⟩ 的操作等同于 ⟨*deal*⟩.

- 第二种，在一个解析器的 2 个状态之间，借用字符对应的命令。假设解析器 ⟨*parser name 1*⟩ 的状态 ⟨*state 2*⟩ 针对 ⟨*symbol meaning 2*⟩ 的操作 (记此操作为 ⟨*deal*⟩) 是已定义的，那么

  ```
  \pgfparserlet{⟨parser name 1⟩}{⟨state 1⟩}{⟨symbol meaning 1⟩}
                                  [⟨state 2⟩]{⟨symbol meaning 2⟩}
  ```

  就使得状态 ⟨*state 1*⟩ 针对 ⟨*symbol meaning 1*⟩ 的操作等同于 ⟨*deal*⟩.

- 第三种，在 2 个解析器的 2 个状态之间，借用字符对应的命令。假设解析器 ⟨*parser name 2*⟩ 的状态 ⟨*state 2*⟩ 针对 ⟨*symbol meaning 2*⟩ 的操作 (记此操作为 ⟨*deal*⟩) 是已定义的，那么

  ```
  \pgfparserlet{⟨parser name 1⟩}{⟨state 1⟩}{⟨symbol meaning 1⟩}
                [⟨parser name 2⟩][⟨state 2⟩]{⟨symbol meaning 2⟩}
  ```

就使得解析器 ⟨*parser name 1*⟩ 的状态 ⟨*state 1*⟩ 针对 ⟨*symbol meaning 1*⟩ 的操作等同于 ⟨*deal*⟩.

\pgfparserdefunknown{⟨*parser name*⟩}{⟨*state*⟩}{⟨*action*⟩}

本命令的作用是，当解析器 ⟨*parser name*⟩ 的状态 ⟨*state*⟩ 遇到未定义的 (或者说无法时别的) letter 时，就执行 ⟨*action*⟩, 不过这需要提前设置选项 /pgfparser/silent 或者 /pgfparser/⟨*parser name*⟩/silent 的值为 true 才行。

```
\newcount\mycount
\pgfparserdef{myparse}{all}a{}
\pgfparserdef{myparse}{all}){\pgfparserswitch{final}}
\pgfparserdefunknown{myparse}{all}{\advance\mycount by 1\relax}
\pgfparserset{myparse/silent=true}%
\pgfparserparse{myparse}abcd)
\the\mycount
```

3

本命令的定义是：

```
\pgfutil@protected\long\def\pgfparserdefunknown#1#2#3%
  {%
    \pgfparser@initiate@parser→P. 510{#1}%
    \pgfutil@namedef{\pgfparser@name{#1} #2 unknown}{#3}%
  }
```

本命令：

- 声明解析器 ⟨*parser name*⟩, 定义关于 ⟨*parser name*⟩-all-⟨空格⟩ 组合的 type 命令和 code 命令
- 定义控制序列 \csname pgfparser ⟨*parser name*⟩ ⟨*state*⟩ unknown\endcsname.

\pgfparserdeffinal{⟨*parser name*⟩}{⟨*action*⟩}

本命令定义解析器 ⟨*parser name*⟩ 的终结状态 final. 当解析器转到终结状态 final 后，执行 ⟨*action*⟩. 本命令定义控制序列 \csname pgfparser ⟨*parser name*⟩ final action\endcsname 的替换文本是 ⟨*action*⟩.

\pgfparserswitch{⟨*state*⟩}

本命令：

- 检查 \csname ifpgfparserconditional pgfparser status\endcsname 的真值，如果是 true, 就向 0 号文件写下信息：(pgfparser) switched to state '⟨*state*⟩'
- 定义状态名 \def\pgfparserstate{⟨*state*⟩}

本命令用在 ⟨*action*⟩ 中，使得当前状态结束后，下一个状态切换至 ⟨*state*⟩.

There are 3 letters.

```
\newcount\mycount
\pgfparserdef{myparser}{initial}{the letter a}{\advance\mycount by 1\relax}
\pgfparserdef{myparser}{initial}.{\advance\mycount by 1\relax}
\pgfparserdef{myparser}{initial}){\advance\mycount by 1\relax \pgfparserswitch{final}}
\pgfparserparse{myparser}a.)

There are \the\mycount\ letters.
```

如果把上面例子中字符串中的) 去掉就不能使得解析器达到 final 状态，从而引发错误。

\pgfparserifmark{⟨*arg*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

前面介绍各个选项类型标示符号时提到了“特殊 mark”, 这个 mark 保存在宏 \pgfparser@mark 中，显示为“-PGFparserXmark-”, 例如：

-PGFparserXmark-

```
\makeatletter
\pgfparser@mark
\makeatother
```

本命令的作用是：用 \ifx 检查 ⟨*arg*⟩ 与 \pgfparser@mark 的定义是否相同，如果相同就执行 ⟨*true code*⟩, 如果不同就执行 ⟨*false code*⟩.

\pgfparserreinsert{⟨*a token*⟩}

本命令会检查它的参数 ⟨*a token*⟩ 是否等于 \pgfparser@getnexttoken, 如果不等于就报错。因此本命令应当用作 \pgfparserdef 或 \pgfparserdefunknown 规定的 ⟨*action*⟩ 中的最后一个记号，因为 ⟨*parser name*⟩-⟨*state*⟩-⟨字符⟩-code 命令的替换文本是“⟨*action*⟩\pgfparser@getnexttoken”。
本命令的作用是：在当前状态 (针对其对应的 letter) 执行完毕 ⟨*action*⟩ 后，再 (针对其对应的 letter) 执行一次 ⟨*action*⟩.

\pgfparserstate

这个宏由 \pgfparserswitch 定义，它展开为一个状态的名称。在命令 \pgfparserparse 的开始部分，会执行 \pgfparserswitch{initial}.
这个宏可以用在 ⟨*action*⟩ 中。

\pgfparsertoken

这个宏被 let 为当前读取的 ⟨*text*⟩ 中的第一个记号 t_0, 定义这个宏后，再定义 \pgfparserletter, 并吃掉记号 t_0; 然后再检查解析器能否处理读取的 t_0; 如果能处理，就执行相应的 ⟨*action*⟩; 如果不能处理，就采取其他操作。所以这个宏可以用在 ⟨*action*⟩ 中。

```
\futurelet\pgfparsertoken\pgfparser@getnexttoken@a
```

\pgfparserletter

这个宏被 \def 为当前已读取的 ⟨*text*⟩ 中的第一个记号 t_0, 定义这个宏后，已读取的 t_0 就被吃掉了；然后再检查解析器能否处理读取的这个 t_0; 如果能处理，就执行相应的 ⟨*action*⟩; 如果不能处理，就采取其他操作。所以这个宏可以用在 ⟨*action*⟩ 中。
多数情况下这个宏保存的记号与 \pgfparsertoken 一样，但也有区别。例如，如果读取的记号是特殊记号 {，}，#，空格 ␣(类代码为 1, 2, 6, 10), 那么有，例如

```
\futurelet\pgfparsertoken\pgfparser@getnexttoken@a{
```

这相当于 \let\pgfparsertoken{, 但是另有

```
\def\pgfparserletter{\bgroup}
```

因为定义 \def\pgfparserletter{{} 会导致错误。

\pgfparserset{⟨*key list*⟩}

此命令的定义是：

```
\long\def\pgfparserset#1%
  {%
    \pgfset{/pgfparser/.cd,#1}%
  }
```

28.2 Parser 模块的选项

/pgfparser/silent=⟨*boolean*⟩ (no default, initially false)

这个键的定义是：

```
\pgfparserset
  {%
    ,silent/.is if=\pgfparser@ifname{every parser silent}%
  }
```

执行 /pgfparser/silent 或 /pgfparser/silent=true 就会把 \csname ifpgfparserconditional every parser silent\endcsname 的真值设为 true, 此时，如果某个字母没有对应的 ⟨*action*⟩, 就不会发出错误信息。本选项对所有解析器有效。

/pgfparser/status=⟨*boolean*⟩ (no default, initially false)

这个键的定义是：

```
\pgfparserset
  {%
    ,status/.is if=\pgfparser@ifname{pgfparser status}%
  }
```

执行 /pgfparser/status 或 /pgfparser/status=true 就会把 \csname ifpgfparserconditional pgfparser status\endcsname 的真值设为 true, 此时，每当执行 ⟨*action*⟩ 时，就向 0 号文件发布一个状态信息，用于 debug.

当使用 \pgfparserdef, \pgfparserdefunknown, \pgfparserlet 对 ⟨*parser name*⟩ 做出规定后，下面的选项可用：

/pgfparser/⟨*parser name*⟩/silent=⟨*boolean*⟩ (no default, initially false)

这个键由 \pgfparser@initiate@parser →P. 510 定义：

```
\pgfparserset
  {%
    ⟨parser name⟩/silent/.is if=\pgfparser@ifname{⟨parser name⟩ silent}%
  }
```

在本选项值为 true 的情况下，当解析器 ⟨*parser name*⟩ 遇到不能处理的字符时，会自动忽略它，而不是报错。

28.3 其他

在 \pgfparserdef 的定义中有：

```
\def\pgfparserdef@argstring@def#1{%
  \def\pgfparserdef@argstring##1#1##2##{##1#1}}%
\expandafter\pgfparserdef@argstring@def\the\pgfparserdef@arg@count% 计数器值
\pgfutil@namedef{\pgfparser@name{#1} #2 #3 code\expandafter}%
  \pgfparserdef@argstring##1##2##3##4##5##6##7##8##9%
  {#5\pgfparser@getnexttoken}%
```

尝试一下：

macro:#17#2{->#17{

macro:#1#2#3#4#5#6#7->xxx

```
\makeatletter
\pgfparserdef@argstring@def 7
\meaning\pgfparserdef@argstring\par
\pgfutil@namedef{xxxcode\expandafter}\pgfparserdef@argstring#1#2#3#4#5#6#7#8#9{xxx}
\meaning\xxxcode
\makeatother
```

\relax
```
\expandafter\def\csname a \space b\endcsname{x}
\expandafter\meaning\csname a \space\space b\endcsname
```

28.4 例子

13 different letters found

```
\mycount=0
\pgfparserdef{different letters}{all};{\pgfparserswitch{final}}%
\pgfparserdefunknown{different letters}{all}
    {\pgfparserdef{different letters}{all}\pgfparsertoken{}\advance\mycount1}%
\pgfparserdeffinal{different letters}{\the\mycount\ different letters found}%
\pgfparserset{different letters/silent=true}%
\pgfparserparse{different letters}udiaternxqlchudiea;
```

nobody **will** *use* Parser

```
\pgfparserdef{arguments}{initial}{the letter a}[d()]
    {\pgfparserifmark{#1}{\textcolor{red}{\textit{use}}}{\textbf{#1}} }%
\pgfparserdef{arguments}{initial}t[m]{\texttt{#1} }%
\pgfparserdef{arguments}{initial}c[t*0{blue}m]
    {\pgfparserifmark{#1}{#3}{\textcolor{#2}{#3}}}%
\pgfparserdef{arguments}{all};{\pgfparserswitch{final}}%
\pgfparserparse{arguments}t{nobody}a(will)ac[green]{P}c*{arse}c{r};
```

28.5 关于 ⟨*text*⟩ 中的\pgfutil@sptoken

按命令 \pgfparserparse 的处理流程，这个命令会先让解析器处于 initial 状态，然后再开始解析过程。对于任何一个状态 ⟨*state*⟩(包括 initial 状态)，如果状态 ⟨*state*⟩ 不能解析 ⟨*text*⟩ 中的某个记号，就会自动去尝试 all 状态的相应命令来解析这个记号 (这个尝试不改变当前状态名称，即 \pgfparserstate 的值仍然是 ⟨*state*⟩)。

当 ⟨*text*⟩ 中有 \pgfutil@sptoken 时，如果希望状态 ⟨*state*⟩ 能解析 \pgfutil@sptoken, 那么需要提前定义：

```
\pgfparserdef{⟨parser name⟩}{⟨state⟩}{blank space}{⟨blank space action⟩}
```

也就是定义 type 命令和 code 命令：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ {⟨state⟩} blank space \space\space type
↪  \endcsname{...}
```

以及

```
\expandafter\def\csname pgfparser ⟨parser name⟩ {⟨state⟩} blank space \space\space code
↪  \endcsname...{...}
```

注意下面命令的输出中，在“blank space”后面有一个空格：

blank space xxx
```
\makeatletter
\meaning\pgfutil@sptoken{}xxx
\makeatother
```

命令 \pgfparserdef, \pgfparserdefunknown 都会定义对应 ⟨*parser name*⟩-all-⟨空格⟩ 的 type 命令和 code 命令 (见 \pgfparser@initiate@parser →P. 510)：

```
\expandafter\def\csname pgfparser ⟨parser name⟩ all blank space \space\space type
↪  \endcsname{noarg}
```

以及

```
\expandafter\def\csname pgfparser ⟨parser name⟩ all blank space \space\space code
↪  \endcsname{\pgfparser@getnexttoken}
```

所以，如果状态 ⟨*state*⟩ 不能解析 ⟨*text*⟩ 中的 `\pgfutil@sptoken`, 就会导致 `all-`⟨空格⟩ 的 code 命令被执行，即执行 `\pgfparser@getnexttoken`, 这会略过 `\pgfutil@sptoken`, 继续在状态 ⟨*state*⟩ 下解析后面的记号；此时如果提前定义了 unknown 命令：

```
\pgfparserdefunknown{⟨parser name⟩}{⟨state⟩}{⟨unknown action⟩}
```

那么 ⟨*unknown action*⟩ 此时也不会被执行，即使提前设置了选项 `silent`，或者 ⟨*parser name*⟩`/silent` 的值为 true.

state initial does not known ';', switch to 'state 1' and reparse it. Semicolon, end.

state initial does not known 'x', switch to 'state 1' and reparse it. A blank space, Semicolon, end.

state initial does not known '\x', switch to 'state 1' and reparse it. A blank space, Semicolon, end.

```
\makeatletter
\pgfparserdefunknown{test space}{initial}{%
  state \pgfparserstate{} does not known '\expandafter\string\pgfparserletter',
  \pgfparserswitch{state 1}%
  switch to '\pgfparserstate{}' and reparse it. \pgfparserreinsert}%
\pgfparserdef{test space}{state 1}{blank space}{A blank space, }%
\pgfparserdef{test space}{state 1}{\meaning;}{Semicolon, end. \pgfparserswitch{final}}%

\pgfkeys{/pgfparser/test space/silent,}%
\pgfparserparse{test space}\pgfutil@sptoken;\par% \pgfutil@sptoken 被忽略
\pgfparserparse{test space}x\pgfutil@sptoken;\par%
\pgfparserparse{test space}\x\pgfutil@sptoken;
\makeatother
```

第二十九章　面向对象

本节介绍 oo 模块。

```
\usepgfmodule{oo}% LaTeX and plain TeX and pure pgf
\usepgfmodule[oo]% ConTeXt and pure pgf
```

这个模块提供一组很少的命令，可用以定义 classes, methods, attributes, objects, 实现简单的对象化编程。见文件《pgfmoduleoo.code.tex》。

PGF 的 datavisualization 模块利用这个模块编程，见文件《pgfmoduledatavisualization.code.tex》。

29.1　Overview

TEX 本身并不支持对象化编程，因为产生 TEX 的年代还没有流行这个概念。简单地讲，oo-system 支持 classes, methods, constructors, attributes, objects, object identities, inheritance, overloading.

假设 ⟨*a class name*⟩ 是一个 class 名称；⟨*method name*⟩ 是一个 method 名称；⟨*attribute name*⟩ 是一个 attribute 名称；\⟨*obj*⟩ 是某个对象；⟨*method name*⟩, ⟨*attribute name*⟩, \⟨*obj*⟩ 属于 ⟨*a class name*⟩, 那么：

- ⟨*method name*⟩ 对应的命令是

  \csname pgfooY⟨*a class name*⟩.⟨*method name*⟩\endcsname

  这个控制序列由 \method “非全局地”定义，可以使得这个命令具有处理参数的能力；这个方法也可能是类 ⟨*a class name*⟩ 从父类那里继承的方法；
- ⟨*attribute name*⟩ 对应的控制序列都是保存某些代码的，它们不是处理参数的命令。⟨*attribute name*⟩ 对应的控制序列有两种：
  - 一种是由命令 \attribute “非全局地”定义的控制序列：

    \csname pgfooY⟨*a class name*⟩@⟨*attribute name*⟩\endcsname

    这是类名称 ⟨*a class name*⟩ 对应的属性；
  - 另一种是由 \pgfoonew → P. 537 “全局地”定义的控制序列，其形式为

    \csname pgfooX⟨*object count value*⟩@⟨*attribute name*⟩\endcsname

    其中 ⟨*object count value*⟩ 是计数器 \pgfoo@objectcount 的值，这是编号为 ⟨*object count value*⟩ 的对象对应的属性；这个控制序列有可能等于

    \csname pgfooY⟨*a class name*⟩@⟨*attribute name*⟩\endcsname

    也可能等于某个控制序列

    \csname pgfooY⟨*parent class name*⟩@⟨*attribute name*⟩\endcsname

    这是类 ⟨*a class name*⟩ 从父类 ⟨*parent class name*⟩ 那里继承的属性。
- 对象 \⟨*obj*⟩ 属于 ⟨*a class name*⟩, 是由于“全局地”定义的控制序列

  \csname pgfooX⟨*object count value*⟩@class\endcsname

  保存了 ⟨*a class name*⟩, 见 \pgfoonew → P. 537.

29.2 声明一个 class

\pgfooclass(⟨*list of superclasses*⟩){⟨*class name*⟩}{⟨*body*⟩}

本命令声明一个名称为 ⟨*class name*⟩ 的 class. 名称 ⟨*class name*⟩ 中可以含有空格，以及多数符号，但不能有句号 (点号)。

(⟨*list of superclasses*⟩) 是可选的。

⟨*body*⟩ 可以是任何通常的 TEX 代码。在 ⟨*body*⟩ 中应当使用命令 \method 和 \attribute 来定义这个 class 的 methods 和 attributes。

本命令的有效范围受到 TEX 组的限制，最好不要把本命令放在组内。若非必要，也不要用组限制命令 \method 和 \attribute.

将使用 C3 算法来解析 Method Resolution Order (MRO)。

本命令的定义是：

```
% base classes are put in parenthesis before the new class name, but
% are optional.
\def\pgfooclass{%
  \pgfutil@ifnextchar ({%
    \pgfooclass@
  }{%
    \pgfooclass@()
  }%
}%
```

本命令调用 \pgfooclass@.

\pgfooclass@(⟨*list of superclasses*⟩){⟨*class name*⟩}{⟨*body*⟩}

本命令的定义是：

```
\long\def\pgfooclass@(#1)#2#3{%
  \def\pgfoo@classname{#2}%
  \expandafter\ifx\csname pgfooY\pgfoo@classname.@pgfooinit
  ↪  \endcsname\relax\else
    \pgferror{class #2 is already defined}%
  \fi
  \pgfoo@ciii{#2}{#1}%
  \expandafter\let\csname pgfooY#2@pgfoo@mro\endcsname\pgfoo@mro
  \let\pgfoo@origmethod=\method%
  \let\pgfoo@origattribute=\attribute%
  \let\method=\pgfoo@declaremethod%
  \let\attribute=\pgfoo@declareattribute%
  \let\pgfoo@attributes=\pgfutil@empty%
  \let\pgfoo@methods=\pgfutil@empty%
  #3%
  % inherit
  \def\pgfoo@temp@baseclasses{#1}%
  \pgfoo@inherit@methods
  \pgfoo@inherit@attributes
  % Always present (and never inherited) methods:
  \expandafter\let\csname pgfooY\pgfoo@classname.get handle\endcsname\pgfoo@obj
  ↪  %
  \expandafter\let\csname pgfooY\pgfoo@classname.get id\endcsname\pgfoo@id%
  % for compatibility with previous versions of pgfoo, define method
  % with the same name as the class as a synonym for init
```

```
  \expandafter\let\expandafter\pgfoo@init\csname pgfooY\pgfoo@classname.init
  ↪  \endcsname
  \expandafter\let\csname pgfooY\pgfoo@classname.\pgfoo@classname
  ↪  \endcsname\pgfoo@init
  % Cleanup
  \let\method=\pgfoo@origmethod%
  \let\attribute=\pgfoo@origattribute%
}%
```

本命令的处理是：

1. 定义

   ```
   \def\pgfoo@classname{⟨class name⟩}
   ```

   在后续处理中，这个宏的值保持不变。
2. 检查 `\csname pgfooY⟨class name⟩.@pgfooinit\endcsname`, 如果这个控制序列与 `\relax` 相同，则继续后面的步骤，否则报错：class ⟨*class name*⟩ is already defined
   不过，文件《pgfmoduleoo.code.tex》似乎没有关于这个控制序列的定义代码。
3. 执行命令 `\pgfoo@ciii{⟨class name⟩}{⟨list of superclasses⟩}`, 即执行 C3 算法。这个命令得到一个由 class 名称组成的列表，这个列表会被保存到宏 `\pgfoo@mro` 中。例如命令
   `\pgfoo@ciii{⟨current class⟩}{⟨class-name-1⟩,⟨class-name-2⟩,⟨class-name-3⟩}`
   导致

   ```
   \def\pgfoo@mro{
       ⟨current class⟩,⟨class-name-1⟩,⟨class-name-2⟩,⟨class-name-3⟩
   }
   ```
4. 令
   `\csname pgfooY⟨class name⟩@pgfoo@mro\endcsname`
   等于 `\pgfoo@mro`.
5. 执行

   ```
   \let\pgfoo@origmethod=\method%
   \let\pgfoo@origattribute=\attribute%
   \let\method=\pgfoo@declaremethod%
   \let\attribute=\pgfoo@declareattribute%
   \let\pgfoo@attributes=\pgfutil@empty%
   \let\pgfoo@methods=\pgfutil@empty%
   ```
6. 执行 ⟨*body*⟩, 在 ⟨*body*⟩ 中应当使用命令 `\method` 和 `\attribute` 来定义这个 class 的 methods 和 attributes。
7. 定义

   ```
   \def\pgfoo@temp@baseclasses{⟨list of superclasses⟩}%
   ```
8. 执行 `\pgfoo@inherit@methods` →P. 528, 这个命令会使得 ⟨*class name*⟩ 从 ⟨*list of superclasses*⟩ 那里继承 method.
9. 执行 `\pgfoo@inherit@attributes` →P. 531, 这个命令会使得 ⟨*class name*⟩ 从 ⟨*list of superclasses*⟩ 那里继承 attribute.
10. 令
    `\csname pgfooY⟨class name⟩.get handle\endcsname`
    等于 `\pgfoo@obj`, 这是定义一个名称为 `get handle` 的 method.
11. 令
    `\csname pgfooY⟨class name⟩.get id\endcsname`

等于 \pgfoo@id, 这是定义一个名称为 get id 的 method.

12. 令 \pgfoo@init 等于

\csname pgfooY⟨class name⟩.init\endcsname

这个控制序列对应名称为 init 的 method, 它一般由命令 \pgfoo@declaremethod →P. 527 定义。

13. 令

\csname pgfooY⟨class name⟩.⟨class name⟩\endcsname

等于 \pgfoo@init, 这是把 ⟨class name⟩ 也做成一个 method, 使之等于方法 init.

14. 执行

```
\let\method=\pgfoo@origmethod%
\let\attribute=\pgfoo@origattribute%
```

29.3 Compute MRO using C3 algorithm

```
\def\pgfoo@emptyinit(){}%
```

\pgfoo@escapeif#1#2\if

本命令与 \if 配合使用。

```
\def\pgfoo@escapeif#1#2\fi{\fi#1}% a little helper
```

\pgfoo@ciii@done(#1,)

本命令定义宏 \pgfoo@mro

```
\def\pgfoo@ciii@done(#1,){%
  \def\pgfoo@mro{#1}%
}%
```

\pgfoo@ciii@clean⟨\macro⟩{⟨string⟩}

命令 \pgfoo@ciii@clean 的作用是，将 ⟨string⟩ 中的符号串“[].”都去掉，变成 ⟨string'⟩, 然后执行 ⟨\macro⟩⟨string'⟩.

```
% cyclically removes all occurrences of "[]," and "[]." (i.e., empty
% sequences) in #2. When done, calls #1 with whatever remains.
\def\pgfoo@ciii@clean#1#2{% clean commas
  \pgfutil@in@{[].}{#2}% clean dots
  \ifpgfutil@in@\pgfoo@escapeif{%
    \pgfoo@ciii@clean@dot#1#2\pgfoo@ciii@clean@END
  }\else\pgfoo@escapeif{%
    #1#2%
  }\fi
}%
\def\pgfoo@ciii@clean@dot#1#2[].#3\pgfoo@ciii@clean@END{%
  \pgfoo@ciii@clean#1{#2#3}%
}%
```

\pgfoo@ciii{⟨current class⟩}{⟨list of base classes⟩}

本命令最终定义宏 \pgfoo@mro, 例如：

macro:->X,A,B,C

```
\makeatletter
\pgfoo@ciii{X}{A,B,C}
\meaning\pgfoo@mro
```

```
\makeatother
```

本命令的定义是：

```
% #1 = current class, #2 = list of base classes
\def\pgfoo@ciii#1#2{%
  % \pgfoo@seq will hold a list of sequences.
  % (i) the first sequence has just one element: the current class
  \def\pgfoo@seq{[#1,].}%
  % (ii) every subclass has a sequence (in the order as they were
  % given): the class' sequence is its mro.
  \pgfutil@for\pgfoo@baseclass:=#2\do{%
    \edef\pgfoo@baseclass@mro{\csname pgfooY\pgfoo@baseclass @pgfoo@mro\endcsname}
    ↪  %
    \expandafter\ifx\expandafter\relax\pgfoo@baseclass@mro\relax\else
      \edef\pgfoo@seq{\pgfoo@seq[\pgfoo@baseclass@mro,].}%
    \fi
  }%
  % The final sequence is a sequence of base classes, in the order as
  % they were given.
  \ifx\relax#2\relax\else
    \edef\pgfoo@seq{\pgfoo@seq[#2,].}%
  \fi
  \edef\pgfoo@temp{()(\pgfoo@seq)}%
  \expandafter\pgfoo@ciii@merge\pgfoo@temp
}%
```

命令 `\pgfoo@ciii{`⟨*current class*⟩`}{`⟨*class-name-1*⟩`,`⟨*class-name-2*⟩`,`⟨*class-name-3*⟩`}` 的处理是：

1. 定义

   ```
   \def\pgfoo@seq{[⟨current class⟩,].}
   ```

2. 执行一个循环操作，如果命令 `\csname pgfooY`⟨*class-name-i*⟩`@pgfoo@mro\endcsname` 等于 `\relax`（这说明 ⟨*class-name-i*⟩ 这个 class 是尚未被命令 `\pgfooclass@`→P. 520 声明的），$i = 1, 2, 3$，则继续后面的步骤；否则定义 `\pgfoo@seq` 为

   ```
   [⟨current class⟩,].[⟨class-name-1⟩,].[⟨class-name-2⟩,].[⟨class-name-3⟩,].
   ```

3. 如果 ⟨*list of base classes*⟩ 非空，则定义 `\pgfoo@seq` 为

   ```
   [⟨current class⟩,].[⟨class-name-1⟩,].[⟨class-name-2⟩,].[⟨class-name-3⟩,].
   [⟨class-name-1⟩,⟨class-name-2⟩,⟨class-name-3⟩,].
   ```

4. 执行

   ```
   \pgfoo@ciii@merge()
   ([⟨current class⟩,].[⟨class-name-1⟩,].[⟨class-name-2⟩,].[⟨class-name-3⟩,].
   [⟨class-name-1⟩,⟨class-name-2⟩,⟨class-name-3⟩,].)
   ```

`\pgfoo@ciii@merge(#1)(#2)`

本命令的定义是：

```
% #1 = MRO so far
% #2 = remaining sequences
% no empty sequences [] should arrive here: they should be removed
% before calling this macro
\def\pgfoo@ciii@merge(#1)(#2){%
  \ifx\relax#2\relax
    \pgfoo@ciii@done(#1)%
  \else\pgfoo@escapeif{% split #2
```

```
    \pgfoo@ciii@merge@A(#1)(#2)%
  }\fi%
}%
```

\pgfoo@ciii@merge@A(#1)(#2.#3)

本命令的定义是：

```
% a splitter: to be only called from \pgfoo@ciii@merge
\def\pgfoo@ciii@merge@A(#1)(#2.#3){%
  \ifx\relax#3\relax % if #3 is empty, we can take the shortcut
    \pgfoo@ciii@mergeA@done(#1)(#2)%
  \else\pgfoo@escapeif{%
    \pgfoo@ciii@merge@checkHead(#1)()(#2)()(#3)%
  }\fi
}%

\def\pgfoo@ciii@mergeA@done(#1)([#2]){%
  \pgfoo@ciii@done(#1#2)%
}%
```

\pgfoo@ciii@merge@checkHead(#1)(#2)([#3,#4])(#5)(#6)

本命令的定义是：

```
% #1 = MRO so far
% #2 = sequences with bad heads (dot after each sequence)
% [#3,#4] = the sequence whose head we're testing right
%           now---obviously, it should not be empty! (#3 has no comma,
%           #4 has a comma after each class name)
% #5 = #3 does not occur in the tail of these sequences (dot after
%      each sequence)
% #6 = the remaining sequences (dot after each sequence)
\def\pgfoo@ciii@merge@checkHead(#1)(#2)([#3,#4])(#5)(#6){%
  \pgfutil@in@{[#3,}{#2}%
  \ifpgfutil@in@\pgfoo@escapeif{% #3 is already blacklisted
    \pgfoo@ciii@clean\pgfoo@ciii@merge@checkHead@P{(#1)(#2[#3,#4].)(#5#6)}%
  }\else\pgfoo@escapeif{%
    \ifx\relax#6\relax\pgfoo@escapeif{
    ↪  % no more sequences to test the head against!
                                % so, #3 is a good head, and we can recurse
      \pgfoo@ciii@goodHeadFound(#1)(#3)(#2[#4].#5)%
    }\else\pgfoo@escapeif{% split #6 and get to work
      \pgfoo@ciii@merge@checkHead@A(#1)(#2)([#3,#4])(#5)(#6)%
    }\fi
  }\fi
}%
% splitter: to be only called from \pgfoo@ciii@merge@checkHead
% we will check if #3 occurs in the tail of #6
\def\pgfoo@ciii@merge@checkHead@A(#1)(#2)([#3,#4])(#5)(#6.#7){%
  \pgfoo@ciii@merge@checkHead@B(#1)(#2)([#3,#4])(#5)(#6)(#7)%
}%
% splitter: to be only called from \pgfoo@ciii@merge@checkHead@A
% &worker: we will check if #3 occurs in #7. If it does, #3 is a bad head.
\def\pgfoo@ciii@merge@checkHead@B(#1)(#2)([#3,#4])(#5)([#6,#7])(#8){%
%  \def\pgfoo@ciii@tempA{#3}\def\pgfoo@ciii@tempB{#5}%
%  \ifx\pgfoo@ciii@tempA\pgfoo@ciii@tempB
%    \pgferror{An attempt to derive from the same class twice}%
%  \else\pgfoo@escapeif{%
```

```
    \pgfutil@in@{,#3,}{,#7}%
    \ifpgfutil@in@\pgfoo@escapeif{% bad head! get next head and restart
                                  % but: need to clean up first!
      \pgfoo@ciii@clean\pgfoo@ciii@merge@checkHead@P
      ↪  {(#1)(#2[#3,#4].)(#5[#6,#7].#8)}%
    }\else\pgfoo@escapeif{% so far, so good
      \ifx\relax#8\relax\pgfoo@escapeif{% if #8 is empty, we won:
        % #3 is a good head! we will move it to MRO
        % but clean-up first!
        \pgfoo@ciii@goodHeadFound(#1)(#3)(#2[#4].#5[#6,#7].)%
      }\else\pgfoo@escapeif{% proceed to the first seq in #8
        \pgfoo@ciii@clean\pgfoo@ciii@merge@checkHead
        ↪  {(#1)(#2)([#3,#4])(#5[#6,#7].)(#8)}%
      }\fi
    }\fi
%  }\fi
}%
```

\pgfoo@ciii@goodHeadFound{⟨*string 1*⟩}{⟨*string 2*⟩}{⟨*string 3*⟩}

本命令的操作是：

- 如果 ⟨*string 3*⟩ 包含符号串 “,⟨*string 2*⟩,”，那么就从 ⟨*string 3*⟩ 中去掉 “⟨*string 2*⟩,”；
- 如果 ⟨*string 3*⟩ 包含符号串 “[⟨*string 2*⟩,”，那么就从 ⟨*string 3*⟩ 中去掉 “⟨*string 2*⟩,”；

最后把 ⟨*string 3*⟩ 变成 ⟨*string 3'*⟩，再执行

```
\pgfoo@ciii@clean\pgfoo@ciii@merge{(⟨string 1⟩⟨string 2⟩,)(⟨string 3'⟩)}
```

本命令的定义是：

```
% #1 = MRO
% #2 = good head
% #3 = the remaining sequences (in need of removing the good head from
% them, and then also some general cleanup)
\def\pgfoo@ciii@goodHeadFound(#1)(#2)(#3){%
  \pgfutil@in@{,#2,}{#3}%
  \ifpgfutil@in@\pgfoo@escapeif{%
    \def\pgfoo@ciii@removeGoodHead(##1)(##2)(##3,#2,##4){%
      \pgfoo@ciii@goodHeadFound(##1)(##2)(##3,##4)}%
    \pgfoo@ciii@removeGoodHead(#1)(#2)(#3)%
  }\else\pgfoo@escapeif{%
    \pgfutil@in@{[#2,}{#3}%
    \ifpgfutil@in@\pgfoo@escapeif{%
      \def\pgfoo@ciii@removeGoodHead(##1)(##2)(##3[#2,##4){%
        \pgfoo@ciii@goodHeadFound(##1)(##2)(##3[##4)}%
      \pgfoo@ciii@removeGoodHead(#1)(#2)(#3)%
    }\else\pgfoo@escapeif{%
      \pgfoo@ciii@clean\pgfoo@ciii@merge{(#1#2,)(#3)}%
    }\fi
  }\fi
}%
```

\pgfoo@ciii@merge@checkHead@P(#1)(#2)(#3)

本命令的定义是：

```
% mediates between the bad head exit of \pgfoo@ciii@merge@checkHead@B
% and \pgfoo@ciii@merge@checkHead; we need the mediator because #3
% below might have been affected by cleaning.
\def\pgfoo@ciii@merge@checkHead@P(#1)(#2)(#3){%
```

```
  \if\relax#3\relax % no more good head candidates!
    \pgferror{Bad MRO: There is no good (monotonic etc.)
      Method Resolution Order for your class hierarchy}%
  \else\pgfoo@escapeif{%
    \pgfoo@ciii@merge@checkHead@Q(#1)(#2)(#3)%
  }\fi
}%
% splitter for \pgfoo@ciii@merge@checkHead@P
% #3 = the sequence containing the newest good head candidate
\def\pgfoo@ciii@merge@checkHead@Q(#1)(#2)(#3.#4){%
  \pgfoo@ciii@merge@checkHead(#1)(#2)(#3)()(#4)%
}%
```

综合以上，命令

\pgfoo@ciii{⟨*current class*⟩}{⟨*class-name-1*⟩,⟨*class-name-2*⟩,⟨*class-name-3*⟩}

导致

```
\pgfoo@ciii@merge()
([⟨current class⟩,].[⟨class-name-1⟩,].[⟨class-name-2⟩,].[⟨class-name-3⟩,].
[⟨class-name-1⟩,⟨class-name-2⟩,⟨class-name-3⟩,].)
```

再得到

```
\pgfoo@ciii@merge
(⟨current class⟩,)
([⟨class-name-1⟩,].[⟨class-name-2⟩,].[⟨class-name-3⟩,].
 [⟨class-name-1⟩,⟨class-name-2⟩,⟨class-name-3⟩,].)
```

再得到

```
\pgfoo@ciii@merge
(⟨current class⟩,⟨class-name-1⟩,⟨class-name-2⟩,⟨class-name-3⟩,)()
```

然后得到

```
\def\pgfoo@mro{
    ⟨current class⟩,⟨class-name-1⟩,⟨class-name-2⟩,⟨class-name-3⟩
}
```

29.4 声明、继承 method

\method ⟨*method name*⟩(⟨*parameter list*⟩){⟨*method body*⟩}

命令 \method 声明一个 method，在 \pgfooclass@ → P. 520 的处理中有

```
\let\method=\pgfoo@declaremethod%
```

这个命令的有效范围受到 TEX 组的限制。

```
\pgfooclass{MyPlot}{

  \attribute x=0;
  \attribute y=0;

  \method MyPlot() {
  }

  \method getX(#1) {
```

```
    \pgfooget{x}{#1}
  }

  \method setPoint(#1,#2) {
    \pgfooset{x}{#1}
    \pgfooset{y}{#2}
  }
}
```

\pgfoo@declaremethod ⟨*method name*⟩(⟨*parameter list*⟩){⟨*method body*⟩}

本命令的定义是：

```
%   define the method macro and append the method to the method collection
\long\def\pgfoo@declaremethod#1(#2)#3{%
  \def\pgfoo@method{#1}%
  \ifx\pgfoo@classname\pgfoo@method
    \def\pgfoo@method{init}%
  \fi
  \expandafter\long\expandafter\def\csname pgfooY\pgfoo@classname.\pgfoo@method
  ↪  \endcsname(#2){#3}%
  \edef\pgfoo@methods{\pgfoo@methods,\pgfoo@method}%
}%
```

在 \pgfooclass@ →P. 520 的处理中有

```
\let\pgfoo@methods=\pgfutil@empty
```

本命令的主要处理是：

- 如果 ⟨*class name*⟩ 与 ⟨*method name*⟩ 相同，则 \long 定义控制序列：

  ```
  \expandafter\long\expandafter\def\csname pgfooY⟨class name⟩.init\endcsname(
  ↪  ⟨parameter list⟩){⟨method body⟩}
  ```

  控制序列

  \csname pgfooY⟨*class name*⟩.init\endcsname

  是能够处理参数的命令，其参数列举格式是“(⟨*parameter list*⟩)”. 命令 \pgfoonew →P. 537 在创建从属于 ⟨*class name*⟩ 的对象时，会用到这个控制序列，因此最好定义它。

- 如果 ⟨*class name*⟩ 与 ⟨*method name*⟩ 不同，则 \long 定义控制序列：

  ```
  \expandafter\long\expandafter\def\csname pgfooY⟨class name⟩.⟨method name⟩
  ↪  \endcsname(⟨parameter list⟩){⟨method body⟩}
  ```

  控制序列

  \csname pgfooY⟨*class name*⟩.⟨*method name*⟩\endcsname

  是能够处理参数的命令，其参数列举格式是“(⟨*parameter list*⟩)”.

- \edef 重定义宏 \pgfoo@methods：

  ```
  \edef\pgfoo@methods{\pgfoo@methods,\pgfoo@method}%
  ```

  在一个组内，每当执行一次命令 \pgfoo@declaremethod, 宏 \pgfoo@methods 就会被重定义一次。由于宏 \pgfoo@methods 的初始值被 let 为 \pgfutil@empty，所以保存在 \pgfoo@methods 中的第一个符号是“逗号”。最后，宏 \pgfoo@methods 保存的是由“逗号”和各个 ⟨*method name*⟩ 组成的列表。

macro:->,ABC,DEF

```
\makeatletter
\let\pgfoo@methods=\pgfutil@empty
\pgfoo@declaremethod ABC(#1#2){$#1^{#2}$}%
\pgfoo@declaremethod DEF(#1){}%
\meaning\pgfoo@methods
\makeatother
```

macro:->init,a,b

\long macro:(#1#2)->$#1^{#2}$

```
\makeatletter
\pgfooclass{A}
{
    \method a(#1#2){$#1^{#2}$}%
    \method b(#1){}
}
\meaning\pgfoo@methods\\
\expandafter\meaning\csname pgfooYA.a\endcsname
\makeatother
```

上一个例子的结果受到命令 \pgfoo@inherit@methods 的影响。

\pgfoo@inherit@methods

本命令用在 \pgfooclass@ → P. 520 的处理中。

```
\expandafter\let\csname pgfooY⟨class name⟩@pgfoo@mro\endcsname\pgfoo@mro%
\def\pgfoo@temp@baseclasses{⟨list of superclasses⟩}%
```

本命令的定义是：

```
\def\pgfoo@inherit@methods{%
  % for non-derived classes:
  \expandafter\ifx\expandafter\relax\pgfoo@temp@baseclasses\relax
    % if init is not defined, define an empty one
    \expandafter\ifx\csname pgfooY\pgfoo@classname.init\endcsname\relax
      \expandafter\let\csname pgfooY\pgfoo@classname.init
      ↪  \endcsname\pgfoo@emptyinit%
      \edef\pgfoo@methods{,init\pgfoo@methods}%
    \fi
  \fi
  \if\relax\pgfoo@methods\relax\else
    % gobble the initial comma
    \edef\pgfoo@methods{\expandafter\pgfutil@gobble\pgfoo@methods}%
  \fi
  % remember the list of collected methods
  \expandafter\let\csname pgfooY\pgfoo@classname @pgfoo@methods
  ↪  \endcsname\pgfoo@methods
  % get MRO
  \expandafter\let\expandafter\pgfoo@mro\csname pgfooY\pgfoo@classname @pgfoo@mro
  ↪  \endcsname
  % loop throough base classes in MRO
  \pgfutil@for\pgfoo@baseclass:=\pgfoo@mro\do{%
    \ifx\pgfoo@baseclass\pgfoo@classname % skip self
    \else\pgfoo@escapeif{%
      % get methods defined in this base class
      \expandafter\let\expandafter\pgfoo@methods\csname pgfooY\pgfoo@baseclass
      ↪  @pgfoo@methods\endcsname
      \pgfutil@for\pgfoo@method:=\pgfoo@methods\do{% for each method
        % check if it is already defined in our class
```

```
        \expandafter\ifx\csname pgfooY\pgfoo@classname.\pgfoo@method
        ↪  \endcsname\relax
          % if not, link the method name from our class to base class
          \edef\pgfoo@temp{\noexpand\let
            \expandafter\noexpand\csname pgfooY\pgfoo@classname.\pgfoo@method
            ↪  \endcsname
            \expandafter\noexpand\csname pgfooY\pgfoo@baseclass.\pgfoo@method
            ↪  \endcsname}%
          \pgfoo@temp
        \fi
      }%
    }\fi
  }%
}%
```

本命令的处理是：

1. 如果 \pgfoo@temp@baseclasses 保存的内容是空的，也就是没有给出 ⟨*list of superclasses*⟩，则
   - 如果 \csname pgfooY⟨*class name*⟩.init\endcsname 等效于 \relax, 则
     (a) 令 \csname pgfooY⟨*class name*⟩.init\endcsname 等于 \pgfoo@emptyinit
     在默认下，命令 \pgfoo@emptyinit 什么也不做。
     (b) \edef 定义

     ```
     \edef\pgfoo@methods{,init\pgfoo@methods}%
     ```

   这一个步骤确保 \csname pgfooY⟨*class name*⟩.init\endcsname 有定义，并且把 init (作为一个 method 名称) 添加到宏 \pgfoo@methods 中。
2. 如果 \pgfoo@methods 保存的内容不是空的，则

   ```
   \edef\pgfoo@methods{\expandafter\pgfutil@gobble\pgfoo@methods}%
   ```

   这个定义会把 \pgfoo@methods 中原来的第一个符号——逗号——去掉。
3. 令

   \csname pgfooY⟨*class name*⟩@pgfoo@methods\endcsname

   等于 \pgfoo@methods, 保存的是由属于 ⟨*class name*⟩ 的那些 method name 组成的列表。
4. 令 \pgfoo@mro 等于

   \csname pgfooY⟨*class name*⟩@pgfoo@mro\endcsname

   这个控制序列保存的是由某些 class name 组成的列表，即

   ⟨*class name*⟩,⟨*super-class-name-1*⟩,⟨*super-class-name-2*⟩,...

   参考 \pgfooclass@ →P. 520 中的 \pgfoo@ciii.
5. 执行一个循环操作：令宏 \pgfoo@baseclass (用作循环变量)“代表”\pgfoo@mro 中的任意一个列表项目 (一个 class name)：
   - 如果 \pgfoo@baseclass 等于 \pgfoo@classname (保存的是 ⟨*class name*⟩), 则什么也不做；
   - 如果 \pgfoo@baseclass 不是 \pgfoo@classname, 而是 ⟨*super-class-name-i*⟩, 则
     (a) 令 \pgfoo@methods 等于

     \csname pgfooY⟨*super-class-name-i*⟩@pgfoo@methods\endcsname

     这个控制序列保存的是由那些属于 ⟨*super-class-name-i*⟩ 的 method name 组成的列表。
     (b) 执行一个循环操作：令宏 \pgfoo@method (用作循环变量)“代表”\pgfoo@methods 中的任意一个列表项目 (一个 method name)：如果

     \csname pgfooY⟨*class name*⟩.\pgfoo@method\endcsname

     等于 \relax，即没有定义，则令

\csname pgfooY⟨*class name*⟩.\pgfoo@method\endcsname

等于

\csname pgfooY⟨*super-class-name-i*⟩.\pgfoo@method\endcsname

这就为 ⟨*class name*⟩ 定义了名称为 \pgfoo@method 的 method.

可见本命令的作用是，例如：

\long macro:(#1#2)->$#1^{#2}$

```
\pgfooclass{A}
{
    \method abc(#1#2){$#1^{#2}$}
}
\pgfooclass(A){B}
{
%    \method abc(#1#2){}
}
\makeatletter
\expandafter\meaning\csname pgfooYB.abc\endcsname
\makeatother
```

上面例子中，尽管没有为 B 定义 method，但它会继承 A 中的 method。

29.5 处理 attribute

29.5.1 声明、继承 attribute 的命令

\attribute ⟨*attribute name*⟩=⟨*initial value*⟩;

命令 \attribute 声明一个 attribute，在 \pgfooclass@ →P. 520 的处理中有

```
\let\attribute=\pgfoo@declareattribute%
\let\pgfoo@attributes=\pgfutil@empty%
```

这个命令的有效范围受到 TEX 组的限制。

\pgfoo@declareattribute ⟨*attribute name*⟩=⟨*initial value*⟩;

本命令的定义是：

```
\def\pgfoo@declareattribute#1;{%
  \pgfutil@in@{ =}{#1}%
  \ifpgfutil@in@%
    \pgfoo@declareunpackspace#1;%
  \else%
    \pgfutil@in@={#1}%
    \ifpgfutil@in@%
      \pgfoo@declareunpack#1;%
    \else%
      \pgfoo@declareattribute@\let{#1}\pgfutil@empty
    \fi%
  \fi%
}%
\def\pgfoo@declareunpack#1=#2;{\pgfoo@declareattribute@\def{#1}{{#2}}}%
\def\pgfoo@declareunpackspace#1 =#2;{\pgfoo@declareattribute@\def{#1}{{#2}}}%

% put the initial value in the class's namespace
% put the attr name in a list
\def\pgfoo@declareattribute@#1#2#3{%
  \expandafter#1\csname pgfooY\pgfoo@classname @#2\endcsname#3%
```

```
  \expandafter\def\expandafter\pgfoo@attributes\expandafter{\pgfoo@attributes,#2}%
}%
```

从定义看，

- 命令
  - \attribute ⟨*attribute name*⟩=⟨*initial value*⟩;
  - \attribute ⟨*attribute name*⟩ =⟨*initial value*⟩;

  导致
  1. 定义控制序列

     \csname pgfooY⟨*class name*⟩@⟨*attribute name*⟩\endcsname

     它保存 ⟨*initial value*⟩;
  2. 重定义宏 \pgfoo@attributes 保存各个 attribute name

     ```
     \expandafter\def\expandafter\pgfoo@attributes\expandafter{
     ↪  \pgfoo@attributes,⟨attribute name⟩}
     ```

- 命令 \attribute ⟨*attribute name*⟩; 导致
  1. 控制序列

     \csname pgfooY⟨*class name*⟩@⟨*attribute name*⟩\endcsname

     被 let 为 \pgfutil@empty;
  2. 重定义宏 \pgfoo@attributes 保存各个 attribute name

     ```
     \expandafter\def\expandafter\pgfoo@attributes\expandafter{
     ↪  \pgfoo@attributes,⟨attribute name⟩}
     ```

每当执行一次命令 \pgfoo@declareattribute，宏 \pgfoo@attributes 就被重定义一次，它保存的是由各个 attribute 的名称 ⟨*attribute name*⟩ 组成的列表。由于宏 \pgfoo@attributes 的初始值是 \pgfutil@empty, 所以最后它保存的第一个符号是“逗号”。

\pgfoo@inherit@attributes

本命令的定义是：

```
\def\pgfoo@inherit@attributes{%
  % store the list of attributes declared in this class for use in
  % derived classes
  \if\relax\pgfoo@attributes\relax\else
    \edef\pgfoo@attributes{\expandafter\pgfutil@gobble\pgfoo@attributes}
    ↪  % gobble comma
  \fi
  \expandafter\let\csname pgfooY\pgfoo@classname.@pgfoo@attributes
  ↪  \endcsname\pgfoo@attributes%
  % get MRO
  \expandafter\let\expandafter\pgfoo@mro\csname pgfooY\pgfoo@classname @pgfoo@mro
  ↪  \endcsname
  \def\pgfoo@allattributes{}
  ↪  % here we will store the attrs declared in base classes
  \pgfutil@for\pgfoo@baseclass:=\pgfoo@mro\do{% for each base class in MRO order
    % the attributes declared in this base class
    \expandafter\let\expandafter\pgfoo@attributes\csname pgfooY
    ↪  \pgfoo@baseclass.@pgfoo@attributes\endcsname
    \pgfutil@for\pgfoo@attribute:=\pgfoo@attributes\do{% for each attribute
      % check if we have already found it in some previous base class
      \edef\pgfoo@temp{{,\pgfoo@attribute=}{\pgfoo@allattributes,}}%
      \expandafter\pgfutil@in@\pgfoo@temp
```

```
      \ifpgfutil@in@\else % don't overwrite
        % append to the list
        \edef\pgfoo@allattributes{\pgfoo@allattributes,\pgfoo@attribute=
        ↪  \pgfoo@baseclass}%
      \fi
    }%
  }%
  \if\relax\pgfoo@allattributes\relax\else
    \edef\pgfoo@allattributes{\expandafter\pgfutil@gobble\pgfoo@allattributes}
    ↪  % gobble comma
  \fi
  % this macro will hold a list of attributes from all base classes
  % we run this macro at init and gc
  \def\pgfoo@process@attributes{}%
  \pgfutil@for\pgfoo@attributeclass:=\pgfoo@allattributes\do{
  ↪  % for each (attribute, class) pair
    % append to the attribute processor
    \expandafter\pgfoo@inherit@attributes@appendattribute\pgfoo@attributeclass.%
  }%
  \expandafter\let\csname pgfooY\pgfoo@classname .@pgfoo@process@attributes
  ↪  \endcsname\pgfoo@process@attributes
}%
```

本命令的处理是：

1. 如果宏 \pgfoo@attributes 保存的内容非空，则重定义它

   ```
   \edef\pgfoo@attributes{\expandafter\pgfutil@gobble\pgfoo@attributes}
   ↪  % gobble comma
   ```

   命令 \pgfutil@gobble 的作用是“吃掉开头的逗号”。

2. 令

   \csname pgfooY⟨*class name*⟩.@pgfoo@attributes\endcsname

   等于 \pgfoo@attributes，保存由各个 attribute 的名称 ⟨*attribute name*⟩ 组成的列表。

3. 令 \pgfoo@mro 等于 \csname pgfooY⟨*class name*⟩@pgfoo@mro\endcsname 这个控制序列保存的是由某些 class name 组成的列表，即

   ⟨*class name*⟩,⟨*super-class-name-1*⟩,⟨*super-class-name-2*⟩,...

   参考 \pgfooclass@ →P. 520 中的 \pgfoo@ciii.

4. 定义 \def\pgfoo@allattributes{}, 初值为空。

5. 执行一个循环操作：令宏 \pgfoo@baseclass (用作循环变量)“代表” \pgfoo@mro 中的任意一个列表项目 (一个 class name)：

   (a) 令 \pgfoo@attributes 等于

   \csname pgfooY\pgfoo@baseclass.@pgfoo@attributes\endcsname

   注意 \pgfoo@attributes 是与 \pgfoo@baseclass 对应的循环变量。

   (b) 执行一个循环操作：令宏 \pgfoo@attribute (用作循环变量)“代表” \pgfoo@attributes 中的任意一个列表项目 (一个 attribute name)：

   - 如果“\pgfoo@attributes,”中包含符号串“,\pgfoo@attribute=”，则什么也不做，这会确保同一个 attribute name 不被重复保存；
   - 否则，定义

     ```
     \edef\pgfoo@allattributes{\pgfoo@allattributes,\pgfoo@attribute=
     ↪  \pgfoo@baseclass}%
     ```

   所以完成上述 (套嵌的) 循环后，宏 \pgfoo@allattributes 保存的是由逗号和

⟨*an attribute name*⟩=⟨ 所属的 *class name*⟩

组成的列表，这个列表可能是：

,
⟨*attribute name 0-1*⟩=⟨*class name*⟩,
⟨*attribute name 0-2*⟩=⟨*class name*⟩,
...
⟨*attribute name 1-1*⟩=⟨*super-class-name-1*⟩,
⟨*attribute name 1-2*⟩=⟨*super-class-name-1*⟩,
...
⟨*attribute name 2-1*⟩=⟨*super-class-name-2*⟩,
⟨*attribute name 2-2*⟩=⟨*super-class-name-2*⟩,
...

注意这个列表的开头是个逗号，其中等号左侧不会有重复的 attribute name, 并且，如果 ⟨*attribute name 1-1*⟩ 既是 ⟨*super-class-name-1*⟩ 的 attribute name, 也是 ⟨*super-class-name-2*⟩ 的 attribute name, 那么，只有 ⟨*attribute name 1-1*⟩=⟨*super-class-name-1*⟩ 出现在上述列表中。

6. 如果 \pgfoo@allattributes 保存的内容非空，则定义

```
\edef\pgfoo@allattributes{\expandafter\pgfutil@gobble\pgfoo@allattributes}
↪  % gobble comma
```

命令 \pgfutil@gobble 的作用是“吃掉开头的逗号”。

7. 定义 \def\pgfoo@process@attributes{}, 这个宏的初始值为空。
8. 执行一个循环操作:令宏 \pgfoo@attributeclass (用作循环变量)“代表”\pgfoo@allattributes 中的任意一个列表项目 (一个 ⟨*an attribute name*⟩=⟨所属的 *class name*⟩)，执行

```
\pgfoo@inherit@attributes@appendattribute ⟨an attribute name⟩=⟨ 所属的 class name⟩.
```

这个循环操作将重定义宏 \pgfoo@process@attributes，见下文。

9. 令

\csname pgfooY⟨*class name*⟩.@pgfoo@process@attributes\endcsname

等于 \pgfoo@process@attributes, 通常，这个控制序列保存一些命令，这些命令可以决定上述列表中各个 ⟨*an attribute name*⟩ 如何被定义。

这个控制序列就是 \pgfoo@inherit@attributes 的结果：保存定义属性的命令，而不是直接定义那些可继承的属性。

使用这个控制序列的是：

- 命令 \pgfoonew →P. 537 (或者说 \pgfoo@@new, \pgfoo@@new@attribute)
- 命令 \pgfoogc →P. 543 (或者说 \pgfoo@dogc)
- object 这个预定义的 class 的定义代码

\pgfoo@inherit@attributes@appendattribute ⟨*an attribute name*⟩=⟨所属的 *class name*⟩.

本命令的定义是：

```
% #1=attr, #2=class which the attr comes from
\def\pgfoo@inherit@attributes@appendattribute#1=#2.{%
  \def\pgfoo@tempA{\pgfoo@attribute@op{#1}}%
  \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter
    \pgfoo@tempB\expandafter\expandafter\expandafter{
      \expandafter\pgfoo@tempA\csname pgfooY#2@#1\endcsname}%
  \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter
    \pgfoo@process@attributes\expandafter\expandafter\expandafter{
```

```
      \expandafter\pgfoo@process@attributes\pgfoo@tempB}%
  % above is just this:
  %   \def\pgfoo@process@attributes{
  %     \pgfoo@process@attributes\pgfoo@attribute@op{#1}%
  %     \csname pgfooY#2@#1\endcsname}
  %  with \pgfoo@process@attributes and \csname expanded exactly once:
}%
```

这个命令重定义宏 \pgfoo@process@attributes, 增加它保存的内容，将命令

```
\pgfoo@attribute@op{⟨an attribute name⟩}{\csname pgfooY⟨ 所属的 class name⟩@
↪  ⟨an attribute name⟩\endcsname}
```

添加到这个宏的原来内容的右侧，宏 \pgfoo@process@attributes 保存的就是数个这样的命令。

不同情况下，命令 \pgfoo@attribute@op 会被 let 为不同的命令，其作用是为属性 ⟨*an attribute name*⟩ 做不同的定义：

- 在 \pgfoo@@new，\pgfoo@@new@attribute 中有：

  ```
  \let\pgfoo@attribute@op\pgfoolet
  ```
  →P. 542

  此时 \pgfoo@process@attributes 实际保存的是由

  ```
  \pgfoolet{⟨an attribute name⟩}
  {\csname pgfooY⟨ 所属的 class name⟩@⟨an attribute name⟩\endcsname}
  ```

  组成的序列 (之间没有分隔符号)。按 \pgfoolet→P. 542 的定义，如果执行 \pgfoo@process@attributes，那么控制序列

  \csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

  将全局地等于

  \csname pgfooY⟨ 所属的 *class name*⟩@⟨*an attribute name*⟩\endcsname

  其中 ⟨*this count value*⟩ 是计数器 \pgfoothis@count 的当前值。

- 在 \pgfoo@dogc 中有：

  ```
  \let\pgfoo@attribute@op\pgfoo@gc@attribute
  ```

  此时 \pgfoo@process@attributes 实际保存的是由

  \pgfoolet{⟨*an attribute name*⟩}\relax

  组成的序列 (之间没有分隔符号)。如果执行 \pgfoo@process@attributes，那么控制序列

  \csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

  将全局地等于 \relax, 其中 ⟨*this count value*⟩ 是计数器 \pgfoothis@count 的当前值。

- 在 object 这个预定义 class 的定义代码中有：

  ```
  \let\pgfoo@attribute@op\pgfoo@copy@attributes
  ```

  此时 \pgfoo@process@attributes 实际保存的是由

  ```
  \pgfoo@copy@attributes{⟨an attribute name⟩}
  {\csname pgfooY⟨ 所属的 class name⟩@⟨an attribute name⟩\endcsname}
  ```

  组成的序列 (之间没有分隔符号)。

例如：

```
c=C,d=C,a=A,b=B
macro:->\pgfoo@attribute@op {c}\pgfooYC@c \pgfoo@attribute@op {d}\pgfooYC@d
\pgfoo@attribute@op {a}\pgfooYA@a \pgfoo@attribute@op {b}\pgfooYB@b
```

```
\pgfooclass{A}{ \attribute a=x; }
\pgfooclass{B}{ \attribute b=y; }
\pgfooclass(A,B){C}{
    \attribute c=z;
    \attribute d=w;
}
\makeatletter\ttfamily
\pgfoo@allattributes\par
\meaning\pgfoo@process@attributes
\makeatother
```

在上面第三个 \pgfooclass 的处理中，宏 \pgfoo@allattributes 保存的是

```
c=C,d=C,a=A,b=B
```

宏 \pgfoo@process@attributes 保存的实际是

```
\pgfoolet{c}{\csname pgfooYC@c\endcsname}
\pgfoolet{d}{\csname pgfooYC@d\endcsname}
\pgfoolet{a}{\csname pgfooYA@a\endcsname}
\pgfoolet{b}{\csname pgfooYB@b\endcsname}
```

29.6 创建一个 object

在 \pgfooclass@ 的处理中有：

```
\expandafter\let\csname pgfooY\pgfoo@classname.get handle\endcsname\pgfoo@obj%
\expandafter\let\csname pgfooY\pgfoo@classname.get id\endcsname\pgfoo@id%
```

```
\newtoks\pgfoo@toks
```

\pgfoo@collect@args

这个命令的使用方式是：

- \pgfoo@collect@args ⟨*some thing*⟩)
- \pgfoo@collect@args{⟨*some thing*⟩})

本命令将 ⟨*some thing*⟩ 保存到记号寄存器 \pgfoo@toks 中，然后执行 \pgfoo@continue(这个宏通常被 let 为其他命令)。

\pgfoo@new@create\⟨*objectname*⟩{⟨*a class name*⟩}

本命令将 \⟨*objectname*⟩ 定义为一个 ⟨*object handle*⟩.

本命令的定义是：

```
\def\pgfoo@new@create#1#2{%
  \advance\pgfoo@objectcount by 1\relax%
  \edef\pgfoolastobj{\noexpand\pgfoo@caller{\the\pgfoo@objectcount}}%
  \expandafter\gdef\csname pgfooX\the\pgfoo@objectcount @class\endcsname{#2}%
  \let#1\pgfoolastobj%
}%
```

这个命令的处理是：

1. 将计数器 \pgfoo@objectcount 的值加 1, 假设此时它的值是 ⟨*object count value*⟩.
2. 把 \pgfoo@caller{⟨*object count value*⟩} 保存到宏 \pgfoolastobj 中。
3. 全局地定义

   \csname pgfooX⟨*object count value*⟩@class\endcsname

   使之保存 ⟨*a class name*⟩.
4. 令 \⟨*objectname*⟩ 等于宏 \pgfoolastobj.

\pgfoo@caller{⟨number⟩}.⟨⟨m⟩ or ⟨c.m⟩⟩(⟨parameters⟩)

本命令的参数 ⟨number⟩ 是个整数。

本命令调用方法来处理参数 ⟨parameters⟩.

参数 ⟨⟨m⟩ or ⟨c.m⟩⟩ 可以是：

- “⟨a method name⟩”，会导致命令

  \csname pgfooY⟨a class name⟩.⟨m⟩\endcsname(⟨parameters⟩)

  其中的 ⟨a class name⟩ 是编号为 ⟨number⟩ 的对象所从属的类名称
- “⟨A Class Name⟩.⟨a method name⟩”，会导致命令

  \csname pgfooY⟨A Class Name⟩.⟨a method name⟩\endcsname(⟨parameters⟩)

参数 (⟨parameters⟩) 应当能用作以上命令的参数，参考 \pgfoo@declaremethod → P. 527.

本命令调用的方法 ⟨a method name⟩ 或者属于“编号为 ⟨number⟩ 的对象所从属的类”，或者属于 ⟨A Class Name⟩ 类。

本命令的定义是：

```
\def\pgfoo@caller#1.#2({%
  \expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooY\csname
  ↪  pgfooX#1@class\endcsname.#2\endcsname%
  \ifx\pgfoo@caller@temp\relax%
    % assume that #2 is of form "superclass.method"
    \expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooY#2\endcsname%
    \ifx\pgfoo@caller@temp\relax%
      \pgferror{Object #1 has no method '#2'}%
    \fi%
  \fi%
  \def\pgf@marshal{%
    \pgfoothis@count=#1\relax%
  }%
  \let\pgfoo@continue\pgfoo@caller@cont%
  \pgfoo@collect@args%
}%
\def\pgfoo@caller@cont{%
  \edef\pgf@marshal{%
    \pgf@marshal%
    \noexpand\pgfoo@caller@temp(\the\pgfoo@toks)%
  }%
  \expandafter\pgf@marshal\expandafter\pgfoothis@count\the\pgfoothis@count\relax%
}%
```

上面的定义代码中，把控制序列 \csname pgfooX⟨number⟩@class\endcsname 保存的 class name 记为 ⟨a class name⟩ (参考 \pgfoo@new@create → P. 535)，这个命令的处理是：

1. 令 \pgfoo@caller@temp 等于 \csname pgfooY⟨a class name⟩.⟨⟨m⟩ or ⟨c.m⟩⟩\endcsname.
2. 如果 \pgfoo@caller@temp 等于 \relax, 即尚未被定义，则
   (a) 令 \pgfoo@caller@temp 等于 \csname pgfooY⟨⟨m⟩ or ⟨c.m⟩⟩\endcsname.
   (b) 如果此时 \pgfoo@caller@temp 还是等于 \relax, 则报错：Object ⟨number⟩ has no method '⟨⟨m⟩ or ⟨c.m⟩⟩'

   这一步是检查参数 ⟨⟨m⟩ or ⟨c.m⟩⟩ 的形式：
   - 如果是“⟨a method name⟩”，则

     \expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooY\csname pgfooX
     ↪ ⟨number⟩@class\endcsname.⟨a method name⟩\endcsname%
     即

```
\expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooY⟨a class name⟩.
↪ ⟨a method name⟩\endcsname%
```

- 如果是“⟨*A Class Name*⟩.⟨*a method name*⟩”，则

```
\expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooY⟨A Class Name⟩.
↪ ⟨a method name⟩\endcsname%
```

3. 将 ⟨*parameters*⟩ 保存到记号寄存器 \pgfoo@toks 中。
4. 执行 \pgfoo@caller@cont, 这个命令的处理是：
   假设此时的计数器值 \the\pgfoothis@count=⟨*this count value*⟩,
   (a) 改变计数器值 \pgfoothis@count=⟨*number*⟩
   (b) 执行 \pgfoo@caller@temp(⟨*parameters*⟩), 实际执行的是
   \csname pgfooY⟨*a class name*⟩.⟨*m*⟩\endcsname(⟨*parameters*⟩)
   或者
   \csname pgfooY⟨*c.m*⟩\endcsname(⟨*parameters*⟩)
   (c) 恢复计数器值 \pgfoothis@count=⟨*this count value*⟩

注意本命令先改变计数器 \pgfoothis@count 的值，然后调用方法，然后恢复这个计数器的值。

\pgfoonew ⟨*object handle or attribute*⟩=new ⟨*a class name*⟩(⟨*constructor arguments*⟩)

这个命令创建一个 object, 每个 object 都对应 (计数器) \pgfoo@objectcount 的一个值 ⟨*object count value*⟩。

这个命令的使用格式是：

格式一 \pgfoonew \⟨*objectname*⟩=new ⟨*a class name*⟩(⟨*constructor arguments*⟩)

这个格式把 \⟨*objectname*⟩ “非全局地”定义为一个 ⟨*object handle*⟩.

格式二 \pgfoonew {⟨*attribute*⟩}=new ⟨*a class name*⟩(⟨*constructor arguments*⟩)

注意这个格式中的花括号。这个格式中的 ⟨*attribute*⟩ 最好是类 ⟨*a class name*⟩ 具有的属性 (包括继承自父类是属性)，否则无法照顾命令 \pgfoogc→P. 543 的作用。

这个格式把 \csname pgfooX⟨*this count*⟩@⟨*attribute*⟩\endcsname “全局地”定义为一个 ⟨*object handle*⟩.

格式三 \pgfoonew new ⟨*a class name*⟩(⟨*constructor arguments*⟩)

这个格式等效于 \pgfoonew \pgfoo@dummy=new ⟨*a class name*⟩(⟨*constructor arguments*⟩)

参数 ⟨*a class name*⟩ 是 object 所从属的 class 名称 (事先由 \pgfooclass→P. 520 声明)。

宏 \⟨*objectname*⟩ 将被定义为一个 ⟨*object handle*⟩。

假设参数 ⟨*constructor arguments*⟩ 的彻底展开是 ⟨*constructor arguments'*⟩, 本命令会执行下面的

\csname pgfooY⟨*a class name*⟩.init\endcsname(⟨*constructor arguments'*⟩)

参考 \pgfoo@declaremethod→P. 527.

本命令的有效范围受到 TEX 组的限制。

本命令的最后一步是执行 \aftergroup\pgfoogc→P. 543, 所以可以考虑把本命令放在一个组中来执行，这个组结束后，命令 \pgfoogc→P. 543 会清理 \pgfoonew 为对象全局定义的那些属性。

29.6.1 格式一

命令 \pgfoonew \⟨*objectname*⟩=new ⟨*a class name*⟩(⟨*constructor arguments*⟩) 的处理是：

1. 检查 \csname pgfooY⟨*a class name*⟩.get id\endcsname 是否已有定义
   - 如果未定义，则报错：Unknown class '⟨*a class name*⟩'
   - 如果已定义，则

(a) 执行命令

```
\pgfoo@new@create\⟨objectname⟩{⟨a class name⟩}
```

这个命令的处理是：

i. 将计数器 \pgfoo@objectcount 的值加 1, 假设此时它的值是 ⟨*object count value*⟩.

ii. 把 “\pgfoo@caller{⟨*object count value*⟩}” 保存到宏 \pgfoolastobj 中。

iii. 全局地定义

\csname pgfooX⟨*object count value*⟩@class\endcsname

使之保存 ⟨*a class name*⟩.

iv. 令 \⟨*objectname*⟩ 等于宏 \pgfoolastobj.

(b) 执行

```
{%
  \pgfoothis@count\pgfoo@objectcount%
  \let\pgfoo@attribute@op\pgfoolet
  \csname pgfooY⟨a class name⟩.@pgfoo@process@attributes\endcsname%
}%
```

参考 \pgfoo@inherit@attributes →P. 531, \pgfoo@inherit@attributes@appendattribute →P. 533. 这导致各个属性被定义，例如控制序列

\csname pgfooX⟨*object count value*⟩@⟨*an attribute name*⟩\endcsname

全局地等于

\csname pgfooY⟨ 所属的 *class name*⟩@⟨*an attribute name*⟩\endcsname

其中 ⟨*this count value*⟩=⟨*object count value*⟩ 是计数器 \pgfoo@objectcount 的值；⟨*an attribute name*⟩ 是当前的 \pgfoo@attributes 中的任意一个 attribute 名称；⟨所属的 *class name*⟩ 是 ⟨*an attribute name*⟩ 所从属的 class name.

(c) 用 \expandafter 将 ⟨*constructor arguments*⟩ 展开一次，得到 ⟨*constructor arguments'*⟩, 将其保存到记号寄存器 \pgfoo@toks 中。

(d) 执行 \pgf@oo@new@cont, 实际是执行

```
\pgfoo@caller{⟨object count value⟩}.init(⟨constructor arguments'⟩)%
\aftergroup\pgfoogc%
```

这个命令的处理是：

i. 令 \pgfoo@caller@temp 等于 \csname pgfooY⟨*a class name*⟩.init\endcsname.

ii. 如果 \pgfoo@caller@temp 等于 \relax, 即尚未被定义，则

A. 令 \pgfoo@caller@temp 等于 \csname pgfooYinit\endcsname.

B. 如果此时 \pgfoo@caller@temp 还是等于 \relax, 则报错：Object ⟨*object count value*⟩ has no method 'init'

iii. 将 ⟨*constructor arguments'*⟩ 保存到记号寄存器 \pgfoo@toks 中。

iv. 执行 \pgfoo@caller@cont，这个命令的处理是：

假设 \the\pgfoothis@count=⟨*this count value*⟩,

A. 赋值 \pgfoothis@count=⟨*object count value*⟩

B. 执行 \pgfoo@caller@temp(⟨*constructor arguments'*⟩), 实际执行的是

\csname pgfooY⟨*a class name*⟩.init\endcsname(⟨*constructor arguments'*⟩)

C. 还原计数器值 \pgfoothis@count=⟨*this count value*⟩

v. 执行 \aftergroup\pgfoogc

所以命令 \pgfoonew \⟨*objectname*⟩=new ⟨*a class name*⟩(⟨*constructor arguments*⟩) 的结果主要是：

1. 全局地定义

   \csname pgfooX⟨*object count value*⟩@class\endcsname

   使之保存 ⟨*a class name*⟩.
2. 定义 ⟨*object handle*⟩，即 \⟨*objectname*⟩，它保存“\pgfoo@caller{⟨*object count value*⟩}”。
3. 定义 \⟨*objectname*⟩ 的那些属性 (包括继承来的属性)，即令控制序列

   \csname pgfooX⟨*object count value*⟩@⟨*an attribute name*⟩\endcsname

   全局地等于

   \csname pgfooY⟨ 所属的 *class name*⟩@⟨*an attribute name*⟩\endcsname
4. 执行

   \csname pgfooY⟨*a class name*⟩.init\endcsname(⟨*constructor arguments'*⟩)
5. 还原计数器值 \pgfoothis@count=⟨*this count value*⟩
6. 在组结束后执行 \pgfoogc。

可见这个格式中，命令 \pgfoonew 利用计数器值 ⟨*object count value*⟩ 将 ⟨*a class name*⟩, ⟨*object handle*⟩, ⟨*an attribute name*⟩ 联系起来。计数器 \pgfoo@objectcount 为对象提供编号，而计数器 \pgfoothis@count 的值最终是被保持的。

29.6.2 格式二

命令 \pgfoonew {⟨*attribute*⟩}=new ⟨*a class name*⟩(⟨*constructor arguments*⟩) 的处理过程与前文的“格式一”是类似地，其结果主要是：

1. 全局地定义

   \csname pgfooX⟨*object count value*⟩@class\endcsname

   使之保存 ⟨*a class name*⟩.
2. 用 \pgfoolet →P. 542 全局地定义控制序列

   \csname pgfooX⟨*this count value*⟩@⟨*attribute*⟩\endcsname

   为一个 ⟨*object handle*⟩, 它全局地保存”\pgfoo@caller{⟨*object count value*⟩}”.

   注意 ⟨*this count value*⟩ 是计数器 \pgfoothis@count 的当前值，未必等于 ⟨*object count value*⟩ (计数器 \pgfoo@objectcount 的当前值)。
3. 定义与 ⟨*object count value*⟩ 相关的属性，例如令控制序列

   \csname pgfooX⟨*object count value*⟩@⟨*an attribute name*⟩\endcsname

   全局地等于

   \csname pgfooY⟨ 所属的 *class name*⟩@⟨*an attribute name*⟩\endcsname
4. 执行

   \csname pgfooY⟨*a class name*⟩.init\endcsname(⟨*constructor arguments'*⟩)
5. 保持计数器值 \pgfoothis@count=⟨*this count value*⟩
6. 在组结束后执行 \pgfoogc。

29.6.3 格式三

命令 \pgfoonew new ⟨*a class name*⟩(⟨*constructor arguments*⟩) 等效于 \pgfoonew \pgfoo@dummy=new ⟨*a class name*⟩(⟨*constructor arguments*⟩), 所以此时的 \pgfoo@dummy 是 ⟨*object handle*⟩.

29.6.4 ⟨*object handle*⟩ 的用法

因为 ⟨*object handle*⟩ 保存的是“\pgfoo@caller{⟨*object count value*⟩}”，所以 ⟨*object handle*⟩ 的用法与 \pgfoo@caller→P. 536 类似。⟨*object handle*⟩ 的用法是：

⟨*object handle*⟩.⟨*a class name*⟩.⟨*method name*⟩(⟨*parameters*⟩)

或者

⟨*object handle*⟩.⟨*method name*⟩(⟨*parameters*⟩)

其中的 (⟨*parameters*⟩) 应当能用作命令

\csname pgfooY⟨*a class name*⟩.⟨*method name*⟩\endcsname

的参数，参考 \pgfoo@declaremethod→P. 527. 实际上 ⟨*object handle*⟩ 的用处就是调用这种与 method 对应的命令来处理参数。

29.7 与 object, attribute 有关的计数器

与 object, attribute 有关的计数器是：

```
\newcount\pgfoo@objectcount
\newcount\pgfoothis@count
```

这 2 个计数器值都不是全局变化的。

每个 object 都对应一个 \pgfoo@objectcount 值。每当使用命令 \pgfoonew→P. 537 创建 object 时，命令 \pgfoo@new@create→P. 535 被执行，导致计数器 \pgfoo@objectcount 的值 (非全局地) 加 1, 此时这个计数器的值就是正在被创建的 object 对应的编号；同时也创建与这个编号对应的各个属性；这样，类、对象、属性，三者就通过这个编号对应起来的了。

计数器 \pgfoothis@count 的值有时参照计数器 \pgfoo@objectcount 的值变化，有时它也独自变化。计数器 \pgfoothis@count 的值 ⟨*this count value*⟩ 用于构成

\csname pgfooX⟨*this count value*⟩@class\endcsname

\csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

的名称，也用于 \pgfoo@id, \pgfoo@obj→P. 541, \pgfoogc→P. 543 等命令的定义中。

当执行一个 ⟨*object handle*⟩, 或者说 \pgfoo@caller→P. 536{⟨*number*⟩} 时，会先

```
\pgfoothis@count=⟨number⟩\relax%
```

然后调用某个方法 ⟨*a method name*⟩ 来处理参数 ⟨*parameters*⟩ (如何利用计数器 \pgfoothis@count 的值要看方法 ⟨*a method name*⟩ 的定义)，然后恢复计数器 \pgfoothis@count 的值。

```
\makeatletter
\pgfooclass{A}{
    \method a(){}
    \attribute x;
}
\pgfooclass(A){B}{}
\pgfoonew \aaaa=new B()
{\pgfoonew \bbbb=new A()}
\the\pgfoo@objectcount\par
\the\pgfoothis@count
\makeatother
```

```
1
0
```

\pgfoo@id(\⟨*macro*⟩)

本命令的定义是：

```
\def\pgfoo@id(#1){%
  \edef#1{\the\pgfoothis@count}%
}%
```

本命令把当前的 object handle 的编号保存到 \⟨*macro*⟩ 中。

\pgfooobj{⟨*number*⟩}

本命令的定义是：

```
\def\pgfooobj#1{%
  \pgfoo@caller{#1}%
}%
```

按 \pgfoo@caller →P. 536 的定义，本命令的使用方式是：

\pgfooobj{⟨*number*⟩}.⟨*method name*⟩(⟨*parameters*⟩)

或者

\pgfooobj{⟨*number*⟩}.⟨*A Class Name*⟩.⟨*method name*⟩(⟨*parameters*⟩)

本命令调用编号为 ⟨*number*⟩ 的对象。

\pgfoo@obj(\⟨*macro*⟩)

本命令的定义是：

```
\def\pgfoo@obj(#1){%
  \edef#1{\noexpand\pgfoo@caller{\the\pgfoothis@count}}%
}%
```

按 \pgfoo@caller →P. 536 的定义，本命令的使用方式是：

```
\pgfoo@obj(\⟨macro⟩)% 定义 \⟨macro⟩
\⟨macro⟩.⟨method name⟩(⟨parameters⟩)% 或者
\⟨macro⟩.⟨A Class Name⟩.⟨method name⟩(⟨parameters⟩)
```

本命令把 \⟨*macro*⟩ 定义为一个 ⟨*object handle*⟩, 它等效于当前的 object handle.

\pgfoosuper(⟨*class*⟩,\⟨*object handle*⟩).⟨*method name*⟩(⟨*parameters*⟩)

本命令的处理是：

1. 获取 \⟨*object handle*⟩ 所属的类名称 ⟨*obj-class*⟩, 假设对 ⟨*obj-class*⟩ 的声明是

   ```
   \pgfooclass{⟨class 1⟩,⟨class 2⟩,...}{⟨obj-class⟩}{...}
   ```

   注意其中的父类列表 ⟨*class 1*⟩,⟨*class 2*⟩,... 不能是空的，否则报错。

   父类列表会被 \pgfoo@ciii →P. 522 处理，处理结果仍然记为 ⟨*class 1*⟩,⟨*class 2*⟩,...

2. 检查列表 ⟨*class 1*⟩,⟨*class 2*⟩,...,

   - 如果其中包含第一个参数 ⟨*class*⟩, 即

     ⟨*class 1*⟩,...,⟨*class*⟩,⟨*class k*⟩,...

     那么后面步骤就只处理列表 ⟨*class k*⟩,⟨*class k+1*⟩,...
   - 如果其中不包含第一个参数 ⟨*class*⟩, 那么后面步骤就处理列表 ⟨*class 1*⟩,⟨*class 2*⟩,...

   注意此时待处理列表不应当是空的，否则报错。

3. 对于待处理列表中的类名称 ⟨*class i*⟩, 检查控制序列

   \csname pgfooY⟨*class i*⟩.⟨*method name*⟩\endcsname

   是否有定义，即检查类 ⟨*class i*⟩ 中是否有方法 ⟨*method name*⟩:

   - 如果有，就执行

     \⟨*object handle*⟩.⟨*class i*⟩.⟨*method name*⟩(⟨*parameters*⟩)

     也就是调用 ⟨*class i*⟩ 中的方法 ⟨*method name*⟩ 来处理 ⟨*parameters*⟩.

     参考 \pgfoo@caller →P. 536.

- 如果没有，就再检查类 ⟨*class i+1*⟩ 中是否有方法 ⟨*method name*⟩, 如此以往。

如果检查彻底失败，会导致报错。

可见第一个参数 ⟨*class*⟩ 是“被排除”的类。

29.8 针对属于某个 object 的 attribute 的操作

声明计数器：

```
\newcount\pgfoo@objectcount
\newcount\pgfoothis@count
```

\pgfoo@gc@attribute{⟨*an attribute name*⟩}⟨*a token*⟩

本命令使得控制序列

\csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

全局地等于 \relax, 参考 \pgfoolet. 注意本命令会吃掉参数 ⟨*a token*⟩。

```
\def\pgfoo@gc@attribute#1#2{\pgfoolet{#1}\relax}%
```

\pgfooset{⟨*an attribute name*⟩}{⟨*value*⟩}

本命令的定义是：

```
\long\def\pgfooset#1#2{%
  \expandafter\gdef\csname pgfooX\the\pgfoothis@count @#1\endcsname{#2}%
}%
```

本命令全局地定义控制序列

\csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

使之保存 ⟨*value*⟩.

\pgfooeset{⟨*an attribute name*⟩}{⟨*value*⟩}

本命令的定义是：

```
\long\def\pgfooeset#1#2{%
  \expandafter\xdef\csname pgfooX\the\pgfoothis@count @#1\endcsname{#2}%
}%
```

\pgfoovalueof{⟨*an attribute name*⟩}

本命令的定义是：

```
\def\pgfoovalueof#1{%
  \csname pgfooX\the\pgfoothis@count @#1\endcsname%
}%
```

本命令导致控制序列

\csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

被执行。注意，这个控制序列有可能是一个 ⟨*object handle*⟩, 见 \pgfoonew → P. 537.

\pgfoolet{⟨*an attribute name*⟩}{⟨*\macro*⟩}

本命令的定义是：

```
\def\pgfoolet#1#2{%
  \expandafter\global\expandafter\let\csname pgfooX\the\pgfoothis@count @#1
  ↪  \endcsname#2%
}%
```

本命令使得控制序列

\csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

全局地等于 ⟨*macro*⟩.

\pgfooget{⟨*an attribute name*⟩}{⟨*macro*⟩}

本命令的定义是：

```
\def\pgfooget#1#2{%
  \expandafter\let\expandafter#2\csname pgfooX\the\pgfoothis@count @#1\endcsname%
}%
```

本命令使得宏 ⟨*macro*⟩ 等于控制序列

\csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

注意，这个控制序列有可能是一个 ⟨*object handle*⟩, 见 \pgfoonew → P. 537.

\pgfooappend{⟨*an attribute name*⟩}{⟨*code*⟩}

本命令的定义是：

```
\def\pgfooappend#1#2{%
  \expandafter\expandafter\expandafter\def%
  \expandafter\expandafter\expandafter\pgf@oo@temp
  ↪  \expandafter\expandafter\expandafter{\csname pgfooX\the\pgfoothis@count @#1
  ↪  \endcsname#2}%
  \expandafter\global\expandafter\let\csname pgfooX\the\pgfoothis@count @#1
  ↪  \endcsname\pgf@oo@temp%
}%
```

本命令重定义控制序列

\csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

在其原来的定义内容的末尾附加 ⟨*code*⟩.

\pgfooprefix{⟨*an attribute name*⟩}{⟨*code*⟩}

本命令的定义是：

```
\def\pgfooprefix#1#2{%
  \pgfooget{#1}\pgf@oo@temp%
  \def\pgf@oo@@temp{#2}%
  \expandafter\expandafter\expandafter\def%
  \expandafter\expandafter\expandafter\pgf@oo@temp%
  \expandafter\expandafter\expandafter{\expandafter\pgf@oo@@temp\pgf@oo@temp}%
  \pgfoolet{#1}\pgf@oo@temp%
}%
```

本命令重定义控制序列

\csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

在其原来的定义内容的开头附加 ⟨*code*⟩.

29.9 Garbage collector

\pgfoogc

本命令的定义是：

```
\def\pgfoogc{%
  {%
    % We do this in a group...
```

```
    \pgfoothis@count\pgfoo@objectcount% this is temporary...
    \let\pgfoo@next=\pgfoo@dogc%
    \pgfoo@next%
  }%
}%
\def\pgfoo@dogc{%
  \advance\pgfoothis@count by 1\relax%
  \expandafter\ifx\csname pgfooX\the\pgfoothis@count @class\endcsname\relax%
    \let\pgfoo@next=\relax%
  \else%
    % Cleanup this object:

    ↪  % The following is the fast version of \pgfooobj{\the\pgfoo@objectcount}.@pgfoogc:
    \let\pgfoo@attribute@op\pgfoo@gc@attribute
    \csname pgfooY\csname pgfooX\the\pgfoothis@count @class
    ↪  \endcsname.@pgfoo@process@attributes\endcsname%
    \expandafter\global\expandafter\let\csname pgfooX\the\pgfoothis@count @class
    ↪  \endcsname\relax%
  \fi%
  \pgfoo@next%
}%
```

从定义代码看，这个命令的所有操作都是限制在一个花括号组内的，这个命令实际上是个循环操作：从计数器 \pgfoo@objectcount 的值 ⟨*object count value*⟩ 开始 (不含这个值)，对于那些大于 ⟨*object count value*⟩ 的整数值 ⟨*this count value*⟩, 全局地取消那些与 ⟨*this count value*⟩ 对应的控制序列，即使得

\csname pgfooX⟨*this count value*⟩@⟨*an attribute name*⟩\endcsname

全局地等于 \relax, 这些控制序列可能是：

- 与 ⟨*this count value*⟩ 对应的属性，参考 \pgfoo@inherit@attributes@appendattribute →P. 533
- 由 ⟨*an attribute name*⟩ 构造的 ⟨*object handle*⟩, 如果 ⟨*an attribute name*⟩ 是被 \pgfoo@inherit@attributes →P 保存的属性的话，参考 \pgfoonew →P. 537

在命令 \pgfoonew →P. 537 (\pgf@oo@new@cont) 的最后会自动执行 \aftergroup\pgfoogc, 所以通常情况下用户无需自己调用 \pgfoogc.

29.10 预定义的内容

29.10.1 object handle \pgfoothis

这个 ⟨*object handle*⟩ 的使用方式是：

1. \pgfoothis.⟨*method name*⟩(⟨*parameters*⟩), 导致

```
\csname pgfooY\csname pgfooX\the\pgfoothis@count @class\endcsname.⟨method name⟩
↪  \endcsname(⟨parameters⟩)
```

其中 \csname pgfooX\the\pgfoothis@count @class\endcsname 保存某个 class name, 是与计数器 \pgfoothis@count 的当前值对应的类名称。

2. \pgfoothis.⟨*A Class Name*⟩.⟨*method name*⟩(⟨*parameters*⟩), 导致

```
\csname pgfooY⟨A Class Name⟩.⟨method name⟩\endcsname(⟨parameters⟩)
```

所以 \pgfoothis 会调用其他方法来处理参数 ⟨*parameters*⟩.

这个 ⟨*object handle*⟩ 的定义是：

```
\def\pgfoothis.#1({%
  \expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooY\csname pgfooX
  ↪  \the\pgfoothis@count @class\endcsname.#1\endcsname%
  \ifx\pgfoo@caller@temp\relax%
    % assume that #1 is of form "superclass.method"
    \expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooY#1\endcsname%
  \fi%
  \let\pgfoo@continue\pgfoothis@cont%
  \pgfoo@collect@args%
}%
\def\pgfoothis@cont{%
  \expandafter\pgfoo@caller@temp\expandafter(\the\pgfoo@toks)%
}%
```

可见 \pgfoothis 首先检查第一种使用方式，如果这个方式行不通，就检查第二种使用方式。

29.10.2 Method get id

在 \pgfooclass@ → P. 520 的处理中有

```
\expandafter\let\csname pgfooY\pgfoo@classname.get id\endcsname\pgfoo@id%
```

参考 \pgfoo@id → P. 540 的定义，这个 method 的用法是：

1. \⟨*object handle*⟩.⟨*A Class Name*⟩.get id(\⟨*macro*⟩)，导致

   ```
   \csname pgfooY⟨A Class Name⟩.get id\endcsname(\⟨macro⟩)
   ```

2. \⟨*object handle*⟩.get id(\⟨*macro*⟩)，导致

   ```
   \csname pgfooY⟨a-class-name⟩.get id\endcsname(\⟨macro⟩)
   ```

   其中 ⟨*a-class-name*⟩ 是 \⟨*object handle*⟩ 所属的类名称。

以上 2 个用法的结果是一样的，其中的 \⟨*macro*⟩ 实际上用作命令 \pgfoo@id → P. 540 的参数，\⟨*object handle*⟩ 的编号被保存到宏 \⟨*macro*⟩ 中。参考 \pgfoo@caller → P. 536.

29.10.3 Method get handle

在 \pgfooclass@ → P. 520 的处理中有

```
\expandafter\let\csname pgfooY\pgfoo@classname.get handle\endcsname\pgfoo@obj%
```

参考 \pgfoo@obj → P. 541 的定义，这个 method 的用法是：

1. \⟨*object handle*⟩.⟨*A Class Name*⟩.get handle(\⟨*macro*⟩)，导致

   ```
   \csname pgfooY⟨A Class Name⟩.get handle\endcsname(\⟨macro⟩)
   ```

2. \⟨*object handle*⟩.get handle(\⟨*macro*⟩)，导致

   ```
   \csname pgfooY⟨a-class-name⟩.get handle\endcsname(\⟨macro⟩)
   ```

   其中 ⟨*a-class-name*⟩ 是 \⟨*object handle*⟩ 所属的类名称。

记号 \⟨*macro*⟩ 实际上用作命令 \pgfoo@obj → P. 541 的参数，宏 \⟨*macro*⟩ 将保存代码

\pgfoo@caller{\⟨*object handle*⟩ 的编号}

也就是说，宏 \⟨*macro*⟩ 被做成一个 object handle，它等效于 \⟨*object handle*⟩. 参考 \pgfoo@caller → P. 536.

29.10.4 Class object

这个 class 的定义是：

```
\pgfooclass{object}{%
  \method copy(#1) {%
    % create a new object
    \edef\pgfoo@temp@classname{\csname pgfooX\the\pgfoothis@count @class\endcsname}%
    \expandafter\pgfoo@new@create\expandafter#1\expandafter{\pgfoo@temp@classname}%
    \def\pgfoo@copy@attributes##1##2{%
      \pgfooget{##1}\pgfoo@attrvalue
      \expandafter\let\csname pgfooX\the\pgfoo@objectcount @##1
      ↪  \endcsname\pgfoo@attrvalue
    }%
    \let\pgfoo@attribute@op\pgfoo@copy@attributes
    \csname pgfooY\pgfoo@temp@classname .@pgfoo@process@attributes\endcsname
    \aftergroup\pgfoogc% cleanup after group
  }%
}%
```

类 object 没有 init 方法。

类 object 的方法 copy 对应命令

\csname pgfooYobject.copy\endcsname

它能处理一个参数，从上述代码看，它的参数将用作 \pgfoo@new@create→P. 535 的参数，所以这个方法的参数应该是一个“宏” \⟨*macro*⟩, 这个宏将被做成一个 object handle.

假设对象 \⟨*Object Handle*⟩ 的编号是 ⟨*Number*⟩, 执行 \⟨*Object Handle*⟩.copy(\⟨*macro*⟩) 导致：

1. 假设计数器 \pgfoothis@count 的当前值是 ⟨*this count value*⟩;
2. 将 ⟨*Number*⟩ 赋予计数器 \pgfoothis@count.
3. 将 \csname pgfooX\the\pgfoothis@count @class\endcsname 保存的类名称记为 ⟨*from class*⟩, 这就是对象 \⟨*Object Handle*⟩ 所属的类名称。
4. 执行

   \pgfoo@new@create\⟨*macro*⟩{⟨*from class*⟩}

   宏 \⟨*macro*⟩ 被做成一个 object handle, 属于类 ⟨*from class*⟩ (与对象 \⟨*Object Handle*⟩ 同类)。假设 \⟨*macro*⟩ 的编号是 ⟨*number*⟩.
5. 定义 \pgfoo@copy@attributes 以及 \let\pgfoo@attribute@op\pgfoo@copy@attributes
6. 执行

   \csname pgfooY⟨*from class*⟩.@pgfoo@process@attributes\endcsname

   参考命令 \pgfoo@inherit@attributes→P. 531, \pgfoo@inherit@attributes@appendattribute→P. 533, 这会导致执行一系列如下的命令

   ```
   \pgfoo@copy@attributes{⟨an attribute name⟩}
   {\csname pgfooY⟨from class⟩@⟨an attribute name⟩\endcsname}
   ```

   也就是执行

   ```
   \pgfooget{⟨an attribute name⟩}\pgfoo@attrvalue
   \expandafter\let\csname pgfooX⟨number⟩@⟨an attribute name⟩\endcsname\pgfoo@attrvalue
   ```

   也就是

   ```
   \expandafter\let\expandafter\pgfoo@attrvalue\csname pgfooX⟨Number⟩@
   ↪  ⟨an attribute name⟩\endcsname%
   \expandafter\let\csname pgfooX⟨number⟩@⟨an attribute name⟩\endcsname\pgfoo@attrvalue
   ```

   结果是从 ⟨*from class*⟩ 那里继承 attribute, 即为 \⟨*macro*⟩ 这个 object handle 定义属性。
7. 还原计数器 \pgfoothis@count 的值 ⟨*this count value*⟩.
8. 执行 \aftergroup\pgfoogc.

29.10.5 Class signal

这个 class 的定义是：

```
\pgfooclass{signal}
{%
  %
  % This class implements signals.
  %
  % After you have created a signal object, you can call
  % connect to connect a slot. Then, whenever the emit method is
  % called, all connected methods get called.

  % Attribute
  \attribute emitter;%
  % Collects the objects that should be called.

  % Constructor
  \method signal() {}%

  % Connect a slot (method) #2 of object #1
  \method connect(#1,#2) {%
    {%
      #1.get id(\pgf@tempid)%
      % Save in emitter
      \pgfooget{emitter}\pgf@temp%
      \let\pgfoo@signal@call=\relax% avoid expansion
      \edef\pgf@temp{\pgf@temp\pgfoo@signal@call{\pgf@tempid}{#2}}%
      \pgfoolet{emitter}\pgf@temp%
    }%
  }%

  \def\pgfoo@signal@call#1#2{%
    \def\pgf@temp{\pgfooobj{#1}.#2}%
    \expandafter\pgf@temp\expandafter(\pgfoo@signal@args)%
  }%

  % Emit a signal
  \method emit(#1) {%
    \def\pgfoo@signal@args{#1}%
    \pgfoovalueof{emitter}
  }%
}%
```

29.10.5.1 method connect

类 signal 的方法 connect 的 2 个参数：

- 第一个参数 #1 可以是任何一个有效的 \⟨*object handle*⟩, 参考 \pgfoo@id →P. 540.
- 第二个参数 #2 可以是，参考 \pgfooobj →P. 541:
  - ⟨*a method name*⟩, 此时参数 \⟨*object handle*⟩ 应当能调用参数 ⟨*a method name*⟩;
  - ⟨*A Class Name*⟩.⟨*a method name*⟩

这个方法的所有操作都被限制在一个组中。

当执行 \⟨*Object Handle*⟩.connect(\⟨*object handle*⟩,⟨*a method name*⟩) 时：

1. 假设计数器 \pgfoothis@count 的当前值是 ⟨*this count value*⟩;

2. 将 \⟨*Object Handle*⟩ 的编号 ⟨*Number*⟩ 赋予计数器 \pgfoothis@count
3. 在组内：
   (a) 获取第一个参数 \⟨*object handle*⟩ 的编号 ⟨*number*⟩, 保存在 \pgf@tempid 中；
   (b) 获取控制序列
   \csname pgfooX⟨*Number*⟩@emitter\endcsname
   的当前值并保存到 \pgf@temp 中，注意其中的 emitter 是与编号 ⟨*Number*⟩, 即 \⟨*Object Handle*⟩ 对应的属性；参考 \pgfooget →P. 543;
   (c) 全局地重定义控制序列
   \csname pgfooX⟨*Number*⟩@emitter\endcsname
   使之保存以下内容：
   ⟨ 此控制序列原来的内容⟩\pgfoo@signal@call{⟨*number*⟩}{⟨*a method name*⟩}
   参考 \pgfoolet →P. 542;
4. 还原计数器 \pgfoothis@count 的值 ⟨*this count value*⟩.

总而言之，其结果是修改与 \⟨*Object Handle*⟩ 对应的属性 emitter 的内容，其内容的作用是调用其他对象的方法。

29.10.5.2 method emit

当执行 \⟨*object handle*⟩.emit(⟨*parameters*⟩) 时：

1. 假设计数器 \pgfoothis@count 的当前值是 ⟨*this count value*⟩;
2. 将 \⟨*object handle*⟩ 的编号 ⟨*number*⟩ 赋予计数器 \pgfoothis@count
3. 保存方法参数
   \def\pgfoo@signal@args{⟨*parameters*⟩}
4. 执行与 \⟨*object handle*⟩ 对应的属性 emitter, 即执行数个 \pgfoo@signal@call 命令。
   参考 \pgfoovalueof →P. 542.
   \pgfoo@signal@call 的作用就是调用方法来处理 ⟨*parameters*⟩:
   \pgfooobj{⟨ 对应某个对象的整数⟩}{⟨*m*⟩ *or* ⟨*c.m*⟩}{⟨*parameters*⟩}
   参考 \pgfooobj →P. 541.
5. 还原计数器 \pgfoothis@count 的值 ⟨*this count value*⟩.

29.10.5.3 使用方式

类 signal 的使用方式可以是：假设 \⟨*object handle x*⟩ 的编号是 ⟨*number x*⟩,

\⟨*object handle A*⟩.connect(\⟨*object handle B*⟩,⟨*method B*⟩)
\⟨*object handle A*⟩.connect(\⟨*object handle C*⟩,⟨*method C*⟩)
以上定义控制序列 \csname pgfooX⟨*number A*⟩@emitter\endcsname
执行 \⟨*object handle A*⟩.emit(⟨*parameters*⟩)
导致
\pgfoo@signal@call{⟨*number B*⟩}{⟨*method B*⟩}
\pgfoo@signal@call{⟨*number C*⟩}{⟨*method C*⟩}
导致
\pgfoobj{⟨*number B*⟩}.{⟨*method B*⟩}(⟨*parameters*⟩)
\pgfoobj{⟨*number C*⟩}.{⟨*method C*⟩}(⟨*parameters*⟩)

参考 \pgfooobj→P. 541.

下面是一个例子：

显示 emitter 的内容：macro:->\pgfoo@signal@call {1}{copy}

显示编号：3

显示定义：macro:->\pgfoo@caller {3}

```
\ttfamily
\makeatletter
\pgfoonew\test@object@usage=new object()% 利用预定义的类 object
\pgfoonew\test@singal@usage=new signal()

% 获取对象 \test@singal@usage 的编号
\test@singal@usage.get id(\test@singal@usage@number)

% 定义对象 \test@singal@usage 的属性 emitter
\test@singal@usage.connect(\test@object@usage,copy)

% 显示对象 \test@singal@usage 的属性 emitter 的内容
显示 emitter 的内容:
\expandafter\meaning\csname pgfooX\test@singal@usage@number @emitter\endcsname

% 下一命令等效于 \test@object@usage.copy(\test@singal@usage@another@obj)
\test@singal@usage.emit(\test@singal@usage@another@obj)

% 显示对象 \test@singal@usage@another@obj 的编号
显示编号:
\the\pgfoo@objectcount

% 显示对象 \test@singal@usage@another@obj 的定义
显示定义:
\meaning\test@singal@usage@another@obj
\makeatother
```

29.11 用对象 A 的方法调用对象 B 的方法

这种操作的核心是：利用计数器 \pgfoothis@count 保存对象的编号。

可以利用 signal 类。

也可以, 例如：

```
\pgfooclass{⟨class A⟩}
{
  ...
  \attribute ⟨an attribute⟩;
  \method ⟨connecting method⟩(...)
  {
    \pgfoonew {⟨an attribute⟩}=new ⟨class B⟩(...)
  }
  \method ⟨calling method⟩(...)
  {
    \pgfoovalueof{⟨an attribute⟩}.⟨called method⟩(...)
  }
}
\pgfoonew \⟨Obj Handle⟩=new ⟨class A⟩(...)
% 假设 \⟨Obj Handle⟩ 的编号是 ⟨N⟩

% 定义对象 \csname pgfooX⟨N⟩@⟨an attribute⟩\endcsname
\⟨Obj Handle⟩.⟨connecting method⟩(...)
```

```
% 调用对象 \csname pgfooX⟨N⟩@⟨an attribute⟩\endcsname 的方法 ⟨called method⟩
\⟨Obj Handle⟩.⟨calling method⟩(...)
```

29.12 总结

29.12.1 \pgfooclass

\pgfooclass→P. 520(⟨list of superclasses⟩){⟨class name⟩}{⟨body⟩} 的处理是：

```
\def\pgfoo@classname{#2}%
\expandafter\ifx\csname pgfooY\pgfoo@classname.@pgfooinit\endcsname\relax\else
    \pgferror{class #2 is already defined}%
\fi
%
%  \pgfoo@ciii{#2}{#1}
% 获取列表 class name,list of superclasses
% 这个列表保存在 \pgfoo@mro
%
\expandafter\let\csname pgfooY#2@pgfoo@mro\endcsname\pgfoo@mro
\let\pgfoo@origmethod=\method%
\let\pgfoo@origattribute=\attribute%
\let\method=\pgfoo@declaremethod%
\let\attribute=\pgfoo@declareattribute%
\let\pgfoo@attributes=\pgfutil@empty%
\let\pgfoo@methods=\pgfutil@empty%
%⟨body⟩%
%
% 定义方法
%
%\method ⟨method name⟩(⟨parameter list⟩){⟨method body⟩}
%
% 定义 ⟨class name⟩ 的方法 ⟨method name⟩，方法是能处理参数的命令
\expandafter\long\expandafter\def\csname pgfooY⟨class name⟩.⟨method name⟩\endcsname(
↪  ⟨parameter list⟩){⟨method body⟩}
%
% 保存方法名称列表到宏 \pgfoo@methods
%
% 定义属性
%
%\attribute ⟨attribute name⟩=⟨initial value⟩;
%
% 定义属性 ⟨attribute name⟩ 的初始值，属性只是保存某些代码
\expandafter\def\csname pgfooY⟨class name⟩@⟨attribute name⟩\endcsname{⟨initial value⟩}
%
% 保存属性名称列表到宏 \pgfoo@attributes
%
% inherit
\def\pgfoo@temp@baseclasses{#1}%
%
% 继承方法
%  \pgfoo@inherit@methods
%
% 将当前类 ⟨class name⟩ 的方法名称列表保存到控制序列
\csname pgfooY⟨class name⟩@pgfoo@methods\endcsname
```

```
%
% 继承父类 ⟨super-class-name-i⟩ 的方法 ⟨method i-j⟩
% 如果方法 ⟨method i-j⟩ 尚不属于 ⟨class name⟩，则令
\csname pgfooY⟨class name⟩.⟨method i-j⟩\endcsname
% 等于控制序列
\csname pgfooY⟨super-class-name-i⟩.⟨method i-j⟩\endcsname
%
% 继承属性
%  \pgfoo@inherit@attributes
%
% 将当前类 ⟨class name⟩ 的属性名称列表保存到控制序列
\csname pgfooY⟨class name⟩.@pgfoo@attributes\endcsname
%
% 定义宏 \pgfoo@attributes 保存 ⟨an attribute name⟩=⟨ 所属的 class name⟩ 的列表
%
% 定义宏 \pgfoo@process@attributes 保存那些定义 ⟨an attribute name⟩ 的命令
%
% 定义控制序列保存那些能够定义各个属性 ⟨an attribute name⟩ 的命令
\csname pgfooY⟨class name⟩ .@pgfoo@process@attributes\endcsname
%
% Always present (and never inherited) methods:
\expandafter\let\csname pgfooY\pgfoo@classname.get handle\endcsname\pgfoo@obj%
\expandafter\let\csname pgfooY\pgfoo@classname.get id\endcsname\pgfoo@id%
\expandafter\let\expandafter\pgfoo@init\csname pgfooY\pgfoo@classname.init\endcsname
\expandafter\let\csname pgfooY\pgfoo@classname.\pgfoo@classname\endcsname\pgfoo@init
% Cleanup
\let\method=\pgfoo@origmethod%
\let\attribute=\pgfoo@origattribute%
```

注意 \pgfooclass 定义的控制序列、宏都不是全局地，因此 \pgfooclass 的作用效果受到 TEX 的限制。

29.12.2 \pgfoonew

29.12.2.1 \⟨*object handle*⟩ 对象

\pgfoonew→P. 537\⟨*object handle*⟩=new ⟨*a class name*⟩(⟨*constructor arguments*⟩) 的处理：

```
\expandafter\ifx\csname pgfooY⟨a class name⟩.get id\endcsname\relax%
  \pgferror{Unknown class '⟨a class name⟩'}%
\else%
%
% 为 \⟨object handle⟩ 计算一个编号
  \advance\pgfoo@objectcount by 1\relax% 这个值是 ⟨object count⟩
%
% 定义 \pgfoolastobj
  \edef\pgfoolastobj{\noexpand\pgfoo@caller{\the\pgfoo@objectcount}}%
%
% 令 \⟨object handle⟩-编号-⟨a class name⟩ 三者关联
% 即“全局地”定义控制序列
  \expandafter\gdef\csname pgfooX⟨object count⟩@class\endcsname{⟨a class name⟩}%
%
% 定义 \⟨object handle⟩
  \let\⟨object handle⟩\pgfoolastobj%
%
  {%
    \pgfoothis@count\pgfoo@objectcount% 这个值是 ⟨this count⟩=⟨object count⟩
%
```

```
% 指定命令 \pgfoo@attribute@op, 用以定义各个 attribute
    \let\pgfoo@attribute@op\pgfoolet
%
% 全局地定义与对象 \⟨object handle⟩ 相关的各个 attribute
% 即属于 ⟨a class name⟩ 的 (包括继承自父类的) attribute
    \csname pgfooY⟨a class name⟩.@pgfoo@process@attributes\endcsname%
% 导致控制序列
%\csname pgfooX⟨this count⟩@⟨an attribute name⟩\endcsname
% 全局地等于
%\csname pgfooY⟨ 所属的 class name⟩@⟨an attribute name⟩\endcsname
  }%
\fi%
%
\expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooY⟨a class name⟩.init
↪  \endcsname%
\ifx\pgfoo@caller@temp\relax%
    % assume that #2 is of form "superclass.method"
  \expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooYinit\endcsname%
  \ifx\pgfoo@caller@temp\relax%
    \pgferror{Object #2 has no method 'init'}%
  \fi%
\fi%
\pgfoothis@count=⟨object count number⟩\relax%
%
% 用 ⟨a class name⟩ 的 init 方法处理参数 ⟨constructor arguments⟩
\pgfoo@caller@temp(彻底展开的 ⟨constructor arguments⟩)
%
\pgfoothis@count⟨this count number⟩\relax%
%
% 在组后收集垃圾
\aftergroup\pgfoogc
```

29.12.2.2 {⟨*attribute*⟩} 对象

\pgfoonew →P.537 {⟨*attribute*⟩}=new ⟨*a class name*⟩(⟨*constructor arguments*⟩) 的处理：

```
\expandafter\ifx\csname pgfooY⟨a class name⟩.get id\endcsname\relax%
  \pgferror{Unknown class '⟨a class name⟩'}%
\else%
%
% 为 object 计算一个编号
  \advance\pgfoo@objectcount by 1\relax% 这个值是 ⟨object count⟩
%
% 定义 \pgfoolastobj
  \edef\pgfoolastobj{\noexpand\pgfoo@caller{\the\pgfoo@objectcount}}%
%
% 令编号 ⟨object count⟩ 与 ⟨a class name⟩ 关联
% 即“全局地”定义控制序列
  \expandafter\gdef\csname pgfooX⟨object count⟩@class\endcsname{⟨a class name⟩}%
%
% 定义 \pgfoo@temp
  \let\pgfoo@temp\pgfoolastobj%
%
  {%
    \pgfoothis@count\pgfoo@objectcount% 这个值是 ⟨this count⟩=⟨object count⟩
%
```

```
% 指定命令 \pgfoo@attribute@op，用以定义各个 attribute
    \let\pgfoo@attribute@op\pgfoolet
%
% 全局地定义与 object 相关的各个 attribute
% 即属于 ⟨a class name⟩ 的（包括继承自父类的）attribute
    \csname pgfooY⟨a class name⟩.@pgfoo@process@attributes\endcsname%
% 导致控制序列
%\csname pgfooX⟨this count⟩@⟨an attribute name⟩\endcsname
% 全局地等于
%\csname pgfooY⟨所属的 class name⟩@⟨an attribute name⟩\endcsname
  }%
\fi%
%
% 令编号 ⟨this count⟩（即计数器值 \the\pgfoothis@count）-⟨attribute⟩-\pgfoo@temp 三者关联
% 即“全局地”定义控制序列
\expandafter\global\expandafter\let\csname pgfooX⟨this count⟩@⟨attribute⟩\endcsname
↪  \pgfoo@temp
% 这个控制序列是一个 object handle
%
\expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooY⟨a class name⟩.init
↪  \endcsname%
\ifx\pgfoo@caller@temp\relax%
    % assume that #2 is of form "superclass.method"
  \expandafter\let\expandafter\pgfoo@caller@temp\csname pgfooYinit\endcsname%
  \ifx\pgfoo@caller@temp\relax%
    \pgferror{Object #2 has no method 'init'}%
  \fi%
\fi%
\pgfoothis@count=⟨object count number⟩\relax%
%
% 用 ⟨a class name⟩ 的 init 方法处理参数 ⟨constructor arguments⟩
\pgfoo@caller@temp(彻底展开的 ⟨constructor arguments⟩)
%
\pgfoothis@count⟨this count number⟩\relax%
%
% 在组后收集垃圾
\aftergroup\pgfoogc
```

29.12.3 用组做限制

命令 \pgfooclass, \method, \attribute, \pgfoonew 可以定义与类、方法、属性、对象有关的诸多宏和命令，其中：

- 保存对象所属的类名称 ⟨*a class name*⟩ 的控制序列

  \csname pgfooX⟨*object count*⟩@class\endcsname

- 与对象相关的属性，即控制序列

  \csname pgfooX⟨*this count*⟩@⟨*an attribute name*⟩\endcsname

- 与某个属性相关的 ⟨*object handle*⟩, 即控制序列

  \csname pgfooX⟨*this count*⟩@⟨*attribute*⟩\endcsname

是全局定义的，命令 \pgfoogc →P. 543 只清理后 2 种控制序列。

如果希望尽量减少以上命令的有效范围，可以在组内执行以上命令，例如：

```
{
  \pgfooclass...
  \pgfooclass...
  \pgfoonew...
  \⟨object handle⟩...
  {
    \pgfoonew...
    \⟨object handle⟩...
  }
  {
    \pgfoonew...
    \pgfoonew...
    \⟨object handle⟩...
  }
}
```

第五部分

库

第三十章　curvilinear 库

```
\usepgflibrary{curvilinear} % LaTeX and plain TeX and pure pgf
\usetikzlibrary{curvilinear} % LaTeX and plain TeX when using TikZ
```

这个程序库利用 Bézier 曲线来定义非线性坐标系，本程序库也用于实现箭头的弯曲效果，即当箭头带有 bend 选项时，箭头会随着路径的弯曲而弯曲。

\pgfsetcurvilinearbeziercurve{⟨*start*⟩}{⟨*first support*⟩}{⟨*second support*⟩}{⟨*end*⟩}

本命令的有效范围受到 TEX 组的限制。为了使用本程序库的其它命令，你需要先使用本命令，本命令的计算结果会被其他命令利用。

本命令的 4 个参数分别是 4 个基本层的点，或者是能给 \pgf@x 和 \pgf@y 赋值的代码。这 4 个点看作是一段控制曲线的控制点，记该控制曲线是 $C(t)$. 本命令会创建一个数据表，这个数据表反应的是曲线长度与参数 t 的对应。

```
\pgfsetcurvilinearbeziercurve
  {\pgfpointorigin}
  {\pgfpoint{1cm}{1cm}}
  {\pgfpoint{2cm}{1cm}}
  {\pgfpoint{3cm}{0cm}}
```

本命令的处理是：

1. 记本命令的 4 个参数，即控制曲线 $C(t)$ 的 4 个控制点依次是 A, B, C, D, 本命令计算：
   (a) $t_a = \frac{1}{|AB|+|BC|+|CD|}$;
   (b) $P = C(t_a)$, $l_a = |P - A|$;
   (c) 如果 $l_a > 1\text{pt}$, 则令：$t_a = \frac{t_a}{l_a}$, $P = C(t_a)$, $l_a = |P - A|$;
   (d) $Q = C(2t_a)$, $l_b = |Q - P| + |P - A|$;
   (e) $R = C(4t_a)$, $l_c = |R - Q| + |Q - P| + |P - A|$;
   (f) $S = C(8t_a)$, $l_d = |S - R| + |R - Q| + |Q - P| + |P - A|$;
   所得的数据表是：

t_a	$2t_a$	$4t_a$	$8t_a$
l_a	l_b	l_c	l_d

   通常有 $0 \leqslant t_a \leqslant 1$, $0 \leqslant l_a \leqslant l_b \leqslant l_c \leqslant l_d$.
   保存计算结果：
   - 宏 \pgf@curvilinear@line@a 保存点 A
   - 宏 \pgf@curvilinear@line@b 保存点 B
   - 宏 \pgf@curvilinear@line@c 保存点 C
   - 宏 \pgf@curvilinear@line@d 保存点 D
   - 尺寸寄存器 \pgf@curvilinear@time@a 保存 $t_a\text{pt}$
   - 尺寸寄存器 \pgf@curvilinear@length@a 保存 $l_a\text{pt}$
   - 尺寸寄存器 \pgf@curvilinear@length@b 保存 $l_b\text{pt}$
   - 尺寸寄存器 \pgf@curvilinear@length@c 保存 $l_c\text{pt}$

- 尺寸寄存器 \pgf@curvilinear@length@d 保存 $l_d$pt

2. 定义命令：

```
\let\pgf@curvilinear@comp@a\pgf@curvilinear@comp@a@initial%
\let\pgf@curvilinear@comp@b\pgf@curvilinear@comp@b@initial%
\let\pgf@curvilinear@comp@c\pgf@curvilinear@comp@c@initial%
\let\pgf@curvilinear@comp@d\pgf@curvilinear@comp@d@initial%
\let\pgf@curvilinear@comp@e\pgf@curvilinear@comp@e@initial%
\let\pgf@curvilinear@point\pgf@curvilinear@curve@point%
```

以上的宏、寄存器、命令不是全局地，所以本命令的结果受到 TEX 组的限制。

\pgf@curvilinear@comp@a@initial

这个命令：

1. 定义命令

```
\let\pgf@curvilinear@comp@a\pgf@curvilinear@comp@a@cont
```

2. 赋值 \pgf@x= $\frac{t_a}{l_a}$\pgf@x

\pgf@curvilinear@comp@b@initial

这个命令：

1. 定义命令

```
\let\pgf@curvilinear@comp@b\pgf@curvilinear@comp@b@cont
```

2. 赋值 \pgf@x= $\frac{t_a}{l_b-l_a}$\pgf@x + $\left(t_a - \frac{t_a l_a}{l_b-l_a}\right)$ pt

\pgf@curvilinear@comp@c@initial

这个命令：

1. 定义命令

```
\let\pgf@curvilinear@comp@c\pgf@curvilinear@comp@c@cont
```

2. 赋值 \pgf@x= $\frac{2t_a}{l_c-l_b}$\pgf@x + $\left(2t_a - \frac{2t_a l_b}{l_c-l_b}\right)$ pt

\pgf@curvilinear@comp@d@initial

这个命令：

1. 定义命令

```
\let\pgf@curvilinear@comp@d\pgf@curvilinear@comp@d@cont
```

2. 赋值 \pgf@x= $\frac{4t_a}{l_d-l_c}$\pgf@x + $\left(4t_a - \frac{4t_a l_c}{l_d-l_c}\right)$ pt

\pgf@curvilinear@comp@e@initial

这个命令：

1. 定义命令

```
\let\pgf@curvilinear@comp@e\pgf@curvilinear@comp@e@cont
```

2. 赋值 \pgf@x= $\frac{8t_a}{l_d}$\pgf@x

\pgfcurvilineardistancetotime{⟨*distance*⟩}

参数 ⟨*distance*⟩ 会被 \pgfmathsetlength 处理，代表一个“行进距离”。

本命令利用 \pgfsetcurvilinearbeziercurve 的计算结果。本命令计算的是：从曲线 $C(t)$ 的起点开始，沿着曲线行进距离为 ⟨*distance*⟩ 时，所对应的参数 t 的值，并将这个值保存在 \pgf@x 中。注

意本命令会在运算速度与精度之间作出权衡。本命令在曲线开端处的精度最好，如果控制曲线的退化程度越高，则精度越差。

本命令的处理是：

1. 执行

```
\pgfmathsetlength{\pgf@x}{⟨distance⟩}%
```

2. 执行一个套嵌的条件分支：
   - 如果 \pgf@x $< l_c$,
     - 如果 \pgf@x $< l_a$, 则执行 \pgf@curvilinear@comp@a
     - 如果 \pgf@x $\geqslant l_a$, 则
       * 如果 \pgf@x $< l_b$, 则执行 \pgf@curvilinear@comp@b
       * 如果 \pgf@x $\geqslant l_b$, 则执行 \pgf@curvilinear@comp@c
   - 如果 \pgf@x $\geqslant l_c$,
     - 如果 \pgf@x $< l_d$, 则执行 \pgf@curvilinear@comp@d
     - 如果 \pgf@x $\geqslant l_d$, 则执行 \pgf@curvilinear@comp@e

   也就是说，假设 ⟨*distance*⟩ 对应的长度是 s, 那么：

长度	执行	结果
$s \in (-\infty, l_a)$	\pgf@curvilinear@comp@a	\pgf@x $= \left(s \cdot \frac{t_a}{l_a}\right)$ pt
$s \in [l_a, l_b)$	\pgf@curvilinear@comp@b	\pgf@x $= \left(s \cdot \frac{t_a}{l_b - l_a} + t_a - l_a \frac{t_a}{l_b - l_a}\right)$ pt
$s \in [l_b, l_c)$	\pgf@curvilinear@comp@c	\pgf@x $= \left(s \cdot \frac{2t_a}{l_c - l_b} + 2t_a - l_b \frac{2t_a}{l_c - l_b}\right)$ pt
$s \in [l_c, l_d)$	\pgf@curvilinear@comp@d	\pgf@x $= \left(s \cdot \frac{4t_a}{l_d - l_c} + 4t_a - l_c \frac{4t_a}{l_d - l_c}\right)$ pt
$s \in [l_d, +\infty)$	\pgf@curvilinear@comp@e	\pgf@x $= \left(s \cdot \frac{8t_a}{l_d}\right)$ pt

\pgfpointcurvilinearbezierorthogonal{⟨*distance*⟩}{⟨*offset expression*⟩}

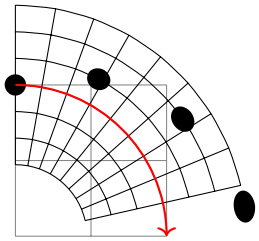
这个命令定义一个非线性变换，其作用受到 TEX 组的限制。

参数 ⟨*distance*⟩ 被 \pgfmathsetlength 处理，处理结果作为一个距离。

参数 ⟨*offset expression*⟩ 被 \pgfmathsetmacro 处理，处理结果作为一个“距离 (默认单位是 pt)”。

先执行 \pgfsetcurvilinearbeziercurve, 再使用本命令。

本命令的作用大致是：从曲线 $C(t)$ 的起点开始，行进距离 ⟨*distance*⟩, 到达点 P; 曲线 $C(t)$ 在点 P 处有法线 l_p (切线方向与法线方向成右手直角系); 自点 P, 再沿着法线 l_p 方向行进距离 ⟨*offset expression*⟩ (如果没有长度单位，就认为是 pt) 到达点 Q, 点 Q 的坐标 (非全局地) 保存在 \pgf@x 与 \pgf@y 中，这就是本命令对参数 ⟨*distance*⟩ 和 ⟨*offset expression*⟩ 的变换结果。



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  {
    \pgfsetcurvilinearbeziercurve{\pgfpoint{0mm}{20mm}}{\pgfpoint{11mm}{20mm}}
                                 {\pgfpoint{20mm}{11mm}}{\pgfpoint{20mm}{0mm}}
    \makeatletter
    \pgftransformnonlinear{\pgfpointcurvilinearbezierorthogonal{\pgf@x}{\pgf@y}}%
    \makeatother
```

```
    \draw (0,-30pt) grid [step=10pt] (80pt,30pt);
    \foreach \i in {0,1,...,3} \fill (\i,\i*10pt) circle [radius=4pt];
  }
  \draw[red,thick,->] (0mm,20mm) .. controls (11mm,20mm) and (20mm,11mm) .. (20mm,0mm);
\end{tikzpicture}
```

本命令的处理是：

1. 处理 ⟨*offset expression*⟩

   ```
   \pgfmathsetmacro\pgf@curvilinear@yfactor{⟨offset expression⟩}
   ```

   即用 \pgfmathparse 解析 ⟨*offset expression*⟩, 解析结果保存到宏 \pgf@curvilinear@yfactor (数值)
2. 计算 ⟨*distance*⟩ 对应的曲线 $C(t)$ 的时刻 t

   ```
   \pgfcurvilineardistancetotime{⟨distance⟩}
   ```

3. 计算时刻 t 对应的曲线 $C(t)$ 上的点 P

   ```
   \pgfpointcurveattime{\pgf@x}{\pgf@curvilinear@line@a}{\pgf@curvilinear@line@b}
   ↪  {\pgf@curvilinear@line@c}{\pgf@curvilinear@line@d}
   ```

   这个命令也给出了曲线 $C(t)$ 在点 P 处的切向量。
4. 检查曲线 $C(t)$ 在点 P 处的切向量的 ∞ 范数，如果这个范数小于 0.0001pt, 就把 $\overrightarrow{AC}$ 用作点 P 处的切向量。如果 $\overrightarrow{AC}$ 的 ∞ 范数还是小于 0.0001pt, 就把 $\overrightarrow{AD}$ 用作点 P 处的切向量。
5. 计算点 P 处的法线方向，使得切线方向与法线方向成右手直角系，并且用 \pgfpointnormalised 计算法线的单位化的方向向量 n_P.
6. 计算 $P+$\pgf@curvilinear@yfactor$\cdot n_P$, 结果 (非全局地) 保存在 \pgf@x 与 \pgf@y 中。

\pgfpointcurvilinearbezierpolar{⟨x⟩}{⟨y⟩}

本命令定义一个非线性变换。

⟨x⟩ 与 ⟨y⟩ 应当是能被 \pgfmathsetlength 处理的参数。

先执行 \pgfsetcurvilinearbeziercurve, 再使用本命令。

此命令的处理是：

1. 用 \pgfmathsetlength 处理参数 ⟨x⟩ 与 ⟨y⟩.
2. 计算单位向量 $e=(e_x\text{pt}, e_y\text{pt})$, $e_x\geqslant 0$, 且与 $(\langle x\rangle, \langle y\rangle)$ 共线。
   如果 $(\langle x\rangle, \langle y\rangle)=(0\text{pt}, 0\text{pt})$, 则令 $e=(1\text{pt}, 0\text{pt})$.
3. 修改 canvas 坐标系的矩阵

   ```
   \pgfsettransformentries%
       {e_x}{e_y}%
       {-e_y}{e_y}{0pt}{0pt}%
   ```

   这是以 $(e_x\text{pt}, e_y\text{pt})$ 为 x 轴的单位向量，以 $(-e_y\text{pt}, e_x\text{pt})$ 为 y 轴的单位向量，构成一个右手直角坐标系。
4. 做平移

   ```
   \pgftransformshift{\pgfpointscale{-1}{\pgf@curvilinear@line@a}}
   ```

   这是把 canvas 坐标系的原点平移到了曲线 $C(t)$ 的第一个控制点处。
5. 计算长度 $d=\frac{\langle x\rangle}{|\langle x\rangle|}\|(\langle x\rangle, \langle y\rangle)\|$, 注意 d 的符号，若 $\langle x\rangle=0$, 则约定 $\frac{\langle x\rangle}{|\langle x\rangle|}=1$.
6. 计算 d 对应的曲线 $C(t)$ 的时刻 t

   ```
   \pgfcurvilineardistancetotime{d}
   ```

7. 计算时刻 t 的曲线点 P, 并对点 P 做线性变换，得到 P'

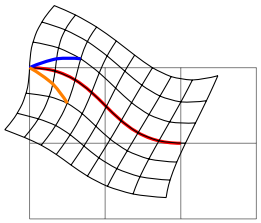
```
\pgfpointtransformed{%
  \pgfpointcurveattime%
    {\pgf@x}%
    {\pgf@curvilinear@line@a}%
    {\pgf@curvilinear@line@b}%
    {\pgf@curvilinear@line@c}%
    {\pgf@curvilinear@line@d}%
}%
```

8. 求和

```
\pgfpointadd{P'}{\pgf@curvilinear@line@a}
```

这是在当前 canvas 坐标系中平移点 P'. 然后把结果全局地保存到 \pgf@x 与 \pgf@y 中。可见参数 $\langle x\rangle$ 与 $\langle y\rangle$ 的作用有 2 点：

- 一是决定基向量：$(\langle x\rangle, \langle y\rangle) \to (e_x\mathrm{pt}, e_y\mathrm{pt})$ 和 $(-e_y\mathrm{pt}, e_x\mathrm{pt})$
- 二是决定距离：$(\langle x\rangle, \langle y\rangle) \to d = \frac{\langle x\rangle}{|\langle x\rangle|}\|(\langle x\rangle, \langle y\rangle)\|$



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw[red, very thick](0mm,20mm) .. controls (10mm,20mm) and (10mm,10mm) .. (20mm,10mm);
  {
    \pgfsetcurvilinearbeziercurve
      {\pgfpoint{0mm}{20mm}}
      {\pgfpoint{10mm}{20mm}}
      {\pgfpoint{10mm}{10mm}}
      {\pgfpoint{20mm}{10mm}}
    \makeatletter
    \pgftransformnonlinear{\pgfpointcurvilinearbezierpolar\pgf@x\pgf@y}%
    \makeatother
    \draw (0,-30pt) grid [step=10pt] (80pt,30pt);
% Add a "barb":
    \draw [blue, very thick] (20pt,10pt) -- (0,0);
    \draw [orange, very thick] (0,0) -- (20pt,-10pt);
  }

\end{tikzpicture}
```

第三十一章　定点算术程序库

```
\usepgflibrary{fixedpointarithmetic} % LaTeX and plain TeX and pure pgf
\usepgflibrary[fixedpointarithmetic] % ConTeXt and pure pgf
\usetikzlibrary{fixedpointarithmetic} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[fixedpointarithmetic] % ConTeXt when using TikZ
```

LaTeX 的宏包 fp 提供了较强的定点算术功能，定点算术程序库提供了使用该宏包的一个“接口”。首先调用宏包 fp,

```
\usepackage{fp}
```

然后再载入定点算术程序库 `fixedpointarithmetic`, 否则导致错误。

31.1　Overview

PGF 的数学引擎在解析表达式时，有较快的速度和弹性，但是精度有时较低。受到 TeX 的计算能力的限制，数学引擎能计算的数据范围不超过 ±16383.99999. 而宏包 fp 提供了更高的精度和更广的数据计算范围（大约是 $\pm 9.999 \times 10^{17}$），但是它工作起来较慢并缺少弹性。

本程序库将宏包 fp 的计算精度与 PGF 数学引擎的弹性结合了起来。使用本程序库时，注意以下几点：

- 宏包 fp 支持很大的数值计算，例如，`2^20` 或 `1.2e10+3.4e10`，但是 PGF 和 Ti*k*Z 并不支持很大的数值，故宏包 fp 所计算的大数值不能直接用于绘图中，需要做一下变通。
- 长度，例如，`10pt`，`10pt+2mm`，仍然由 PGF 的数学引擎来处理，故关于长度的计算仍然不能超过 ±16383.99999 pt. 所以，表达式 `3*10000cm` 不能接受，但 `3cm*10000` 却可以接受。
- 在关于数学引擎的介绍中介绍了许多函数，使用本程序库后，其中多数函数都会被转换为 fp 版本的函数，具备 fp 的计算能力，但其行为可能与不使用本程序库时的行为有所不同，具体参考宏包 fp 的说明文档。
- 在 PGF 中，三角函数，例如，`sin`，`cos`，其参数默认为角度制下的数值；反三角函数，例如，`asin`，`acos`，其函数值也是默认为角度制下的数值。尽管宏包 fp 总是使用弧度制，不过 PGF 会自动做好相应的转换，以维持 PGF 偏好角度制的习惯。
- 使用本程序库后，总体上 PGF 的数学运算变慢了。为了利用宏包 fp 的精度，不得不牺牲一点速度。

31.2　在 PGF 和 TikZ 中使用定点算术

通过使用以下 key，可以在 PGF 和 Ti*k*Z 中使用宏包 fp.

`/pgf/fixed point arithmetic=`⟨*options*⟩　(no default)

/tikz/fixed point arithmetic=⟨*options*⟩ (no default)

这个选项会把 ⟨*options*⟩ 中的选项冠以前缀 /pgf/fixed point/ 来执行。本选项最好用作环境选项，例如，用作环境 {tikzpicture}，{pgfpicture}，{scope} 的选项. 当本选项用作环境选项时，就限制在该环境中使用宏包 fp 来执行计算，在该环境之外，PGF 的数学引擎恢复到原本的行为状态，这样能节省一些时间。

本选项的定义是：

```
\pgfkeys{/pgf/.cd,
    fixed point arithmetic/.code={%
        \pgfmathfp@plots@install%
        \pgfmathfp@parser@install%
        \let\pgfmathparse=\pgfmathfpparse%
        \pgfkeys{/pgf/fixed point/.cd, #1}%
    },%
}
```

其中的 \pgfmathfp@plots@install 可以改造 \pgfplotxyfile, \pgfplotxyzfile, 使之能利用 fp 宏包的命令来处理文件中的数据。而 \pgfmathfp@parser@install 则可以把 PGF 的“私人版”数学函数改造为 fp 版本，例如，

```
\let\pgfmathadd@=\pgfmathfpadd@%
```

\pgfmathfpparse 可以配合选项 /pgf/fixed point/scale results.

\pgfmathfpparse{⟨*expression*⟩}

在适当的情形下,本命令利用 fp 版本的 PGF 数学函数来解析 ⟨*expression*⟩, 将结果保存在 \pgfmathresult.

本命令会检查参数 ⟨*expression*⟩ 是否以星号 * 开头：

- 如果是，就

  ```
  \pgfmathparse@{\pgfmathfpscale⟨expression⟩}
  ```

  例如 \pgfmathfpparse{*99999-88888} 导致

  ```
  \pgfmathparse@{\pgfmathfpscale*99999-88888}
  ```

  其中的 \pgfmathparse@ 应该能调用 fp 版本的 PGF 数学函数来解析 ⟨*expression*⟩.

  注意，只有 ⟨*expression*⟩ 的第一个因式与 \pgfmathfpscale 相乘。

- 如果不是，就

  ```
  \pgfmathparse@{⟨expression⟩}
  ```

\pgfmathfpscale

这个宏的初始值是 1.

目前，在 ⟨*options*⟩ 中能使用的选项很少，列举如下：

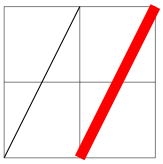
/pgf/fixed point/scale results=⟨*factor* ⟩ (no default)

本选项重定义 \pgfmathfpscale 的值。

本选项的定义是：

```
\pgfkeys{/pgf/fixed point/.cd,
    scale results/.code={%
        \pgfmathfpparse{#1}%
        \let\pgfmathfpscale=\pgfmathresult%
    }%
}%
```

看一个例子：



```
\tikz[fixed point arithmetic={scale results=10^-6}]{
  \draw [help lines] grid (2,2);
  \draw (0,0) -- (1,2);
  \draw [red, line width=4pt] (*1.0e6,0) -- (*2.0e6,*2.0e6);}
```

在上面的代码中，环境选项中用了 `fixed point arithmetic={scale results=10^-6}`，表示在该环境中使用宏包 fp，并且设置一个比例因子 10^{-6}. 在绘图命令中用了很大的坐标数值，每个大数值都前缀一个星号“*”，这个星号会引导程序使用宏包 fp 来处理这个大数值，即给大数值乘上前面选项设置的比例因子 10^{-6}，从而把大数值变通为普通的 PGF 数学引擎能处理的“小数值”。如果一个数值前面不带星号“*”，就不会使用比例因子 10^{-6} 来处理该数值，结果将超出普通的 PGF 数学引擎的处理范围。

有时候会把数据点保存在一个外部文件中，绘图时调用该文件，将文件中的数据点变成可视的图形，例如 `plot[`⟨*options*⟩`]file{`⟨*file name*⟩`}`, 以及数据可视化命令使用的外部文件。在编辑包含数据点的外部文件时，可以给大数值前面带上星号“*”，然后在绘图环境的选项中，或在绘图命令的选项中调用宏包 fp 来处理大数值。如果手工给大数值添加前缀“*”太麻烦，还可以使用下面的选项：

`/pgf/fixed point/scale file plot x=`⟨*factor*⟩ (no default)

本选项将外部数据文件的第一列数据 (对应 x 轴的数据) 乘上比例因子 ⟨*factor*⟩, 并用 fp 版本的 PGF 数学函数来计算乘积，乘积应当能被普通的 PGF 数学引擎的处理。

本选项不依赖选项 `scale results`. 本选项的定义是：

```
\pgfkeys{/pgf/fixed point/.cd,
  scale file plot x/.code=\pgfmathfpparse{#1}\edef\pgfmathfpplotscalex{
  ↪  \pgfmathresult*},
  scale file plot y/.code=\pgfmathfpparse{#1}\edef\pgfmathfpplotscaley{
  ↪  \pgfmathresult*},
  scale file plot z/.code=\pgfmathfpparse{#1}\edef\pgfmathfpplotscalez{
  ↪  \pgfmathresult*},
}%
```

`/pgf/fixed point/scale file plot y=`⟨*factor*⟩ (no default)

本选项将外部数据文件的第二列数据乘上比例因子 ⟨*factor*⟩, 并用 fp 版本的 PGF 数学函数来计算乘积，乘积应当能被普通的 PGF 数学引擎的处理。

`/pgf/fixed point/scale file plot z=`⟨*factor*⟩ (no default)

本选项将外部数据文件的第三列数据乘上比例因子 ⟨*factor*⟩, 并用 fp 版本的 PGF 数学函数来计算乘积，乘积应当能被普通的 PGF 数学引擎的处理。

31.3 关于 fp 宏包

在使用 `fixedpointarithmetic` 库的情况下，PGF 的数学函数将使用 fp 宏包的命令来计算，例如，

```
\let\pgfmathadd@=\pgfmathfpadd@%
\def\pgfmathfpadd#1#2{%
    \pgfmathfpparse{#2}\let\pgfmath@result=\pgfmathresult%
    \pgfmathfpparse{#1}%
    \pgfmathfpadd@{\pgfmathresult}{\pgfmath@result}%
}%
\def\pgfmathfpadd@#1#2{%
    \begingroup%
        \FPadd\pgfmathresult{#1}{#2}%
        \pgfmath@smuggleone\pgfmathresult%
```

```
    \endgroup%
}%
```

也就是说，“公共版”函数 \pgfmathadd 调用的“个人版”命令 \pgfmathadd@ 会等于 \pgfmathfpadd@，而 \pgfmathfpadd 则是 fp 版本的函数。上面代码中的 \FPadd 是 fp 宏包的函数。

下面是 fp 宏包手册的内容。

\usepackage[⟨*options*⟩]fp

宏包选项 ⟨*options*⟩ 可以是：

- nomessages, 只是计算函数值，不打印关于函数的信息
- debug, 打印 debug messages (mainly for \FPupn).

\FPset\⟨*macro*⟩{⟨*something*⟩}

将 ⟨*something*⟩ 保存到 \⟨*macro*⟩ 中，保存前会先把 ⟨*something*⟩ 中的宏展开。

3+3+abc-5

```
\newcount\cccc
\cccc5
\def\aaaa{3}
\FPset\bbbb{\aaaa+3+abc-\the\cccc}
\bbbb
```

\FPprint\⟨*macro*⟩

打印保存在 \⟨*macro*⟩ 中的内容。

3+5+6

```
\def\aaaa{3+5+6}
\FPprint\aaaa\par
\FPprint{abc}
```

31.3.1 基本算术运算

\FPadd\⟨*macro*⟩{⟨*A*⟩}{⟨*B*⟩}

将 ⟨*A*⟩ + ⟨*B*⟩ 保存到 \⟨*macro*⟩ 中

\FPdiv\⟨*macro*⟩{⟨*A*⟩}{⟨*B*⟩}

将 ⟨*A*⟩ / ⟨*B*⟩ 保存到 \⟨*macro*⟩ 中

\FPmul\⟨*macro*⟩{⟨*A*⟩}{⟨*B*⟩}

将 ⟨*A*⟩ · ⟨*B*⟩ 保存到 \⟨*macro*⟩ 中

\FPsub\⟨*macro*⟩{⟨*A*⟩}{⟨*B*⟩}

将 ⟨*A*⟩ − ⟨*B*⟩ 保存到 \⟨*macro*⟩ 中

\FPabs\⟨*macro*⟩{⟨*A*⟩}

将 ⟨*A*⟩ 的绝对值保存到 \⟨*macro*⟩ 中

\FPneg\⟨*macro*⟩{⟨*A*⟩}

将 ⟨*A*⟩ 的相反数保存到 \⟨*macro*⟩ 中

\FPsgn\⟨*macro*⟩{⟨*A*⟩}

将 ⟨*A*⟩ 的符号 (可能是 0, −1, 1) 保存到 \⟨*macro*⟩ 中

\FPmin\⟨*macro*⟩{⟨*A*⟩}{⟨*B*⟩}

将 ⟨*A*⟩ 与 ⟨*B*⟩ 中的较小者保存到 \⟨*macro*⟩ 中

\FPmax\⟨*macro*⟩{⟨*A*⟩}{⟨*B*⟩}

将 ⟨*A*⟩ 与 ⟨*B*⟩ 中的较大者保存到 \⟨*macro*⟩ 中

31.3.2 基本的比较运算

\FPiflt{⟨A⟩}{⟨B⟩} ⟨true code⟩ \else ⟨false code⟩ \fi
若 ⟨A⟩ 小于 ⟨B⟩, 则执行 ⟨true code⟩, 否则执行 ⟨false code⟩

\FPifeq{⟨A⟩}{⟨B⟩} ⟨true code⟩ \else ⟨false code⟩ \fi
若 ⟨A⟩ 等于 ⟨B⟩, 则执行 ⟨true code⟩, 否则执行 ⟨false code⟩

\FPifgt{⟨A⟩}{⟨B⟩} ⟨true code⟩ \else ⟨false code⟩ \fi
若 ⟨A⟩ 大于 ⟨B⟩, 则执行 ⟨true code⟩, 否则执行 ⟨false code⟩

\FPifneg{⟨A⟩} ⟨true code⟩ \else ⟨false code⟩ \fi
若 ⟨A⟩ 是负数，则执行 ⟨true code⟩, 否则执行 ⟨false code⟩

\FPifpos{⟨A⟩} ⟨true code⟩ \else ⟨false code⟩ \fi
若 ⟨A⟩ 是正数，则执行 ⟨true code⟩, 否则执行 ⟨false code⟩

\FPifzero{⟨A⟩} ⟨true code⟩ \else ⟨false code⟩ \fi
若 ⟨A⟩ 是 0，则执行 ⟨true code⟩, 否则执行 ⟨false code⟩

\FPifint{⟨A⟩} ⟨true code⟩ \else ⟨false code⟩ \fi
若 ⟨A⟩ 是整数，则执行 ⟨true code⟩, 否则执行 ⟨false code⟩

\ifFPtest⟨true code⟩ \else ⟨false code⟩ \fi
若之前的一个真值测试为真, 则执行 ⟨true code⟩, 否则执行 ⟨false code⟩

```
\FPiflt{25}{30} y \else n \fi\par
\ifFPtest 是 \else 否 \fi\par
```

y
是

31.3.3 幂、对数

\FPe
这个宏保存数值 2.718281828459045235, 作为自然对数底 e

\FPexp\⟨macro⟩{⟨x⟩}
将 $\exp(\langle x\rangle)$ 保存到宏 \⟨macro⟩ 中

\FPln\⟨macro⟩{⟨x⟩}
将 $\ln(\langle x\rangle)$ 保存到宏 \⟨macro⟩ 中

\FPpow\⟨macro⟩{⟨x⟩}{⟨y⟩}
将 $\langle x\rangle^{\langle y\rangle}$ 保存到宏 \⟨macro⟩ 中

\FProot\⟨macro⟩{⟨x⟩}{⟨y⟩}
将 $\langle x\rangle^{1/\langle y\rangle}$ 保存到宏 \⟨macro⟩ 中

```
\FProot\aaaa{4}{2}\aaaa
```

1.999999999999999990

31.3.4 三角函数运算

注意，以下的三角函数只接受带有小数点的数，不接受整数。

\FPpi

代表圆周率，3.141592653589793238

\FPsin\⟨*macro*⟩{⟨*A*⟩}

将 sin(⟨*A*⟩) 保存到 \⟨*macro*⟩ 中

\FPcos\⟨*macro*⟩{⟨*A*⟩}

将 cos(⟨*A*⟩) 保存到 \⟨*macro*⟩ 中

\FPsincos\⟨*macro 1*⟩\⟨*macro 2*⟩{⟨*A*⟩}

将 sin(⟨*A*⟩) 保存到 \⟨*macro 1*⟩ 中，将 cos(⟨*A*⟩) 保存到 \⟨*macro 2*⟩ 中

\FPtan\⟨*macro*⟩{⟨*A*⟩}

将 tan(⟨*A*⟩) 保存到 \⟨*macro*⟩ 中

\FPcot\⟨*macro*⟩{⟨*A*⟩}

将 cot(⟨*A*⟩) 保存到 \⟨*macro*⟩ 中

\FPtancot\⟨*macro 1*⟩\⟨*macro 2*⟩{⟨*A*⟩}

将 tan(⟨*A*⟩) 保存到 \⟨*macro 1*⟩ 中，将 cot(⟨*A*⟩) 保存到 \⟨*macro 2*⟩ 中

\FParcsin\⟨*macro*⟩{⟨*A*⟩}

将 arcsin(⟨*A*⟩) 保存到 \⟨*macro*⟩ 中

\FParccos\⟨*macro*⟩{⟨*A*⟩}

将 arccos(⟨*A*⟩) 保存到 \⟨*macro*⟩ 中

\FParcsincos\⟨*macro 1*⟩\⟨*macro 2*⟩{⟨*A*⟩}

将 arcsin(⟨*A*⟩) 保存到 \⟨*macro 1*⟩ 中，将 arccos(⟨*A*⟩) 保存到 \⟨*macro 2*⟩ 中

\FParctan\⟨*macro*⟩{⟨*A*⟩}

将 arctan(⟨*A*⟩) 保存到 \⟨*macro*⟩ 中

\FParccot\⟨*macro*⟩{⟨*A*⟩}

将 arccot(⟨*A*⟩) 保存到 \⟨*macro*⟩ 中

\FParctancot\⟨*macro 1*⟩\⟨*macro 2*⟩{⟨*A*⟩}

将 arctan(⟨*A*⟩) 保存到 \⟨*macro 1*⟩ 中，将 arccot(⟨*A*⟩) 保存到 \⟨*macro 2*⟩ 中

31.3.5 关于解代数方程

\FPlsolve\⟨*macro*⟩{⟨*A*⟩}{⟨*B*⟩}

将 $\langle A\rangle x + \langle B\rangle = 0$ 的解保存到 \⟨*macro*⟩ 中。如果 $\langle A\rangle = 0, \langle B\rangle \neq 0$, 则报错。

\FPqsolve\⟨*macro 1*⟩\⟨*macro 2*⟩{⟨*A*⟩}{⟨*B*⟩}{⟨*C*⟩}

将 $\langle A\rangle x^2 + \langle B\rangle x + \langle C\rangle = 0$ 的解保存到 \⟨*macro 1*⟩, \⟨*macro 2*⟩ 中。

如果 $\langle A\rangle = 0$, 则解方程 $\langle B\rangle x + \langle C\rangle = 0$.

如果 $\langle B\rangle^2 - 4\langle A\rangle\langle C\rangle < 0$, 则报错。

对于方程 $ax^2 + bx + c = 0$, 解是：\⟨*macro 1*⟩$= \frac{-b+\sqrt{b^2-4ac}}{2a}$, \⟨*macro 2*⟩$= \frac{-b-\sqrt{b^2-4ac}}{2a}$.

\FPcsolve\⟨macro 1⟩\⟨macro 2⟩\⟨macro 3⟩{⟨A⟩}{⟨B⟩}{⟨C⟩}{⟨D⟩}

将 $\langle A\rangle x^3+\langle B\rangle x^2+\langle C\rangle x+\langle D\rangle=0$ 的解保存到 \⟨macro 1⟩, \⟨macro 2⟩, \⟨macro 3⟩ 中。
如果三次方程只有一个实根，则 \⟨macro 1⟩=\⟨macro 2⟩=\⟨macro 3⟩.

\FPqqsolve\⟨macro 1⟩\⟨macro 2⟩\⟨macro 3⟩\⟨macro 4⟩{⟨A⟩}{⟨B⟩}{⟨C⟩}{⟨D⟩}{⟨E⟩}

将 $\langle A\rangle x^4+\langle B\rangle x^3+\langle C\rangle x^2+\langle D\rangle x+\langle E\rangle=0$ 的解保存到 \⟨macro 1⟩, \⟨macro 2⟩, \⟨macro 3⟩, \⟨macro 4⟩ 中。
如果四次方程没有实根，则报错。
如果四次方程只有一个实根，则 \⟨macro 1⟩=\⟨macro 2⟩=\⟨macro 3⟩=\⟨macro 4⟩.

31.3.6 随机数

\FPseed

这是个计数器名称，用作计算随机数的种子。

```
\newcount\FPseed
```

\FPrandom\⟨macro⟩

0.2184182969939048 85

```
\FPrandom\aaaa
\aaaa
```

31.3.7 表达式的值

\FPeval⟨toks⟩{⟨expression⟩}

将表达式 ⟨expression⟩ 的值保存到 ⟨toks⟩ 中。

- 参数 ⟨toks⟩ 可以是以反斜线“\”开头的宏的形式，也可以是一串符号，这一串符号会被做成一个控制序列，保存计算结果。
- 如果要在表达式 ⟨expression⟩ 中使用保存数值的宏，那么可以使用 \⟨macroname⟩, 也可以写成 \⟨macroname⟩{}, 或者只写出宏名称 ⟨macroname⟩, 例如：

18.000000000000000000

```
\def\k{5}
\def\m{12}
\FPeval{result}{1+m+\k}
\result
```

但是：

+2000000
13.000000000000000000

```
\def\k{3+2}
\def\x{12}
\FPeval{result}{1+x+\k}
\result
```

上面代码中 \k 保存一个运算式，导致的结果有点意外。

- 在表达式 ⟨expression⟩ 中，符号“-”不能被识别，因此要把“-2”写成“neg(2)”, 把“-x”写成“neg(x)”

-11.000000000000000000

```
\def\x{12}
\FPeval{result}{1+neg(x)}
\result
```

- 命令 \FPeval 得到的结果是带小数点的。

可用在表达式 ⟨expression⟩ 中的函数有：

- +
- -
- *
- /
- abs(#1)
- neg(#1)
- pow(#1,#2), 计算 #2 的 #1 次幂
- root(#1,#2)
- exp(#1)
- ln(#1)
- min(#1,#2)
- max(#1,#2)
- e, e = 2.718281828459045235
- pi, π = 3.141592653589793238
- round(#1:#2), 按照精度 #2 对 #1 做舍入
- trunc(#1:#2), 按照精度 #2 对 #1 做截断
- clip(#1), 将 #1 的“尾巴”中的符号“0”去掉
- sin(#1)
- cos(#1)
- tan(#1)
- cot(#1)
- arcsin(#1)
- arccos(#1)
- arctan(#1)
- arccot(#1)

```
\FPeval{result}{pow(2,3)}% 3 的 2 次方
\result
```

8.999999999999999733

```
\FPeval{result}{round(3.1415926,2)}\result
```

3.14

```
\FPeval{result}{3.14}\result\par
\FPeval{result}{3.14+0.}\result\par
\FPeval{result}{clip(3.14+0.)}\result
```

3.14
3.140000000000000000
3.14

参数 ⟨*toks*⟩ 可以是这种形式：

```
% 斐波那契数列
%\usepackage{pgfplotstable}
\FPeval{a1}{round(1:0)}
\FPeval{a2}{round(1:0)}
\pgfplotsforeachungrouped \i in {3,...,9}
{
    \FPeval{m}{round(\i+neg(1):0)}
    \FPeval{n}{round(\i+neg(2),0)}
    \FPeval{a\i}{round(a\m+a\n,0)}
    \csname a\i\endcsname\par
}
```

2
3
5
8
13
21
34

\FPupn⟨*toks*⟩{⟨*upn-expression*⟩}

此命令与 \FPeval 类似，不过参数 ⟨*upn-expression*⟩ 用于构造一个“栈”。在 ⟨*upn-expression*⟩ 中列举的可以是数值、保存数值的宏、运算符，相邻的列举项目之间用空格分隔，其中可用的运算有：
+, add, -, sub, *, mul, /, div, abs, neg, sgn, min, max, round, trunc, clip, e(常数), exp, ln, pow, root, seed(种子), random, pi(常数), sin, cos, sincos, tan, cot, tancot, arcsin, arccos, arcsincos, arctan, arccot, arctancot, pop, swap, copy.
解析 ⟨*upn-expression*⟩ 所得到的数值依次保存到 \FP@stack 中，把 \FP@stack 看作一个栈。

\FP@stack

这个宏 (临时地) 保存解析 ⟨*upn-expression*⟩ 时得到的数值。

```
\edef\FP@stack{(⟨ 解析 ⟨upn-expression⟩ 时得到的数值⟩)}
```

例如解析 {1 2 3} 后，\FP@stack 就是 (3,2,1)，\FP@stack 的左侧是栈的“顶端”，即数值 3 位于“顶端”。
当遇到 ⟨*upn-expression*⟩ 中的运算符号时，就对保存在 \FP@stack 中的数值做运算，运算结果还保存到 \FP@stack 中。例如

- 解析 {1 2 3 mul} 得到的是 (mul(2,3),1)，即 (6,1)
- 解析 {1 2 3 div} 得到的是 (div(2,3),1)，即 (2/3,1)
- 解析 {1 2 3 sub} 得到的是 (sub(2,3),1)，即 (-1,1)

运算符号 pop, swap, copy 的意义是：
▶ pop, 删除 \FP@stack 顶端的那一个数值，将那个数值保存到 \FP@vala 中
▶ swap, 交换 \FP@stack 顶端两个数值的位置,此时栈的顶端数值是 \FP@valb, 其次是 \FP@vala
▶ copy, 复制 \FP@stack 顶端的那一个数值，并放在顶端，此时栈的顶端数值是 \FP@vala
所以，如果把 ⟨*upn-expression*⟩ 看作一个类似“栈”的处理过程的话，那么其特点是：

1. 从左向右处理 ⟨*upn-expression*⟩ 的各个项目
2. 数值被按处理次序从左向右放置
3. 数值被直接读取
4. 运算符号针对其左侧的数值进行计算，吃掉被运算的数值，然后放置其计算结果
5. 运算符号 pop 删除的是：pop 左侧的那一个数值
6. 运算符号 swap 交换的是：swap 左侧的两个数值的位置
7. 运算符号 copy 复制的是：copy 左侧的那一个数值，并放置复制结果

\FP@getstack\⟨*macro*⟩

删除 \FP@stack 顶端的那一个元素，将那个元素保存到 \⟨*macro*⟩ 中

\FP@putstack\⟨*macro*⟩

将 \⟨*macro*⟩ 中保存的值放到 \FP@stack 的顶端

\FP@put\⟨*macro*⟩\@upnend

给出 debug 信息，将 \⟨*macro*⟩ 中保存的值放到 \FP@stack 的顶端

```
%put value to stack
\def\FP@put#1\@upnend{%
  \FP@upnc@result{#1}%
  \FP@putstack{#1}%
}
```

解析 ⟨*upn-expression*⟩ 最终得到一个数值，将此数值保存到 ⟨*toks*⟩ 中。

第三十二章　浮点单元程序库

```
\usepgflibrary{fpu} % LaTeX and plain TeX and pure pgf
\usepgflibrary[fpu] % ConTeXt and pure pgf
\usetikzlibrary{fpu} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[fpu] % ConTeXt when using TikZ
```

Floating Point Unit （fpu）程序库能够为 PGF 的科学计算提供足够大的数值范围，其核心是 PGF 中处理尾数的数学程序，并在精度与速度之间实现某种平衡。本程序库不需要第三方宏包或外部程序的支持。

32.1　Overview

fpu 能提供足够大的数值范围以及合理的数据精度。它至少能提供 IEEE 标准下的双精度数值范围，即从 $-1\cdot10^{324}$ 到 $1\cdot10^{324}$. 大于 0 的最小数值是 $1\cdot10^{-324}$. fpu 的相对精度至少是 $1\cdot10^{-4}$，而且对于像加法那样的运算，相对精度是 $1\cdot10^{-6}$.

注意本程序库还没有与绘图命令一起进行测试，fpu 的计算结果应当转换到通常情况下 PGF 能接受的数值范围内，即 ±16383.99999，TEX 允许的数值范围，然后再用于绘图，PGF 可以自己做到这一点。

首先注意本程序库所使用的表达浮点数的格式。fpu 使用一种低层次的表数格式，该格式包括：标记、尾数、幂指数、分隔符号，例如，`1Y2.1e4]`，有的命令会把这种格式的浮点数保存在 `\pgfmathresult` 中，有的命令只接受这种格式的浮点数作为其参数。可以为命令手工编写这种格式的参数。具体说明如下：

- 标记是个数字符号，标记 0 代表数值 $\pm0\cdot10^0$；标记 1 代表正号；标记 2 代表负号；标记 3 代表“非数”；标记 4 代表 $+\infty$；标记 5 代表 $-\infty$.
- 尾数是 1 到 10 之间的实数，而且总是带有小数点，至少有一个小数数字。
- 幂指数是一个整数。
- 标记与尾数之间用一个大写字母“Y”分隔。
- 尾数之后是小写字母“e”，代表单词 exponent.
- 小写字母“e”之后是幂指数。
- 幂指数之后是右方括号“]”，表示数值表达格式的结束。

后文把 fpu 程序库使用的表达浮点数的格式简称为“浮点数格式”，所提到的浮点数都是这种格式的。可以把浮点数格式的数值转换为适合于 PGF 处理的格式，fpu 提供了相应的转换办法。

32.2　用法

使用 fpu 程序库的方式一般有两种，一种方式是使用程序库的命令，另一种方式是通过选项设置，将 fpu 程序库的作用植入其它程序库的命令或者绘图命令中。这里介绍第 2 种方式。

/pgf/fpu={⟨*boolean*⟩} (default true)

这个选项决定是否在当前分组中启用 fpu 的功能。若启用，那些标准的 PGF 数学解析过程 (routine) 会被换成 fpu 版本。

选项 fpu 或 fpu=true 的处理是：

1. 检查 \ifpgfmathfloatparseactive → P. 594 的真值，如果是 true, 则表明已经载入 fpu 库的函数，此时什么也不做；如果是 false, 则表明尚未载入 fpu 库的函数，此时
   (a) 检查命令 \pgfmathdeclarefunction 是否已定义
   - 如果未定义，则执行

     ```
     \pgfmathfloat@parser@install@pgf@two@null@null
     ```

   - 如果已定义，则执行

     ```
     \pgfmathfloat@parser@install
     ```

   (b) 设置真值 \pgfmathfloatparseactivetrue.
   (c) 则执行

   ```
   \pgfkeysalso{/pgf/fixed point arithmetic/.prefix style={/pgf/fpu=false}}
   ```

注意：

- 解析表达式的命令 \pgfmathparse 将等于 \pgfmathfloatparse,在文件《pgflibraryfpu.code.tex》中有

  \pgfmathfloat@install\pgfmathparse=\pgfmathfloatparse

  此时被解析的表达式仍可以是通常的 (不使用 fpu 库时的) 形式，也可以是浮点数形式，但表达式的解析结果——通常是一个不带单位的浮点数——仍然保存在 \pgfmathresult 中。

  1Y2.0e0]
  ```
  \pgfkeys{/pgf/fpu}
  \pgfmathparse{1+1} \pgfmathresult
  ```

  注意例子中的命令没有用在绘图环境中。

- 标准的 PGF 数学函数会具有 fpu 的计算能力（被转成了 fpu 版本），PGF 的“公共版”数学函数 \pgfmath⟨*name*⟩ 可以接受通常的表达式，也可以接受浮点数，但输出的是浮点数。PGF 的“私人版”数学函数将只接受浮点数，例如函数 \pgfmathadd@ 将等于 \pgfmathfloatadd@，文件《pgflibraryfpu.code.tex》中有

  \pgfmathfloat@install\pgfmathadd@=\pgfmathfloatadd@

  fpu 的“公共版”数学函数 \pgfmathfloat⟨*name*⟩，“私人版”数学函数 \pgfmathfloat⟨*name*⟩@ 基本上都只接受浮点数，把定点数传递给 fpu 的数学函数前，要先转换为浮点数，例如

  1Y5.59931293e9]
  ```
  \makeatletter
  \pgfkeys{/pgf/fpu,}%
  \pgfmathsetmacro{\aaa}{98989}% 保存在 \aaa 中的是浮点数
  \pgfmathsetmacro{\bbb}{56565}% 保存在 \bbb 中的是浮点数
  \pgfmathmultiply@{\aaa}{\bbb}
  \pgfmathresult
  \makeatother
  ```

  所以，如果不确定被运算的数值是否已经 (或是否能够) 被转换为浮点数，那么就使用 PGF 的“公共版”数学函数。如果遇到这样的错误：

  ! Package PGF Math Error: Sorry, an internal routine of the floating point unit got an ill-formatted floating point number ‘000.0’. The unreadable part was near ’000.0’..

  那么可以尝试使用 PGF 的“公共版”数学函数。

- 在设置 /pgf/fpu=true 的情况下，有的数学引擎命令、基本层命令会发生改变，例如数学引擎

的命令 \pgfmathsetlength, 这个命令先用 \pgfmathparse 解析一个表达式，再将解析结果加上 pt 赋予某个尺寸寄存器。在 \pgfmathparse 等于 \pgfmathfloatparse 的情况下，解析表达式的结果是浮点数格式，而浮点数格式不能作为尺寸寄存器的值，因此会导致错误：

```
! Illegal unit of measure (pt inserted).
```

基本层的坐标点命令 \pgfpoint 的定义中使用了 \pgfmathsetlength, 因此也会出现这种问题。不少基本层的命令都有这种问题，而命令 \pgfqpoint, \pgfqpointxy, \pgfqpointxyz 就没有这种问题 (这种 quick 版的命令直接给寄存器赋值，不利用 \pgfmathparse)。

但 \pgfqpointpolar 会出问题,因为这个命令的定义中使用了函数 \pgfmathcos@, \pgfmathsin@, 在 /pgf/fpu=true 的情况下，这两个函数都只接受浮点数。

用 fpu=false 取消 fpu 的作用，恢复到 PGF 原来的状态。注意 fpu 的作用受到 TEX 组的限制，故若只是在组内使用 fpu，无需使用 fpu=false.

选项 fpu=false 的处理是：检查 \ifpgfmathfloatparseactive 的真值，如果为 true, 则

1. 执行 \pgfmathfloat@uninstall → P. 596
2. 设置真值 \pgfmathfloatparseactivefalse

注意：如果先启用了 fpu，而后又启用定点算术程序库 fixed point arithmetics，那么 fpu 会被自动取消。

/pgf/fpu/output format=float|sci|fixed (no default, initially float)

这个选项设置保存在 \pgfmathresult 中的数值格式。

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=float}
\pgfmathparse{exp(50)*42} \pgfmathresult
```

1Y2.17765411e23]

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=sci}
\pgfmathparse{exp(50)*42} \pgfmathresult
```

2.17765411e23

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=fixed}
\pgfmathparse{exp(50)*42} \pgfmathresult
```

217765411000000000000000.0

注意 fixed 格式会被选项 scale results={⟨*scale*⟩} 强制选定。

/pgf/fpu/scale results={⟨*scale*⟩} (no default)

本选项的作用有两个：(1) 计算过程、计算结果都采用 fixed 数值格式，便于 PGF 对计算结果做进一步的处理；(2) 以星号“*”为前缀的表达式都会乘上因子 ⟨*scale*⟩.

/pgf/fpu/scale file plot x={⟨*scale*⟩} (no default)

/pgf/fpu/scale file plot y={⟨*scale*⟩} (no default)

/pgf/fpu/scale file plot z={⟨*scale*⟩} (no default)

这几个选项针对外部的数据文件，与程序库 fixed point arithmetics 中的相应选项类似。

\pgflibraryfpuifactive{⟨*true-code*⟩}{⟨*false-code*⟩}

如果已经启用 fpu，则执行 ⟨*true-code*⟩; 如果没有启用 fpu 或者它已经被取消，则执行 ⟨*false-code*⟩.

```
\pgflibraryfpuifactive
  {\tikz\draw circle(5mm);}
  {\tikz\fill circle(5mm);}
```

下面的例子说明选项 fpu 的作用受到 TEX 分组的限制：

开启；关闭
```
{\pgfkeys{/pgf/fpu}
\pgflibraryfpuifactive {开启}{关闭}};
\pgflibraryfpuifactive {开启}{关闭}
```

下面的代码：

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=fixed}
\begin{tikzpicture}[x=1/10000 cm, y=1/10000 cm]
  \draw [red] (0,0)--(10000,20000);
\end{tikzpicture}
```

会导致错误：！Dimension too large.
上面代码中，尽管设置了选项 /pgf/fpu, 但 fpu 程序库并不能处理带单位的尺寸 x=1/10000 cm，这个带单位的尺寸仍然由 PGF 按 TEX 的计算能力来处理，所以导致错误。

注意，仅仅调用 fpu 程序库并不能使得 Ti*k*Z 命令具有 fpu 的计算能力。

32.3 与定点算术程序库的比较

- fpu 至少支持 IEEE 标准下的双精度数值范围，而 fp 覆盖的数值范围是 $\pm 1 \cdot 10^{17}$.
- fpu 有一致的相对精度，使用 4 个或 5 个纠正数字。定点算术程序库使用绝对精度，当计算过程处于数值范围的极限附近时，计算可能失败。
- fpu 使用 PGF 的数学程序（使用 TEX 的寄存器）来处理数值的尾数，它的运行速度有可能比 fp 更快。

32.4 命令与编程参考

以下介绍文件《pgfmathfloat.code.tex》中的一些命令。

```
\let\pgfmathfloat@a@S=\c@pgf@counta
↪  % 计数器，0 ---> 0，1 ---> 正， 2 ---> 负， 3 ---> nan，4 ---> 正无穷， 5 ---> 负无穷
\let\pgfmathfloat@a@M=\pgf@xa% 尺寸寄存器，保存尾数
\let\pgfmathfloat@a@E=\c@pgf@countb% 计数器，保存指数
\let\pgfmathfloat@b@S=\c@pgf@countc% 计数器
\let\pgfmathfloat@b@M=\pgf@xb% 尺寸寄存器
\let\pgfmathfloat@b@E=\c@pgf@countd% 计数器
```

```
\def\pgfmathfloat@round@precision{2}
```

定义精度的初始值为 2.

32.4.1 浮点数的创建与转换

\pgfmathfloatparsenumber{⟨*x*⟩}

这个命令是 fpu 读取数值的主要命令。对参数 ⟨*x*⟩ 的要求：

- 可以带有正负号"±"，但不能是运算式，不能带有长度单位。
- 可以是定点数格式。
- 可以是使用 e 或 E 的科学计数法格式，也可以是浮点数格式，其中的幂指数应处于 TEX 允许的整数范围内，即不超过 31 位的整数。
- 可以使用数字符号“0”开头，例如 000123.123.
- 可以是“nan，nAn，nAN，Nan，NaN，NAN，inf，iNf，inF，iNF，Inf，INf，InF，INF，±inf，±iNf ……”等。

- 可以是保存以上格式的数值的宏。

如果在参数 ⟨x⟩ 中出现了不能解析的符号，一般会报错。

由于本命令以纯文本形式读取、解析 ⟨x⟩, 所以参数 ⟨x⟩ 可以是任意数量级和任意精度的数值。本命令把解析结果以纯文本形式保存在命令 \pgfmathresult 中，默认以 float 格式保存数值。

标记: 1; 尾数 5.21513; 幂指数 -11.

```
\pgfmathfloatparsenumber{5.21513e-11}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
标记: \F; 尾数 \M; 幂指数 \E.
```

上面例子中，命令 \pgfmathfloatparsenumber 读取数值并保存在 \pgfmathresult 中，然后用命令 \pgfmathfloattomacro 读取 \pgfmathresult 中的值，并将值的标记保存在宏 \F 中，将尾数保存在宏 \M 中，将幂指数保存在宏 \E 中。然后用这 3 个宏分别输出标记、尾数、幂指数。

本命令的定义是：

```
\def\pgfmathfloatparsenumber#1{%
    \begingroup
    \edef\pgfmathresult{#1}%
    \expandafter\pgfflt@impl\pgfmathresult\pgfflt@EOI
    \ifpgfmathfloatparsenumberpendingperiod
        \pgfmathfloat@a@Mtok=\expandafter{\the\pgfmathfloat@a@Mtok.0}%
    \fi
    \pgfmathfloatcreate{\the\pgfmathfloat@a@S}{\the\pgfmathfloat@a@Mtok}{
    ↪  \the\pgfmathfloat@a@E}%
    \pgfmath@smuggleone\pgfmathresult
    \endgroup
}
```

本命令先用 \pgfflt@impl...\pgfflt@EOI 解析 ⟨x⟩, 将：

- ⟨x⟩ 的标记符号保存在计数器 \pgfmathfloat@a@S 中；
- ⟨x⟩ 的尾数保存在记号寄存器 \pgfmathfloat@a@Mtok 中；
- ⟨x⟩ 的指数保存在计数器 \pgfmathfloat@a@E 中，

然后再用命令 \pgfmathfloatcreate→P. 576 将浮点数格式的结果保存到 \pgfmathresult 中。

注意本命令设置了一个组，除了结果 \pgfmathresult 外，本命令的其他操作都限制在组中。

\pgfmathfloat@decompose⟨*tok 1*⟩⟨*tok 2*⟩

本命令可以处理一个浮点数，将其标记符号、尾数、指数分别保存。此命令的定义是：

```
% #4: integer register for the flags.
% #5: dimen registers for the mantissa.
% #6: integer register for the exponent.
\def\pgfmathfloat@decompose#1#2{% sanitize!
    \ifx#2Y%
        \expandafter\pgfmathfloat@decompose@%
    \else
        \expandafter\pgfmathfloat@decompose@error%
    \fi
    #1#2%
}
\def\pgfmathfloat@decompose@#1Y#2e#3]\relax#4#5#6{%
   #4=#1\relax
   #5=#2pt % keep space here.
   #6=#3\relax%
}
```

通常这么用：

```
\expandafter\pgfmathfloat@decompose\pgfmathresult\relax\pgfmathfloat@a@S
↪  \pgfmathfloat@a@M\pgfmathfloat@a@E
```

其中 \pgfmathresult 保存一个浮点数。

\pgfmathfloat@decompose@tok⟨*tok 1*⟩⟨*tok 2*⟩

本命令类似 \pgfmathfloat@decompose, 通常这么用：

```
\expandafter\pgfmathfloat@decompose@tok\pgfmathresult\relax\pgfmathfloat@a@S
↪  \pgfmathfloat@a@Mtok\pgfmathfloat@a@E
```

浮点数的尾数保存在记号寄存器 \pgfmathfloat@a@Mtok 中 (而不是保存在尺寸寄存器中)。

还有一些类似上面两个命令的命令，例如：

```
\def\pgfmathfloat@decompose@F#1#2{%
    \ifx#2Y%
        \expandafter\pgfmathfloat@decompose@F@%
    \else
        \expandafter\pgfmathfloat@decompose@F@error%
    \fi
    #1#2%
}%
\def\pgfmathfloat@decompose@F@#1Y#2e#3]\relax#4{#4=#1\relax}%
```

/pgf/fpu/handlers/empty number={⟨*input*⟩}{⟨*unreadable part*⟩} (no default)

如果执行 \pgfkeys{/pgf/fpu/handlers/empty number={⟨*input*⟩}{⟨*unreadable part*⟩}}, 就会得到错误信息：

! Package PGF Math Error: Could not parse input '⟨*input*⟩' as a floating point number, sorry. The unreadable part was near '⟨*unreadable part*⟩'..

See the PGF Math package documentation for explanation.

/pgf/fpu/handlers/invalid number={⟨*input*⟩}{⟨*unreadable part*⟩} (no default)

/pgf/fpu/handlers/wrong lowlevel format={⟨*input*⟩}{⟨*unreadable part*⟩} (no default)

\pgfmathfloatqparsenumber{⟨*x*⟩}

等于 \pgfmathfloatparsenumber→P. 573, 在《pgfmathfloat.code.tex》中定义：

```
\let\pgfmathfloatqparsenumber=\pgfmathfloatparsenumber
```

\pgfmathfloattofixed{⟨*x*⟩}

这里的参数 ⟨*x*⟩ 是浮点数格式的数值，本命令以纯文本形式解析 ⟨*x*⟩, 把它转换为定点数格式并以纯文本形式保存在命令 \pgfmathresult 中。

1Y5.2e-4]; 0.00052;

```
\pgfmathfloatparsenumber{0.00052} \pgfmathresult; \quad
\pgfmathfloattofixed{\pgfmathresult} \pgfmathresult;
```

1Y1.23456e6]; 1234560.00000000;

```
\pgfmathfloatparsenumber{123.456e4} \pgfmathresult; \quad
\pgfmathfloattofixed{\pgfmathresult} \pgfmathresult;
```

11111.11111111111111111111000000000;

```
\pgfmathfloatparsenumber{1.11111111111111111111111e4}
\pgfmathfloattofixed{\pgfmathresult} \pgfmathresult;
```

上面例子中得到的定点数末尾有 9 个“0”，这 9 个“0”是内部命令自己加的，一般都会带有 9 个“0”。

命令 \pgfmathfloattofixed 大体上是根据指数来处理小数点位置的：把尾数的整数部分与小数部分分别处理，该添加符号“0”的地方就添加，逐步决定小数点位置；如果指数的绝对值较大，就使用循环操作。

\pgfmathfloattoint⟨*x*⟩

这里的参数 ⟨*x*⟩ 是浮点数格式的数值，本命令将 ⟨*x*⟩ 的小数部分直接去掉（无舍入），只保留整数部分，并以纯文本形式保存在 \pgfmathresult 中。

1Y1.23456e6]; 1234560;

```
\pgfmathfloatparsenumber{123.456e4} \pgfmathresult; \quad
\pgfmathfloattoint{\pgfmathresult} \pgfmathresult;
```

本命令先用 \pgfmathfloattofixed{⟨*x*⟩} 得到定点数，再提取其整数部分，其小数部分被直接吃掉。

\pgfmathfloattosci{⟨*float*⟩}

这里的参数 ⟨*float*⟩ 是浮点数格式的数值，本命令将 ⟨*float*⟩ 变成科学计数法格式，并以纯文本形式保存在 \pgfmathresult 中。

2.586949e3
```
\pgfmathfloattosci{1Y2.586949e3]}
\pgfmathresult
```

\pgfmathfloatvalueof{⟨*float*⟩}

本命令将浮点数格式的 ⟨*float*⟩ 转换为科学记数法格式的数，并将该数值输出到当前位置。

1.1e2
```
\pgfmathfloatvalueof{1Y1.1e2]}
```

\pgfmathfloatcreate{⟨*flags*⟩}{⟨*mantissa*⟩}{⟨*exponent*⟩}

本命令创建一个浮点数值，⟨*flags*⟩ 是所要创建的浮点数的标记，⟨*mantissa*⟩ 是尾数，⟨*exponent*⟩ 是幂指数。所创建的浮点数保存在 \pgfmathresult 中。本命令的参数都应该是纯粹的字符，或者是展开值为纯字符的宏（本命令会使用 \edef 将其展开）。

标记: 1; 尾数 1.0; 指数 327

```
\pgfmathfloatcreate{1}{1.0}{327}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
标记: \F; 尾数 \M; 指数 \E
```

本命令见文件《pgfmathfloat.code.tex》：

```
\def\pgfmathfloatcreate#1#2#3{%
    \edef\pgfmathresult{#1Y#2e#3]}%
}%
```

\pgfmathfloatifflags{⟨*floating point number*⟩}{⟨*flag*⟩}{⟨*true-code*⟩}{⟨*false-code*⟩}

参数 ⟨*floating point number*⟩ 是浮点数，或者是展开为浮点数的宏，可以是 \pgfmathresult.

如果浮点数 ⟨*floating point number*⟩ 的标记等于 ⟨*flag*⟩，则执行 ⟨*true-code*⟩，否则执行 ⟨*false-code*⟩.

⟨*flag*⟩ 有以下选择：

0 对应数值 0.

1 对应正数。

+ 对应正数。

2 对应负数。

- 对应负数。
3 对应“非数”。
4 对应 $+\infty$.
5 对应 $-\infty$.

It’s not zero!

```
\pgfmathfloatparsenumber{42}
\pgfmathfloatifflags{\pgfmathresult}{0}{It’s zero!}{It’s not zero!}
```

本命令见文件《pgflibraryfpu.code.tex》。

\pgfmathfloattomacro{⟨x⟩}{⟨*flags macro*⟩}{⟨*mantissa macro*⟩}{⟨*exponent macro*⟩}

参数 ⟨x⟩ 是浮点数，或者是展开为浮点数的宏，可以是 \pgfmathresult.
本命令创建 3 个宏：⟨*flags macro*⟩, ⟨*mantissa macro*⟩, ⟨*exponent macro*⟩, 这 3 个宏分别保存 ⟨x⟩ 的标记、尾数、幂指数。
本命令见文件《pgfmathfloat.code.tex》：

```
\def\pgfmathfloattomacro#1#2#3#4{%
    \begingroup
    \expandafter\pgfmathfloat@decompose@tok#1\relax\pgfmathfloat@a@S
    ↪  \pgfmathfloat@a@Mtok\pgfmathfloat@a@E
    \xdef\pgfmathfloat@glob@TMP{%
        \noexpand\def\noexpand#2{\the\pgfmathfloat@a@S}%
        \noexpand\def\noexpand#3{\the\pgfmathfloat@a@Mtok}%
        \noexpand\def\noexpand#4{\the\pgfmathfloat@a@E}%
    }%
    \endgroup
    \pgfmathfloat@glob@TMP
}
```

\pgfmathfloattoregisters{⟨x⟩}{⟨*flags count*⟩}{⟨*mantissa dimen*⟩}{⟨*exponent count*⟩}

参数 ⟨x⟩ 是浮点数，或者是展开为浮点数的宏，可以是 \pgfmathresult.
⟨*flags count*⟩ 是已定义的整数寄存器，⟨*mantissa dimen*⟩ 是已定义的尺寸寄存器，⟨*exponent count*⟩ 是已定义的整数寄存器，在本命令的处理下，这 3 个寄存器分别保存 ⟨x⟩ 的标记、尾数、幂指数。注意，本命令用到的寄存器需要提前定义。本命令会按照 TEX 的精度对尾数做截断（无舍入），然后保存在尺寸寄存器 ⟨*mantissa dimen*⟩ 中，因为寄存器都是 TEX 的寄存器。
本命令见文件《pgfmathfloat.code.tex》：

```
\def\pgfmathfloattoregisters#1#2#3#4{%
    \expandafter\pgfmathfloat@decompose#1\relax{#2}{#3}{#4}%
}
```

\pgfmathfloattoregisterstok{⟨x⟩}{⟨*flags count*⟩}{⟨*mantissa toks*⟩}{⟨*exponent count*⟩}

参数 ⟨x⟩ 是浮点数，或者是展开为浮点数的宏，可以是 \pgfmathresult.
⟨*flags count*⟩ 是已定义的整数寄存器，⟨*mantissa toks*⟩ 是已定义的 token 寄存器，⟨*exponent count*⟩ 是已定义的整数寄存器，这 3 个寄存器分别保存 ⟨x⟩ 的标记、尾数、幂指数。与上一个命令不同，本命令会保存 ⟨x⟩ 的尾数的完整精度。
本命令见文件《pgfmathfloat.code.tex》：

```
\def\pgfmathfloattoregisterstok#1#2#3#4{%
    \expandafter\pgfmathfloat@decompose@tok#1\relax{#2}{#3}{#4}%
}
```

\pgfmathfloatgetflags{⟨x⟩}{⟨*flags count*⟩}

参数 ⟨*x*⟩ 是浮点数，或者是展开为浮点数的宏，可以是 \pgfmathresult. ⟨*flags count*⟩ 是已定义的整数寄存器，⟨*x*⟩ 的标记会被保存在 ⟨*flags count*⟩ 中。

本命令见文件《pgfmathfloat.code.tex》：

```
\def\pgfmathfloatgetflags#1#2{%
    \expandafter\pgfmathfloat@decompose@F#1\relax{#2}%
}
%......
\def\pgfmathfloat@decompose@F#1#2{%
    \ifx#2Y%
        \expandafter\pgfmathfloat@decompose@F@%
    \else
        \expandafter\pgfmathfloat@decompose@F@error%
    \fi
    #1#2%
}%
%......
\def\pgfmathfloat@decompose@F@#1Y#2e#3]\relax#4{#4=#1\relax}%
```

\pgfmathfloatgetflagstomacro{⟨*x*⟩}{⟨*macro*⟩}

参数 ⟨*x*⟩ 是浮点数，或者是展开为浮点数的宏，可以是 \pgfmathresult. 本命令定义宏 ⟨*macro*⟩, 并将 ⟨*x*⟩ 的标记保存在宏 ⟨*macro*⟩ 中。

本命令见文件《pgfmathfloat.code.tex》：

```
\def\pgfmathfloatgetflagstomacro#1#2{%
    \expandafter\pgfmathfloat@decompose@Fmacro#1\relax{#2}%
}%
%......
\def\pgfmathfloat@decompose@Fmacro#1#2{%
    \ifx#2Y%
        \expandafter\pgfmathfloat@decompose@Fmacro@%
    \else
        \expandafter\pgfmathfloat@decompose@F@error%
    \fi
    #1#2%
}%
%......
\def\pgfmathfloat@decompose@Fmacro@#1Y#2e#3]\relax#4{\def#4{#1}}%
```

\pgfmathfloatgetmantissa{⟨*x*⟩}{⟨*mantissa dimen*⟩}

参数 ⟨*x*⟩ 是浮点数，或者是展开为浮点数的宏，可以是 \pgfmathresult. ⟨*mantissa dimen*⟩ 是已定义的尺寸寄存器，⟨*x*⟩ 的尾数会被保存在 ⟨*mantissa dimen*⟩ 中。

\pgfmathfloatgetmantissatok{⟨*x*⟩}{⟨*mantissa toks*⟩}

参数 ⟨*x*⟩ 是浮点数，或者是展开为浮点数的宏，可以是 \pgfmathresult. ⟨*mantissa toks*⟩ 是已定义的 token 寄存器，⟨*x*⟩ 的尾数会被保存在 ⟨*mantissa toks*⟩ 中。

\pgfmathfloatgetexponent{⟨*x*⟩}{⟨*exponent count*⟩}

参数 ⟨*x*⟩ 是浮点数，或者是展开为浮点数的宏，可以是 \pgfmathresult. ⟨*exponent count*⟩ 是已定义的整数寄存器，⟨*x*⟩ 的幂指数会被保存在 ⟨*exponent count*⟩ 中。

32.4.2 符号舍入操作

下面的命令会以纯文本形式处理其输入数值，对数值的舍入操作其实是对文本符号的操作，因此可以接受任意大、任意精度的输入数值。

\pgfmathroundto{⟨x⟩}

本命令的参数 ⟨x⟩ 可以是定点数，也可以是浮点数。本命令按选项 /pgf/number format/precision → P. 160 设置的输出精度，对 ⟨x⟩ 做舍入，小数部分中多余的 0 字符会被去掉，并把结果保存在 \pgfmathresult 中；如果 ⟨x⟩ 是定点数，则保存的结果也是定点数；如果 ⟨x⟩ 是浮点数，则保存的结果也是浮点数。本命令会把无效的符号 0 丢掉，这一点与下面的命令 \pgfmathroundtozerofill 不同。

仅当该命令保存在 \pgfmathresult 中的结果包含小数点时，该命令会把 TEX 条件判断宏 \ifpgfmathfloatroundhasperiod 的值设为 true，这个条件判断宏是全局布尔变量。

如果该命令保存在 \pgfmathresult 中的结果的幂指数大于 ⟨x⟩, 那么该命令会把 TEX 条件判断命令 \ifpgfmathfloatroundmayneedrenormalize 的值设为 true，否则这个条件判断命令的值保持为 false.

20000
```
\pgfmathroundto{19999.9996}
\pgfmathresult
```

2Y2.568462
```
\pgfkeys{/pgf/number format/precision=6}
\pgfmathroundto{2Y2.568462105e7]}
\pgfmathresult
```

2
```
\pgfkeys{/pgf/number format/precision=6}
\pgfmathroundto{2.000000006}
\pgfmathresult
```

\pgfmathroundtozerofill{⟨x⟩}

按选项 /pgf/number format/precision → P. 160 设置的输出精度，对 ⟨x⟩ 做舍入，如果小数部分的位数少于精度规定的位数，则用 0 补足，并把结果保存在 \pgfmathresult 中，保存结果的格式（按有关选项的设置，见 §92）是定点数格式或科学计数法格式。

20000.00
```
\pgfmathroundtozerofill{19999.9996}
\pgfmathresult
```

\pgfmathfloatround{⟨x⟩}

参数 ⟨x⟩ 是浮点数格式的，按选项 /pgf/number format/precision 设置的输出精度，调用命令 \pgfmathroundto 对 ⟨x⟩ 做舍入，小数部分中多余的 0 字符会被去掉，并把结果保存在 \pgfmathresult 中，保存的结果仍然是浮点数格式。

仅当该命令保存在 \pgfmathresult 中的结果包含小数点时，该命令会把 TEX 条件判断命令 \ifpgfmathfloatroundhasperiod 的值设为 true，这个条件判断命令是全局布尔变量。

5.3e1
```
\pgfmathfloatparsenumber{52.965}
\pgfmathfloatround{\pgfmathresult}
\pgfmathfloattosci{\pgfmathresult}
\pgfmathresult
```

1e0
```
\pgfmathfloatparsenumber{1}
\pgfmathfloatround{\pgfmathresult}
\pgfmathfloattosci{\pgfmathresult}
\pgfmathresult
```

\pgfmathfloatroundzerofill{⟨x⟩}

类似 \pgfmathfloatround, 不同的是，如果小数部分的位数少于精度规定的位数，则用 0 补足。

1.00e0
```
\pgfmathfloatparsenumber{1}
\pgfmathfloatroundzerofill{\pgfmathresult}
\pgfmathfloattosci{\pgfmathresult}
\pgfmathresult
```

\pgfmathfloatgetfrac{⟨*number*⟩}

本命令需要 fpu 库的支持，某些情况下也需要 fp 宏包的支持。

参数 ⟨*number*⟩ 是浮点数。本命令计算 ⟨*number*⟩ 所对应的分数形式，即符号或整数部分、分子、分母，并把“{符号或整数部分}{分子}{分母}”保存到 \pgfmathresult 中。

macro:->{-1}{1}{4}
```
\pgfmathfloatgetfrac{2Y1.25e0]}
\meaning\pgfmathresult
```

macro:->{-0}{1}{4}
```
\pgfmathfloatgetfrac{2Y0.25e0]}
\meaning\pgfmathresult
```

macro:->{025}{1}{4}
```
\pgfmathfloatgetfrac{1Y0.2525e2]}
\meaning\pgfmathresult
```

本命令的定义是：

```
% Defines \pgfmathresult to contain three sets of braces containing
% the sign (optionally containing any components >1), the numerator and the denominator for #
%
% \pgfmathfloatgetfrac{0.5}  -> \pgfmathresult contains {}{1}{2}
% \pgfmathfloatgetfrac{-0.5} -> \pgfmathresult contains {-}{1}{2}
% \pgfmathfloatgetfrac{1.5} ->  \pgfmathresult contains {1}{1}{2}
%
% special cases:
% \pgfmathfloatgetfrac{0}   ->  \pgfmathresult contains {0}{0}{1}
% \pgfmathfloatgetfrac{1}   ->  \pgfmathresult contains {1}{0}{1}
%
% The numerator and denominator is always a number (not empty)
\def\pgfmathfloatgetfrac#1{%
    \pgfutil@ifundefined{pgfmathfloatmultiply@}{%
        \pgfmath@PackageError{Sorry, the number format 'frac' requires '
        ↪  \string\usetikzlibrary{fpu}' (and, optionally, \string\usepackage{fp})
        ↪  in order to work correctly}%
        \edef\pgfmathresult{{#1}{0}{1}}%
    }{%
        \pgfmathfloatgetfrac@{#1}%
    }%
}%
\def\pgfmathfloatgetfrac@#1{%
    \begingroup
    %..... 省略
    \endgroup
}%
\def\pgfmathfloat@gobble@until@relax#1\relax{}
```

命令 \pgfmathfloatgetfrac@{⟨*number*⟩} 的处理是：

本命令设置一个组，所有操作都在组内进行。

把 ⟨*number*⟩ 的标记记为 S_0, 尾数记为 M_0, 指数记为 E_0,

- 如果 S_0 是 0，则定义

```
\def\pgfmathresult{{0}{0}{1}}%
```

- 如果 S_0 是 3，则定义

```
\edef\pgfmathresult{{}{NaN}{1}}%
```

- 如果 S_0 是 4，则定义

```
\edef\pgfmathresult{{}{\infty}{1}}%
```

- 如果 S_0 是 5，则定义

```
\edef\pgfmathresult{{-}{\infty}{1}}%
```

- 如果 S_0 是 1 或 2，那么做下面的处理。

1. 如果 S_0 是 1, 则赋值 \pgfmathfloat@a@S=1; 如果 S_0 是 2, 则赋值 \pgfmathfloat@a@S=-1.
2. 检查 E_0 的值，
   - 如果 $E_0 < 0$, 则

```
\def\pgfmathfloat@wholenumber{}%
```

   - 如果不是 $E_0 < 0$, 则

   (a) 执行

```
\pgfmathfloatcreate{1}{M_0}{E_0}%
```

   得到绝对值 $|\langle number\rangle|$ 的的浮点数格式。

   (b) 用 \pgfmathfloattofixed 将 $|\langle number\rangle|$ 转为定点数。
   - 把这个定点数的整数部分保存在宏 \pgfmathfloat@wholenumber 中；
   - 把这个定点数的小数部分，记为 $f_1 = 0.n_{11}n_{12}\cdots$.

   (c) 将 f_1 转为浮点数格式，将这个浮点数的标记记为 S_1, 尾数记为 M_1, 指数记为 E_1,
   - M_1 的整数部分保存到 \pgfmathfloat@mantissa@first 中；
   - M_1 的小数部分记为 $f_2 = 0.n_{21}n_{22}\cdots$，保存到 \pgfmathfloat@mantissa@ltone 中。

   (d) 检查 /pgf/number format/frac denom →P. 160 的值，
   - 如果 /pgf/number format/frac denom →P. 160 的值是空的，则
     i. 赋值 \pgfmathfloat@a@E=-\pgfmathfloat@a@E= $-E_1$.
     ii. 检查 f_2 是否等于 0,
     ▶ 如果 $f_2 = 0$, 则

```
\def\pgfmathfloat@factor{1}%
\edef\pgfmathfloat@scaled@numerator{\the\pgfmathfloat@a@Mtok}
↪  % 这是 M_1
```

     ▶ 如果 $f_2 \neq 0$, 那么：
     ⟨1⟩ 如果没有载入 fp 宏包，则把 f_2 转为浮点数格式，用命令 \pgfmathfloatreciprocal@ 计算倒数 $\frac{1}{f_2}$. 如果已经载入 fp 宏包，则用命令 \FPdiv →P. 564 计算倒数 $\frac{1}{f_2}$.
     ⟨2⟩ 把计算的倒数转为浮点数并保存在 \pgfmathfloat@inv 中。
     ⟨3⟩ 将浮点数 10^k 保存到 \pgfmathfloat@scalebaseten 中，其中 k 是选项 /pgf/number format/frac shift →P. 160 的值。
     ⟨4⟩ 计算 $\frac{1}{f_2} \cdot 10^k$, 结果保存在宏 \pgfmathfloat@factor 中。
     ⟨5⟩ 计算 $\frac{1}{f_2} \cdot 10^k \cdot M_1$, 结果转为定点数，保存在宏 \pgfmathfloat@scaled@numerator 中。
     ⟨6⟩ 将 \pgfmathfloat@factor, 即 $\frac{1}{f_2} \cdot 10^k$ 转为定点数，将其小数部分舍入到

整数部分，再把整数部分，即 $\mathrm{round}(\frac{1}{f_2}\cdot 10^k)$ 保存在宏 \pgfmathfloat@factor 中。

iii. 把宏 \pgfmathfloat@factor 保存的值赋予计数器 \pgfmathresultdenom.

iv. 如果 $-E_1 > 0$, 则把 \pgfmathresultdenom (此时也就是 \pgfmathfloat@factor) 与 $10^{|E_1|}$ 的乘积保存在计数器 \pgfmathresultdenom 中，保存的是 $\left[\mathrm{round}(\frac{1}{f_2}\cdot 10^k)\cdot 10^{|E_1|}\right]$, 这个值作为分母。

v. 将 \pgfmathfloat@scaled@numerator, 即 $\frac{1}{f_2}\cdot 10^k\cdot M_1$ 的小数部分舍入到整数部分，再把整数部分，即 $\mathrm{round}(\frac{1}{f_2}\cdot 10^k\cdot M_1)$ 保存在计数器 \pgfmathresultnumerator 中，这个值作为分子。

vi. 执行

```
\pgfmathgreatestcommondivisor{\pgfmathresultnumerator}{
↪  \pgfmathresultdenom}
```

将 \pgfmathresultnumerator 与 \pgfmathresultdenom 的最大公因子保存在 \pgfmathresult 中。

vii. 执行

```
\divide\pgfmathresultnumerator by\pgfmathresult\relax
\divide\pgfmathresultdenom by\pgfmathresult\relax
```

约去最大公因子。

viii. 如果 \ifpgfmathprintnumber@frac@warn 的真值是 true, 参考 /pgf/number format/frac warning→P. 160, 则

```
\ifnum\pgfmathresultdenom>1000
    \pgfutil@ifundefined{FPdiv}{%
        \pgfmathfloattosci@\pgfmathfloat@arg
        \pgf@typeout{! Package pgf /pgf/number format/frac
        ↪  warning=true: /pgf/number format/frac of `
        ↪  \pgfmathresult' = \the\pgfmathresultnumerator\space /
        ↪  \the\pgfmathresultdenom\space might be large due to
        ↪  instabilities. Try \string\usepackage{fp} to improve
        ↪  accuracy.}%
    }{}%
\fi
```

也就是说，如果此时的分母大于 1000, 而且还没有载入 fp 宏包，也就说，前面计算 $\frac{1}{f_2}$ 时用的是命令 \pgfmathfloatreciprocal@, 就会报错，提醒计算过程是不稳定的。

– 如果 /pgf/number format/frac denom→P. 160 的值不是空的，则

i. 规定分母 \pgfmathresultdenom,

```
\pgfmathresultdenom=\pgfmath@target@denominator\relax
```

ii. 计算 $|f_1|$ 的浮点数格式

```
\pgfmathfloatcreate{1}{M_1}{E_1}%
```

iii. 然后计算分子 \pgfmathresultnumerator,

```
\pgfmathfloattofixed\pgfmathresult
\pgf@xa=\pgfmathresult pt
\multiply\pgf@xa by\pgfmathresultdenom
\edef\pgfmathfloat@scaled@numerator{\pgf@sys@tonumber\pgf@xa}%
```

```
\expandafter\pgfmathfloat@loc@@to@int\pgfmathfloat@scaled@numerator
↪  \relax{\pgfmathresultnumerator}%
```

(e) 检查 \ifpgfmathprintnumber@frac@whole 的真值，即选项 /pgf/number format/frac whole→P. 160的真值，

– 如果真值是 true, 则输出带分数，此时不做调整。

– 如果真值是 false, 则输出假分数形式，此时检查是否有整数部分，如果有，则把整数部分 \pgfmathfloat@wholenumber 与分母 \pgfmathresultdenom 相乘，乘积加到分子 \pgfmathresultnumerator 上，再清空宏 \pgfmathfloat@wholenumber.

(f) 检查 \pgfmathfloat@a@S 的值 (此时它的值是 1 或 −1)，

– 如果 \pgfmathfloat@a@S<0, 则

```
\edef\pgfmathresult{{-\pgfmathfloat@wholenumber}{\the
↪  \pgfmathresultnumerator}{\the\pgfmathresultdenom}}%
```

– 如果不是 \pgfmathfloat@a@S<0, 则

```
\edef\pgfmathresult{{\pgfmathfloat@wholenumber}{\the
↪  \pgfmathresultnumerator}{\the\pgfmathresultdenom}}%
```

3. 将得到的 \pgfmathresult 推到 \endgroup 之外，结束命令。

总结 当把小数 $0.n_1n_2\cdots$ 转为浮点数后，得到与之对应的标记 S, 尾数 M, 指数 E, 也就是说，有对应关系：

$$0.n_1n_2\cdots \to (S, M, E)$$

综合以上，假设 N 是正的定点数，命令 \pgfmathfloatgetfrac 使用的算法对 N 的处理是：

1. 获取 N 的整数部分 w.
2. 获取 N 的小数部分 $f_1 \to (S_1, M_1, E_1)$.
3. 获取 M_1 的小数部分 f_2.
4. 假设 $f_2 \neq 0$, 计算 $\frac{1}{f_2}$,
   - 如果未载入 fp 宏包，则使用命令 \pgfmathfloatreciprocal@ 计算 $\frac{1}{f_2}$.
   - 如果已载入 fp 宏包，则使用命令 \FPdiv 计算 $\frac{1}{f_2}$.
5. 计算分母 D
   - 如果 $E_1 \geqslant 0$, 则令 $D = \text{round}(\frac{1}{f_2} \cdot 10^k)$.
   - 如果 $E_1 < 0$, 则令 $D = \text{round}(\frac{1}{f_2} \cdot 10^k) \cdot 10^{|E_1|}$.
6. 计算分子 $R = \text{round}(\frac{1}{f_2} \cdot 10^k \cdot M_1)$.
7. 计算分子 R 与分母 D 的最大公因子，约掉，得到分子 r, 分母 d.
8. 如果允许警告，那么，若之前使用命令 \pgfmathfloatreciprocal@ 计算 $\frac{1}{f_2}$, 并且 $d > 1000$, 则给出“计算不稳定”警告。
9. 最终得到分数 $w + \frac{r}{d}$.

32.4.3 数学运算命令

下面介绍《pgflibraryfpu.code.tex》中的一些命令。

无论是否载入 fpu 程序库，以下命令都可用。

\pgfmathfloat⟨*op*⟩

这是个一般的形式，⟨*op*⟩ 是数学引擎能够接受的函数名称，例如

```
\pgfmathfloatadd
\pgfmathfloatneg
\pgfmathfloatabs
\pgfmathfloatcos
\pgfmathfloatceil
\pgfmathfloatnotequal
\pgfmathfloatveclen
```

等等，都是数学命令的 fpu 版本。注意，fpu 版本的数学命令只接受浮点数格式的参数，它们保存在 \pgfmathresult 中的结果也是浮点数格式的。

```
\pgfmathfloatadd{1Y2.0e1]}{2Y3.256789e2]}
\pgfmathresult
```

2Y3.056789e2]

1Y4.5476e-1]; 2Y9.9619e-1]; 2Y5.4143001e-1];

```
\pgfmathfloatsin{1Y2.705e1]} \edef\s{\pgfmathresult} \s; \quad
\pgfmathfloatcos{2Y1.75e2]} \edef\c{\pgfmathresult} \c; \quad
\pgfmathfloatadd{\s}{\c} \pgfmathresult;
```

下面的代码可以正常输出：

```
\pgfmathfloatparsenumber{0.123456}
\let\bili=\pgfmathresult
\pgfmathfloatparsenumber{123456}
\pgfmathfloatmultiply{\pgfmathresult}{\bili}
\pgfmathfloattofixed{\pgfmathresult}
\pgfmathresult
```

15241.374000000000

但是若把上面代码中的 \bili 与 \pgfmathresult 交换位置，即

```
\pgfmathfloatparsenumber{0.123456}
\let\bili=\pgfmathresult
\pgfmathfloatparsenumber{123456}
\pgfmathfloatmultiply{\bili}{\pgfmathresult} % 交换了位置导致错误
\pgfmathfloattofixed{\pgfmathresult}
\pgfmathresult
```

会导致错误：

! Package PGF Math Error: Sorry, an internal routine of the floating point unit got an ill-formatted floating point number '12.3456000'. The unreadable part w as near '12.3456000'..

\pgfmathfloattoextentedprecision{⟨x⟩}

参数 ⟨x⟩ 是个浮点数或保存浮点数的宏。本命令对参数 ⟨x⟩ 的浮点表示式做一些修改，本命令的定义是：

```
\def\pgfmathfloattoextentedprecision#1{%
    \begingroup
    \pgfmathfloattoextentedprecision@a{#1}%
    \pgfmathfloatcreate{\pgfmathfloat@a@S}{\pgfmathresult}{\pgfmathfloat@a@E}%
    \pgfmath@smuggleone\pgfmathresult
    \endgroup
}%
```

为了演示本命令的作用，本文将此命令的定义修改为：

```
\def\pgfmathfloattoextentedprecision#1{%
    \begingroup
    \pgfmathfloattoextentedprecision@a{#1}%
    \pgfmathfloatcreate{\the\pgfmathfloat@a@S}{\pgfmathresult}{
    ↪  \the\pgfmathfloat@a@E}%
```

```
    \pgfmath@smuggleone\pgfmathresult
    \endgroup
}%
```

例如

1Y1.23456789e8]
1Y123.456789000e6]

```
\pgfmathfloatparsenumber{123456789}\pgfmathresult\par
\pgfmathfloattoextentedprecision{\pgfmathresult}
\pgfmathresult
```

可见本命令对 ⟨*x*⟩ 的浮点表示式中的尾数和指数做调整。在默认下，本命令将尾数的小数点右移 2 位，将指数“减去 2”。有的命令只针对浮点数的尾数做计算的，有时可能需要调整尾数来提高计算的精度或有效性。也可以使用本命令将尾数的小数点右移 1 位、或 3 位，只需要提前使用命令\pgfmathfloatsetextprecision 来做出设置，见下一个命令。

\pgfmathfloatsetextprecision{⟨*shift*⟩}

本命令规定命令 \pgfmathfloattoextentedprecision → P. 584 将尾数的小数点右移的位数。参数 ⟨*shift*⟩ 有以下可选值：

- 0, 不移动尾数的小数点；
- 1, 将尾数的小数点右移的 1 位，将指数“减去 1”；
- 2, 将尾数的小数点右移的 2 位，将指数“减去 2”；
- 3, 将尾数的小数点右移的 3 位，将指数“减去 3”；

1Y1.23456789e8]
1Y1234.56789000e5]

```
\pgfmathfloatsetextprecision{3}
\pgfmathfloatparsenumber{123456789}\pgfmathresult\par
\pgfmathfloattoextentedprecision{\pgfmathresult}
\pgfmathresult
```

本命令有默认设置：

```
\pgfmathfloatsetextprecision{2}%
```

\pgfmathfloattoextentedprecision@a{⟨*x*⟩}

参数 ⟨*x*⟩ 是个浮点数或保存浮点数的宏。

```
\def\pgfmathfloattoextentedprecision@a#1{%
    \edef\pgfmathresult{#1}%
    \expandafter\pgfmathfloat@decompose@tok\pgfmathresult\relax\pgfmathfloat@a@S
    ↪  \pgfmathfloat@a@Mtok\pgfmathfloat@a@E
    \ifnum\pgfmathfloat@a@S<3
        \advance\pgfmathfloat@a@E by-\pgfmathfloatextprec@shift\relax
        ↪  % compensate for shift
        \expandafter\pgfmathfloattoextentedprecision@@\the\pgfmathfloat@a@Mtok 000
        ↪  \pgfmathfloat@EOI
    \fi
}%
```

本命令先获取浮点数 ⟨*x*⟩ 的标记 \pgfmathfloat@a@S、尾数 \pgfmathfloat@a@Mtok、指数 \pgfmathfloat@a@E，若标记值是 0, 1, 2, 则按命令 \pgfmathfloatsetextprecision 设置的值 ⟨*shift*⟩, 将指数“减去 ⟨*shift*⟩”，将尾数的小数点右移 ⟨*shift*⟩ 位，再把尾数保存在 \pgfmathresult 中。

尾数：001.23456000
指 数：1

```
\def\aaa{1Y0.0123456e3]}
\makeatletter
\pgfmathfloattoextentedprecision@a\aaa
尾数：\pgfmathresult\par
指数：\the\pgfmathfloat@a@E
\makeatother
```

\pgfmathfloattoextentedprecision@b{⟨x⟩}

类似上一命令。

```
\def\pgfmathfloattoextentedprecision@b#1{%
    \edef\pgfmathresult{#1}%
    \expandafter\pgfmathfloat@decompose@tok\pgfmathresult\relax\pgfmathfloat@b@S
    ↪  \pgfmathfloat@a@Mtok\pgfmathfloat@b@E
    \ifnum\pgfmathfloat@b@S<3
        \advance\pgfmathfloat@b@E by-\pgfmathfloatextprec@shift\relax
        \expandafter\pgfmathfloattoextentedprecision@@\the\pgfmathfloat@a@Mtok 00
        ↪  \pgfmathfloat@EOI
    \fi
}%
```

\pgfmathfloatlessthan{⟨x⟩}{⟨y⟩}

参数 ⟨x⟩, ⟨y⟩ 都是浮点数，或者是经过 \pgfmathfloatparsenumber 处理过的数。

- 如果 ⟨x⟩ < ⟨y⟩, 则定义 \pgfmathresult 的值是 1.0, 且 \global\pgfmathfloatcomparisontrue;
- 否则定义 \pgfmathresult 的值是 0.0 并且 \global\pgfmathfloatcomparisonfalse.

0.0

```
\pgfmathfloatlessthan{1Y2.1e6]}{1Y2.1e6]}
\pgfmathresult
```

本命令的定义是：

```
\def\pgfmathfloatlessthan@#1#2{%
%\def\pgfmathfloatlessthan#1#2#3\and#4#5#6{%
    \global\pgfmathfloatcomparisonfalse
    \begingroup
    \edef\pgfmathfloat@loc@TMPa{#1}%
    \edef\pgfmathfloat@loc@TMPb{#2}%
    \expandafter\pgfmathfloat@decompose\pgfmathfloat@loc@TMPa
    ↪  \relax\pgfmathfloat@a@S\pgfmathfloat@a@M\pgfmathfloat@a@E
    \expandafter\pgfmathfloat@decompose\pgfmathfloat@loc@TMPb
    ↪  \relax\pgfmathfloat@b@S\pgfmathfloat@b@M\pgfmathfloat@b@E
    \ifcase\pgfmathfloat@a@S
        % x = 0 -> (x<y <=> y >0)
        \ifcase\pgfmathfloat@b@S
            % y = 0
        \or% y > 0
            \global\pgfmathfloatcomparisontrue
        \or% y < 0
        \or% y = nan
        \or% y = + infty
            \global\pgfmathfloatcomparisontrue
        \or% y = -infty
        \fi
    \or
        % x > 0 -> (x<y <=> ( y > 0 && |x| < |y|) )
        \ifcase\pgfmathfloat@b@S
            % y = 0
        \or% y>0:
            \pgfmathfloatlessthan@positive
        \or% y < 0
        \or% y = nan
        \or% y = + infty
            \global\pgfmathfloatcomparisontrue
```

```
        \or% y = -infty
        \fi
    \or
        % x < 0 -> (x<y <=> (y >= 0 ||   |x| > |y|) )
        \ifcase\pgfmathfloat@b@S
            % y = 0
            \global\pgfmathfloatcomparisontrue
        \or%y > 0
            \global\pgfmathfloatcomparisontrue
        \or% 'y<0':
            \pgfmathfloatgreaterthan@positive
        \or% y = nan
        \or% y = + infty
            \global\pgfmathfloatcomparisontrue
        \or% y = -infty
        \fi
    \or
        % x = nan.
    \or
        % x = +infty
    \or
        % x = -infty
        \ifnum\pgfmathfloat@b@S=3
        \else
            \global\pgfmathfloatcomparisontrue
        \fi
    \fi
    \endgroup
    \ifpgfmathfloatcomparison
        \def\pgfmathresult{1.0}%
    \else
        \def\pgfmathresult{0.0}%
    \fi
}%
\let\pgfmathfloatlessthan=\pgfmathfloatlessthan@
\let\pgfmathfloatless@=\pgfmathfloatlessthan@
%......
% compares \pgfmathfloat@a@[SME] < \pgfmathfloat@b@[SME]
\def\pgfmathfloatlessthan@positive{%
    \ifnum\pgfmathfloat@a@E<\pgfmathfloat@b@E
        \global\pgfmathfloatcomparisontrue
    \else
        \ifnum\pgfmathfloat@a@E=\pgfmathfloat@b@E
            \ifdim\pgfmathfloat@a@M<\pgfmathfloat@b@M
                \global\pgfmathfloatcomparisontrue
            \fi
        \fi
    \fi
}%
```

可见本命令是通过比较指数和尾数来判断大小的。宏 `\pgfmathresult` 可以用作本命令的第一个、或者第二个参数。

`\pgfmathfloatnotless@{`⟨x⟩`}{`⟨y⟩`}`

参数 ⟨x⟩, ⟨y⟩ 都是浮点数。

```
\def\pgfmathfloatnotless@#1#2{%
    \pgfmathfloatless@{#1}{#2}%
    \ifpgfmathfloatcomparison
        \def\pgfmathresult{0.0}%
    \else
        \def\pgfmathresult{1.0}%
    \fi
}%
```

\pgfmathfloatifapproxequalrel{⟨x⟩}{⟨y⟩}{⟨true code⟩}{⟨false code⟩}

参数 ⟨x⟩, ⟨y⟩ 先被命令 \pgfmathfloatparsenumber→P. 573 解析。本命令判断 ⟨x⟩ 与 ⟨y⟩ 是否足够接近。

- 如果 ⟨y⟩= 0, 那么，若 |⟨x⟩| <\pgfmathfloat@relthresh, 则执行 ⟨true code⟩; 否则执行 ⟨false code⟩.
- 如果 ⟨y⟩≠ 0, 那么，若 $\left|\frac{\langle x\rangle-\langle y\rangle}{\langle y\rangle}\right|$ <\pgfmathfloat@relthresh, 则执行 ⟨true code⟩; 否则执行 ⟨false code⟩.

\pgfmathfloat@relthresh 的初始值是：

```
\pgfqkeys{/pgf}{
    fpu/rel thresh/.code={%
        \pgfmathfloatparsenumber{#1}%
        \let\pgfmathfloat@relthresh=\pgfmathresult
    },
    fpu/rel thresh=1e-4,
}
```

本命令的定义是：

```
\long\def\pgfmathfloatifapproxequalrel#1#2#3#4{%
    \begingroup
    \pgfmathfloatparsenumber{#1}%
    \let\pgfmathfloatarga=\pgfmathresult
    \pgfmathfloatparsenumber{#2}%
    \let\pgfmathfloatargb=\pgfmathresult
    \pgfmathfloatrelerror@\pgfmathfloatarga\pgfmathfloatargb
    \let\pgfmathfloatarga=\pgfmathresult
    \pgfmathfloatlessthan@\pgfmathfloatarga\pgfmathfloat@relthresh
    \ifpgfmathfloatcomparison
        \def\pgfmathfloat@loc@TMPa{#3}%
    \else
        \def\pgfmathfloat@loc@TMPa{#4}%
    \fi
    \expandafter\endgroup
    \pgfmathfloat@loc@TMPa
}%
```

可见不要把宏 \pgfmathresult 用作本命令的第二个参数。

\pgfmathfloatequal@{⟨x⟩}{⟨y⟩}

参数 ⟨x⟩, ⟨y⟩ 先被命令 \pgfmathfloatparsenumber→P. 573 解析。本命令利用 \pgfmathfloatifapproxequalrel 判断 ⟨x⟩ 与 ⟨y⟩ 是否足够接近：

- 若足够接近，就认为 ⟨x⟩=⟨y⟩, 定义 \pgfmathresult 的值是 1;
- 否则定义 \pgfmathresult 的值是 0.

本命令的定义是：

```
\def\pgfmathfloatequal@#1#2{%
    \pgfmathfloatifapproxequalrel{#1}{#2}{%
        \def\pgfmathresult{1}%
        \pgfmathfloatcomparisontrue
    }{%
        \def\pgfmathresult{0}%
        \pgfmathfloatcomparisonfalse
    }%
}%
\let\pgfmathfloatequalto@=\pgfmathfloatequal@
```

可见不要把宏 \pgfmathresult 用作本命令的第二个参数。

\pgfmathfloatnotequal@{⟨x⟩}{⟨y⟩}

```
\def\pgfmathfloatnotequal@#1#2{%
    \pgfmathfloatifapproxequalrel{#1}{#2}{%
        \def\pgfmathresult{0}%
        \pgfmathfloatcomparisonfalse
    }{%
        \def\pgfmathresult{1}%
        \pgfmathfloatcomparisontrue
    }%
}%
```

\pgfmathfloatsign{⟨*float point*⟩}

参数 ⟨*float point*⟩ 是浮点数，或保存浮点数的宏。本命令解析 ⟨*float point*⟩ 的符号，将符号以浮点数格式保存在 \pgfmathresult 中，保存的可能是：0Y1.0e0], 1Y1.0e0], 2Y1.0e0], 3Y0.0e0].

1Y1.0e0]
```
\pgfmathfloatparsenumber{3}
\pgfmathfloatsign{\pgfmathresult}
\pgfmathresult
```

3Y0.0e0]
```
\pgfmathfloatparsenumber{nan}
\pgfmathfloatsign{\pgfmathresult}
\pgfmathresult
```

\pgfmathfloatmultiplyfixed{⟨*float*⟩}{⟨*fixed*⟩}

⟨*float*⟩ 是浮点数，⟨*fixed*⟩ 是定点数，本命令计算二者的乘积，并把结果保存在 \pgfmathresult 中，保存的结果是浮点数格式。本命令的计算采用浮点数算法，即计算 $m \cdot \langle fixed \rangle$, 其中 m 是 ⟨*float*⟩ 的尾数，然后把计算结果规范化。本命令首先利用命令 \pgfmathfloattoextentedprecision 处理 ⟨*float*⟩, ⟨*fixed*⟩, 然后再执行计算。

/pgf/fpu/rel thresh={⟨*number*⟩} (no default, initially 1e-4)

本选项为命令 \pgfmathfloatifapproxequalrel 设置一个阈值（threshold），用以判断两个数值是否足够近似。

按阈值近似
```
\pgfmathfloatifapproxequalrel{1Y1.123e-5]}{0}
  {按阈值近似} {按阈值不近似}
```

注意上面例子中，⟨*b*⟩ 是 0，但仍然能计算。

\pgfmathfloatshift{⟨*x*⟩}{⟨*num*⟩}

⟨*x*⟩ 是浮点数，⟨*num*⟩ 是整数。本命令将 $\langle x \rangle \cdot 10^{\langle num \rangle}$ 保存在 \pgfmathresult 中，保存的结果是浮点数格式。

```
2Y2.0e5] \pgfmathfloatshift{2Y2.0e3]}{2}
\pgfmathresult
```

\pgfmathfloatabserror{⟨x⟩}{⟨y⟩}

⟨x⟩, ⟨y⟩ 都是浮点数，本命令计算 ⟨x⟩ 与 ⟨y⟩ 的绝对误差 $|x-y|$，并保存在 \pgfmathresult 中，保存的结果是浮点数格式。

\pgfmathfloatrelerror{⟨x⟩}{⟨y⟩}

⟨x⟩, ⟨y⟩ 都是浮点数，本命令计算 ⟨x⟩ 与 ⟨y⟩ 的相对误差 $\frac{|x-y|}{|y|}$，并保存在 \pgfmathresult 中，保存的结果是浮点数格式。

\pgfmathfloatint{⟨x⟩}

⟨x⟩ 是浮点数，本命令将 ⟨x⟩ 的小数部分直接去掉（无舍入），将整数部分以浮点数格式保存在 \pgfmathresult 中,保存的结果是浮点数格式。被去掉的小数部分全局地保存在宏 \pgfmathfloatintremainder 中。

```
1Y1.23e2]„ 456789 \pgfmathfloatparse{123.456789}
\pgfmathfloatint\pgfmathresult
\pgfmathresult,, \pgfmathfloatintremainder
```

\pgfmathlog{⟨x⟩}

目前，这里的 ⟨x⟩ 必须是数值，不能是表达式。本命令计算 ⟨x⟩ 的自然对数值，等效于 \pgfmathln，不同的是，本命令以浮点数格式读取 ⟨x⟩, 利用等式

$$\ln(m\cdot 10^e)=\ln(m)+e\cdot\ln(10),$$

来计算结果。ln(10) 是个常数，使用 PGF 的标准数学程序计算 ln(m)，这里 $1\leqslant m<10$，最后的计算结果保存在 \pgfmathresult 中，保存的格式是定点数格式或整数。

如果 ⟨x⟩ 不是有效的数值，例如 ⟨x⟩ 不是正实数，则 \pgfmathresult 的值是空值（empty），而且没有错误提示信息。

```
9.21031 \pgfmathlog{1Y10e3]}
\pgfmathresult
```

```
-15.7452 \pgfmathlog{1.452e-7}
\pgfmathresult
```

\ifpgfmathfloatcomparison

这是个 TEX-if, 其初始值为 false, 其定义在文件《pgfmathfloat.code.tex》中：

```
\global\newif\ifpgfmathfloatcomparison
```

有的命令的结果就是设置这个 TEX-if 的真值，例如：

- \pgfmathfloatlessthan@{⟨x⟩}{⟨y⟩},
  - 如果 ⟨x⟩<⟨y⟩, 则 \global\pgfmathfloatcomparisontrue.
  - 否则保持初始值 \pgfmathfloatcomparisonfalse.
- \pgfmathfloatnotless@, 本命令调用 \pgfmathfloatlessthan@ 工作，所以
  - 如果 ⟨x⟩⩾⟨y⟩, 则 \global\pgfmathfloatcomparisontrue.
  - 否则保持初始值 \pgfmathfloatcomparisonfalse.
- \pgfmathfloatequal@{⟨x⟩}{⟨y⟩}, 本命令调用 \pgfmathfloatifapproxequalrel 来工作，所以
  - 如果判断为真，则

```
\def\pgfmathresult{1}%
\pgfmathfloatcomparisontrue
```

– 如果判断为假，则

```
\def\pgfmathresult{0}%
\pgfmathfloatcomparisonfalse
```

- \pgfmathfloatnotequal@{⟨x⟩}{⟨y⟩}, 类似 \pgfmathfloatequal@.
- \pgfmathfloatifapproxequalrel{⟨x⟩}{⟨y⟩}{⟨*true code*⟩}{⟨*false code*⟩}, 本命令调用 \pgfmathfloatlessthan@ 来工作，所以
  – 如果判断为真，则 \global\pgfmathfloatcomparisontrue.
  – 否则保持初始值 \pgfmathfloatcomparisonfalse.

在某些命令的处理过程中,为了比较数值,也会 (多次) 改变这个 TEX-if 的真值,例如 \pgfmathfloatadd@.

\pgfmathfloatmax@#1

这个命令的定义是：

```
\def\pgfmathfloatmax@#1{%
    \begingroup
        \pgfmathfloatcreate{2}{1.0}{2147483644}%
        \let\pgfmathmaxsofar=\pgfmathresult
        \pgfmathfloatmax@@#1{}%
}%
\def\pgfmathfloatmax@@#1{%
    \def\pgfmath@temp{#1}%
    \ifx\pgfmath@temp\pgfmath@empty%
        \expandafter\pgfmathfloatmax@@@%
    \else%
        \pgfmathfloatlessthan{\pgfmathmaxsofar}{#1}%
        \ifpgfmathfloatcomparison
            \edef\pgfmathmaxsofar{#1}%
        \fi
        \expandafter\pgfmathfloatmax@@%
    \fi%
}%
\def\pgfmathfloatmax@@@{%
    \let\pgfmathresult=\pgfmathmaxsofar
    \pgfmath@smuggleone{\pgfmathresult}%
    \endgroup
}%
```

可见这个命令的使用方式可以是：

```
\pgfmathfloatmax@{⟨x⟩}{⟨y⟩}...\pgfmath@empty
```

参数 ⟨x⟩, ⟨y⟩ ... 都是浮点数。这个命令采用逐项比较的方式，从参数 ⟨x⟩, ⟨y⟩ ... 中选出最大者，但最大者不会小于 −2147483644.

\pgfmathfloatmin@#1

这个命令的定义是：

```
\def\pgfmathfloatmin@#1{%
    \begingroup
        \pgfmathfloatcreate{1}{1.0}{2147483644}%
        \let\pgfmathminsofar=\pgfmathresult
        \pgfmathfloatmin@@#1{}%
}%
```

```
\def\pgfmathfloatmin@@#1{%
    \def\pgfmath@temp{#1}%
    \ifx\pgfmath@temp\pgfmath@empty%
        \expandafter\pgfmathfloatmin@@@%
    \else%
        \pgfmathfloatlessthan{#1}{\pgfmathminsofar}%
        \ifpgfmathfloatcomparison
            \edef\pgfmathminsofar{#1}%
        \fi
        \expandafter\pgfmathfloatmin@@%
    \fi%
}%
\def\pgfmathfloatmin@@@{%
    \let\pgfmathresult=\pgfmathminsofar
    \pgfmath@smuggleone{\pgfmathresult}%
    \endgroup
}%
```

可见这个命令的使用方式可以是：

```
\pgfmathfloatmin@{⟨x⟩}{⟨y⟩}...\pgfmath@empty
```

参数 ⟨x⟩, ⟨y⟩ ... 都是浮点数。这个命令采用逐项比较的方式，从参数 ⟨x⟩, ⟨y⟩ ... 中选出最小者，但最小者不会大于 2147483644.

\pgfmathfloatmax{⟨x⟩}{⟨y⟩}

参数 ⟨x⟩, ⟨y⟩ 都是浮点数。

```
\let\pgfmathfloatmax=\pgfmathfloatmaxtwo
```

\pgfmathfloatmaxtwo{⟨x⟩}{⟨y⟩}

参数 ⟨x⟩, ⟨y⟩ 都是浮点数。

```
\def\pgfmathfloatmaxtwo#1#2{%
    \pgfmathfloatlessthan{#1}{#2}%
    \ifpgfmathfloatcomparison
        \edef\pgfmathresult{#2}%
    \else
        \edef\pgfmathresult{#1}%
    \fi
}%
```

\pgfmathfloatmin{⟨x⟩}{⟨y⟩}

参数 ⟨x⟩, ⟨y⟩ 都是浮点数。

```
\let\pgfmathfloatmin=\pgfmathfloatmintwo
```

\pgfmathfloatmintwo{⟨x⟩}{⟨y⟩}

参数 ⟨x⟩, ⟨y⟩ 都是浮点数。

```
\def\pgfmathfloatmintwo#1#2{%
    \pgfmathfloatlessthan{#1}{#2}%
    \ifpgfmathfloatcomparison
        \edef\pgfmathresult{#1}%
    \else
        \edef\pgfmathresult{#2}%
    \fi
}%
```

32.4.4 用于编程的原始数学程序

当载入 fpu 库后，原始的（私人版的）数学函数命令 (在 /pgf/fpu=false 情况下的数学命令) 被保存在形式为

\pgfmath@basic@⟨*name*⟩@

的命令中。例如在 fpu 库中有定义

```
\let\pgfmath@basic@add@=\pgfmathadd@
```

\pgfmathadd@ 是命令 \pgfmathadd 的“私人版”，参考“数学引擎”中关于“自定义函数”的部分。当设置 /pgf/fpu=true 后，\pgfmathadd@ 等于 fpu 版本的命令 (不是原始的数学命令)，此时如果想使用原始的数学命令，就得使用 \pgfmath@basic@add@.

ptpt 11.94

```
\makeatletter
\pgfkeys{/pgf/fpu,}
\pgfmath@basic@add@{5.23pt}{6.71pt}
\pgfmathresult
\makeatother
```

但是注意，有的命令，例如 \pgfmath@basic@sign@, 在 fpu 库中并没有定义，详情参照文件《pgflibraryfpu.code.tex》。这可能是因为，由 fpu 库得到的数值一般都很大或很小，此时用原始的 \pgfmathsign@ 命令来处理这个很大或很小的数值并不合适，可能会出错。

实际上，在执行 /pgf/fpu=true 后，原始的（私人版的）数学函数、命令会被全局地保存为如下形式的控制序列：

\csname pgfmathfloat@backup@\string\⟨*origin command*⟩\endcsname

例如：

```
\csname pgfmathfloat@backup@\string\pgfmathadd@\endcsname
以及原始的解析命令
\csname pgfmathfloat@backup@\string\pgfmathparse\endcsname
```

1Y2.0e0]

```
\makeatletter
\pgfkeys{/pgf/fpu,}
\expandafter\let\expandafter\pgfmathparse\csname pgfmathfloat@backup@
↪  \string\pgfmathparse\endcsname
\pgfmathparse{1+1}
\pgfmathresult
\makeatother
```

上面例子的输出是浮点数格式，是因为执行“1+1”运算的还是 fpu 版本的函数。

32.4.5 例子

数学引擎的命令 \pgfmathveclen{x}{y}(即函数 veclen(x,y)) 计算向量 (x, y) 的长度，但是向量的分量 x，y 的绝对值不能超出 TEX 允许的精度范围 (不能太大或太小)。下面定义命令 \vectorlength 用来计算向量 (x, y) 的长度，其中的分量 x，y 是 fpu 程序库允许的数值。

```
\def\vectorlength[#1](#2,#3){%
  \pgfmathfloatparsenumber{#2}%
  \let\hengbiao=\pgfmathresult%
  \pgfmathfloatparsenumber{#3}%
  \let\zongbiao=\pgfmathresult%
  \pgfmathfloatmultiply{\hengbiao}{\hengbiao}%
  \let\hengbiaopingfang=\pgfmathresult%
  \pgfmathfloatmultiply{\zongbiao}{\zongbiao}%
  \let\zongbiaopingfang=\pgfmathresult%
  \pgfmathfloatadd{\hengbiaopingfang}{\zongbiaopingfang}%
```

```
  \pgfmathfloatsqrt{\pgfmathresult}%
  \pgfkeys{/pgf/number format/.cd,#1}%
  \pgfmathprintnumber{\pgfmathresult}%
}
```

上面定义的命令直接使用命令 \pgfmathfloatparsenumber 来解析向量的分量，所以向量分量中不能带有长度单位。用上面定义的命令来计算向量 (999999.123456, 999999.654321) 的长度：

sci 格式：$1.4142123 \cdot 10^{6}$； fixed 格式：1,414,212.3

```
sci 格式: \vectorlength[sci,precision=20](999999.123456,999999.654321); \quad
fixed 格式: \vectorlength[fixed,precision=20](999999.123456,999999.654321)
```

32.5 其他

32.5.1 几个命令

\ifpgfmathfloatparseactive

这是个 TEX-if, 当装备 fpu 的函数、命令时，同时会把这个 TEX-if 的真值设为 true; 当卸载 fpu 的函数、命令时，同时会把这个 TEX-if 的真值设为 false. 可以用这个 TEX-if 检查当前状态中是否已经了装备 fpu 的函数、命令。

命令 \pgflibraryfpuifactive → P. 572 就是利用这个 TEX-if 工作的：

```
\def\pgflibraryfpuifactive#1#2{%
    \ifpgfmathfloatparseactive
        #1%
    \else
        #2%
    \fi
}%
```

\pgfmathfloat@parser@install

这个命令的定义是：

```
\def\pgfmathfloat@parser@install{%
    \pgfmathfloat@plots@checkuninstallcmd
    \pgfmathfloat@plots@install%
    \pgfmathfloat@parser@install@functions
    \pgfmath@tokens@make{exponent}{\pgfmathfloat@POSTFLAGSCHAR}%
    \pgfmathfloat@uninstall@appendcmd{%
        \expandafter\let\csname pgfmath@token@exponent@\pgfmathfloat@POSTFLAGSCHAR
        ↪  \endcsname=\relax
    }%
    \let\pgfmath@basic@parse@exponent=\pgfmath@parse@exponent%
    \let\pgfmath@basic@stack@push@operand=\pgfmath@stack@push@operand
    \pgfmathfloat@install\pgfmath@stack@push@operand=
    ↪  \pgfmathfloat@stack@push@operand
    \pgfmathfloat@install\pgfmath@parse@operand@quote=
    ↪  \pgfmathfloat@parse@operand@quote
    \pgfmathfloat@install\pgfmath@parse@exponent=
    ↪  \pgfmathfloat@parse@float@or@exponent
    %
    \pgfmathfloat@install\pgfmathparse=\pgfmathfloatparse%

    ↪  %\pgfmathfloat@install\pgfmathparse@trynumber@token=\pgfmathfloat@parse@trynumber@toke
```

```
    \pgfmathfloat@install\pgfmathparse@expression@is@number=
    ↪  \pgfmathfloat@parse@expression@is@number
}%
```

其中：

- \pgfmathfloat@parser@install@functions 把“私人版”的数学引擎的函数转成 fpu 版本的函数
- \pgfmathfloat@install 把数学引擎的命令转成 fpu 版本的命令，例如使得 \pgfmathparse 等于 \pgfmathfloatparse
- \pgfmathfloat@uninstall@appendcmd 全局地向 \pgfmathfloat@uninstall 中添加代码（重定义它），而执行 \pgfmathfloat@uninstall 的作用是把 fpu 版本的函数、命令还原为原始的数学引擎的函数、命令

\pgfmathfloat@install\⟨*cmd 1*⟩=\⟨*cmd 2*⟩

本命令将原始的数学引擎的函数、命令“全局地另存”，再把数学引擎的函数、命令转成 fpu 版本；还会全局地向 \pgfmathfloat@uninstall 中添加代码（重定义它）。

这个命令的定义是：

```
\def\pgfmathfloat@install#1=#2{%
    \pgfmathfloat@prepareuninstallcmd{#1}%
    \let#1=#2%
}%
```

例如，当执行

```
\pgfmathfloat@install\pgfmathparse=\pgfmathfloatparse%
```

时，会导致：

- 将原始的 \pgfmathparse 另存：

  ```
  \expandafter\global\expandafter\let\csname pgfmathfloat@backup@\string
  ↪  \pgfmathparse\endcsname=\pgfmathparse%
  ```

  即另存为 \csname pgfmathfloat@backup@\string\pgfmathparse\endcsname.
- 全局地重定义 \pgfmathfloat@uninstall:

  ```
  \expandafter\gdef\expandafter\pgfmathfloat@uninstall\expandafter{
  ↪  \pgfmathfloat@uninstall
      \expandafter\let\expandafter\pgfmathparse\csname pgfmathfloat@backup@
      ↪  \string\pgfmathparse\endcsname%
  }%
  ```

- 将原始的 \pgfmathparse 转成 fpu 版本的 \pgfmathfloatparse:

  ```
  \let\pgfmathparse=\pgfmathfloatparse%
  ```

\pgfmathfloat@prepareuninstallcmd\⟨*cmd*⟩

这个命令的定义是：

```
\def\pgfmathfloat@prepareuninstallcmd#1{%
    % and store backup information (globally - I don't want to do that
    % all the time when the FPU is used!):
    \expandafter\global\expandafter\let\csname pgfmathfloat@backup@\string#1
    ↪  \endcsname=#1%
    \expandafter\gdef\expandafter\pgfmathfloat@uninstall\expandafter{
    ↪  \pgfmathfloat@uninstall
        \expandafter\let\expandafter#1\csname pgfmathfloat@backup@\string#1
        ↪  \endcsname%
```

```
    }%
}%
```

\pgfmathfloat@uninstall

这个宏保存一系列“全局定义代码”，例如保存着

```
\expandafter\let\expandafter\pgfmathparse\csname pgfmathfloat@backup@\string
↪  \pgfmathparse\endcsname%
```

执行 \pgfmathfloat@uninstall 的作用是把数学引擎的函数、命令还原为原始的（不使用 fpu 库时）的意义。

\pgfmathfloatparse{⟨*expression*⟩}

这个命令是 \pgfmathparse →P. 110 的变体，它把解析 ⟨*expression*⟩ 的结果 (一个浮点格式的数) 以纯文本形式保存在宏 \pgfmathresult 中。⟨*expression*⟩ 可以是含有复杂成分的表达式 (就像原本的命令 \pgfmathparse 所解析的表达式那样)。

此命令的定义是：

```
\def\pgfmathfloatparse{%
    \begingroup%
        % disable any dimension-dependant scalings:
        \let\pgfmathpostparse=\relax%
        \pgfmath@catcodes%
        \pgfmath@quickparsefalse%
        \ifpgfmathfloatparseactive\else
          \pgfmathfloat@parser@install
        \fi
        \pgfmathfloatparse@}%
\def\pgfmathfloatparse@#1{%
    \edef\pgfmathfloat@expression{#1}%
    \expandafter\pgfmathfloatparse@@\pgfmathfloat@expression\pgfmathfloat@
    \ifpgfmathfloat@scaleactive
        \expandafter\pgfmathfloatmultiply@\expandafter{\pgfmathresult}{
        ↪  \pgfmathfloatscale}
        \pgfmathfloattofixed{\pgfmathresult}%
    \else
        \pgfmathfloatparse@output
    \fi
}%
\def\pgfmathfloat@char@asterisk{*}%
\def\pgfmathfloatparse@@#1#2\pgfmathfloat@{%
    \def\pgfmathfloat@test{#1}%
    \ifx\pgfmathfloat@test\pgfmathfloat@char@asterisk%
        \def\pgfmathfloat@expression{#2}%
        \pgfmathfloat@scaleactivetrue
    \fi%
    \expandafter\pgfmathparse@\expandafter{\pgfmathfloat@expression}%
    % \endgroup provided by \pgfpathmarse@end
}%
```

命令 \pgfmathfloatparse 也会导致执行命令 \pgfmathfloat@parser@install →P. 594，把数学引擎的函数转成 fpu 版本的函数。上面代码中的命令 \pgfmathparse@ 在文件《pgfmathparser.code.tex》中定义。

注意上面代码中的一句注释：\endgroup provided by \pgfpathmarse@end。

\

32.5.2 一个错误

10.000000000

```
\pgfset{%
  bili/.initial=1/100,
  yuanshichicun/.initial=1000,
}
\pgfmathparse{\pgfkeysvalueof{/pgf/bili}}
\pgfmathfloatparsenumber{\pgfmathresult}%
\let\floatbili=\pgfmathresult%
\pgfmathfloatparsenumber{\pgfkeysvalueof{/pgf/yuanshichicun}}%
% 注意下一行的 \floatbili 与 \pgfmathresult 不能换位
\pgfmathfloatmultiply{\pgfmathresult}{\floatbili}%
\pgfmathfloattofixed{\pgfmathresult}%
\pgfmathresult
```

如果将上面代码中的 \floatbili 与 \pgfmathresult 交换位置会导致错误：

! Package PGF Math Error: Sorry, an internal routine of the floating point unit got an ill-formatted floating point number '10.000'. The unreadable part was near '10.000'..

导致这个错误的原因是：函数 \pgfmathfloatmultiply 的两个参数都必须是浮点数格式的数值，或者是保存浮点数的宏；此命令依次处理它的两个参数，在处理它的第一个参数时，会把处理结果 (一个定点数) 保存在 \pgfmathresult 中，然后再处理第二个参数；如果第二个参数是宏 \pgfmathresult, 那么就会导致错误，因为此时 \pgfmathresult 保存一个定点数。

当使用 fpu 版数学函数做计算时，尽量不要直接把宏 \pgfmathresult 用作函数的参数，如果要用的话，一定要参考函数的定义。

再如：

2Y4.0e0]
```
\pgfmathfloatparse{1}%
\pgfmathfloatsubtract{\pgfmathresult}{1Y2.0e0]}% 实际计算 -2+(-2)=-4
\pgfmathresult
```

2Y1.0e0]
```
\pgfmathfloatparse{1}%
\let\aaaa=\pgfmathresult%
\pgfmathfloatsubtract{\aaaa}{1Y2.0e0]}% 计算 1-2=-1
\pgfmathresult
```

000.0
```
\pgfmathfloatparse{1}%
\pgfmathfloatadd{\pgfmathresult}{0Y0.0e0]}% 不是 1+0
\pgfmathresult
```

32.5.3 一个关于计算精度的例子

假设 $f(x)=x^2+38x-645$, 用其他的计算器得到：

$$f(-50.717503054307414)\approx 0$$
$$f(-50.717545)\approx 0.002660827025$$
$$f(-50.717468)\approx -0.002223668976$$

选定区间 $I=[-50.717545,-50.717468]$, 采用二分法来求 $f(x)=0$ 的一个根，使用 fpu 的函数计算如下：

近似计算零值： $f(-50.717503054307414)\approx 2Y2.7e-4]$,
近似计算零值： $f(-50.7175)\approx 2Y2.7e-4]$,
右端值：$f(2Y5.0717468e1])\approx 2Y2.21e-3]$,
左端值：$f(2Y5.0717545e1])\approx 1Y2.62e-3]$,
中间值：$f(2Y5.0717545e1])\approx 1Y2.62e-3]$,

```
\makeatletter
% 定义命令 \quadraticfunction@value 计算 (ax+b)x+c
\def\quadraticfunction@value#1#2#3#4{%
    \begingroup%
    \pgfmathsetmacro{\temp@x}{#1}%
    \pgfmathsetmacro{\temp@a}{#2}%
    \pgfmathsetmacro{\temp@b}{#3}%
    \pgfmathsetmacro{\temp@c}{#4}%
    \pgfmathmultiply@{\temp@a}{\temp@x}%
    \let\@temp\pgfmathresult%
    \pgfmathadd@{\@temp}{\temp@b}%
    \pgfmathmultiply@{\pgfmathresult}{\temp@x}%
    \let\@temp\pgfmathresult%
    \pgfmathadd@{\@temp}{\temp@c}%
    \pgfmath@smuggleone\pgfmathresult\endgroup
}%
\ttfamily
\pgfkeys{/pgf/fpu}
近似计算零值：\quadraticfunction@value{-50.717503054307414}{1}{38}{-645}
$f(-50.717503054307414)\approx \pgfmathresult$,\\
近似计算零值：\quadraticfunction@value{-50.7175}{1}{38}{-645}
$f(-50.7175)\approx \pgfmathresult$,\\
\pgfmathparse{-50.717545}%
\let\left@temp\pgfmathresult%
\pgfmathparse{-50.717468}%
\let\right@temp\pgfmathresult%
\quadraticfunction@value{\right@temp}{1}{38}{-645}%
右端值：$f(\right@temp)\approx \pgfmathresult$, \\
\quadraticfunction@value{\left@temp}{1}{38}{-645}%
左端值：$f(\left@temp)\approx \pgfmathresult$, \\
\pgfmathparse{0.5}%
\let\@temp\pgfmathresult%
\pgfmathadd@{\right@temp}{\left@temp}%
\pgfmathmultiply@{\pgfmathresult}{\@temp}%
\let\mid@temp\pgfmathresult%
\quadraticfunction@value{\mid@temp}{1}{38}{-645}%
中间值：$f(\mid@temp)\approx \pgfmathresult$,
\pgfkeys{/pgf/fpu=false}
\makeatother
```

上面的输出表明：

- 用自定义的命令 \quadraticfunction@value 计算二次函数的零值时，自变量值 -50.717503054307414 与 -50.7175 产生的结果一样，说明小数部分中的有效数字“3054307414”对计算“结果”没有贡献；
- 计算二次函数的零值时，计算结果是 -0.00027, 这显示了误差；
- 用二分法处理区间 $I=[-50.717545,-50.717468]$ 时，中间值与左端值一样，这会导致二分法陷入无限循环。

产生这种状况的原因是 fpu 函数能处理的有效数字个数过少，限制了精度。

按文件《pgfmathfunctions.basic.code.tex》对数学函数的定义，fpu 的数学函数在对尾数做计算时，主要步骤是：

1. 为了运算需要，平移尾数的小数点，例如加减运算，如果两个数的指数不一样，会把小指数转换为大指数，此时可能损失有效数字；
2. 为了改善精度，平移尾数的小数点，参考 \pgfmathfloatsetextprecision→P. 585, 有的函数不需要这一步骤；
3. 调用普通的 PGF 数学函数来处理尾数，也就是利用 TEX 的寄存器来计算。

命令

```
\pgfmathfloatadd{2Y5.0717468e1]}{2Y5.0717545e1]}
```

针对尾数的计算是：

```
\pgfmath@basic@add@{-507.17468}{-507.17545}%
```

比较：

-1014.35013
2Y1.01435013e2]

```
\pgfmathadd{-507.17468}{-507.17545}%
\pgfmathresult
\par
\pgfmathfloatadd{2Y5.0717468e1]}{2Y5.0717545e1]}
\pgfmathresult
```

命令

```
\pgfmathfloatmultiply{2Y1.01435013e2]}{1Y5.0e-1]}
```

针对尾数的计算是：

```
\pgfmath@basic@multiply@{-10.1435013}{50}%
```

比较：

-507.17545
2Y5.0717545e1]

```
\pgfmathmultiply{-10.1435013}{50}%
\pgfmathresult
\par
\pgfmathfloatmultiply{2Y1.01435013e2]}{1Y5.0e-1]}
\pgfmathresult
```

使用 fp 宏包的命令 \FPqsolve → P. 566 来解 $x^2 + 38x - 645 = 0$, 结果是：

12.717503054307411653
-50.717503054307411653

```
\FPqsolve{\aaaa}{\bbbb}{1}{38}{-645}
\aaaa\par
\bbbb
```

第三十三章　Lindenmayer System 分形图

PGF 有《pgflibrarylindenmayersystems.code.tex》。

Ti*k*Z 有《tikzlibrarylindenmayersystems.code.tex》。

```
\usepgflibrary{lindenmayersystems} % LaTeX and plain TeX and pure pgf
\usepgflibrary[lindenmayersystems] % ConTeXt and pure pgf
\usetikzlibrary{lindenmayersystems} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[lindenmayersystems] % ConTeXt when using TikZ
```

本程序库提供构造平面 L-S 的方法。受到 TEX 内存的限制，能画出的 L-S 分形图形都不是非常复杂。

33.1　pgf 中的 L-S

33.1.1　声明一个 L-S

\pgfdeclarelindenmayersystem{⟨*name*⟩}{⟨*specification*⟩}

⟨*name*⟩ 是所声明的 L-S 的名称，它可以是一串符号，也可以是保存某些符号的宏。

⟨*specification*⟩ 是 L-S 的定义，其中通常使用命令 \symbol 和 \rule 来定义。\symbol 指定 ⟨*name*⟩ 使用的符号以及符号对应的命令。\rule 指定符号替换规则。

本命令对 ⟨*name*⟩ 的声明是全局的。

本命令检查控制序列 \csname pgf@lsystem@⟨*name*⟩\endcsname 是否已定义，如果已定义就报错；如果未定义就：

1. 用 \begingroup 开启一个组
2. 保存 ⟨*name*⟩ 的彻底展开值到 \pgf@lsystem@name

   ```
   \edef\pgf@lsystem@name{⟨name⟩}
   ```

3. 全局地定义控制序列 \csname pgf@lsystem@⟨*name*⟩\endcsname, 使之保存 ⟨*name*⟩ 的彻底展开值
4. 定义命令 \symbol 和 \rule

   ```
   \let\symbol=\pgf@lsystem@symbol%
   \let\rule=\pgf@lsystem@rule%
   ```

5. 执行 ⟨*specification*⟩
6. 用 \endgroup 结束组

\symbol{⟨*symbol*⟩}{⟨*code*⟩}

在命令 \pgfdeclarelindenmayersystem 中，它等于 \pgf@lsystem@symbol.

⟨*symbol*⟩ 是一个或一串符号，或者是保存一些符号的宏。

⟨*code*⟩ 是 ⟨*symbol*⟩ 对应的命令。

本命令全局地定义控制序列

\csname pgf@lsystem@⟨name⟩@symbol@⟨symbol⟩\endcsname

使之保存 ⟨code⟩.

```
\def\pgf@lsystem@symbol#1#2{%
    \expandafter\gdef\csname pgf@lsystem@\pgf@lsystem@name @symbol@#1\endcsname{#2
    ↪ }%
}%
```

例如

```
\symbol{A}{\pgflsystemdrawforward}
使得符号 A 执行命令 \pgflsystemdrawforward，即令画笔从当前点向前移动且在移动时画线
```

\rule{⟨head⟩->⟨body⟩}

在命令 \pgfdeclarelindenmayersystem 中，它等于 \pgf@lsystem@rule.

⟨head⟩ 是一个或一串符号，或者是保存一些符号的宏。本命令在处理 ⟨head⟩ 前，会先用 \expandafter 将它展开一次。

本命令只读取 ⟨head⟩ 的第一个 (非空格) 记号——仍然记为 ⟨head⟩——而其余记号会被忽略。

本命令指定一个替换规则，即在构建 L-S 的过程中，用 ⟨body⟩ 替换 ⟨head⟩.

可以多次使用本命令，规定多个替换规则。

本命令全局地定义控制序列

\csname pgf@lsystem@⟨name⟩@rule@⟨head⟩\endcsname

使之保存 ⟨body⟩.

注意，命令 \rule 的参数 ⟨head⟩ 中的符号可以与命令 \symbol 的参数 ⟨symbol⟩ 中的符号相同，也可以不同。

33.1.2 创建一个 L-S

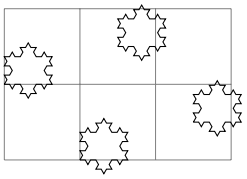
\pgflindenmayersystem{⟨name⟩}{⟨axiom⟩}{⟨order⟩}

⟨name⟩ 是已经声明的 L-S 名称。

参数 ⟨axiom⟩ 规定初始符号。命令 \rule 的参数 ⟨head⟩ 中的符号，命令 \symbol 的参数 ⟨symbol⟩ 中的符号，以及本命令参数 ⟨axiom⟩ 中的符号，可以有相同的，也可以有不同的。

参数 ⟨order⟩ 规定符号替换的次数。初始符号 ⟨axiom⟩ 对应第 0 次替换。⟨order⟩ 应当是能被 \pgfmathtruncatemacr 处理的表达式。

本命令的所有操作都被限制在一个 \begingroup 与 \endgroup 的组合中。



```
\begin{tikzpicture}
  \pgfdeclarelindenmayersystem{Koch curve}{
    \rule{F -> F-F++F-F}
  }
  \draw [help lines] grid (3,2);
  \pgfset{lindenmayer system/.cd, angle=60, step=2pt}
  \foreach \x/\y in {0cm/1cm, 1.5cm/1.5cm, 2.5cm/0.5cm, 1cm/0cm}{
    \pgftransformshift{\pgfqpoint{\x}{\y}}
    \pgfpathmoveto{\pgfpointorigin}
    \pgflindenmayersystem{Koch curve}{F++F++F}{2}
    \pgfusepath{stroke}
  }
\end{tikzpicture}
```

注意上面例子中的命令 \pgfpathmoveto 是必要的，因为其他命令都不会引入 move-to.

本命令的定义是：

```
\def\pgflindenmayersystem#1#2#3{%
    \begingroup%
        \edef\pgf@lsystem@name{#1}%
        \edef\pgf@lsystem@axiom{#2}%
        \pgfmathtruncatemacro\pgf@lsystem@order{#3}%
        %
        \let\pgf@lsystem@current@symbol=\relax%
        %
        \c@pgf@lsystem@iteration=0\relax%
        %
        \ifnum\pgf@lsystem@order=0\relax%
            \expandafter\pgf@lsystem@draw\pgf@lsystem@axiom\pgf@stop
            \let\pgf@lsystem@next=\pgf@lsystem@end%
        \else%
            \let\pgf@lsystem@next=\pgf@lsystem@run%
        \fi%
        \expandafter\pgf@lsystem@next\pgf@lsystem@axiom\pgf@lsystem@stop%
}%
```

其中执行符号替换的是 \pgf@lsystem@run, 将符号转为命令的是 \pgf@lsystem@draw.

\pgf@lsystem@run⟨*a token*⟩

本命令的处理是：

- 如果 \ifx⟨*a token*⟩\pgf@lsystem@stop 为 true, 则意味着结束了针对所有符号的“替换、转为命令”的操作，执行

  ```
  \def\pgf@lsystem@token{\pgf@lsystem@stop}%
  \let\pgf@lsystem@next=\pgf@lsystem@end%
  \expandafter\pgf@lsystem@next\pgf@lsystem@token
  ```

  以上 3 行最后导致 \endgroup.

- 如果 \ifx⟨*a token*⟩\pgf@stop 为 true, 则意味着完成了针对当前符号的“替换、转为命令”的操作，执行

  ```
  \advance\c@pgf@lsystem@iteration by-1\relax%
  \pgf@lsystem@run%
  ```

  注意 \pgf@stop 对应着计数器 \c@pgf@lsystem@iteration 的值减 1.

- 如果 ⟨*a token*⟩ 不是以上 2 个情况，则把 \pgf@lsystem@token let 为控制序列

  \csname pgf@lsystem@⟨*name*⟩@rule@⟨*a token*⟩\endcsname

  – 如果 \pgf@lsystem@token 等于 \relax, 即在 ⟨*name*⟩ 的声明中没有替换规则：

  ```
  \rule{⟨a token⟩ -> ⟨body⟩}
  ```

  那么就

  ```
  \pgf@lsystem@draw⟨a token⟩\pgf@stop%
  \pgf@lsystem@run%
  ```

  – 如果 \pgf@lsystem@token 不等于 \relax, 即在 ⟨*name*⟩ 的声明有替换规则：

  ```
  \rule{⟨a token⟩ -> ⟨body⟩}
  ```

  那么就

  1. 计数器 \c@pgf@lsystem@iteration 的值加 1,

     ```
     \advance\c@pgf@lsystem@iteration by1\relax%
     ```

  2. 检查计数器 \c@pgf@lsystem@iteration 的值，

* 如果计数器 \c@pgf@lsystem@iteration 的值等于 ⟨order⟩, 则意味着不必再对当前符号做进一步的替换，于是

```
\pgf@lsystem@draw⟨a token⟩\pgf@stop% 转为命令
\advance\c@pgf@lsystem@iteration by-1\relax% 计数器值减 1
\pgf@lsystem@run% 继续处理输入流中的下一个记号
```

* 如果计数器 \c@pgf@lsystem@iteration 的值不等于 ⟨order⟩, 则意味着还需要对当前符号做进一步的替换，于是

```
\def\pgf@lsystem@token{⟨a token⟩\pgf@stop}%
\expandafter\pgf@lsystem@run\pgf@lsystem@token%
```

注意其中添加了 \pgf@stop, 它影响计数器 \c@pgf@lsystem@iteration 的值。

\pgf@lsystem@draw⟨*a token*⟩⟨*a token*⟩...\pgf@stop

本命令是个循环处理：

- 如果 \ifx⟨*a token*⟩\pgf@stop 为 true, 则无需转换，执行 \relax.
- 如果 \ifx⟨*a token*⟩\pgf@stop 为 false, 则
  1. 把 \pgf@lsystem@current@symbol let 为控制序列

     \csname pgf@lsystem@⟨*name*⟩@symbol@⟨*a token*⟩\endcsname
  2. 检查 \pgf@lsystem@current@symbol,
     - 如果 \pgf@lsystem@current@symbol 等于 \relax, 这意味着在 ⟨*name*⟩ 的声明中没有

       ```
       \symbol{⟨a token⟩}{⟨code⟩}
       ```

       于是把 \pgf@lsystem@current@symbol let 为控制序列

       \csname pgf@lsystem@symbol@default@⟨*a token*⟩\endcsname

       如果这个控制序列没有定义，那么它等于 \relax. 这个控制序列会在后面被利用。
  3. 执行 \pgf@lsystem@@draw, 即

```
\edef\pgflsystemcurrentstep{\the\pgflsystemstep}%
\let\pgflsystemcurrentrightangle=\pgflsystemrightangle%
\let\pgflsystemcurrentrightangle=\pgflsystemleftangle%
\pgf@lsystem@current@symbol% 执行符号对应的命令，可能是 \relax
\pgf@lsystem@draw% 循环
```

33.1.2.1 \pgflindenmayersystem 总结

\pgflindenmayersystem{⟨*name*⟩}{⟨*axiom*⟩}{⟨*order*⟩} 会对 ⟨*axiom*⟩ 中的符号做“替换——转换为相应命令、代码”的操作。

- 当 ⟨*order*⟩ 等于 0 时，直接执行 ⟨*axiom*⟩ 对应的命令：

```
\pgf@lsystem@draw⟨axiom⟩\pgf@stop
```

- 当 ⟨*order*⟩ 大于 0 时，执行

```
\pgf@lsystem@run⟨axiom⟩\pgf@lsystem@stop
```

当 \pgf@lsystem@run →P. 602 解析到 \pgf@lsystem@stop 时，就结束组，并结束所有操作。

- 命令 \pgf@lsystem@run →P. 602 是对它之后的“单个记号”做“替换——转换为相应命令、代码”这种操作的。
- 在对某个记号 (符号) ⟨*S*⟩ 做替换时，如果 ⟨*S*⟩ 是未被 \rule 声明的符号，则直接执行

```
\pgf@lsystem@draw⟨S⟩\pgf@stop
```

即执行 S 对应的命令 (可能等于 \relax)，然后再继续处理之后的记号 (符号)。

- 在对某个记号 (符号) ⟨S⟩ 做替换时，如果 ⟨S⟩ 是被 \rule{⟨S⟩ -> ⟨*rule S*⟩} 声明的符号，则
  1. 计数器 \c@pgf@lsystem@iteration 加 1
  2. 检查计数器 \c@pgf@lsystem@iteration 的值，
     - 如果计数器 \c@pgf@lsystem@iteration 的值等于 ⟨*order*⟩, 则
       (a) 执行符号序列 ⟨*rule S*⟩ 中各个符号对应的命令

```
\pgf@lsystem@draw⟨rule S⟩\pgf@stop% 转为命令
```

       (b) 计数器值减 1

```
\advance\c@pgf@lsystem@iteration by-1\relax%
```

       (c) 循环

```
\pgf@lsystem@run% 继续处理输入流中的下一个记号
```

     - 如果计数器 \c@pgf@lsystem@iteration 的值不等于 ⟨*order*⟩, 则做替换：

```
\def\pgf@lsystem@token{⟨rule S⟩\pgf@stop}%
\expandafter\pgf@lsystem@run\pgf@lsystem@token%
```

注意其中添加了记号 \pgf@stop.

注意区别 2 种 \pgf@stop:

- 记号 \pgf@stop 是命令 \pgf@lsystem@draw →P. 603 的参数定界标志，使用这个命令时要附加这个记号；
- 在符号替换时，要附加记号 \pgf@stop, 这个记号将被命令 \pgf@lsystem@run →P. 602 吃掉，它就使得计数器 \c@pgf@lsystem@iteration 的值减 1，这是控制迭代次数的关键。

假设有

```
\symbol{A}{⟨code A⟩} \symbol{B}{⟨code B⟩}
\rule{A->AB} \rule{B->BA} \rule{X->AYB} \rule{F->BFA}
⟨axiom⟩ 是 XF
⟨order⟩ 是 2
```

那么“替换——转换”的操作就是：

1. 计数器 \c@pgf@lsystem@iteration=0
2. 把 ⟨*axiom*⟩, 即 XF 当作当前待处理的输入，处理输入中的第一个记号 X, 执行替换得到输入

```
计数器     \c@pgf@lsystem@iteration=1
待处理输入 AYB\pgf@stop F\pgf@lsystem@stop
```

3. 增值计数器，读取第一个记号 A, 处理 A 的替换

```
\c@pgf@lsystem@iteration=2
\pgf@lsystem@draw AB\pgf@stop YB\pgf@stop F\pgf@lsystem@stop
\c@pgf@lsystem@iteration=1
```

4. 之后面临的是

```
计数器     \c@pgf@lsystem@iteration=1
待处理输入 YB\pgf@stop F\pgf@lsystem@stop
```

5. 处理 Y, 由于 Y 未被 \rule 声明，所以相当于 \relax, 之后面临的是

```
计数器     \c@pgf@lsystem@iteration=1
待处理输入 B\pgf@stop F\pgf@lsystem@stop
```

6. 增值计数器，然后读取第一个记号 B, 处理 B 的替换

```
\c@pgf@lsystem@iteration=2
\pgf@lsystem@draw BA\pgf@stop\pgf@stop F\pgf@lsystem@stop
\c@pgf@lsystem@iteration=1
```

7. 之后面临的是

```
计数器     \c@pgf@lsystem@iteration=1
待处理输入 \pgf@stop F\pgf@lsystem@stop
```

8. 处理 \pgf@stop

```
\c@pgf@lsystem@iteration=1
\pgf@lsystem@run\pgf@stop F\pgf@lsystem@stop
```

9. 减值计数器，之后面临的是

```
计数器     \c@pgf@lsystem@iteration=0
待处理输入 F\pgf@lsystem@stop
```

10. 处理 F

```
\c@pgf@lsystem@iteration=0
\pgf@lsystem@run F\pgf@lsystem@stop
```

11. 增值计数器，替换 F, 之后面临的是

```
计数器     \c@pgf@lsystem@iteration=1
待处理输入 BFA\pgf@stop\pgf@lsystem@stop
```

12. 如上继续。

每当开始一个迭代层次后，都把计数器 \c@pgf@lsystem@iteration 的值加 1，然后执行符号对应的命令；每当完成一个迭代层次后，都把该计数器值减 1；当所有迭代层次完成后，这个计数器的值是 0.

33.1.3 符号对应的命令

一个符号对应的命令，就是在“把符号转换为命令”时，所得到的命令、代码。命令 \symbol 的参数 ⟨*code*⟩ 是一种符号命令。预定义的符号 F，f，+，-，[，] 对应的命令也是一种符号命令。

33.1.3.1 关于控制序列 \csname pgf@lsystem@symbol@default@⟨*symbol*⟩\endcsname

lindenmayersystems 库预定义了符号 F，f，+，-，[，] 对应的命令 (默认的命令)，它们是以下命令：

\pgflsystemdrawforward

本命令将画笔从当前点向前移动且在移动时画线，移动距离由 \pgflsystemcurrentstep 规定。

默认符号 F 对应这个命令。

```
\expandafter\def\csname
pgf@lsystem@symbol@default@F\endcsname{\pgflsystemdrawforward}%
%
\def\pgflsystemdrawforward{%
    \pgflsystemradonmizestep
    \pgftransformxshift{+\pgflsystemcurrentstep}%
    \pgfpathlineto{\pgfpointorigin}}%
```

其中命令 \pgflsystemradonmizestep 的最终作用是决定宏 \pgflsystemcurrentstep 的值 (一个尺寸值)。

本命令沿着 canvas 坐标系的 x 轴方向执行一个平移 (以 \pgflsystemcurrentstep 为平移距离)，并插入一个 line-to 操作。

\pgflsystemradonmizestep

本命令的定义是：

```
\def\pgflsystemradonmizestep{%
    \ifpgf@lsystem@randomize@step%
        \pgfmathrand%
        \pgf@x=\pgflsystemrandomizesteppercent pt\relax%
        \pgf@x=\pgfmathresult\pgf@x%
        \divide\pgf@x by20\relax%
        \advance\pgf@x by\pgflsystemstep\relax%
        \edef\pgflsystemcurrentstep{\the\pgf@x}%
    \else%
        \edef\pgflsystemcurrentstep{\the\pgflsystemstep}%
    \fi%
}%
```

随机函数 \pgfmathrand 得到的结果 \pgfmathresult= r 是 $[-1,1]$ 中的伪随机数。

\ifpgf@lsystem@randomize@step 的真值由选项 /pgf/lindenmayer system/randomize step percent→P. 609 的值决定。

宏 \pgflsystemrandomizesteppercent= p 是由选项 /pgf/lindenmayer system/randomize step percent→P. 609 定义的数值。

尺寸寄存器 \pgflsystemstep= $s$pt 是由选项 /pgf/lindenmayer system/step→P. 609 定义的尺寸。

命令 \pgflsystemradonmizestep 检查 \ifpgf@lsystem@randomize@step 的真值：

- 如果它的真值是 true, 则定义宏 \pgflsystemcurrentstep 为

  ```
  \edef\pgflsystemcurrentstep{r×p/20 + s pt}%
  ```

- 如果它的真值是 false, 则定义宏 \pgflsystemcurrentstep 为

  ```
  \edef\pgflsystemcurrentstep{s pt}%
  ```

宏 \pgflsystemcurrentstep 的值通常是尺寸寄存器 \pgflsystemstep= $s$pt 的当前值。

\pgflsystemmoveforward

本命令将画笔从当前点向前移动且在移动时不画线，移动距离由 \pgflsystemcurrentstep 规定。

默认符号 f 对应这个命令。

```
\expandafter\def\csname
pgf@lsystem@symbol@default@f\endcsname{\pgflsystemmoveforward}%
%
\def\pgflsystemmoveforward{%
    \pgflsystemradonmizestep
    \pgftransformxshift{+\pgflsystemcurrentstep}%
    \pgfpathmoveto{\pgfpointorigin}}%
```

本命令沿着 canvas 坐标系的 x 轴方向执行一个平移 (以 \pgflsystemcurrentstep 为平移距离)，并插入一个 move-to 操作。

\pgflsystemturnleft

本命令将画笔的前方方向逆时针旋转，旋转角度由 \pgflsystemcurrentleftangle 规定。

默认符号 + 对应这个命令。

```
\expandafter\def\csname
pgf@lsystem@symbol@default@+\endcsname{\pgflsystemturnleft}%
%
```

```
\def\pgflsystemturnleft{%
    \pgflsystemranomizeleftangle
    \pgftransformrotate{\pgflsystemcurrentleftangle}}%
```

其中命令 \pgflsystemranomizeleftangle 决定宏 \pgflsystemcurrentleftangle 的值 (一个数值)。

本命令将 canvas 坐标系旋转一个角度 \pgflsystemcurrentleftangle.

\pgflsystemranomizeleftangle

本命令的定义是:

```
\def\pgflsystemranomizeleftangle{%
    \ifpgf@lsystem@randomize@angle%
        \pgf@x=\pgflsystemrandomizeanglepercent pt\relax%
        \divide\pgf@x by20\relax%
        \pgfmathrand%
        \pgf@x=\pgfmathresult\pgf@x%
        \advance\pgf@x by\pgflsystemleftangle pt\relax%
        \edef\pgflsystemcurrentleftangle{\pgfmath@tonumber{\pgf@x}}%
    \else%
        \let\pgflsystemcurrentleftangle=\pgflsystemleftangle%
    \fi%
}%
```

随机函数 \pgfmathrand 得到的结果 \pgfmathresult= r 是 $[-1,1]$ 中的伪随机数。

\ifpgf@lsystem@randomize@angle 的真值由选项 /pgf/lindenmayer system/randomize angle percent →P. 610 的值决定。

宏 \pgflsystemrandomizeanglepercent= p 是由选项 /pgf/lindenmayer system/randomize angle percent →P. 610 定义的数值。

宏 \pgflsystemleftangle= a 是由选项 /pgf/lindenmayer system/left angle →P. 609 定义的数值。

命令 \pgflsystemranomizeleftangle 检查 \ifpgf@lsystem@randomize@angle 的真值:

- 如果它的真值是 true, 则定义宏 \pgflsystemcurrentleftangle 为

  \edef\pgflsystemcurrentleftangle{$\frac{r\times p}{20}+a$ pt}%

- 如果它的真值是 false, 则定义宏 \pgflsystemcurrentleftangle 为

  \let\pgflsystemcurrentleftangle=\pgflsystemleftangle

\pgflsystemturnright

本命令将画笔的前方方向顺时针旋转，旋转角度由 \pgflsystemcurrentrightangle 规定。

默认符号 - 对应这个命令。

```
\expandafter\def\csname
pgf@lsystem@symbol@default@-\endcsname{\pgflsystemturnright}%
%
\def\pgflsystemturnright{%
    \pgflsystemranomizerightangle
    \pgftransformrotate{-\pgflsystemcurrentrightangle}}%
```

其中命令 \pgflsystemranomizerightangle 决定宏 \pgflsystemcurrentrightangle 的值 (一个数值)。

本命令将 canvas 坐标系旋转一个负的角度 \pgflsystemcurrentleftangle.

\pgflsystemranomizerightangle

本命令的定义是：

```
\def\pgflsystemranomizerightangle{%
    \ifpgf@lsystem@randomize@angle%
        \pgf@x=\pgflsystemrandomizeanglepercent pt\relax%
        \divide\pgf@x by20\relax%
        \pgfmathrand%
        \pgf@x=\pgfmathresult\pgf@x%
        \advance\pgf@x by\pgflsystemrightangle pt\relax%
        \edef\pgflsystemcurrentrightangle{\pgfmath@tonumber{\pgf@x}}%
    \else%
        \let\pgflsystemcurrentrightangle=\pgflsystemrightangle%
    \fi%
}%
```

随机函数 \pgfmathrand 得到的结果 \pgfmathresult$= r$ 是 $[-1, 1]$ 中的伪随机数。

\ifpgf@lsystem@randomize@angle 的真值由选项 /pgf/lindenmayer system/randomize angle percent→P. 610 的值决定。

宏 \pgflsystemrandomizeanglepercent$= p$ 是由选项 /pgf/lindenmayer system/randomize angle percent→P. 610 定义的数值。

宏 \pgflsystemrightangle$= a$ 是由选项 /pgf/lindenmayer system/right angle→P. 610 定义的数值。

命令 \pgflsystemranomizerightangle 检查 \ifpgf@lsystem@randomize@angle 的真值：

- 如果它的真值是 true, 则定义宏 \pgflsystemcurrentrightangle 为

  \edef\pgflsystemcurrentrightangle{$\frac{r\times p}{20}+a$ pt}%

- 如果它的真值是 false, 则定义宏 \pgflsystemcurrentrightangle 为

  \let\pgflsystemcurrentrightangle=\pgflsystemrightangle

\pgflsystemsavestate

将画笔的当前状态暂时封存到另一个地方，它实际上开启一个组 \begingroup.

默认符号 [对应这个命令。

```
\expandafter\def\csname
pgf@lsystem@symbol@default@[\endcsname{\pgflsystemsavestate}%
%
\def\pgflsystemsavestate{\begingroup}%
```

\pgflsystemrestorestate

调出封存的画笔状态，它实际上结束一个组 \endgroup.

默认符号] 对应这个命令。

```
\expandafter\def\csname
pgf@lsystem@symbol@default@]\endcsname{\pgflsystemrestorestate}%
%
\def\pgflsystemrestorestate{\endgroup\pgfpathmoveto{\pgfpointorigin}}%
```

可见符号 [,] 的作用是设置一个组，限制组内的非全局操作，例如变换命令。

符号 F, f, +, -, [,] 可以直接用在 \pgflindenmayersystem 的参数 ⟨*axiom*⟩ 中，也可以直接用在命令 \rule 的参数 ⟨*head*⟩, ⟨*body*⟩ 中，不必经由命令 \symbol 来声明。当然可以用命令 \symbol 重定义这些符号对应的命令。

如果在命令 \rule 的参数 ⟨*head*⟩, ⟨*body*⟩ 中使用了某一个符号 X, 而这个 X 未经命令 \symbol 声明，

也不属于 F，f，+，-，[，] 之一，那么这个 X 就只是对“符号替换”过程有作用，而对“把符号转换为命令”的过程无作用。

如果在 \pgflindenmayersystem 的参数 ⟨*axiom*⟩ 中使用了某一个符号 X，而这个 X 未经命令 \symbol 声明，也不属于 F，f，+，-，[，] 之一，那么这个 X 不对应什么命令，但可以在符号替换中起到作用。

33.1.3.2 命令 \symbol 声明的符号命令

\symbol{⟨*symbol*⟩}{⟨*code*⟩} 声明了符号 ⟨*symbol*⟩ 对应的命令 ⟨*code*⟩. 参考 \pgflsystemdrawforward → P. 605 等命令的定义，在 ⟨*code*⟩ 中可以：

- 设置组
- 使用变换命令
- 如果是在 pgfpicture 环境中，可以插入 PGF 的构建路径的操作，如 move-to, line-to, curve-to 等操作，或者它们的组合
- 如果是在 tikzpicture 环境中，可以使用 TikZ 的路径命令，可以插入 scope 环境
- 执行其他命令

33.1.3.3 选项

/pgf/lindenmayer system/step=⟨*length*⟩ (no default, initially 5pt)

这个选项设置画笔向前移动的距离。

参数 ⟨*length*⟩ 会被 \pgfmathsetlength 处理，处理结果保存在 TEX 尺寸寄存器 \pgflsystemstep 中。

```
\pgfkeys{/pgf/lindenmayer system/.cd,%
    step/.code={\pgfmathsetlength\pgflsystemstep{#1}},%
}
```

/pgf/lindenmayer system/randomize step percent=⟨*percentage*⟩ (no default, initially 0)

⟨*percentage*⟩ 是百分比下的数值，从 0 到 100，该值会被 \pgfmathparse 处理，处理结果保存在宏 \pgflsystemrandomizesteppercent 中。

另外，如果处理结果是 0, 就设置 \ifpgf@lsystem@randomize@step 的真值为 false; 如果处理结果不是 0, 就设置 \ifpgf@lsystem@randomize@step 的真值为 true.

```
\pgfkeys{/pgf/lindenmayer system/.cd,%
    randomize step percent/.code={%
        \pgfmathparse{#1}%
        \let\pgflsystemrandomizesteppercent=\pgfmathresult%
        \ifdim\pgfmathresult pt=0pt\relax%
            \pgf@lsystem@randomize@stepfalse%
        \else%
            \pgf@lsystem@randomize@steptrue%
        \fi%
    },%
}
```

/pgf/lindenmayer system/left angle=⟨*angle*⟩ (no default, initially 90)

⟨*angle*⟩ 会被 \pgfmathparse 处理，处理结果保存在宏 \pgflsystemrleftangle 中。这个选项设置逆时针旋转画笔“前方”方向的角度。

```
\pgfkeys{/pgf/lindenmayer system/.cd,%
    left angle/.code={\pgfmathparse{#1}\let\pgflsystemleftangle=\pgfmathresult},%
}
```

/pgf/lindenmayer system/right angle=⟨*angle*⟩ (no default, initially 90)

⟨*angle*⟩ 会被 \pgfmathparse 处理，处理结果保存在宏 \pgflsystemrrightangle 中。这个选项设置顺时针旋转画笔“前方”方向的角度。

```
\pgfkeys{/pgf/lindenmayer system/.cd,%
    right angle/.code={\pgfmathparse{#1}\let\pgflsystemrightangle=\pgfmathresult},
    ↪  %
}
```

/pgf/lindenmayer system/angle=⟨*angle*⟩ (no default)

同时设置选项 left angle=⟨*angle*⟩ 和 right angle=⟨*angle*⟩.

```
\pgfkeys{/pgf/lindenmayer system/.cd,%
    angle/.style={/pgf/lindenmayer system/left angle=#1, /pgf/lindenmayer
    ↪  system/right angle=#1},%
}
```

/pgf/lindenmayer system/randomize angle percent=⟨*percentage*⟩ (no default, initially 0)

⟨*percentage*⟩ 是百分比下的数值，从 0 到 100，该值会被 \pgfmathparse 处理，处理结果保存在宏 \pgflsystemrandomizeanglepercent 中。

另外，如果处理结果是 0, 就设置 \ifpgf@lsystem@randomize@angle 的真值为 false; 如果处理结果不是 0, 就设置 \ifpgf@lsystem@randomize@angle 的真值为 true.

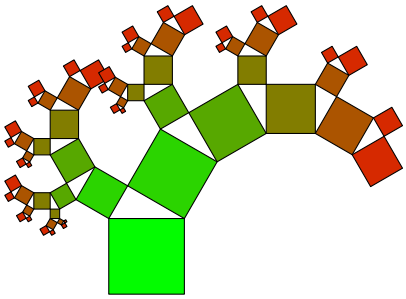
```
\pgfkeys{/pgf/lindenmayer system/.cd,%
    randomize angle percent/.code={%
        \pgfmathparse{#1}%
        \let\pgflsystemrandomizeanglepercent=\pgfmathresult%
        \ifdim\pgfmathresult pt=0pt\relax%
            \pgf@lsystem@randomize@anglefalse%
        \else%
            \pgf@lsystem@randomize@angletrue%
        \fi%
    }%
}
```

库文件如下执行选项：

```
\pgfkeys{/pgf/lindenmayer system/.cd,
    step=5pt,% 寄存器 \pgflsystemstep 的尺寸
    randomize step percent=0,
        % 宏 \pgflsystemrandomizesteppercent 的值,
        %\ifpgf@lsystem@randomize@step 的真值为 false
    angle=90,% 宏 \pgflsystemleftangle, \pgflsystemrightangle 的值
    randomize angle percent=0
        % 宏 \pgflsystemrandomizeanglepercent 的值
        %\ifpgf@lsystem@randomize@angle 的真值为 false
}%
```

33.1.4 例子

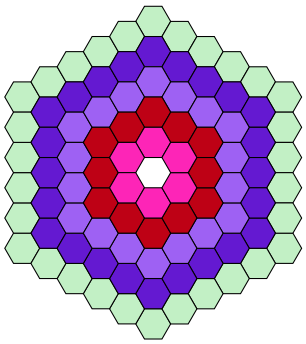
33.1.4.1 正方形树



```
\makeatletter
\pgfdeclarelindenmayersystem{rec-tri}{%
  \symbol{A}{
    % 宏 \pgf@lsystem@order 是迭代总次数，见 \pgflindenmayersystem 的定义
    % 计数器 \c@pgf@lsystem@iteration 是当前完成的层次数
    % 计算一个数值，用于填充色
    \expandafter\def\expandafter\TemLevelNum\expandafter{\the\c@pgf@lsystem@iteration}
    \pgfmathparse{int(\TemLevelNum/(\pgf@lsystem@order+1)*100)+1}
    \let\TempLevelNumCalForColor\pgfmathresult
% 需要提前给出点 (\TempPax,\TempPay)，点 (\TempPbx,\TempPby)
    \tikz@scan@one@point\pgf@process([rotate around={90:(\TempPax,\TempPay)}]\TempPbx,\TempPby)%
    \pgfgetlastxy\TempQax\TempQay
    \tikz@scan@one@point\pgf@process([rotate around={-90:(\TempPbx,\TempPby)}]\TempPax,\TempPay)%
    \pgfgetlastxy\TempQbx\TempQby
    \pgfpointscale{0.5}{\pgfpointadd{\pgfpoint{\TempQax}{\TempQay}}{\pgfpoint{\TempQbx}{\TempQby}}}
    \tikz@scan@one@point\pgf@process($(\TempQax,\TempQay)!(\TempQbx,\TempQby)!60:(\the\pgf@x,\the\pgf
    ↪  @y)$)
    \pgfgetlastxy\TempQcx\TempQcy
    \draw [line join=round,fill/.expand once={red!\TempLevelNumCalForColor!green}] (\TempPax,
    ↪  \TempPay)--(\TempPbx,\TempPby)--(\TempQbx,\TempQby)--(\TempQax,\TempQay)--cycle;
  }
  \symbol{B}{
    \let\TempPax=\TempQax
    \let\TempPay=\TempQay
    \let\TempPbx=\TempQcx
    \let\TempPby=\TempQcy
  }
  \symbol{C}{
    \let\TempPax=\TempQcx
    \let\TempPay=\TempQcy
    \let\TempPbx=\TempQbx
    \let\TempPby=\TempQby
  }
  \rule{X->[BAX][CAX]}
}
\makeatother
\begin{tikzpicture}
  \def\TempPax{0cm}
  \def\TempPay{0cm}
  \def\TempPbx{1cm}
  \def\TempPby{0cm}
  \pgflindenmayersystem{rec-tri}{AX}{5}
\end{tikzpicture}
```

上面例子中，宏 \pgf@lsystem@order 和计数器 \c@pgf@lsystem@iteration 是 \pgflindenmayersystem → P. 601 的内部处理使用的控制序列。上面图形中，每一个较大的正方形之后会有左右 2 个分支，[BAX] 绘制左侧分支，[CAX] 绘制右侧分支。在绘制过程中，由于未被 \rule 声明的符号都会被立即执行，所以符号 B，A，C 都是立即执行的，这样决定的绘制次序是：从根到末、先左后右。

33.1.4.2 蜂巢



```
\pgfmathsetmacro\LShoneycombminsize{2/sqrt(3)}% 转换尺寸，使得小六边形高度等于 1
\makeatletter
\pgfdeclarelindenmayersystem{L-S honeycomb}{%
  \symbol{X}{
    \pgfmathparse{rnd}\let\templevelcolorR\pgfmathresult
    \pgfmathparse{rnd}\let\templevelcolorG\pgfmathresult
    \pgfmathparse{rnd}\let\templevelcolorB\pgfmathresult
    \definecolor{templevelcolor}{rgb}{\templevelcolorR,\templevelcolorG,\templevelcolorB}
  }
  \symbol{A}{
    \pgfmathtruncatemacro\templevel{\c@pgf@lsystem@iteration}
    \pgftransformrotate{60}
    \node [fill=templevelcolor](\templevel-1) at (30:\templevel) {};
    \ifnum\templevel=1\relax\else
      \foreach \i[remember=\i as \j (initially 1)] in {2,...,\templevel}
      {
        \node[fill=templevelcolor,below=0pt and 0pt of \templevel-\j] (\templevel-\i) {};
      }
    \fi
  }
  \rule{Y->[YXAAAAAA]}
}
\makeatother

\begin{tikzpicture}
  [
    scale=0.4,
    every node/.style={
      outer sep=0pt,
      regular polygon,
      regular polygon sides=6,
      minimum size=\LShoneycombminsize cm,
      draw,
      transform shape
    }
  ]
  \pgflindenmayersystem{L-S honeycomb}{Y}{5}
\end{tikzpicture}
```

上面例子中，六边形用 node 画出；符号 X 设置随机的颜色，用于填充各个层次的 node；由于 PGF 的伪随机函数默认的种子是 \time×\year, 所以颜色可能随着系统时间的分钟数的变化而变化 (当然需要编译一下)；符号 A 画出数个自上而下排布的六边形，然后将其旋转一个或几个 60°，在一个层次内，因为没有组的限制，旋转角度是累计的；符号 Y 的替换符号中的方括号代表一个组，用于限制一个层次。

下面看一下，按规则 \rule{Y -> [YXAAAAAA]} 将 Y 替换 3 次是什么样子：

```
第 1 次
1[YXA^6]\pgf@stop\pgf@lsystem@stop
第 2 次
1[2[YXA^6]\pgf@stop XA^6]\pgf@stop\pgf@lsystem@stop
```

```
第 3 次
1[2[3[YXA^6]\pgf@stop 2 XA^6]\pgf@stop 1 XA^6]\pgf@stop 0\pgf@lsystem@stop
```

上面各行的符号中：

- 用 A^6 代表 AAAAAA;
- 方括号代表 \begingroup 与 \endgroup 的组合；方括号可以明显地标识各个层次，但在这个例子中，方括号不是必须的，可以省去的；
- 下标数字代表计数器 \c@pgf@lsystem@iteration 的值；
- 记号 \pgf@stop 是符号替换时添加的，是会被 \pgf@lsystem@run 解析的记号，这种 \pgf@stop 会导致计数器 \c@pgf@lsystem@iteration 的值减 1.

第三十四章　patterns

第三十五章　patterns.meta

文件《pgflibrarypatterns.meta.code.tex》与《pgfcorepatterns.code.tex》有关系，但二者基本上相互独立，并且前者能兼容后者。

参考基本层命令 \pgfdeclarepatternformonly→P. 352, \pgfdeclarepatterninherentlycolored→P. 353, \pgfsetfillpattern→P. 355.

35.1　自定义图样

```
\usepgflibrary{patterns.meta} % LaTeX and plain TeX and pure pgf
\usepgflibrary[patterns.meta] % ConTeXt and pure pgf
\usetikzlibrary{patterns.meta} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[patterns.meta] % ConTeXt when using TikZ
```

这两个库提供一些命令、选项来创建图样。注意这两个库是实验性的，将来可能会有改变。

库文件《tikzlibrarypatterns.meta.code.tex》会调用文件《pgflibrarypatterns.meta.code.tex》.

PGF 填充图样的过程参考 \pgfdeclarepatternformonly→P. 352.

35.1.1　基本命令

\pgfdeclarepattern{⟨*config*⟩}

本命令定义一个图样。在 ⟨*config*⟩ 中利用键做出定义，其中的键应当有前缀路径 /pgf/patterns, 见下文。

本命令的所有操作都限制在一个组中。

本命令：

1. 设置 \nullfont,
2. 保存 ⟨*config*⟩

   ```
   \def\pgf@pat@options{,⟨config⟩}
   ```

3. 执行 ⟨*config*⟩ 中的键值对，
4. 然后如下检查：
   - 如果不使用选项 name 给出名称，则导致错误；
   - 如果不使用选项 bottom left, top right 给出边界盒子的边界坐标，则导致错误；
   - 如果不使用选项 tile size 给出“图样砖盒子”的右上角点，则导致错误；
   - 如果不使用选项 code 给出绘制图样的代码，则导致错误。
5. 然后执行 \pgf@declarepattern@meta→P. 616.

图样可以分为 colored 与 uncolored 两类，也可以分为 mutable 与 unmutable 两类。

利用 \pgfdeclarepattern{⟨*config*⟩} 声明一个图样时：

- 如果在 ⟨*config*⟩ 中给出了选项

```
name=⟨name⟩,% 用户给出的图样名称
parameters={⟨list⟩},%
...
```

并且参数 ⟨*list*⟩ 非空，那么声明的图样 ⟨*name*⟩ 是 mutable 类型的，此时命令 `\pgfdeclarepattern` 只是全局地定义控制序列

`\csname pgf@pattern@name@meta@`⟨*name*⟩`\endcsname`

使之保存关于图样 ⟨*name*⟩ 的设置 ⟨*config*⟩, 并不直接声明图样。

只有在使用图样的命令 `\pgfsetfillpattern{`⟨*name*⟩`[`⟨*options*⟩`]}{`⟨*color*⟩`}` 那里，才会声明图样 ⟨*name*⟩ (调用 `\pgf@pat@declare`)，然后引入图样做填充。此时参数 ⟨*name*⟩⟨*list*⟩ 被看作一个整体组合，对应全局控制序列

`\csname pgf@pattern@name@meta@`⟨*name*⟩⟨*list*⟩`\endcsname`

(这个控制序列与 `\csname pgf@pattern@name@meta@`⟨*name*⟩`\endcsname` 保存的内容相同)。这个组合只被声明一次。当第 2 次使用这个组合时，直接利用，不再次做声明。

- 如果在 ⟨*config*⟩ 中没有给出选项 `parameters={`⟨*list*⟩`}` 或者参数 ⟨*list*⟩ 是空的，那么声明的图样是 unmutable 类型的，此时命令 `\pgfdeclarepattern` 会 (调用 `\pgf@pat@declare`) 直接声明图样，也会全局地定义控制序列

`\csname pgf@pattern@name@meta@`⟨*name*⟩`\endcsname`

使之保存关于图样 ⟨*name*⟩ 的设置 ⟨*config*⟩.

关于 mutable 类型的图样：

1. 用 `\pgfdeclarepattern` 声明 mutable 类型的图样时：
   (a) 那些能够改变图样外观的变量最好是以 `/pgf/pattern keys` 为前缀路径的键；
   (b) 在 ⟨*config*⟩ 中，应当给出选项 `parameters={`⟨*list*⟩`}`, 那些能够改变图样外观的键应当全部列举在 ⟨*list*⟩ 中；
   (c) 在 ⟨*config*⟩ 中，应当给出选项 `defaults={`⟨*default options*⟩`}`, 那些能够改变图样外观的键的默认值，最好以键值对的形式列举在 ⟨*default options*⟩ 中 (以逗号为分隔符号)；
   (d) 在 ⟨*config*⟩ 中，选项 `code={`⟨*code*⟩`}` 指定绘制图样的代码，在 ⟨*code*⟩ 中，可以利用那些以 `/pgf/pattern keys` 为前缀路径的键；
2. 使用命令 `\pgfsetfillpattern{`⟨*name*⟩`[`⟨*options*⟩`]}{`⟨*color*⟩`}` 引入 mutable 类型的图样时，如果需要修改图样的外观，那么就需要重新设置那些以 `/pgf/pattern keys` 为前缀路径的键的值，那些键值对应当在命令选项 ⟨*options*⟩ 中列出。

`\pgf@declarepattern@meta`

本命令检查选项 `name=`⟨*name*⟩ 指定的名称 ⟨*name*⟩ 是否已经由 `\pgfdeclarepattern` 声明过，也就是检查控制序列

`\csname pgf@pattern@name@meta@`⟨*name*⟩`\endcsname`

是否有定义 (不等于 `\relax`)。如果已经声明过，则报错；否则，检查选项 `parameters=`⟨*list*⟩ 指定的列表 ⟨*list*⟩ 是否是空的 (这个列表的默认值是空的，等于 `\pgfutil@empty`)：

- 如果是空的，则
  1. `\pgf@pat@declare`
  2. 重定义选项组合

```
\pgf@pat@addto@macro\pgf@pat@options{,number ..= 展开的 \pgf@pattern@number
↪  }
```

- 如果不是空的，则什么也不做。

然后全局定义一个控制序列

\csname pgf@pattern@name@meta@⟨*name*⟩\endcsname

使之保存 ⟨*config*⟩:

```
\expandafter\global\expandafter\let\csname\pgf@pat@name@prefix\pgf@pat@name
↪  \endcsname=\pgf@pat@options%
```

这个控制序列就代表了名称为 ⟨*name*⟩ 的图样，它保存的是

⟨*config*⟩，以及 number ..= 展开的 \pgf@pattern@number

其中的 \pgf@pattern@number 在 \pgf@pat@declare 那里定义，它保存 ⟨*name*⟩ 对应的编号，这个编号实际是 ⟨*name*⟩ 的底层名称。

选项 /pgf/patterns/number .. 的定义是:

```
\pgfkeys{/pgf/patterns/number ../.store in=\pgf@pat@number}
```

这个选项是内部操作利用的选项。

\pgf@pat@declare

本命令的定义是:

```
\def\pgf@pat@declare{%
   \pgfsysprotocol@getcurrentprotocol\pgf@pattern@temp%
   {%
     % Set up x and y vectors. Should use a scope rather than TeX group?
     % Vectors may be needed when the tile bounding box is
     % calculated.
     \pgfsetxvec{\pgfpoint{\pgf@pat@xvec}{+0pt}}%
     \pgfsetyvec{\pgfpoint{+0pt}{\pgf@pat@yvec}}%
     \pgf@pat@doifnotempty\pgf@pat@declarebefore%
     \pgfinterruptpath%
       \pgfpicturetrue%
       \pgf@relevantforpicturesizetrue%
       \pgftransformreset%
       \pgfsysprotocol@setcurrentprotocol\pgfutil@empty%
       \pgfsysprotocol@bufferedtrue%
       \pgfsys@beginscope%
         \pgfinterruptboundingbox%
           \pgfsetarrows{-}%
           \pgf@pat@doifnotempty\pgf@pat@codebefore
           \pgf@pat@code%
           \pgf@pat@doifnotempty\pgf@pat@codeafter%
         \endpgfinterruptboundingbox%
       \pgfsys@endscope%
       \pgfsysprotocol@getcurrentprotocol\pgf@pattern@code%
       \global\let\pgf@pattern@code=\pgf@pattern@code%
     \endpgfinterruptpath%
     \pgf@pat@doifnotempty\pgf@pat@declareafter%
     \pgf@pat@processpoint{\pgf@pat@bottomleft}%
     \pgf@xa=\pgf@x%
     \pgf@ya=\pgf@y%
     \pgf@pat@processpoint{\pgf@pat@topright}%
     \pgf@xb=\pgf@x%
     \pgf@yb=\pgf@y%
     \pgf@pat@processpoint{\pgf@pat@tilesize}%
     \pgf@xc=\pgf@x%
     \pgf@yc=\pgf@y%
     \begingroup%
```

```
38        \pgftransformreset%
39        \pgf@pat@processtransformations\pgf@pat@transformation%
40        \pgfgettransformentries\aa\ab\ba\bb\shiftx\shifty%
41        \global\edef\pgf@pattern@matrix{{\aa}{\ab}{\ba}{\bb}{\shiftx}{\shifty}}%
42      \endgroup%
43      % Now, build a name for the pattern
44      \pgfutil@tempcnta=\pgf@pattern@number\relax%
45      \advance\pgfutil@tempcnta by1\relax%
46      \xdef\pgf@pattern@number{\the\pgfutil@tempcnta}%
47      \xdef\pgf@marshal{\noexpand\pgfsys@declarepattern%
48        {\pgf@pattern@number}%
49        {\the\pgf@xa}{\the\pgf@ya}{\the\pgf@xb}{\the\pgf@yb}{\the\pgf@xc}{
          ↪  \the\pgf@yc}\pgf@pattern@matrix{\pgf@pattern@code}{\pgf@pat@type}}%
50    }%
51    \pgf@marshal%
52    \pgfsysprotocol@setcurrentprotocol\pgf@pattern@temp%
53  }%
```

2 行 转存当前的 protocol 缓存到宏 \pgf@pattern@temp.

7, 8 行 设置 xyz 坐标系的单位向量，见选项 /pgf/patterns/x，/pgf/patterns/y.

9 行 执行 \pgf@pat@declarebefore, 如果这个宏不等于 \pgfutil@empty 的话。这个宏的初始值等于 \pgfutil@empty, 见选项 /pgf/patterns/set up code.

12 行 开启边界盒子的计算。

14 行 清空当前的 protocol 缓存。

18 行 取消箭头。

19 行 执行 \pgf@pat@codebefore, 如果这个宏不等于 \pgfutil@empty 的话。这个宏的初始值等于 \pgfutil@empty.

20 行 执行 \pgf@pat@code, 见选项 /pgf/patterns/code.

21 行 执行 \pgf@pat@codeafter, 如果这个宏不等于 \pgfutil@empty 的话。这个宏的初始值等于 \pgfutil@empty.

24 行 将当前的 protocol 缓存保存到宏 \pgf@pattern@code.

27 行 执行 \pgf@pat@declareafter, 如果这个宏不等于 \pgfutil@empty 的话。这个宏的初始值等于 \pgfutil@empty.

28-36 行 处理 3 个点坐标，见选项 /pgf/patterns/bottom left, top right, tile size.

38 行 将变换矩阵设为单位矩阵。

39 执行 \pgf@pat@transformation 保存的变换命令,见选项 /pgf/patterns/tile transformation.

40 行 保存当前的变换矩阵。

41 行 将变换矩阵全局化。

44-46 行 为图样设置一个编号，这个编号全局地保存在 \pgf@pattern@number.

47-51 行 用系统层命令 \pgfsys@declarepattern 声明一个图样,其名称是宏 \pgf@pattern@number 保存的一个整数，可以看作是编号，这个编号实际是图样 ⟨*name*⟩ 的底层名称。

52 行 将 \pgf@pattern@temp 保存的 protocol 缓存作为当前的缓存。

\pgf@pat@declarebefore

初始值下这个宏等于 \pgfutil@empty.

\pgf@pat@declareafter

初始值下这个宏等于 \pgfutil@empty.

\pgf@pat@codebefore

初始值下这个宏等于 \pgfutil@empty.

\pgf@pat@codeafter

初始值下这个宏等于 \pgfutil@empty.

\pgfpatternalias{⟨name⟩}{⟨new name⟩}

本命令为已声明的图样 ⟨name⟩ 指定一个新的别名 ⟨new name⟩.

注意图样 ⟨name⟩ 应当是由 \pgfdeclarepattern →P. 615 声明的。

35.1.2 基本选项

/pgf/patterns/name=⟨name⟩ (no default)

本选项给定义的图样命名，⟨name⟩ 是其名称。通常，一个名称只能使用一次。

/pgf/patterns/type=⟨type⟩ (default uncolored)

本选项规定图样的类型，⟨type⟩ 的可用值是：

- uncolored 可变色的 (未着色的)，这个值导致宏 \pgf@pat@type 的定义：

```
\let\pgf@pat@type=\pgf@pat@type@uncolored% 保存整数 0
```

使得这个宏保存整数 0.

- colored 不可变色的 (已着色的)，这个值导致宏 \pgf@pat@type 的定义：

```
\let\pgf@pat@type=\pgf@pat@type@colored% 保存整数 1
```

使得这个宏保存整数 1.

- form only 可变色的，等效于 uncolored
- inherently colored 不可变色的，等效于 colored

/pgf/patterns/x=⟨dimension⟩ (default 1cm)

本选项规定 xyz 坐标系的 x 轴的单位长度，这个坐标系可以用作“图样砖坐标系”。

/pgf/patterns/y=⟨dimension⟩ (default 1cm)

本选项规定 xyz 坐标系的 y 轴的单位长度，这个坐标系可以用作“图样砖坐标系”。

/pgf/patterns/parameters={⟨comma separated list⟩} (default empty)

如果所声明的图样是 mutable 类型的，那么，那些能够修改图样外观的变量应当在本选项的参数 ⟨comma separated list⟩ 中列举出来，各个变量之间可以有分隔符号，也可以没有分隔符号，但最好采用统一的列举风格 (一般采用逗号作为分隔符号)。

注意，⟨comma separated list⟩ 中各个可展开的变量应当是“足够简单的”，因为选项 name=⟨name⟩ 的参数 ⟨name⟩, 与本选项的参数 ⟨comma separated list⟩ 将被用于构成控制序列

\csname pgf@pattern@name@meta@⟨name⟩⟨comma separated list⟩\endcsname

的名称。参数 ⟨comma separated list⟩ 的意义就在于值得这个控制序列的名称具有“独特性”。

参数 ⟨name⟩⟨comma separated list⟩ 这个组合被当作是一个整体，这个整体只被声明一次。当第 2 次使用这个组合时，直接利用这个组合，而不会再次声明这个组合。

本选项的定义是：

```
\pgfkeys{/pgf/patterns/.cd,
  parameters/.store in=\pgf@pat@parameters,
}
```

如果 \ifx\pgf@pat@parameters\pgfutil@empty 的判断为真，那么在绘制图样的代码中就没有需要展开的宏，一般情况下对这种图样的外观不能 (或者说无需) 做出任何修改，这种图样是 unmutable 类型的。

/pgf/patterns/defaults={⟨*default options*⟩} (default empty)

⟨*default options*⟩ 是选项列表，其中选项的路径应当是 /pgf/pattern keys, 这个选项列表会被保存到宏 \pgf@pat@defaults.

例如

```
defaults={
  points/.store in=\tikzstarpoints,points=5,
  radius/.store in=\tikzstarradius,radius=3pt,
  rotate/.store in=\tikzstarrotate,rotate=0,
  tile size/.store in=\tikztilesize,tile size=10pt,
}
```

将导致定义

```
\def\tikzstarpoints{5}
\def\tikzstarradius{3pt}
\def\tikzstarrotate{0}
\def\tikztilesize{10pt}
```

这些宏的定义会在绘制图样前被给出，因此在绘制图样的代码中可以使用这些宏，改变这些宏的值就可以修改图样的外观。

/pgf/patterns/bottom left=⟨*pgfpoint*⟩ (no default)

在“图样砖坐标系”中规定“边界盒子”的左下角点，⟨*pgfpoint*⟩ 是类似 \pgfpoint{-.1pt}{-.1pt} 这样的点。

/pgf/patterns/top right=⟨*pgfpoint*⟩ (no default)

在“图样砖坐标系”中规定“边界盒子”的右上角点，⟨*pgfpoint*⟩ 是类似 \pgfpoint{3.1pt}{3.1pt} 这样的点。

/pgf/patterns/tile size=⟨*pgfpoint*⟩ (no default)

在“图样砖坐标系”中规定“图样砖盒子”的右上角点，“图样砖盒子”的左下角点默认为“图样砖坐标系”的原点，⟨*pgfpoint*⟩ 是类似 \pgfpoint{3pt}{3pt} 这样的点。

/pgf/patterns/tile transformation=⟨*pgftransformation*⟩ (default empty)

⟨*pgftransformation*⟩ 会被保存到 \pgf@pat@transformation.

⟨*pgftransformation*⟩ 是类似 \pgftransformrotate{30} 这样的变换命令，是针对图样砖图形的。

/pgf/patterns/code=⟨*code*⟩ (no default)

⟨*code*⟩ 被保存到 \pgf@pat@code. ⟨*code*⟩ 是能够被“protocolled”的 PGF 命令，绘制图样砖图形，其中不能含有颜色代码或 node.

/pgf/patterns/set up code=⟨*code*⟩ (default empty)

⟨*code*⟩ 会被保存到 \pgf@pat@declarebefore. 在运行绘制图样的代码前执行 ⟨*code*⟩.

35.2 使用图样

文件《pgfcorepatterns.code.tex》定义的 \pgfsetfillpattern → P. 355 被保存为 \pgfsetfillpattern@old:

```
\let\pgfsetfillpattern@old=\pgfsetfillpattern
```

文件《pgflibrarypatterns.meta.code.tex》重定义了命令 \pgfsetfillpattern.

新命令的格式是

\pgfsetfillpattern{⟨*name*⟩[⟨*online options list*⟩]}{⟨*color*⟩}

其中的可选项 ⟨*online options list*⟩ 是一个选项列表，各个选项的路径是 /pgf/pattern keys, 作为图样名称 ⟨*name*⟩ 的选项。列表 ⟨*online options list*⟩ 会被保存到宏 \pgf@pat@onlineoptions.

新命令如下处理：

- 如果图样 ⟨*name*⟩ 是由文件《pgfcorepatterns.code.tex》的命令，如 \pgfdeclarepatternformonly → P. 352 等命令声明的，则用 \pgfsetfillpattern@old 处理它，此时可选项 ⟨*online options*⟩ 会被忽略。
- 如果图样 ⟨*name*⟩ 是由 \pgfdeclarepattern → P. 615 声明的，那么
  1. 清空 \pgf@pat@parameters

     ```
     \let\pgf@pat@parameters=\pgfutil@empty
     ```

  2. 执行 \pgf@pat@options 中保存的选项，这些选项包括 \pgfdeclarepattern{⟨*config*⟩} 中的 ⟨*config*⟩, 以及“number ..= 展开的 \pgf@pattern@number”。
  3. 检查 \pgf@pat@parameters 是否等于 \pgfutil@empty,
     - 如果等于 \pgfutil@empty (即 ⟨*config*⟩ 中没有给出选项 /pgf/patterns/parameters 的非空参数)，则什么也不做；
     - 如果不等于 \pgfutil@empty, 即 ⟨*config*⟩ 中给出了选项 /pgf/patterns/parameters 的非空参数，则
       (a) 用 \begingroup 开启一个组
       (b) 执行 \pgf@pat@defaults 保存的选项

       ```
       \pgf@pat@macroaskeys{/pgf/pattern keys/.cd}{\pgf@pat@defaults}
       ```

       见 /pgf/patterns/defaults → P. 620.
       (c) 执行 \pgf@pat@onlineoptions 保存的选项 ⟨*online options list*⟩

       ```
       \pgf@pat@macroaskeys{/pgf/pattern keys/.cd}{\pgf@pat@onlineoptions}
       ```

       (d) 定义 \pgf@pat@name

       ```
       \edef\pgf@pat@current@parameters{\pgf@pat@parameters}%
       \edef\pgf@pat@onlinename{\pgf@pat@onlinename\pgf@pat@current@parameters
       ↪  }%
       \let\pgf@pat@name=\pgf@pat@onlinename%
       ```

       \pgf@pat@name 保存的是 ⟨*name*⟩⟨*comma separated list*⟩, 其中的 ⟨*comma separated list*⟩ 是选项 /pgf/patterns/parameters → P. 619 的参数。此处的 ⟨*name*⟩⟨*comma separated list*⟩ 这个组合被当作是一个整体，这个整体只被声明一次。
       (e) 检查控制序列

       \csname pgf@pattern@name@meta@⟨*name*⟩⟨*comma separated list*⟩\endcsname

       是否有定义，如果有定义就什么也不做；如果没有定义 (等于 \relax)，就清空 \pgf@pat@parameters

       ```
       \let\pgf@pat@parameters=\pgfutil@empty
       ```

       然后执行 \pgf@declarepattern@meta → P. 616, 这会全局地定义控制序列

       \csname pgf@pattern@name@meta@⟨*name*⟩⟨*comma separated list*⟩\endcsname

       它保存的是选项

⟨*config*⟩，以及 number ..= 展开的 \pgf@pattern@number
与 \csname pgf@pattern@name@meta@⟨*name*⟩\endcsname 保存的内容相同。

(f) \endgroup 结束组

(g) 结束组前定义

```
\def\pgf@pat@onlinename{⟨name⟩⟨comma separated list⟩}
```

(h) 执行前述控制序列保存的选项

```
\pgf@pat@macroaskeys{/pgf/patterns/.cd}{\pgf@pat@options}
```

4. 检查 \pgf@pat@type 的值 \ifx\pgf@pat@type\pgf@pat@type@uncolored,
   – 如果 true, 则

   ```
   \pgf@pat@setpatternuncolored{\pgf@pat@number}{⟨color⟩}%
   ```

   其中 \pgf@pat@number 是 ⟨*name*⟩ 对应的编号。这个命令引入图样图形，并用颜色 ⟨*color*⟩ 为图样着色。如果使用了 xxcolor 宏包，那么颜色 ⟨*color*⟩ 会受到影响。
   – 如果 false, 则

   ```
   \pgfsys@setpatterncolored{\pgf@pat@number}%
   ```

   其中 \pgf@pat@number 是 ⟨*name*⟩ 对应的编号。这个命令引入图样图形。

见 /pgf/patterns/type →P. 619.

35.3 一些预定义的图样

文件《pgflibrarypatterns.meta.code.tex》预定义了一些图样，如 Lines，Hatch，Dots，Stars，这些图样都是 mutable 类型的图样，它们都可以接受一些选项来修改其外观，这些选项的路径都是 /pgf/pattern keys，详情参考手册。这些选项可以用作命令 \pgfsetfillpattern 中的图样名称的选项，来修改图样的外观。


```
\begin{tikzpicture}
  \pgfsetfillpattern{Stars[points=8,radius=2mm,distance=5mm]}{red}
  \filldraw (0,0) rectangle (2,2);
  \pgfsetfillpattern{Stars[points=5, yshift=-2mm]}{green!80!black}
  \pgfpathrectangle{\pgfpoint{3cm}{0cm}}{\pgfpoint{2cm}{2cm}}
  \pgfusepath{fill,draw}
\end{tikzpicture}
```

第三十六章　intersections

```
\usepgflibrary{intersections} % LaTeX and plain TeX and pure pgf
\usepgflibrary[intersections] % ConTeXt and pure pgf
\usetikzlibrary{intersections} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[intersections] % ConTeXt when using TikZ
```

使用 Ti*k*Z 计算两个路径的交点时要调用《tikzlibraryintersections.code.tex》, 而这个文件会调用《pgflibraryintersections.code.tex》.

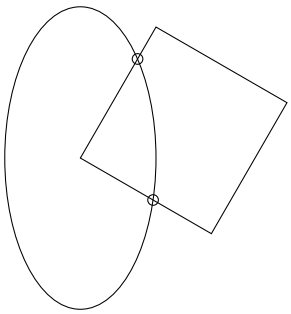
利用 PGF 的 intersections 库计算两个路径的交点的一般步骤是：

1. 获取第 1 个路径的软路径并保存到宏 \⟨*soft path A*⟩, 参考 \pgfgetpath→P. 275
2. 获取第 2 个路径的软路径并保存到宏 \⟨*soft path B*⟩
3. 如果希望沿着第 1 个路径对交点做排序，就执行 \pgfintersectionsortbyfirstpath
4. 如果希望沿着第 2 个路径对交点做排序，就执行 \pgfintersectionsortbysecondpath
5. 执行命令

   \pgfintersectionofpaths{⟨*code A*⟩}{⟨*code B*⟩}

   其中，在参数 ⟨*code A*⟩ 中可以使用各种代码，但一定要包含命令 \pgfsetpath\⟨*soft path A*⟩; 在参数 ⟨*code B*⟩ 中可以使用各种代码，但一定要包含命令 \pgfsetpath\⟨*soft path B*⟩. 也就是说，命令 \pgfintersectionofpaths 针对软路径来计算交点。

   参考 \pgfsetpath→P. 275, \pgfsyssoftpath@setcurrentpath→P. 222.
6. 计算交点后，宏 \pgfintersectionsolutions (局部地) 保存交点总数
7. 计算交点后，命令 \pgfpointintersectionsolution{⟨*number*⟩} 返回第 ⟨*number*⟩ 个交点，如果没有这个交点，就返回原点。本命令返回的第 ⟨*number*⟩ 个交点是非全局地定义的。



```
\begin{pgfpicture}
  \pgfintersectionofpaths
  {
    \pgfpathellipse{\pgfpointxy{0}{0}}{\pgfpointxy{1}{0}}{\pgfpointxy{0}{2}}
    \pgfgetpath\temppath
    \pgfusepath{stroke}
    \pgfsetpath\temppath
  }{
    \pgftransformrotate{-30}
    \pgfpathrectangle{\pgfpointorigin}{\pgfpointxy{2}{2}}
    \pgfgetpath\temppath
```

```
    \pgfusepath{stroke}
    \pgfsetpath\temppath
  }
  \foreach \s in {1,...,\pgfintersectionsolutions}
  {\pgfpathcircle{\pgfpointintersectionsolution{\s}}{2pt}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

36.1 基本思路

注意在文件《pgflibraryintersections.code.tex》的开头写道：

```
% Note: at the time of this writing, the library has quadratic runtime.
%   Experimentally, it performed well while computing ~12 intersections of two
%   plots, each with 600 samples. It failed when the number of samples exceeded 700.
```

一个路径由先后相继的数个“片段”(segment) 构成，片段是由 moveto, lineto, curveto, closepath 这几种操作创建的，在计算两个路径的交点时，只需要考虑由 lineto, curveto, closepath 操作创建的片段，例如，假设一个路径是

```
\path (0,0)--(1,3)--(2,0);
```

则此路径由片段 (lineto 操作创建的线段) `(0,0)--(1,3)` 和 `(1,3)--(2,0)` 构成。

一个路径的片段的编号从 0 开始，而交点的编号则从 1 开始。

把路径的各个片段看作是参数曲线。假设路径 A 由片段 $a_i(s), 0 \le s \le 1, i = 0, 1, 2, \cdots, n$ 组成，路径 B 由片段 $b_j(t), 0 \le t \le 1, j = 0, 1, 2, \cdots, m$ 组成。在计算路径 A 与 B 的交点时，依次计算

$$\begin{array}{llll} a_1(s) \cap b_1(t), & a_1(s) \cap b_2(t), & \cdots & a_1(s) \cap b_m(t), \\ a_2(s) \cap b_1(t), & a_2(s) \cap b_2(t), & \cdots & a_2(s) \cap b_m(t), \\ \cdots & \cdots & & \cdots \\ a_n(s) \cap b_1(t), & a_n(s) \cap b_2(t), & \cdots & a_n(s) \cap b_m(t) \end{array}$$

的交点，这要依次处理 $n \times m$ 对片段。在处理 $a_i \cap b_j$ 这对片段时，可能是处理线段与线段，线段与曲线，曲线与线段，曲线与曲线——有这 4 种情况——本程序库概括为线段与线段，曲线与曲线两种情况。

当两个路径的片段都是线段时，计算交点的过程直观一些，例如，假设路径 A 是

```
\path (1,0)--(1,3)--(2,3)--(2,0);
```

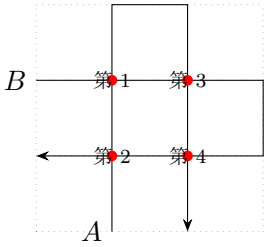
假设路径 B 是

```
\path (0,2)--(3,2)--(3,1)--(0,1);
```

那么求路径 A 与 B 的交点就是计算

$$\begin{aligned} (a_0 \cup a_1 \cup a_2) \cap (b_0 \cup b_1 \cup b_2) = & (a_0 \cap b_0) \cup (a_0 \cap b_1) \cup (a_0 \cap b_2) \cup \\ & (a_1 \cap b_0) \cup (a_1 \cap b_1) \cup (a_1 \cap b_2) \cup \\ & (a_2 \cap b_0) \cup (a_2 \cap b_1) \cup (a_2 \cap b_2) \end{aligned}$$

依次计算 $a_0 \cap b_0, a_0 \cap b_2, \cdots$ 所得交点依次是第一、第二、第三、第四个交点，如下图：




```
\begin{tikzpicture}
  \draw [help lines,dotted] (0,0) grid (3,3);
  \path [-Stealth,draw,name path=A] (1,0) node [left] {$A$}
  ↪  --(1,3)--(2,3)--(2,0);
  \path [-Stealth,draw,name path=B]
    (0,2) node [left] {$B$} --(3,2)--(3,1)--(0,1);
  \fill [fill=red,name intersections={of=A and B,name=i,total=\t}]
    \foreach \k in {1,...,\t}
    {(i-\k) circle (2pt) node[font=\footnotesize] {第\k}};
\end{tikzpicture}
```

当需要按照某个路径的走向对交点排序时，会为各个交点附加一个“时间值”。假设按照路径 A 的走向对交点排序，交点 $P_{i,s}$ 位于路径 A 的片段 a_i 上，即 $P_{i,s} = a_i(s)$, 那么交点 $P_{i,s}$ 的“时间值”是 $i + s$. 按照“时间值”的大小来对交点排序。

把片段看作是参数曲线主要是为了照顾“时间值”这个概念，未必真的要解一个参数方程组来求出交点。当检查线段与线段的交点时，使用解一次方程组的办法寻找交点。当检查控制曲线与控制曲线的交点时，使用几何化的方法，利用控制点寻找交点 (当然这也可以看作是求解 3 次方程组的几何方法)。

36.2 代码分析

36.2.1 辅助代码

```
\usepgflibrary{fpu}%
```

调用 fpu 程序库，在计算线段与曲线、曲线与曲线的交点时使用这个库。

```
\newcount\pgf@intersect@solutions
```

这个计数器用于记录直到当前所计算出来的交点个数 (不计重复的交点)。

```
\newif\ifpgf@intersect@sort
\newif\ifpgf@intersect@sort@by@second@path

\def\pgfintersectionsortbyfirstpath{%
  \pgf@intersect@sorttrue%
  \pgf@intersect@sort@by@second@pathfalse%
}%

\def\pgfintersectionsortbysecondpath{%
  \pgf@intersect@sorttrue%
  \pgf@intersect@sort@by@second@pathtrue%
}%
```

\pgfpointintersectionsolutions

此命令在 \pgfintersectionofpaths 的内部被局部地定义：

```
\edef\pgfintersectionsolutions{\the\pgf@intersect@solutions}
```

其值是最终得到的交点个数。在得到各个交点之后、对交点做排序之前，这个宏就被定义了。

\pgfpointintersectionsolution{⟨*number*⟩}

此命令的结果是：

- 如果 ⟨*number*⟩ 小于 1 或者大于当前的 \pgfintersectionsolutions 值，则此命令得到原点，也就是保存在

  ```
  \pgfpoint@intersect@solution@orgin
  ```

  中的值。

- 否则，得到第 ⟨*number*⟩ 个交点，也就是保存在

  ```
  \csname pgfpoint@intersect@solution@⟨number⟩\endcsname
  ```

  中的值。

\csname pgfpoint@g@intersect@solution@⟨*number*⟩\endcsname

在执行 \pgf@intersectionoflines, \pgf@@@intersectionofcurves 的过程中，这个命令被全局地定义。

本命令全局地保存当前计算出来的 (无重复的) 交点，即第 \the\pgf@intersect@solutions 个交点。

```
\expandafter\global\expandafter\let\csname pgfpoint@g@intersect@solution@
↪  \the\pgf@intersect@solutions\endcsname=\pgf@intersect@solution@candidate
```

其中的 \pgf@intersect@solution@candidate 保存了当前计算出来的交点坐标。

\csname pgfpoint@intersect@solution@⟨*number*⟩\endcsname

在执行 \pgfintersectionofpaths 的过程中，这个命令被 (非全局地) 定义。

本命令保存第 \pgfmathcounter 个交点。

```
\pgfutil@namelet{pgfpoint@intersect@solution@\pgfmathcounter}%
    {pgfpoint@g@intersect@solution@\pgfmathcounter}%
```

命令 \pgfutil@namelet, \pgfutil@namedef 的定义见文件《pgfutil-common.tex》.

\pgf@intersect@solution@candidate

此命令的定义是

```
\pgfextract@process\pgf@intersect@solution@candidate{}
或者
\pgfextract@process\pgf@intersect@solution@candidate{关于 \pgf@x, \pgf@y 的计算}
```

这个命令用于保存寄存器 \pgf@x, \pgf@y 的值，作为当前计算出来的交点坐标。此命令保存交点坐标但并不直接引起 \pgf@intersect@solutions 的值的增长，只有在确定此交点与之前得到的交点无重复后，才正式接受此交点为“一个交点”，然后再让 \pgf@intersect@solutions 的值加 1.

检查交点是否重复的命令是 \pgf@ifsolution@duplicate → P. 633.

\pgf@intersect@path@reset@a

执行此命令后会定义两个宏

```
\def\pgf@intersect@path@reset@a{%
    \def\pgf@intersect@time@offset{0}%
    \def\pgf@intersect@time@a{}%
}%
```

\pgf@intersect@time@offset

执行 \pgf@intersect@path@reset@a 后会定义这个宏，它用于保存第一个路径的当前片段的编号。

\pgf@intersect@time@a

见前一命令。

假设当前得到的交点是第一个路径的第 i 片段上的 s 点，记为 $a_i(s)$; 也是第二个路径的第 j 片段上的 t 点，记为 $b_j(t)$, 于是 $a_i(s) = b_j(t)$:

- 宏 \pgf@intersect@time@offset 的值是片段 a_i 的编号 i
- 宏 \pgf@intersect@time@offset@b 的值是片段 b_j 的编号 j
- 宏 \pgf@intersect@time@a 的值应该是 $i+s$, 这个值看作是此交点在第一个路径上对应的时间。这个宏的值可能等于 \pgfutil@empty.

- 宏 \pgf@intersect@time@b 的值应该是 $j+t$, 这个值看作是此交点在第二个路径上对应的时间。这个宏的值可能等于 \pgfutil@empty.

\pgf@intersect@path@reset@b

执行此命令后会定义两个宏

```
\def\pgf@intersect@path@reset@b{%
    \def\pgf@intersect@time@offset@b{0}%
    \def\pgf@intersect@time@b{}%
}%
```

\pgf@intersect@time@offset@b

执行 \pgf@intersect@path@reset@b 后定义此命令，它用于保存第二个路径的当前片段的编号。

\pgf@intersect@time@b

见前面的命令。

\pgf@intersection@set@properties{⟨*code*⟩}

本命令用于定义 \csname pgf@intersect@solution@props@\pgfmathcounter\endcsname.

```
\def\pgf@intersection@set@properties#1{%
    \pgfutil@namedef{pgf@intersect@solution@props@\pgfmathcounter}{#1}%
}%
```

而 \pgfutil@namedef 的定义是

```
\def\pgfutil@namedef#1{\expandafter\def\csname #1\endcsname}
```

\pgf@intersection@store@properties{⟨*prefix*⟩}

本命令的定义是：

```
\def\pgf@intersection@store@properties#1{%
    \expandafter\xdef\csname #1@props\endcsname{{\pgf@intersect@time@offset}{
    ↪  \pgf@intersect@time@offset@b}{\pgf@intersect@time@a}{\pgf@intersect@time@b
    ↪  }}%
}%
```

本命令用法如下：

```
\pgf@intersection@store@properties{%
  pgfpoint@g@intersect@solution@\the\pgf@intersect@solutions}
```

导致全局定义

```
\expandafter\xdef%
  \csname pgfpoint@g@intersect@solution@\the\pgf@intersect@solutions @props
  ↪  \endcsname%
  {{\pgf@intersect@time@offset}{\pgf@intersect@time@offset@b}{
  ↪  \pgf@intersect@time@a}{\pgf@intersect@time@b}}
```

再执行

```
\edef\pgf@marshal{\noexpand\pgf@intersection@set@properties{%
  \csname pgfpoint@g@intersect@solution@\pgfmathcounter @props\endcsname}}%
\pgf@marshal
```

就把

```
\csname pgf@intersect@solution@props@\pgfmathcounter\endcsname
```

定义为

```
\csname pgfpoint@g@intersect@solution@\pgfmathcounter @props\endcsname
```

的展开值。

\csname pgfpoint@g@intersect@solution@⟨*number*⟩@props\endcsname

这个宏由 \pgf@intersection@store@properties →P. 627 全局地定义，这个命令全局地保存第 ⟨*number*⟩ 个交点的在路径上的片段编号、时间数据，其定义内容是：

```
{\pgf@intersect@time@offset}{\pgf@intersect@time@offset@b}{\pgf@intersect@time@a}{
↪  \pgf@intersect@time@b}
```

不过：

- 在 \ifpgf@intersect@sort 的判断为真的前提下，这个命令保存的 \pgf@intersect@time@offset 和 \pgf@intersect@time@a, 即第 ⟨*number*⟩ 个交点的，属于第一个路径 (即由 \pgfintersectionsortbyfirstpath 或 \pgfintersectionsortbysecondpath 选定的路径) 的片段编号、时间数据，才是具体的数值；而 \pgf@intersect@time@offset@b 和 \pgf@intersect@time@b 都是空的，即等于 \pgfutil@empty.
- 在 \ifpgf@intersect@sort 的判断为假的前提下，这个命令保存的 \pgf@intersect@time@offset 是具体的数值,其余 3 个参数都是空的,即等于 \pgfutil@empty.

\csname pgf@intersect@solution@props@⟨*number*⟩\endcsname

这个宏由 \pgf@intersection@set@properties →P. 627 定义，它保存第 ⟨*number*⟩ 个交点的编号、时间数据，与 \csname pgfpoint@g@intersect@solution@⟨*number*⟩@props\endcsname 保存的内容一样，不过这个宏不是全局定义的。

\csname pgf@g@intersect@solution@⟨*number*⟩@time@a\endcsname

在执行 \pgf@intersectionoflines, \pgf@@@intersectionofcurves 的过程中，这个命令被全局地定义。

本命令的值是当前 \pgf@intersect@time@a 的值，即第 ⟨*number*⟩ 个交点在第一个路径上的时间数据。

只有在 \ifpgf@intersect@sort 的判断为真的前提下，也就是使用 \pgfintersectionsortbyfirstpath 或 \pgfintersectionsortbysecondpath 的前提下这个命令才会被全局地定义。在 \ifpgf@intersect@sort 的判断为假的前提下，这个命令的值是空的，即等于 \pgfutil@empty.

■
```
\expandafter\global\expandafter\let\csname pgf@g@intersect@solution@
↪  \the\pgf@intersect@solutions @time@a\endcsname=\pgf@intersect@time@a
```

\csname pgf@intersect@solution@⟨*number*⟩@time@a\endcsname

在执行 \pgfintersectionofpaths 的过程中，在 \ifpgf@intersect@sort 的判断为真的前提下，这个命令被定义。

本命令的值是与 \csname pgf@g@intersect@solution@⟨*number*⟩@time@a\endcsname 保存的内容一样，不过这个命令不是全局定义的。当按照路径的走向来对各个交点做排序时会用到这个命令。

■
```
\ifpgf@intersect@sort%
    \pgfutil@namelet{pgf@intersect@solution@\pgfmathcounter @time@a}%
        {pgf@g@intersect@solution@\pgfmathcounter @time@a}%
\fi%
```

\pgfintersectionsolutionsortbytime@swap{⟨*one*⟩}{⟨*two*⟩}

本命令交换 ⟨*one*⟩ 与 ⟨*two*⟩ 的定义内容。

■
```
\def\pgfintersectionsolutionsortbytime@swap#1#2{%
    \pgfutil@namelet{pgf@intersect@temp}{#1}%
    \pgfutil@namelet{#1}{#2}%
```

```
    \pgfutil@namelet{#2}{pgf@intersect@temp}%
}%
```

\pgfintersectionsolutionsortbytime

本命令是 \pgfintersectionofpaths 的处理过程的最后一步 (如果 \ifpgf@intersect@sort 的判断为 true 的话)。

本命令以各个交点的 \csname pgf@intersect@solution@⟨*number*⟩@time@a\endcsname 数据为指标，按照从小到大的次序，将各交点排序。

本命令通过对相邻两项做比较、做换位来实现排序，不过一般要多次使用本命令才能完成排序。例如，将 $\{3,2,1\}$ 按从小到大的次序排序，就是：

- 第一次执行 \pgfintersectionsolutionsortbytime：$\{3,2,1\} \to \{2,3,1\} \to \{2,1,3\}$
- 第二次执行 \pgfintersectionsolutionsortbytime：$\{2,1,3\} \to \{1,2,3\} \to \{1,2,3\}$
- 第三次执行 \pgfintersectionsolutionsortbytime：$\{1,2,3\} \to \{1,2,3\} \to \{1,2,3\}$

尽管第二次执行本命令后已完成排序，但因为 \pgf@intersect@solutions@sortfinishfalse，所以还要第三次执行本命令。

排序过程只用到了交点在第一个路径上的时间数据，这是因为在命令 \pgfintersectionofpaths 的处理过程的早期有这种设置：

```
\ifpgf@intersect@sort@by@second@path%
    \let\pgf@intersect@temp=\pgf@intersect@path@a%
    \let\pgf@intersect@path@a=\pgf@intersect@path@b%
    \let\pgf@intersect@path@b=\pgf@intersect@temp%
\fi%
```

也就是说，按照哪个路径走向来排序，哪个路径就是“第一个路径”。如果不指定按照哪个路径走向来排序，那么“先给出的路径就是第一个路径”。

此命令的定义是：

```
\newif\ifpgf@intersect@solutions@sortfinish
\def\pgfintersectionsolutionsortbytime{%
    \pgf@intersect@solutions@sortfinishtrue%
    \pgfmathloop%
        \ifnum\pgfmathcounter<\pgfintersectionsolutions\relax%
            \pgfutil@tempcnta=\pgfmathcounter%
            \advance\pgfutil@tempcnta by1\relax%
            \ifdim\csname pgf@intersect@solution@\pgfmathcounter @time@a\endcsname
            ↪  pt>%
                \csname pgf@intersect@solution@\the\pgfutil@tempcnta @time@a
                ↪  \endcsname pt\relax%
                \pgf@intersect@solutions@sortfinishfalse%
                %
% 将 \csname pgfpoint@intersect@solution@\pgfmathcounter\endcsname 与
%\csname pgfpoint@intersect@solution@\the\pgfutil@tempcnta\endcsname 互换
                \pgfintersectionsolutionsortbytime@swap
                ↪  {pgfpoint@intersect@solution@\pgfmathcounter}%
                    {pgfpoint@intersect@solution@\the\pgfutil@tempcnta}%
                %
% 将 \csname pgf@intersect@solution@\pgfmathcounter @time@a\endcsname 与
%\csname pgf@intersect@solution@\the\pgfutil@tempcnta @time@a\endcsname 互换
                \pgfintersectionsolutionsortbytime@swap{pgf@intersect@solution@
                ↪  \pgfmathcounter @time@a}%
                    {pgf@intersect@solution@\the\pgfutil@tempcnta @time@a}%
                %
```

```
% 将 \csname pgf@intersect@solution@props@\pgfmathcounter\endcsname 与
%\csname pgf@intersect@solution@props@\the\pgfutil@tempcnta\endcsname 互换
                \pgfintersectionsolutionsortbytime@swap
                ↪  {pgf@intersect@solution@props@\pgfmathcounter}%
                   {pgf@intersect@solution@props@\the\pgfutil@tempcnta}%
            \fi%
    \repeatpgfmathloop%
    \ifpgf@intersect@solutions@sortfinish%
    \else%
        \expandafter\pgfintersectionsolutionsortbytime%
    \fi%
}%
```

命令 \pgfintersectionsolutionsortbytime@swap 做互换的方法是：假设 $A = x$, $B = y$, 逐步令 $T = A = x$, $A = B = y$, $B = T = x$, 实现 A 与 B 值的互换，它实际调用 \pgfutil@namelet 来做互换，而 \pgfutil@namelet 实际调用 \let 来工作 (没有\global)。

\pgfintersectiongetsolutionsegmentindices{⟨num⟩}{⟨macro A⟩}{⟨macro B⟩}

此命令的效果是：检查 ⟨num⟩ 的值，

- 如果 ⟨num⟩ 小于 1 或者大于当前的 \pgfintersectionsolutions 值，则宏 ⟨macro A⟩, ⟨macro B⟩ 都等于 \pgfutil@empty, 在文件《pgfutil-common.tex》中有定义：

  ```
  \def\pgfutil@empty{}
  ```

- 否则：假设第 ⟨num⟩ 个交点位于第一个路径的第 i 片段 a_i 上，并且位于第二个路径的第 j 片段 b_j 上，那么宏 ⟨macro A⟩ 的值等于片段编号 i, 宏 ⟨macro B⟩ 的值等于片段编号 j.

\pgfintersectiongetsolutiontimes{⟨num⟩}{⟨macro A⟩}{⟨macro B⟩}

此命令的效果是：检查 ⟨num⟩ 的值，

- 如果 ⟨num⟩ 小于 1 或者大于当前的 \pgfintersectionsolutions 值，则宏 ⟨macro A⟩, ⟨macro B⟩ 都等于 \pgfutil@empty.
- 否则：
  把第 ⟨num⟩ 个交点在第一个路径上对应的时间，即该交点对应的宏 \pgf@intersect@time@a 的值，保存到宏 ⟨macro A⟩ 中；把第 ⟨num⟩ 个交点在第二个路径上对应的时间，即该交点对应的宏 \pgf@intersect@time@b 的值，保存到宏 ⟨macro B⟩ 中。
  但如果宏 \pgf@intersect@time@a 的值是 \pgfutil@empty, 就把宏 \pgf@intersect@time@offset 的值保存到宏 ⟨macro A⟩ 中；如果宏 \pgf@intersect@time@b 的值是 \pgfutil@empty, 就把宏 \pgf@intersect@time@offset@b 的值保存到宏 ⟨macro B⟩ 中。

\pgf@intersect@boundingbox@reset

此命令的定义是：

■
```
\def\pgf@intersect@boundingbox@reset{%
    \pgf@xa=16000pt\relax%
    \pgf@ya=16000pt\relax%
    \pgf@xb=-16000pt\relax%
    \pgf@yb=-16000pt\relax%
}%
```

\pgf@intersect@boundingbox@update{⟨code⟩}

这里的 ⟨code⟩ 应该是能给出尺寸寄存器 \pgf@x, \pgf@y 的值的代码，例如，\pgfpoint{⟨x⟩}{⟨y⟩}.
本命令做重定义：

- \pgf@xa = min{\pgf@x, \pgf@xa}

- \pgf@ya = min{\pgf@y, \pgf@ya}
- \pgf@xb = max{\pgf@x, \pgf@xb}
- \pgf@yb = max{\pgf@y, \pgf@yb}

此命令的定义是：

```
\def\pgf@intersect@boundingbox@update#1{%
    #1\relax%
    \ifdim\pgf@x<\pgf@xa\pgf@xa=\pgf@x\fi%
    \ifdim\pgf@y<\pgf@ya\pgf@ya=\pgf@y\fi%
    \ifdim\pgf@x>\pgf@xb\pgf@xb=\pgf@x\fi%
    \ifdim\pgf@y>\pgf@yb\pgf@yb=\pgf@y\fi%
}%
```

\pgf@intersect@boundingbox@assign@a

本命令对 \pgf@intersect@boundingbox@a 作定义。

使用本命令前要给出 \pgf@xa, \pgf@ya, \pgf@xb, \pgf@yb 的值。

```
\def\pgf@intersect@boundingbox@assign@a{%
    \edef\pgf@intersect@boundingbox@a{%
        % lower left:
        \noexpand\pgf@xb=\the\pgf@xa\space%
        \noexpand\pgf@yb=\the\pgf@ya\space%
        % upper right:
        \noexpand\pgf@xc=\the\pgf@xb\space%
        \noexpand\pgf@yc=\the\pgf@yb\space%
    }%
}%
```

\pgf@intersect@boundingbox@assign@b

本命令对 \pgf@intersect@boundingbox@b 作定义。

使用本命令前要给出 \pgf@xa, \pgf@ya, \pgf@xb, \pgf@yb 的值。

```
\def\pgf@intersect@boundingbox@assign@b{%
    \edef\pgf@intersect@boundingbox@b{%
        % lower left:
        \noexpand\global\noexpand\pgf@x=\the\pgf@xa\space%
        \noexpand\global\noexpand\pgf@y=\the\pgf@ya\space%
        % upper right:
        \noexpand\pgf@xa=\the\pgf@xb\space%
        \noexpand\pgf@ya=\the\pgf@yb\space%
    }%
}%
```

以上几个命令用于确定一组坐标点的边界矩形的左下角点、右上角点。假设 $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, 那么

```
1 \pgf@intersect@boundingbox@reset
2 \pgf@intersect@boundingbox@update{⟨P1⟩}
3 \pgf@intersect@boundingbox@update{⟨P2⟩}
4 \pgf@intersect@boundingbox@assign@a
```

的结果是：

第二行 做赋值：

- \pgf@xa 等于 $\min\{16000\text{pt}, x_1\}$
- \pgf@ya 等于 $\min\{16000\text{pt}, y_1\}$

- \pgf@xb 等于 $\max\{-16000\text{pt}, x_1\}$
- \pgf@yb 等于 $\max\{-16000\text{pt}, y_1\}$

第三行 做赋值：

- \pgf@xa 等于 $\min\{x_2, x_1\}$
- \pgf@ya 等于 $\min\{y_2, y_1\}$
- \pgf@xb 等于 $\max\{x_2, x_1\}$
- \pgf@yb 等于 $\max\{y_2, y_1\}$

此时点组 $\{P_1, P_2\}$ 的边界矩形的左下角点的坐标分量是 \pgf@xa 与 \pgf@ya; 右上角点的坐标分量是 \pgf@xb 与 \pgf@yb.

第四行 令点组 $\{P_1, P_2\}$ 的边界矩形的左下角点的坐标分量是 \pgf@xb 与 \pgf@yb; 右上角点的坐标分量是 \pgf@xc 与 \pgf@yc, 将这四个坐标分量 (尺寸寄存器) 保存在 \pgf@intersect@boundingbox@a 中。

假如第四行执行的是

```
\pgf@intersect@boundingbox@assign@b
```

那么点组 $\{P_1, P_2\}$ 的边界矩形的左下角点的坐标分量是 \pgf@x 与 \pgf@y; 右上角点的坐标分量是 \pgf@xa 与 \pgf@ya, 这四个坐标分量保存在 \pgf@intersect@boundingbox@b 中。

\pgf@intersect@boundingbox@ifoverlap{⟨*first code*⟩}{⟨*second code*⟩}

本命令判断两个矩形是否有重叠。使用本命令前要给出：第一个矩形的左下角点 (\pgf@xb,\pgf@yb)，右上角点 (\pgf@xc,\pgf@yc)；第二个矩形的左下角点 (\pgf@x,\pgf@y)，右上角点 (\pgf@xa,\pgf@ya).

在两个矩形无重叠的情况下执行 \pgf@intersect@next (等于 \pgfutil@secondoftwo).

在两个矩形有重叠的情况下执行 \pgf@intersect@next (等于 \pgfutil@firstoftwo).

命令 \pgfutil@firstoftwo, \pgfutil@secondoftwo 见文件《pgfutil-common.tex》.

■

```
\def\pgf@intersect@boundingbox@ifoverlap{%
    \def\pgf@intersect@next{\pgfutil@secondoftwo}%
    %
    \ifdim\pgf@xa<\pgf@xb%
    \else%
        \ifdim\pgf@x>\pgf@xc%
        \else%
            \ifdim\pgf@ya<\pgf@yb%
            \else%
                \ifdim\pgf@y>\pgf@yc%
                \else%
                    \def\pgf@intersect@next{\pgfutil@firstoftwo}%
                \fi
            \fi
        \fi
    \fi
    \pgf@intersect@next
}%
```

\pgfintersectiontolerance

定义是：

```
\def\pgfintersectiontolerance{0.1pt}%
```

\pgfintersectiontoleranceupperbound

定义是：

```
\def\pgfintersectiontoleranceupperbound{1pt}%
```

\pgfintersectiontolerancefactor

定义是：

```
\def\pgfintersectiontolerancefactor{0.1}%
```

\pgf@intersect@boundingbox@ifoverlap@upperbound{⟨*first code*⟩}{⟨*second code*⟩}

本命令判断两个矩形是否有重叠。使用本命令前要给出：第一个矩形的左下角点 (\pgf@xb,\pgf@yb)，右上角点 (\pgf@xc,\pgf@yc)；第二个矩形的左下角点 (\pgf@x,\pgf@y)，右上角点 (\pgf@xa,\pgf@ya)。判断过程使用了阈限值 \pgfintersectiontolerance→P. 632，也就是说，把第二个矩形的边界向外扩展尺寸 \pgfintersectiontolerance，再判断两个矩形是否有重叠。

在两个矩形无重叠的情况下执行 \pgf@intersect@next (等于 \pgfutil@secondoftwo)。

在两个矩形有重叠的情况下执行 \pgf@intersect@next (等于 \pgfutil@firstoftwo)。

■

```
\def\pgf@intersect@boundingbox@ifoverlap@upperbound{%
    \begingroup
    \def\pgf@intersect@next{\pgfutil@secondoftwo}%
    %
    \advance\pgf@xa by+\pgfintersectiontolerance\relax
    \ifdim\pgf@xa<\pgf@xb%
    \else%
        \global\advance\pgf@x by-\pgfintersectiontolerance\relax
        \ifdim\pgf@x>\pgf@xc%
        \else%
            \advance\pgf@ya by\pgfintersectiontolerance\relax
            \ifdim\pgf@ya<\pgf@yb%
            \else%
                \global\advance\pgf@y by-\pgfintersectiontolerance\relax
                \ifdim\pgf@y>\pgf@yc%
                \else%
                    \def\pgf@intersect@next{\pgfutil@firstoftwo}%
                \fi
            \fi
        \fi
    \fi
    \expandafter
    \endgroup
    \pgf@intersect@next
}%
```

此命令使用 \begingroup, \endgroup 设置一个组，限制寄存器运算的范围。

\pgf@ifsolution@duplicate{⟨*code*⟩}{⟨*first code*⟩}{⟨*second code*⟩}

这里的 ⟨*code*⟩ 应该是能确定寄存器 \pgf@x, \pgf@y 的值的代码，例如，\pgfpoint{⟨*x*⟩}{⟨*y*⟩}，实际用的是 \pgf@intersect@solution@candidate→P. 626。

本命令检查 ⟨*code*⟩ 所确定的坐标点 (即 \pgf@x, \pgf@y 的值) 是否与之前得到的某个交点重合。这个检查是用循环

\pgfmathloop ... \repeatpgfmathloop

实现的，循环开头定义计数宏 \pgfmathcounter 来控制循环过程。参考《pgfmathutil.code.tex》。宏 \pgfmathcounter 的值从 1 变到 \pgf@intersect@solutions (之前得出的交点总数)，对于每个 \pgfmathcounter 值，检查 ⟨*code*⟩ 所确定的坐标点是否与第 \pgfmathcounter 个交点重合。

循环内使用命令 \pgf@ifsolution@duplicate@ 来判断两个点是否重合。
如果有重合则执行 \pgf@intersect@next (等于 \pgfutil@firstoftwo).
如果无重合则执行 \pgf@intersect@next (等于 \pgfutil@secondoftwo).

```
\def\pgf@ifsolution@duplicate#1{%
    #1%
    \pgf@xa=\pgf@x%
    \pgf@ya=\pgf@y%
    \let\pgf@intersect@next=\pgfutil@secondoftwo%
    \pgfmathloop%
        \ifnum\pgfmathcounter>\pgf@intersect@solutions\relax%
        \else%
            \pgf@ifsolution@duplicate@{\pgfmathcounter}%
    \repeatpgfmathloop%
    \pgf@intersect@next%
}%
```

\pgf@ifsolution@duplicate@{⟨*number*⟩}

本命令用在 \pgf@ifsolution@duplicate 的内部循环中，检查目前得到的交点是否与之前得到的第 ⟨*number*⟩ 个交点重合。假设目前计算出来的交点是 (x_c, y_c), 之前计算出来的某个交点是 (x_k, y_k), 如果

$$\texttt{\textbackslash pgfintersectiontolerancefactor}|x_k - x_c| < \texttt{\textbackslash pgfintersectiontolerance}$$

并且

$$\texttt{\textbackslash pgfintersectiontolerancefactor}|y_k - y_c| < \texttt{\textbackslash pgfintersectiontolerance}$$

就认为 (x_c, y_c) 与 (x_k, y_k) 重合。

```
\def\pgf@ifsolution@duplicate@#1{%
    \pgf@process{\csname pgfpoint@g@intersect@solution@#1\endcsname}%
    \advance\pgf@x by-\pgf@xa%
    \advance\pgf@y by-\pgf@ya%
    \ifdim\pgf@x<0pt\relax\pgf@x=-\pgf@x\fi%
    \ifdim\pgf@y<0pt\relax\pgf@y=-\pgf@y\fi%
    %
    \pgf@x=\pgfintersectiontolerancefactor\pgf@x%
    \pgf@y=\pgfintersectiontolerancefactor\pgf@y%
    \ifdim\pgf@x<\pgfintersectiontolerance\relax%
        \ifdim\pgf@y<\pgfintersectiontolerance\relax%
            \let\pgf@intersect@next=\pgfutil@firstoftwo%
        \fi%
    \fi%
}%
```

可见执行 \pgf@ifsolution@duplicate 后，寄存器 \pgf@x, \pgf@y, \pgf@xa, \pgf@ya 的值会被改变。假设目前计算出来的交点是 (x_c, y_c), 此前被正式确认的交点是第 e 个交点 (x_e, y_e), 注意此时 (x_c, y_c) 尚未被正式确认为一个交点，所以 e 就是当前的 \pgf@intersect@solutions 的值；那么

- \pgf@xa 的值变成 x_c
- \pgf@ya 的变成 y_c
- \pgf@x 的变成 $\texttt{\textbackslash pgfintersectiontolerancefactor}|x_c - x_e|$
- \pgf@y 的变成 $\texttt{\textbackslash pgfintersectiontolerancefactor}|y_c - y_e|$

这 4 个寄存器的值会被用到 ⟨*first code*⟩ 或者 ⟨*second code*⟩ 阿中。
注意按照 \pgf@process 的定义，\pgf@process 在一个花括号组内执行动作，其最终作用只是全局

地为 \pgf@x, \pgf@y 赋值。参照文件《pgfcorepoints.code.tex》。

\pgf@float@adapter@setxy

使用本命令前应该提供 \pgf@x, \pgf@y 的值。这个命令把 \pgf@x 的数值部分转成浮点数保存在 \pgf@fpu@x 中；把 \pgf@y 的数值部分转成浮点数保存在 \pgf@fpu@y 中。

```
\def\pgf@float@adapter@setxy{%
    \pgfmathfloatparsenumber{\pgf@sys@tonumber\pgf@x}\let\pgf@fpu@x=\pgfmathresult
    \pgfmathfloatparsenumber{\pgf@sys@tonumber\pgf@y}\let\pgf@fpu@y=\pgfmathresult
}%
```

\pgf@float@adapter@mult{⟨*macro*⟩}={⟨*fixed point*⟩}*{⟨*float point*⟩}

本命令将浮点数 ⟨*float point*⟩ 与定点数 ⟨*fixed point*⟩ 的乘积 (浮点数) 保存在 ⟨*macro*⟩ 中。

```
\def\pgf@float@adapter@mult#1=#2*#3{%
    \pgfmathfloatmultiplyfixed@{#3}{#2}%
    \let#1=\pgfmathresult
}%
```

\pgf@float@adapter@advance{⟨*macro*⟩}by{⟨*fixed point*⟩}*{⟨*float point*⟩}

本命令将浮点数 ⟨*float point*⟩ 与定点数 ⟨*fixed point*⟩ 的乘积 (浮点数)，再加上 ⟨*macro*⟩ 的和保存在 ⟨*macro*⟩ 中。

```
\def\pgf@float@adapter@advance#1by#2*#3{%
    \pgfmathfloatmultiplyfixed@{#3}{#2}%
    \let\pgfutil@temp=\pgfmathresult
    \pgfmathfloatadd@{#1}{\pgfutil@temp}%
    \let#1=\pgfmathresult
}%
```

\pgf@float@adapter@tostring{⟨*macro*⟩}

本命令将浮点数 ⟨*macro*⟩ 转换为定点数，然后再保存在 ⟨*macro*⟩ 中。

```
\def\pgf@float@adapter@tostring#1{%
    \pgfmathfloattofixed{#1}\edef#1{\pgfmathresult pt }%
}%
```

36.2.2 命令 \pgfintersectionofpaths

有定义 \long\def\pgfintersectionofpaths#1#2{ ... }, 其中变量 #1 代表第一个路径的代码，变量 #2 代表第二个路径的代码。

命令 \pgfintersectionofpaths{⟨*code of path A*⟩}{⟨*code of path B*⟩} 的处理过程是:

1. 将 ⟨*code of path A*⟩ 对应的软路径保存到命令 \pgf@intersect@path@a 中。

```
\begingroup%
    \pgfinterruptpath%
    #1%
    \pgfgetpath\pgf@intersect@path@a%
    \global\let\pgf@intersect@path@temp=\pgf@intersect@path@a%
    \endpgfinterruptpath%
\endgroup%
\let\pgf@intersect@path@a=\pgf@intersect@path@temp%
```

2. 将 ⟨*code of path B*⟩ 对应的软路径保存到命令 \pgf@intersect@path@b 中。

3. 执行 \pgf@intersect@solutions=0\relax.

4. 执行 \pgf@intersect@path@reset@a, 定义两个宏

```
\def\pgf@intersect@path@reset@a{%
    \def\pgf@intersect@time@offset{0}%
    \def\pgf@intersect@time@a{}%
}%
```

5. 执行

```
\ifpgf@intersect@sort@by@second@path%
  \let\pgf@intersect@temp=\pgf@intersect@path@a%
  \let\pgf@intersect@path@a=\pgf@intersect@path@b%
  \let\pgf@intersect@path@b=\pgf@intersect@temp%
\fi%
```

6. 执行 \pgfprocessround\pgf@intersect@path@a\pgf@intersect@path@a.

   如果有圆角选项，命令 \pgfprocessround 把路径的尖角变成圆角，路径仍然是软路径，仍然保存在 \pgf@intersect@path@a 中。

7. 执行 \pgfprocessround\pgf@intersect@path@b\pgf@intersect@path@b.

8. 执行 \let\pgf@intersect@token@after=\pgf@intersect@path@process@a.

9. 执行 \expandafter\pgf@intersectionofpaths\pgf@intersect@path@a\pgf@stop.

   这个执行过程计算两个路径的交点，其处理过程是：

   (a) 按照命令 \pgf@intersectionofpaths 的定义：

■

```
\def\pgf@intersectionofpaths#1{%
    \ifx#1\pgf@stop%
        \let\pgf@intersect@next=\relax%
    \else%
        \ifx#1\pgfsyssoftpath@movetotoken%
            \let\pgf@intersect@next=\pgf@intersect@token@moveto%
        \else%
            \ifx#1\pgfsyssoftpath@linetotoken%
                \let\pgf@intersect@next=\pgf@intersect@token@lineto%
            \else%
                \ifx#1\pgfsyssoftpath@closepathtoken%
                    \let\pgf@intersect@next=\pgf@intersect@token@lineto%
                \else%
                    \ifx#1\pgfsyssoftpath@curvetosupportatoken%
                        \let\pgf@intersect@next=\pgf@intersect@token@curveto%
                    \else%
                        \ifx#1\pgfsyssoftpath@rectcornertoken%
                            \let\pgf@intersect@next=\pgf@intersect@token@rect%
                        \fi%
                    \fi%
                \fi%
            \fi%
        \fi%
    \fi%
    \pgf@intersect@next}%
```

可见命令 \pgf@intersectionofpaths 会吃掉一个形式为 \pgfsyssoftpath@*token 的记号，执行命令 \pgf@intersect@next(等于形式为 \pgf@intersect@token@* 的记号或者 \relax).

(b) 形式为 \pgf@intersect@token@* 的记号如下定义：

```
\def\pgf@intersect@token@moveto#1#2{%
    \def\pgfpoint@intersect@start{\pgfqpoint{#1}{#2}}%
    \pgf@intersectionofpaths%
}%

\def\pgf@intersect@token@lineto#1#2{%
    \def\pgfpoint@intersect@end{\pgfqpoint{#1}{#2}}%
    \def\pgf@intersect@type{line}%
    \pgf@intersect@token@after%
}%
\def\pgf@intersect@token@curveto#1#2\pgfsyssoftpath@curvetosupportbtoken#3#4
↪   \pgfsyssoftpath@curvetotoken#5#6{%
    \def\pgfpoint@intersect@firstsupport{\pgfqpoint{#1}{#2}}%
    \def\pgfpoint@intersect@secondsupport{\pgfqpoint{#3}{#4}}%
    \def\pgfpoint@intersect@end{\pgfqpoint{#5}{#6}}%
    \def\pgf@intersect@type{curve}%
    \pgf@intersect@token@after%
}%

\def\pgf@intersect@token@rect#1#2\pgfsyssoftpath@rectsizetoken#3#4{%
    \pgf@xa=#1\relax%
    \advance\pgf@xa by#3\relax%
    \pgf@ya=#2\relax%
    \advance\pgf@ya by#4\relax%
    \edef\pgf@marshal{%
        \noexpand\pgfsyssoftpath@movetotoken{#1}{#2}%
        \noexpand\pgfsyssoftpath@linetotoken{#1}{\the\pgf@ya}%
        \noexpand\pgfsyssoftpath@linetotoken{\the\pgf@xa}{\the\pgf@ya}%
        \noexpand\pgfsyssoftpath@linetotoken{\the\pgf@xa}{#2}%
        \noexpand\pgfsyssoftpath@closepathtoken{#1}{#2}%
    }%
    \expandafter\pgf@intersectionofpaths\pgf@marshal%
}%
```

可见执行一次命令 \pgf@intersect@token@* 就可以转化 (吃掉) 路径的一个片段，得到

- \pgfpoint@intersect@start
- \pgfpoint@intersect@end
- \pgfpoint@intersect@firstsupport
- \pgfpoint@intersect@secondsupport
- \pgf@intersect@type, 片段类型，有 line, curve 两种

然后执行 \pgf@intersect@token@after, 等于执行 \pgf@intersect@path@process@a.

(c) 命令 \pgf@intersect@path@process@a 的定义是：

```
\def\pgf@intersect@path@process@a{%
    \pgf@intersect@path@getpoints@a%
    \let\pgf@intersect@token@after=\pgf@intersect@path@process@b%
    \pgf@intersect@path@reset@b
    \expandafter\pgf@intersectionofpaths\pgf@intersect@path@b\pgf@stop%
    \let\pgfpoint@intersect@start=\pgfpoint@intersect@end@a%
    \let\pgf@intersect@token@after=\pgf@intersect@path@process@a%
```

```
    \c@pgf@counta=\pgf@intersect@time@offset\relax%
    \advance\c@pgf@counta by1\relax%
    \edef\pgf@intersect@time@offset{\the\c@pgf@counta}%
    \pgf@intersectionofpaths%
}%
```

可见命令 \pgf@intersect@path@process@a 的处理过程是：

i. 执行 \pgf@intersect@path@getpoints@a, 此命令的定义是

```
\def\pgf@intersect@path@getpoints@a{%
    \let\pgfpoint@intersect@start@a=\pgfpoint@intersect@start%
    \let\pgfpoint@intersect@end@a=\pgfpoint@intersect@end%
    \let\pgfpoint@intersect@firstsupport@a=
    ↪  \pgfpoint@intersect@firstsupport%
    \let\pgfpoint@intersect@secondsupport@a=
    ↪  \pgfpoint@intersect@secondsupport%
    \let\pgf@intersect@type@a=\pgf@intersect@type%
}%
```

此命令把坐标点、片段类型转存。

ii. 执行 \let\pgf@intersect@token@after=\pgf@intersect@path@process@b.

iii. 执行 \pgf@intersect@path@reset@b, 此命令的定义是

```
\def\pgf@intersect@path@reset@b{%
    \def\pgf@intersect@time@offset@b{0}%
    \def\pgf@intersect@time@b{}%
}%
```

iv. 执行 \expandafter\pgf@intersectionofpaths\pgf@intersect@path@b\pgf@stop, 这会把第二个路径的各个片段逐个处理掉，每处理一个片段就计算一次交点。处理一个片段时：

A. 先把形式为 \pgfsyssoftpath@*token 的记号转成形式为 \pgf@intersect@token@* 的记号并执行，得到

- \pgfpoint@intersect@start
- \pgfpoint@intersect@end
- \pgfpoint@intersect@firstsupport
- \pgfpoint@intersect@secondsupport
- \pgf@intersect@type

B. 然后执行 \pgf@intersect@token@after, 这等于执行 \pgf@intersect@path@process@b, 其处理过程是：

第一，执行 \pgf@intersect@path@getpoints@b, 此命令的定义是

```
\def\pgf@intersect@path@getpoints@b{%
    \let\pgfpoint@intersect@start@b=\pgfpoint@intersect@start%
    \let\pgfpoint@intersect@end@b=\pgfpoint@intersect@end%
    \let\pgfpoint@intersect@firstsupport@b=
    ↪  \pgfpoint@intersect@firstsupport%
    \let\pgfpoint@intersect@secondsupport@b=
    ↪  \pgfpoint@intersect@secondsupport%
    \let\pgf@intersect@type@b=\pgf@intersect@type%
}%
```

第二，执行

```
\csname pgf@intersect@\pgf@intersect@type@a @and@\pgf@intersect@type@b
↪  \endcsname
```

计算两个片段的交点。

第三，执行

```
\let\pgfpoint@intersect@start=\pgfpoint@intersect@end@b%
\c@pgf@counta=\pgf@intersect@time@offset@b\relax%
\advance\c@pgf@counta by1\relax%
\edef\pgf@intersect@time@offset@b{\the\c@pgf@counta}%
```

让终点变起点，并且让片段编号加 1.

第四，执行 \pgf@intersectionofpaths, 处理第二个路径的下一个片段。

如此处理完第二个路径的各个片段。

v. 回到 \pgf@intersect@path@process@a 的处理过程，执行

```
\let\pgfpoint@intersect@start=\pgfpoint@intersect@end@a%
\let\pgf@intersect@token@after=\pgf@intersect@path@process@a%
\c@pgf@counta=\pgf@intersect@time@offset\relax%
\advance\c@pgf@counta by1\relax%
\edef\pgf@intersect@time@offset{\the\c@pgf@counta}%
\pgf@intersectionofpaths%
```

让终点变起点，并且让片段编号加 1, 处理第一个路径的下一个片段。

如此处理完各个片段对。

以上步骤计算出所有交点。注意此时 \pgf@intersect@time@offset 的值刚刚超出第一个路径所需的片段编号范围。也就是说，假如第一个路径有 3 个片段，则此时 \pgf@intersect@time@offset 的值是 3，因为片段编号从 0 开始，所以说其值刚刚超出所需的片段编号范围。

下面将之前全局定义的交点的坐标、编号、时间数据进行转存，以便于引用、排序。

10. 执行 \edef\pgfintersectionsolutions{\the\pgf@intersect@solutions}, 保存交点的总数。

11. 执行循环 \pgfmathloop ... \repeatpgfmathloop.

    这个循环的开头有定义 \def\pgfmathcounter{1}.

    \pgfmathcounter 的值会从 1 变化到 \pgfintersectionsolutions, 对于每个 \pgfmathcounter 的值，执行以下步骤：

    (a) 执行

    ```
    \pgfutil@namelet{pgfpoint@intersect@solution@\pgfmathcounter}%
      {pgfpoint@g@intersect@solution@\pgfmathcounter}%
    ```

    将第 \pgfmathcounter 个交点坐标保存到

    \csname pgfpoint@intersect@solution@\pgfmathcounter\endcsname 中。

    (b) 执行

    ```
    \edef\pgf@marshal{\noexpand\pgf@intersection@set@properties{\csname
    ↪  pgfpoint@g@intersect@solution@\pgfmathcounter @props\endcsname}}%
    \pgf@marshal
    ```

    将 \csname pgfpoint@g@intersect@solution@\pgfmathcounter @props\endcsname 的展开值保存到 \csname pgf@intersect@solution@props@\pgfmathcounter\endcsname 中。这是唯一使用命令 \pgf@intersection@set@properties 的地方。

    (c) 执行

```
\ifpgf@intersect@sort%
  \pgfutil@namelet{pgf@intersect@solution@\pgfmathcounter @time@a}%
  {pgf@g@intersect@solution@\pgfmathcounter @time@a}%
\fi%
```

将 \csname pgf@g@intersect@solution@\pgfmathcounter @time@a\endcsname 的展开值保存到 \csname pgf@intersect@solution@\pgfmathcounter @time@a\endcsname 中。

12. 执行

```
\ifpgf@intersect@sort%
  \pgfintersectionsolutionsortbytime%
\fi%
```

命令 \pgfintersectionofpaths 至此结束。

36.2.3 计算交点

执行命令

```
\csname pgf@intersect@\pgf@intersect@type@a @and@\pgf@intersect@type@b\endcsname
```

计算两个片段的交点，因为 \pgf@intersect@type@a 与 \pgf@intersect@type@b 都有 line，curve 两种类型，所以有

```
\pgf@intersect@line@and@line
\pgf@intersect@line@and@curve
\pgf@intersect@curve@and@line
\pgf@intersect@curve@and@curve
```

四种片段相交的情况，不过后三种情况统一为“curve and curve”情况。

36.2.3.1 分析线段与线段的交点

假设第一个路径的第 i 片段 $a_i(s), 0 \le s \le 1$ 是线段，第二个路径的第 j 片段 $b_j(t), 0 \le t \le 1$ 是线段，考虑这两个线段的交点。

- 点 $a_i(0)$ 保存在 \pgfpoint@intersect@start@a 中，记 $a_i(0) = P_1 = (x_1, y_1)$.
- 点 $a_i(1)$ 保存在 \pgfpoint@intersect@end@a 中，记 $a_i(1) = P_2 = (x_2, y_2)$.
- 点 $b_j(0)$ 保存在 \pgfpoint@intersect@start@b 中，记 $b_j(0) = P_3 = (x_3, y_3)$.
- 点 $b_j(1)$ 保存在 \pgfpoint@intersect@end@b 中，记 $b_j(1) = P_4 = (x_4, y_4)$.

直线 $P_1 + (P_2 - P_1)s$ 与直线 $P_3 + (P_4 - P_3)t$ 的交点是

$$s = \frac{\begin{vmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{vmatrix}}{\begin{vmatrix} x_4 - x_3 & x_2 - x_1 \\ y_4 - y_3 & y_2 - y_1 \end{vmatrix}} = \frac{A}{C}, \quad t = \frac{\begin{vmatrix} x_4 - x_3 & x_3 - x_1 \\ y_4 - y_3 & y_3 - y_1 \end{vmatrix}}{\begin{vmatrix} x_4 - x_3 & x_2 - x_1 \\ y_4 - y_3 & y_2 - y_1 \end{vmatrix}} = \frac{B}{C}$$

当 $C \neq 0, 0 \le s, t \le 1$ 时两个线段有交点，记为 $P_c = (x_c, y_c)$.

命令 \pgf@intersect@line@and@line 的定义是

```
\def\pgf@intersect@line@and@line{%
    \pgf@intersectionoflines{\pgfpoint@intersect@start@a}{\pgfpoint@intersect@end@a}%
        {\pgfpoint@intersect@start@b}{\pgfpoint@intersect@end@b}%
}%
```

可见命令 \pgf@intersect@line@and@line 本身不带参数，它导致执行

\pgf@intersectionoflines{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩} 命令

命令 \pgf@intersectionoflines 的处理是：

1. 以 P_1P_2 为对角线的矩形是线段 $a_i(s)$ 的边界矩形，记为 Box_{a_i}. 以 P_3P_4 为对角线的矩形是线段 $b_j(t)$ 的边界矩形，记为 Box_{b_j}. 先检查 Box_{a_i} 与 Box_{b_j} 是否有重叠，如果有重叠再尝试计算交点。
   (a) 执行 \pgf@intersect@boundingbox@reset.
   (b) 执行 \pgf@intersect@boundingbox@update{⟨P_1⟩}.
   (c) 执行 \pgf@intersect@boundingbox@update{⟨P_2⟩}.
   (d) 执行 \pgf@intersect@boundingbox@assign@b, 将 Box_{a_i} 的左下角点的坐标、右上角点的坐标保存在 \pgf@intersect@boundingbox@b 中，此矩形的左下角坐标分量是 \pgf@x 和 \pgf@y, 右上角坐标分量是 \pgf@xa 和 \pgf@ya.
   (e) 执行 \pgf@intersect@boundingbox@assign@a, 将 Box_{b_j} 的左下角点的坐标、右上角点的坐标保存在 \pgf@intersect@boundingbox@a 中，此矩形的左下角坐标分量是 \pgf@xb 和 \pgf@yb, 右上角坐标分量是 \pgf@xc 和 \pgf@yc.
   (f) 执行 \pgf@intersect@boundingbox@a, \pgf@intersect@boundingbox@b, 调出盒子的对角点。
   (g) 执行 \pgf@intersect@boundingbox@ifoverlap@upperbound.
   判断盒子 Box_{a_i} 与 Box_{b_j} 是否有重叠，再执行 \pgf@intersect@next.
   (h) 此时有两个情况：
   - 如果 Box_{a_i} 与 Box_{b_j} 无重叠,即当前 \pgf@intersect@next 的值是 \pgfutil@secondoftwo, 则执行

     \let\pgf@intersect@next=\pgfutil@secondoftwo

   - 如果 Box_{a_i} 与 Box_{b_j} 有重叠,即当前 \pgf@intersect@next 的值是 \pgfutil@firstoftwo, 则执行

     \pgf@iflinesintersect@{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩} 命令

     此命令的处理过程是
     i. 把 $P_1P_2 = (x_2 - x_1, y_2 - y_1)$ 的 (L_1) 范数 $\|P_1P_2\|_1 = |x_2 - x_1| + |y_2 - y_1|$ (不带长度单位的数值) 全局地保存在 \pgf@intersect@len@a 中。
     ii. 把 $P_3P_4 = (x_4 - x_3, y_4 - y_3)$ 的 (L_1) 范数 $\|P_3P_4\|_1 = |x_4 - x_3| + |y_4 - y_3|$ (不带长度单位的数值) 全局地保存在 \pgf@intersect@len@b 中。
     iii. 使用命令 \pgfutilsolvetwotwoleqfloat (参考文件《pgfutil-common.tex》) 解方程

     $$(P_2 - P_1)s + (P_3 - P_4)t = P_3 - P_1$$

     未知量 s, t 的数值解 (不带长度单位) 保存在 \pgfmathresult 中。
     \pgfmathresult 的展开值是 {s 的值}{t 的值}, 或者是 \pgfutil@empty.
     iv. 执行 \let\pgf@intersect@next=\pgfutil@secondoftwo.
     v. 执行一个 \ifx, 判断 \pgfmathresult 的值是否是 \pgfutil@empty. 如果是就什么也不做。如果不是，那么
     A. 把 s 的值加上长度单位 pt 保存在 \pgf@x 中，把 t 的值加上长度单位 pt 保存在 \pgf@y 中。
     B. 检查 \pgf@x 的值，以排除 \pgf@x<0sp 或者 \pgf@x>1pt 的情况：
     第一，如果 \pgf@x<0sp, 并且

     -\pgf@intersect@len@a\pgf@x<\pgfintersectiontolerance

     则定义

```
\def\pgf@marshal{1}
```

否则定义

```
\def\pgf@marshal{0}
```

第二，如果 \pgf@x>1pt, 并且

```
\pgf@intersect@len@a(\pgf@x-1pt)<\pgfintersectiontolerance
```

则定义

```
\def\pgf@marshal{1}
```

否则定义

```
\def\pgf@marshal{0}
```

第三，如果 \pgf@x 的值是其他情况，则定义 \def\pgf@marshal{1}.

C. 检查当前 \pgf@marshal 的值。

如果 \pgf@marshal 的值是 1, 则检查 \pgf@y 的值,以排除 \pgf@y<0sp 或者 \pgf@y>1pt 的情况，办法如上，同时也重定义 \pgf@marshal 的值。

如果 \pgf@marshal 的值不是 1, 则什么也不做。

D. 再检查当前 \pgf@marshal 的值。

如果 \pgf@marshal 的值是 1, 则线段 $a_i(s)$ 与 $b_j(t)$ 有交点，把交点的横标 (全局地) 保存在 \pgf@x 中，把交点的纵标 (全局地) 保存在 \pgf@y 中。然后执行

```
\let\pgf@intersect@next=\pgfutil@firstoftwo
```

如果 \pgf@marshal 的值不是 1, 则什么也不做。

2. 执行 \pgf@intersect@next.
3. 此时有两个情况：
   - 如果 \pgf@intersect@next 等于 \pgfutil@secondoftwo, 即没有交点，则什么也不做。
   - 如果 \pgf@intersect@next 等于 \pgfutil@firstoftwo, 即有交点，则
     (a) 执行 \pgfextract@process\pgf@intersect@solution@candidate{}.
     命令 \pgfextract@process{⟨*macro*⟩}{⟨*code*⟩} 的效果是:执行 ⟨*code*⟩, 然后把当前的 \pgf@x 与 \pgf@y 全局地转存到 ⟨*macro*⟩ 中，见文件《pgfcorepoints.code.tex》.
     此时，宏 \pgf@intersect@solution@candidate 保存当前的交点坐标。
     (b) 执行 \pgf@ifsolution@duplicate{\pgf@intersect@solution@candidate}, 检查当前计算出来的交点是否与之前计算出来的交点有重合。如果有重合，则什么也不做；否则
     (c) 执行 \global\advance\pgf@intersect@solutions by1\relax
     (d) 执行

```
\expandafter\global\expandafter\let\csname pgfpoint@g@intersect@solution@
↪  \the\pgf@intersect@solutions\endcsname=
↪  \pgf@intersect@solution@candidate
```

     全局地保存当前交点的坐标。
     (e) 执行条件判断 \ifpgf@intersect@sort,
     – 如果 \pgf@intersect@sorttrue, 则 (使用 L_2 范数，即命令 \pgfmathveclen@) 计算 $t_c = \frac{|P_1P_c|}{|P_1P_2|}$, 再定义 \pgf@intersect@time@a→ P. 626 的值是当前的 \pgf@intersect@time@offset $+t_c$. 再执行

```
\expandafter\global\expandafter\let\csname pgf@g@intersect@solution@
↪  \the\pgf@intersect@solutions @time@a\endcsname=
↪  \pgf@intersect@time@a
```

全局地保存当前交点在第一个路径上的时间数据。

– 如果 \pgf@intersect@sortfalse, 则 \let\pgf@intersect@time@a=\pgfutil@empty.

(f) 执行 \let\pgf@intersect@time@b=\pgfutil@empty.

(g) 执行

```
\pgf@intersection@store@properties{pgfpoint@g@intersect@solution@\the
↪  \pgf@intersect@solutions}
```

前文解释过，这导致全局定义

■

```
\expandafter\xdef%
  \csname pgfpoint@g@intersect@solution@\the\pgf@intersect@solutions
  ↪  @props\endcsname%
  {{\pgf@intersect@time@offset}{\pgf@intersect@time@offset@b}{
  ↪  \pgf@intersect@time@a}{\pgf@intersect@time@b}}
```

36.2.3.2 分析曲线与曲线的交点

当处理的一对片段是曲线与曲线时，算法复杂一些。

将曲线 $C(s), 0 \le s \le 1$, 在 $s = 0.5$ 处分成两段，记 $C(s)(0 \le s \le 0.5)$ 为 $Cl(u), 0 \le u \le 1$, 称 Cl 为 C 的左侧部分；记 $C(s)(0.5 \le s \le 1)$ 为 $Cr(v), 0 \le v \le 1$, 称 Cr 为 C 的右侧部分。由 C 得到 Cl 和 Cr 的过程是对 C 的一次“分割”，对 Cl 和 Cr 也可以做分割，得到 Cll, Clr, Crl , Crr; 这样不断分割下去就能把 C 分割地足够细密。这个分割过程可以利用 C 的控制点递推地实现。

假设曲线 C 的 4 个控制点是 P_1, P_2, P_3, P_4, 那么 Cl 的 4 个控制点是

$$P_1, \quad 0.5(P_1 + P_2), \quad 0.25(P_1 + 2P_2 + P_3), \quad 0.125(P_1 + 3P_2 + 3P_3 + P_4)$$

也就是

$$M_l = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{4} & \frac{2}{4} & \frac{1}{4} & 0 \\ \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{bmatrix}, \quad M_l \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

其中 $0.125(P_1 + 3P_2 + 3P_3 + P_4) = C(0.5)$ 是 C 的“中点”。Cl 由命令 \pgf@curve@subdivide@left 得到。

Cr 的 4 个控制点是

$$0.125(P_1 + 3P_2 + 3P_3 + P_4), \quad 0.25(P_2 + 2P_3 + P_4), \quad 0.5(P_3 + P_4), \quad P_4$$

也就是

$$M_r = \begin{bmatrix} \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \\ 0 & \frac{1}{4} & \frac{2}{4} & \frac{1}{4} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad M_r \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

Cr 由命令 \pgf@curve@subdivide@right 得到。如果要再把 Cr 分成左右两部分，只要继续使用矩阵乘

积即可得到相应的控制点：

$$M_l M_r \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}, \quad M_r M_r \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}$$

寻找曲线 U 与 V 的交点时采用下面的准则：

1. 如果 U 与 V 的边界矩形无重叠，则得到坐标系的原点。
2. 如果 U 与 V 的边界矩形有重叠,且两个边界矩形的尺寸 (宽、高) 都小于 \pgfintersectiontolerance→P. 632，则规定交点坐标是

$$x = \frac{x_1 + x_2 + x_3 + x_4}{4}, \quad y = \frac{y_1 + y_2 + y_3 + y_4}{4}$$

其中 $(x_1, y_1), (x_2, y_2)$ 分别是曲线 U 的 4 个控制点的边界矩形的左下角点、右上角点；$(x_3, y_3), (x_4, y_4)$ 分别是曲线 V 的 4 个控制点的边界矩形的左下角点、右上角点。

如果 U 与 V 的边界矩形有重叠,但并非两个边界矩形的尺寸 (宽、高) 都小于 \pgfintersectiontolerance→P. 632，那就需要对曲线 U 或 V 做分割，并在分割时检查是否满足准则。这个分割——检查过程可以用下面的递推模式来描述：

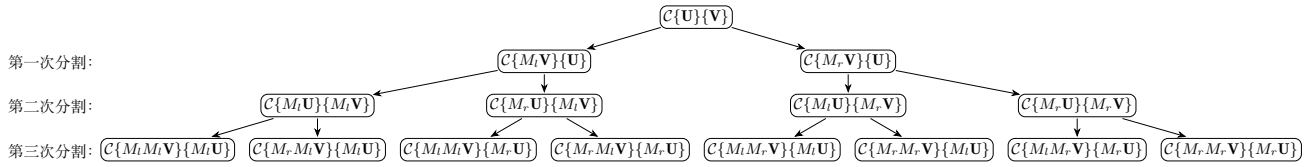
```
x=U, y=V % U, V 代表分割过程中的曲线
if x 与 y 满足准则
  得出一组坐标，结束计算
else
  tempx=x, tempy=y, x=tempyl, y=tempx % tempyl 表示 temp 的左侧部分
  if x 与 y 满足准则
    得出一组坐标，
    x=tempyr, y=tempx，回到第 2 行 % tempyr 表示 temp 的右侧部分
  else 回到第 5 行
```

这个分割——检查过程由命令 \pgf@@intersectionofcurves 完成。

对曲线 V 的分割由命令 \pgf@intersect@subdivide@curve@b 完成。

对曲线 U 的分割由命令 \pgf@intersect@subdivide@curve@a 完成。

设曲线 U 和 V 的控制点分别是 $\mathbf{U} = \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix}$ 和 $\mathbf{V} = \begin{bmatrix} V_1 \\ V_2 \\ V_3 \\ V_4 \end{bmatrix}$，把命令 \pgf@@intersectionofcurves{\mathbf{U}}{\mathbf{V}} 简单记为 $\mathcal{C}\{\mathbf{U}\}\{\mathbf{V}\}$，则分割过程可以表示为：



假设第一个路径的第 i 片段 $a_i(s), 0 \le s \le 1$ 是控制曲线，其控制点是 $P_i = (x_i, y_i), i = 1, 2, 3, 4$; 第二个路径的第 j 片段 $b_j(t), 0 \le t \le 1$ 是控制曲线，其控制点是 $P_j = (x_j, y_j), j = 5, 6, 7, 8$; 考虑这两个曲线的交点。

\pgf@intersect@curve@and@curve

命令 \pgf@intersect@curve@and@curve 的定义是：

```
\def\pgf@intersect@curve@and@curve{%
    \pgf@intersectionofcurves%
        {\pgf@process{\pgfpoint@intersect@start@a}}{\pgf@process{
        ↪  \pgfpoint@intersect@firstsupport@a}}%
        {\pgf@process{\pgfpoint@intersect@secondsupport@a}}{\pgf@process{
        ↪  \pgfpoint@intersect@end@a}}%
```

```
        {\pgf@process{\pgfpoint@intersect@start@b}}{\pgf@process{
        ↪  \pgfpoint@intersect@firstsupport@b}}%
        {\pgf@process{\pgfpoint@intersect@secondsupport@b}}{\pgf@process{
        ↪  \pgfpoint@intersect@end@b}}%
}%
```

此命令利用 \pgf@intersectionofcurves 工作。

\pgf@intersectionofcurves{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨Q_1⟩}{⟨Q_2⟩}{⟨Q_3⟩}{⟨Q_4⟩}

命令 \pgf@intersectionofcurves 的定义是：

```
\def\pgf@intersectionofcurves#1#2#3#4#5#6#7#8{%
    \begingroup%
        \dimendef\pgf@time@a=2\relax%
        \dimendef\pgf@time@aa=4\relax%
        \dimendef\pgf@time@b=6\relax%
        \dimendef\pgf@time@bb=8\relax%
        \pgf@time@a=0pt\relax%
        \pgf@time@aa=1pt\relax%
        \pgf@time@b=0pt\relax%
        \pgf@time@bb=1pt\relax%
        \let\pgf@intersect@subdivide@curve=\pgf@intersect@subdivide@curve@b%
        \let\pgf@curve@subdivde@after=\pgf@@intersectionofcurves%
        \pgf@@intersectionofcurves{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}%
    \endgroup%
}%
```

其中的变量 #1#2#3#4 代表 $a_i(s)$ 的 4 个控制点；变量 #5#6#7#8 代表 $b_j(t)$ 的 4 个控制点。此命令设置一个组，其所有处理都在这个组内进行。

此命令利用 \pgf@@intersectionofcurves 工作。

\pgf@@intersectionofcurves{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨P_5⟩}{⟨P_6⟩}{⟨P_7⟩}{⟨P_8⟩}

命令 \pgf@@intersectionofcurves 的定义是：

```
\def\pgf@@intersectionofcurves#1#2#3#4#5#6#7#8{%
    \pgf@intersect@boundingbox@reset%
    \pgf@intersect@boundingbox@update{#1}%
    \pgf@intersect@boundingbox@update{#2}%
    \pgf@intersect@boundingbox@update{#3}%
    \pgf@intersect@boundingbox@update{#4}%
    \pgf@intersect@boundingbox@assign@b%
    %
    \pgf@intersect@boundingbox@reset%
    \pgf@intersect@boundingbox@update{#5}%
    \pgf@intersect@boundingbox@update{#6}%
    \pgf@intersect@boundingbox@update{#7}%
    \pgf@intersect@boundingbox@update{#8}%
    \pgf@intersect@boundingbox@assign@a%
    %
    \pgf@intersect@boundingbox@a%
    \pgf@intersect@boundingbox@b%
    %
    \pgf@intersect@boundingbox@ifoverlap{%
        \pgf@@@intersectionofcurves{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}%
    }{%
        % no overlap -- no intersection.
    }%
```

```
}%
```

命令 \pgf@@intersectionofcurves{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨P_5⟩}{⟨P_6⟩}{⟨P_7⟩}{⟨P_8⟩} 的处理是：

1. 用 \pgf@intersect@boundingbox@b 保存点组 $\{P_1, P_2, P_3, P_4\}$ 的边界矩形。
2. 用 \pgf@intersect@boundingbox@a 保存点组 $\{P_5, P_6, P_7, P_8\}$ 的边界矩形。
3. 调出两个边界矩形的对角点。
4. 判断两个边界矩形是否有重叠。若无重叠，则什么也不做。若有重叠，则执行

```
\pgf@@@intersectionofcurves{⟨P1⟩}{⟨P2⟩}{⟨P3⟩}{⟨P4⟩}{⟨P5⟩}{⟨P6⟩}{⟨P7⟩}{⟨P8⟩}
```

\pgf@@@intersectionofcurves{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨P_5⟩}{⟨P_6⟩}{⟨P_7⟩}{⟨P_8⟩}

在两个边界矩形有重叠的情况下执行此命令，其处理过程是

1. 计算矩形 \pgf@intersect@boundingbox@a 的对角向量 ⟨右上角点⟩ − ⟨左下角点⟩：

```
\advance\pgf@xc by-\pgf@xb%
\advance\pgf@yc by-\pgf@yb%
```

   计算结果保存在 \pgf@xc 与 \pgf@yc 中。

2. 计算矩形 \pgf@intersect@boundingbox@b 的对角向量 ⟨右上角点⟩ − ⟨左下角点⟩：

```
\advance\pgf@xa by-\pgf@x%
\advance\pgf@ya by-\pgf@y%
```

   计算结果保存在 \pgf@xa 与 \pgf@ya 中。

3. 执行 \let\pgf@intersect@subdivde=\relax.
4. 执行条件判断：
   - 如果两个边界矩形的宽度和高度都小于 \pgfintersectiontolerance(用矩形的对角向量检查宽度、高度)，则
     (a) 令

$$\pgf@x = \frac{\pgf@x + \pgf@xa + \pgf@xb + \pgf@xc}{4}$$

$$\pgf@y = \frac{\pgf@y + \pgf@ya + \pgf@yb + \pgf@yc}{4}$$

     并把 \pgf@x 与 \pgf@y 全局地保存到宏 \pgf@intersect@solution@candidate 中，作为当前得到的交点。
     (b) 执行 \let\pgf@intersect@subdivde=\pgf@stop.
     (c) 执行 \pgf@ifsolution@duplicate 检查当前得到的交点 \pgf@intersect@solution@candidate 是否与之前的某个交点重合。如果有重合，则什么也不做。如果无重合，则
     (d) 执行 \global\advance\pgf@intersect@solutions by1\relax.
     (e) 用 \begingroup 开启一个组。
     (f) 定义 \pgf@intersect@time@a 的值是

$$\pgf@intersect@time@offset + \frac{\pgf@time@a + \pgf@time@aa}{2}$$

     的数值部分。此时这个数值也是寄存器 \pgf@time@a 的数值部分。
     (g) 定义 \pgf@intersect@time@b 的值是

$$\pgf@intersect@time@offset@b + \frac{\pgf@time@b + \pgf@time@bb}{2}$$

     的数值部分。此时这个数值也是寄存器 \pgf@time@b 的数值部分。
     (h) 执行

```
\pgf@intersection@store@properties{pgfpoint@g@intersect@solution@\the
↪  \pgf@intersect@solutions}%
```

全局地保存当前交点的片段编号、时间数据。

(i) 执行

```
\expandafter\global\expandafter\let%
  \csname pgfpoint@g@intersect@solution@\the\pgf@intersect@solutions
  ↪  \endcsname=%
  \pgf@intersect@solution@candidate%
```

全局地保存当前交点的坐标。

(j) 执行

```
\ifpgf@intersect@sort%
  \expandafter\xdef%
  \csname pgf@g@intersect@solution@\the\pgf@intersect@solutions @time@a
  ↪  \endcsname%
  {\pgf@intersect@time@a}%
\fi%
```

在 \ifpgf@intersect@sort 判断为 true 的情况下，将当前交点在第一个路径上的时间数据全局地保存到

```
\csname pgf@g@intersect@solution@\the\pgf@intersect@solutions @time@a
↪  \endcsname
```

中，提供给排序命令使用。

(k) 用 \endgroup 结束之前开启的组。在这个组内，关于的寄存器 \pgf@time@a, \pgf@time@aa, \pgf@time@b, \pgf@time@bb 的运算只在组内有效，这 4 个寄存器用于计算交点的“时间”数据。

- 如果并非两个边界矩形的宽度、高度都小于 \pgfintersectiontolerance, 什么也不做。

5. 执行

```
\ifx\pgf@intersect@subdivde\pgf@stop%
\else%
  \pgf@intersect@subdivide@curve{#1}{#2}{#3}{#4}{#5}{#6}{#7}{#8}%
\fi%
```

这实际上是检查：是否“并非两个边界矩形的宽度及高度都小于 \pgfintersectiontolerance”.

- 如果是 (至少有一个矩形的宽度或高度大于等于阈值 \pgfintersectiontolerance), 则执行 \pgf@intersect@subdivide@curve, 此命令一开始在 \pgf@intersectionofcurves →P. 645 中被 let 为 \pgf@intersect@subdivide@curve@b;之后在 \pgf@intersect@subdivide@curve@b 中被 let 为 \pgf@intersect@subdivide@curve@a;之后在 \pgf@intersect@subdivide@curve@a →P. 648 中被 let 为 \pgf@intersect@subdivide@curve@b, 形成循环。
- 如果不是 (两个矩形的宽度及高度都小于阈值 \pgfintersectiontolerance)，则什么也不做。

\pgf@intersect@subdivide@curve@b{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨Q_1⟩}{⟨Q_2⟩}{⟨Q_3⟩}{⟨Q_4⟩}

命令 \pgf@intersect@subdivide@curve@b 的定义是：

```
\def\pgf@intersect@subdivide@curve@b#1#2#3#4#5#6#7#8{%
    \begingroup%
        \advance\pgf@time@bb by\pgf@time@b\relax%
```

```
        \divide\pgf@time@bb by2\relax%
        \let\pgf@intersect@subdivide@curve=\pgf@intersect@subdivide@curve@a%
        \pgf@curve@subdivide@left{#5}{#6}{#7}{#8}{#1}{#2}{#3}{#4}%
    \endgroup%
    \begingroup%
        \advance\pgf@time@b by\pgf@time@bb\relax%
        \divide\pgf@time@b by2\relax%
        \let\pgf@intersect@subdivide@curve=\pgf@intersect@subdivide@curve@a%
        \pgf@curve@subdivide@right{#5}{#6}{#7}{#8}{#1}{#2}{#3}{#4}%
    \endgroup%
}%
```

命令 \pgf@intersect@subdivide@curve@b 用于分割第二个路径的片段。每次分割都把分割对象分成“左侧、右侧”两部分，左侧部分由命令 \pgf@curve@subdivide@left 得到，右侧部分由命令 \pgf@curve@subdivide@right 得到。关于左侧的处理在一个组内进行，关于右侧的处理也在一个组内进行，两个组的处理互不干涉 (例如寄存器值互不干涉)。

\pgf@intersect@subdivide@curve@a{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨Q_1⟩}{⟨Q_2⟩}{⟨Q_3⟩}{⟨Q_4⟩}

命令 \pgf@intersect@subdivide@curve@a 的定义是：

```
\def\pgf@intersect@subdivide@curve@a#1#2#3#4#5#6#7#8{%
    \begingroup%
        \advance\pgf@time@aa by\pgf@time@a\relax%
        \divide\pgf@time@aa by2\relax%
        \let\pgf@intersect@subdivide@curve=\pgf@intersect@subdivide@curve@b%
        \pgf@curve@subdivide@left{#5}{#6}{#7}{#8}{#1}{#2}{#3}{#4}%
    \endgroup%
    \begingroup%
        \advance\pgf@time@a by\pgf@time@aa\relax%
        \divide\pgf@time@a by2\relax%
        \let\pgf@intersect@subdivide@curve=\pgf@intersect@subdivide@curve@b%
        \pgf@curve@subdivide@right{#5}{#6}{#7}{#8}{#1}{#2}{#3}{#4}%
    \endgroup%
}%
```

以上两个命令对曲线做分割——检查的处理。分割操作由命令 \pgf@curve@subdivide@left (得到曲线的左侧部分)，\pgf@curve@subdivide@right(得到曲线的右侧部分) 完成。

\pgf@curve@subdivide@left{⟨Q_1⟩}{⟨Q_2⟩}{⟨Q_3⟩}{⟨Q_4⟩}

命令 \pgf@curve@subdivide@left 的定义是：

```
\def\pgf@curve@subdivide@left#1#2#3#4{%
    %
    % The left curve (from t=0 to t=.5)
    %
    \begingroup
    #1\relax%
    \pgfutil@tempdima=\pgf@x%
    \pgfutil@tempdimb=\pgf@y%
    \pgf@float@adapter@setxy
    \pgf@float@adapter@mult\pgf@fpu@xa=.5*\pgf@fpu@x  \pgf@float@adapter@mult
    ↪  \pgf@fpu@ya=.5*\pgf@fpu@y%
    \pgf@float@adapter@mult\pgf@fpu@xb=.25*\pgf@fpu@x \pgf@float@adapter@mult
    ↪  \pgf@fpu@yb=.25*\pgf@fpu@y%
    \pgf@float@adapter@mult\pgf@fpu@xc=.125*\pgf@fpu@x\pgf@float@adapter@mult
    ↪  \pgf@fpu@yc=.125*\pgf@fpu@y%
```

```
    #2\relax%
    \pgf@float@adapter@setxy
    \pgf@float@adapter@advance\pgf@fpu@xa by.5*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@fpu@ya by.5*\pgf@fpu@y%
    \pgf@float@adapter@advance\pgf@fpu@xb by.5*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@fpu@yb by.5*\pgf@fpu@y%
    \pgf@float@adapter@advance\pgf@fpu@xc by.375*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@fpu@yc by.375*\pgf@fpu@y%
    #3\relax%
    \pgf@float@adapter@setxy
    \pgf@float@adapter@advance\pgf@fpu@xb by.25*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@fpu@yb by.25*\pgf@fpu@y%
    \pgf@float@adapter@advance\pgf@fpu@xc by.375*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@fpu@yc by.375*\pgf@fpu@y%
    #4\relax%
    \pgf@float@adapter@setxy
    \pgf@float@adapter@advance\pgf@fpu@xc by.125*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@fpu@yc by.125*\pgf@fpu@y%
    %
    \pgf@float@adapter@tostring\pgf@fpu@xa
    \pgf@float@adapter@tostring\pgf@fpu@ya
    \pgf@float@adapter@tostring\pgf@fpu@xb
    \pgf@float@adapter@tostring\pgf@fpu@yb
    \pgf@float@adapter@tostring\pgf@fpu@xc
    \pgf@float@adapter@tostring\pgf@fpu@yc
    \edef\pgf@marshal{%
        \noexpand\pgf@curve@subdivde@after%
            {\noexpand\pgf@x=\the\pgfutil@tempdima\noexpand\pgf@y=\the
            ↪  \pgfutil@tempdimb}%
            {\noexpand\pgf@x=\pgf@fpu@xa\noexpand\pgf@y=\pgf@fpu@ya}%
            {\noexpand\pgf@x=\pgf@fpu@xb\noexpand\pgf@y=\pgf@fpu@yb}%
            {\noexpand\pgf@x=\pgf@fpu@xc\noexpand\pgf@y=\pgf@fpu@yc}%
    }%
    \expandafter
    \endgroup
    \pgf@marshal%
}%
```

在 \pgf@intersectionofcurves 的定义中有

```
\let\pgf@curve@subdivde@after=\pgf@@intersectionofcurves%
```

\pgf@curve@subdivide@right{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}

命令 \pgf@curve@subdivide@right 的定义是：

■

```
\def\pgf@curve@subdivide@right#1#2#3#4{%
    %
    % The right curve (from t=0.5 to t=1)
    %
    \begingroup
    #1\relax%
    \pgf@float@adapter@setxy
    \pgf@float@adapter@mult\pgf@float@tmpa=.125*\pgf@fpu@x\pgf@float@adapter@mult
    ↪  \pgf@float@tmpb=.125*\pgf@fpu@y%
    #2\relax%
    \pgf@float@adapter@setxy
```

```
    \pgf@float@adapter@advance\pgf@float@tmpa by.375*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@float@tmpb by.375*\pgf@fpu@y%
    \pgf@float@adapter@mult\pgf@fpu@xa=.25*\pgf@fpu@x\pgf@float@adapter@mult
    ↪  \pgf@fpu@ya=.25*\pgf@fpu@y%
    #3\relax%
    \pgf@float@adapter@setxy
    \pgf@float@adapter@advance\pgf@float@tmpa by.375*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@float@tmpb by.375*\pgf@fpu@y%
    \pgf@float@adapter@advance\pgf@fpu@xa by.5*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@fpu@ya by.5*\pgf@fpu@y%
    \pgf@float@adapter@mult\pgf@fpu@xb=.5*\pgf@fpu@x\pgf@float@adapter@mult
    ↪  \pgf@fpu@yb=.5*\pgf@fpu@y%
    #4\relax%
    \pgf@float@adapter@setxy
    \pgf@float@adapter@advance\pgf@float@tmpa by.125*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@float@tmpb by.125*\pgf@fpu@y%
    \pgf@float@adapter@advance\pgf@fpu@xa by.25*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@fpu@ya by.25*\pgf@fpu@y%
    \pgf@float@adapter@advance\pgf@fpu@xb by.5*\pgf@fpu@x
    ↪  \pgf@float@adapter@advance\pgf@fpu@yb by.5*\pgf@fpu@y%
    \let\pgf@fpu@xc=\pgf@fpu@x\let\pgf@fpu@yc=\pgf@fpu@y%
    %
    \pgf@float@adapter@tostring\pgf@float@tmpa
    \pgf@float@adapter@tostring\pgf@float@tmpb
    \pgf@float@adapter@tostring\pgf@fpu@xa
    \pgf@float@adapter@tostring\pgf@fpu@ya
    \pgf@float@adapter@tostring\pgf@fpu@xb
    \pgf@float@adapter@tostring\pgf@fpu@yb
    \pgf@float@adapter@tostring\pgf@fpu@xc
    \pgf@float@adapter@tostring\pgf@fpu@yc
    \edef\pgf@marshal{%
        \noexpand\pgf@curve@subdivde@after%
            {\noexpand\pgf@x=\pgf@float@tmpa\noexpand\pgf@y=\pgf@float@tmpb}%
            {\noexpand\pgf@x=\pgf@fpu@xa\noexpand\pgf@y=\pgf@fpu@ya}%
            {\noexpand\pgf@x=\pgf@fpu@xb\noexpand\pgf@y=\pgf@fpu@yb}%
            {\noexpand\pgf@x=\pgf@fpu@xc\noexpand\pgf@y=\pgf@fpu@yc}%
    }%
    \expandafter
    \endgroup
    \pgf@marshal%
}%
```

分割过程是：

第一次分割
\pgf@intersect@subdivide@curve@b{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨Q_1⟩}{⟨Q_2⟩}{⟨Q_3⟩}{⟨Q_4⟩}
导致
\begingroup%
  \pgf@curve@subdivide@left{⟨Q_1⟩}{⟨Q_2⟩}{⟨Q_3⟩}{⟨Q_4⟩}{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}
\endgroup%
\begingroup%
  \pgf@curve@subdivide@right{⟨Q_1⟩}{⟨Q_2⟩}{⟨Q_3⟩}{⟨Q_4⟩}{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}
\endgroup%
这会导致把 {⟨Q_1⟩}{⟨Q_2⟩}{⟨Q_3⟩}{⟨Q_4⟩} 分裂为
{⟨Q_{1L}⟩}{⟨Q_{2L}⟩}{⟨Q_{3L}⟩}{⟨Q_{4L}⟩} 和 {⟨Q_{1R}⟩}{⟨Q_{2R}⟩}{⟨Q_{3R}⟩}{⟨Q_{4R}⟩}
\begingroup%
  \pgf@@intersectionofcurves→P. 645{⟨Q_{1L}⟩}{⟨Q_{2L}⟩}{⟨Q_{3L}⟩}{⟨Q_{4L}⟩}{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}

```
\endgroup%
\begingroup%
  \pgf@@intersectionofcurves{⟨Q_{1R}⟩}{⟨Q_{2R}⟩}{⟨Q_{3R}⟩}{⟨Q_{4R}⟩}{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}
\endgroup%
然后进入第二次分割
\begingroup%
  \pgf@intersect@subdivide@curve@a → P. 648
  ↪  {⟨Q_{1L}⟩}{⟨Q_{2L}⟩}{⟨Q_{3L}⟩}{⟨Q_{4L}⟩}{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}
\endgroup%
\begingroup%
  \pgf@intersect@subdivide@curve@a{⟨Q_{1R}⟩}{⟨Q_{2R}⟩}{⟨Q_{3R}⟩}{⟨Q_{4R}⟩}{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}
\endgroup%
导致
\begingroup%
  \begingroup%
    \pgf@curve@subdivide@left{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨Q_{1L}⟩}{⟨Q_{2L}⟩}{⟨Q_{3L}⟩}{⟨Q_{4L}⟩}
  \endgroup%
  \begingroup%
    \pgf@curve@subdivide@right{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨Q_{1L}⟩}{⟨Q_{2L}⟩}{⟨Q_{3L}⟩}{⟨Q_{4L}⟩}
  \endgroup%
\endgroup%
\begingroup%
  \begingroup%
    \pgf@curve@subdivide@left{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨Q_{1R}⟩}{⟨Q_{2R}⟩}{⟨Q_{3R}⟩}{⟨Q_{4R}⟩}
  \endgroup%
  \begingroup%
    \pgf@curve@subdivide@right{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨Q_{1R}⟩}{⟨Q_{2R}⟩}{⟨Q_{3R}⟩}{⟨Q_{4R}⟩}
  \endgroup%
\endgroup%
这会导致把 {⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩} 分裂为
{⟨P_{1L}⟩}{⟨P_{2L}⟩}{⟨P_{3L}⟩}{⟨P_{4L}⟩} 和 {⟨P_{1R}⟩}{⟨P_{2R}⟩}{⟨P_{3R}⟩}{⟨P_{4R}⟩}
......
```

36.2.3.3 其他命令

\pgfintersectionoflines{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}

此命令判断线段 P_1P_2 与 P_3P_4 是否有交点，并计算交点。此命令的定义是

```
\def\pgfintersectionoflines#1#2#3#4{%
    \pgf@intersect@solutions=0\relax%
    \pgf@intersectionoflines{#1}{#2}{#3}{#4}%
}%
```

交点个数保存在 \pgfintersectionsolutions 中。

宏 \pgfpointintersectionsolution{⟨*number*⟩} 展开为交点坐标。

\pgfintersectionoflineandcurve{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨P_5⟩}{⟨P_6⟩}

此命令判断线段 P_1P_2 与曲线 $P_3P_4P_5P_6$ 是否有交点，并计算交点。此命令的定义是

```
\def\pgfintersectionoflineandcurve#1#2#3#4#5#6{%
    \pgf@intersect@solutions=0\relax%
    \pgf@intersectionoflineandcurve{#1}{#2}{#3}{#4}{#5}{#6}%
}%
```

其中命令 \pgf@intersectionoflineandcurve 调用 \pgf@intersectionofcurves 工作。

交点个数保存在 \pgfintersectionsolutions 中。

宏 \pgfpointintersectionsolution{⟨number⟩} 展开为交点坐标。

\pgfpointintersectionofcurves{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨P_5⟩}{⟨P_6⟩}{⟨P_7⟩}{⟨P_8⟩}{⟨number⟩}

此命令判断曲线 $P_1P_2P_3P_4$ 与曲线 $P_5P_6P_7P_8$ 是否有交点，并计算交点，还返回第 ⟨number⟩ 个交点。此命令的定义是

```
\def\pgfpointintersectionofcurves#1#2#3#4#5#6#7#8#9{%
    \pgf@intersect@solutions=0\relax%
    \pgf@intersectionofcurves%
        {\pgf@process{#1}}{\pgf@process{#2}}{\pgf@process{#3}}{\pgf@process{#4}}%
        {\pgf@process{#5}}{\pgf@process{#6}}{\pgf@process{#7}}{\pgf@process{#8}}%
    \pgfpointintersectionsolution{#9}%
}%
```

交点个数保存在 \pgfintersectionsolutions 中。

宏 \pgfpointintersectionsolution{⟨number⟩} 展开为交点坐标。

\pgfintersectionofcurves{⟨P_1⟩}{⟨P_2⟩}{⟨P_3⟩}{⟨P_4⟩}{⟨P_5⟩}{⟨P_6⟩}{⟨P_7⟩}{⟨P_8⟩}

此命令判断曲线 $P_1P_2P_3P_4$ 与曲线 $P_5P_6P_7P_8$ 是否有交点，并计算交点。此命令的定义是

```
\def\pgfintersectionofcurves#1#2#3#4#5#6#7#8{%
    \pgf@intersect@solutions=0\relax%
    \pgf@intersectionofcurves%
        {\pgf@process{#1}}{\pgf@process{#2}}{\pgf@process{#3}}{\pgf@process{#4}}%
        {\pgf@process{#5}}{\pgf@process{#6}}{\pgf@process{#7}}{\pgf@process{#8}}%
}%
```

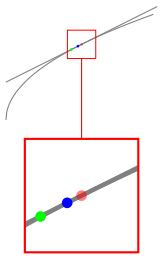
交点个数保存在 \pgfintersectionsolutions 中。

宏 \pgfpointintersectionsolution{⟨number⟩} 展开为交点坐标。

36.3 如果用 intersections 库求切点

在目前的 Ti*k*Z/PGF 中，曲线都是用 3 次 Bézier 曲线来近似的，对于这种近似而言，切点与交点有很大的不同。从数学的角度看，如果两个曲线有交点，那么他们的近似曲线几乎总有交点。但对于切点的情况，如果曲线的近似精度不足，那么近似曲线就可能没有切点。即使按算法能得到一个“切点”，这个切点的位置也可能不够准确。如果要提高近似的精度，同时缩小阈值 \pgfintersectiontolerance，那么必定增大计算量，同时还要考虑 TEX 本身的计算精度的限制。

下面的例子显示了切点情况下阈值 \pgfintersectiontolerance 的影响：



```
\begin{tikzpicture}[line width=0.1pt,
    spy using outlines={lens={scale=4}, size=1.5cm, connect spies}]
  \draw[name path=A,domain=0:2,draw=black!50!white] plot (\x,{0.5*\x+0.5});
  \draw[name path=B,draw=black!50!white,]
    plot[samples at={0,0.01,...,0.15},smooth] (\x,{sqrt(\x)})--
    plot[samples at={0.15,0.25,...,2},smooth] (\x,{sqrt(\x)});
  \fill[fill opacity=0.5,fill=red](1,1)circle (0.5pt);
  \def\pgfintersectiontolerance{0.1pt}
  \fill[fill=green,name intersections={of=A and B,by=P}](P)circle(0.5pt);
  \def\pgfintersectiontolerance{0.01pt}
  \fill[fill=blue,name intersections={of=A and B,by=P}](P)circle(0.5pt);
  \spy [red] on (1,1) in node at (1,-1);
\end{tikzpicture}
```

第三十七章　luamath 库

```
\usepgflibrary{luamath} % LaTeX and plain TeX and pure pgf
\usetikzlibrary{luamath} % LaTeX and plain TeX when using TikZ
```

使用这个库时，需要用 LuaTEX 或 LuaLATEX 来编译。

这个库会载入 /generic/pgf/libraries/luamath/pgf/luamath 下的 functions.lua, parser.lua 模块。

在正常编译的情况下，\pgfutil@directlua 被 let 为 \directlua, 而 \pgfutil@luaescapestring 等效于 \luaescapestring.

使用 luamath 库的情况下，命令 \pgfmathparse{⟨*expression*⟩} 的处理方式会被改变。

37.1　角度制与弧度制

键 /pgf/trig format→P. 122 的可选值对应的代码会被重定义：

```
\pgfkeys{
    /pgf/trig format/deg/.add code={}{\directlua
    ↪  {pgfluamathfunctions.setTrigFormat("deg")}\aftergroup\pgfmath@settrigformat},
    /pgf/trig format/rad/.add code={}{\directlua
    ↪  {pgfluamathfunctions.setTrigFormat("rad")}\aftergroup\pgfmath@settrigformat},
}%
```

仍然默认角度制。

37.2　luamath 库的工作方式

luamath 库有 2 种工作方式：

1. 仍然由 PGF 的 (纯 TEX 的) 解析命令 \pgfmathparse 解析表达式，而 luamath 库只负责函数计算 (采用 lua 函数做计算)；这是默认的工作方式。
2. 解析表达式、计算函数值 (采用 lua 函数做计算) 的工作都由 luamath 库完成。

无论哪一种工作方式，被解析的表达式都应当是普通的 PGF 数学引擎能接受的表达式。

/pgf/luamath/only computation　(no value)

这个键使得 luamath 库采用第 1 种工作方式。这是默认的。

此时，命令 \pgfmathparse 仍然是 PGF 的 (纯 TEX 的) 解析命令，但“私人版”的数学函数会被改为 lua 版函数。

/pgf/luamath/parser and computation　(no value)

这个键使得 luamath 库采用第 2 种工作方式。

此时，命令 \pgfmathparse 被 let 为 \pgfluamathparse→P. 655，“私人版”的数学函数会被改为 lua 版函数。

/pgf/luamath/parser (no value)

这个键使得表达式由 luamath 库来解析。

/pgf/luamath/off (no value)

这个键取消 luamath 库的解析、计算工作。

/pgf/luamath=only computation|parser and computation|parser|off (default, initially only computation)

这个键决定 luamath 库的工作方式，如上述，默认值和初始值都是 only computation.

/pgf/luamath/output format/fixed (no value)

这个键使得 luamath 库最后计算出来的结果采用定点数形式。这是默认的。

/pgf/luamath/output format/float (no value)

这个键使得 luamath 库最后计算出来的结果采用浮点数形式，即 fpu 库所采用的浮点数形式。如果需要向 fpu 库的命令传递计算结果，可以使用这个键。

/pgf/luamath/output format=fixed|float (initially fixed)

这个键决定 luamath 库最后计算出来的结果采用定点数形式还是浮点数形式，如上述，初始之下是 fixed.

当 luamath 库的解析、计算失败时，有 3 种选择：

1. 发布错误信息，中止编译，这对应真值：

```
\pgfluamathshowerrormessagetrue
```

2. 不发布错误信息，采用 PGF 的 (纯 TEX 的) 的数学命令来做解析、计算，这对应 2 个真值：

```
\pgfluamathenableTeXfallbacktrue
\pgfluamathshowerrormessagefalse
```

这是默认的。

3. 不发布错误信息，也不采用 PGF 的 (纯 TEX 的) 的数学命令来做解析、计算，这对应 2 个真值：

```
\pgfluamathenableTeXfallbackfalse
\pgfluamathshowerrormessagefalse
```

此时 \pgfmathresult 是空的 (等于 \pgfutil@empty)，并且 \ifpgfmathunitsdeclared 的真值是 false.

/pgf/luamath/show error message=true|false (initially true)

这个键决定 \ifpgfluamathshowerrormessage 的真值。

/pgf/luamath/enable TeX fallback/true (no value)

这个键决定这 2 个真值：

```
\pgfluamathenableTeXfallbacktrue
\pgfluamathshowerrormessagefalse
```

/pgf/luamath/enable TeX fallback/false (no value)

这个键决定这 2 个真值：

```
\pgfluamathenableTeXfallbackfalse
\pgfluamathshowerrormessagetrue
```

/pgf/luamath/enable TeX fallback=true|false (default, initially true)

这个键决定：当 luamath 库的解析、计算失败时，做什么选择，如上述，默认值和初始值都是 true.

37.3 luamath 库支持的函数

文件《pgflibraryluamath.code.tex》有以下定义：

```
\def\pgfluamath@install@function#1=#2{%
  \pgfluamath@prepareuninstallcmd{#1}%
  \let#1=#2%
}%
\def\pgfluamath@install{%
  \pgfluamath@install@function\pgfmathadd@=\pgfluamathadd@%
  ...
}
\def\pgfluamathgetresult#1{%
  \edef\pgfmathresult{\pgfutil@directlua{tex.print(-1,#1)}}}%
\def\pgfluamathadd@#1#2{%
  \pgfluamathgetresult{pgfluamathfunctions.add(#1,#2)}}%
```

以上代码定义了 \pgfluamathadd@, 并且把“私人版”的 \pgfmathadd@ 命令 let 为 \pgfluamathadd@. 当使用“公共版”命令 \pgfmathadd 时，实际上会调用 \pgfluamathadd@.

注意，luamath 库支持的函数是有限的，详见文件《pgflibraryluamath.code.tex》。

通常，luamath 库不能识别用户使用命令 \pgfmathdeclarefunction →P. 129 自定义的函数。

37.4 luamath 库的解析命令

\pgfluamathparse{⟨*expression*⟩}

这个命令是 luamath 库的解析或计算表达式 ⟨*expression*⟩ 的命令。

在选项 luamath/parser and computation, parser 要求 luamath 库解析表达式的情况下，会有

```
\let\pgfmathparse\pgfluamathparse
```

\pgfluamath@pgfmathparse

这个命令等于 PGF 的 (纯 TEX 的) 解析命令 \pgfmathparse.

```
\let\pgfluamath@pgfmathparse\pgfmathparse
```

luamath 库的解析、计算结果仍然保存在宏 \pgfmathresult 中。

\pgfluamathgetresult{⟨*lua math expression*⟩}

⟨*lua math expression*⟩ 是 lua 的数学表达式，或者保存 lua 数学表达式的宏。

这个命令调用 \directlua 计算 ⟨*lua math expression*⟩, 结果被保存到宏 \pgfmathresult 中。

-0.95886

```
\pgfkeys{/pgf/trig format/rad}
\pgfmathparse{sin(5)}\pgfmathresult% 纯 TeX 的计算结果
```

-0.958924 对比 -0.95892427466314

```
\pgfkeys{/pgf/trig format/rad,/pgf/luamath/parser and computation}
\pgfmathparse{sin(5)}\pgfmathresult
对比
\pgfluamathgetresult{math.sin(5)}\pgfmathresult
```

注意上面例子中，两个输出数值的小数位数不同，其原因在于：`\pgfluamathparse` 在输出最后的计算结果前，会用

```
discardTrailingZeros(stringformat("%f", x))
```

处理计算结果 (其中参数 `x` 代表计算结果) 中多余的“0”字符，而 `stringformat("%f", x)` 就是函数 `string.format("%f", x)`, 在 lua 程序中，这个函数返回 6 位的小数部分 (舍入后的结果)。

37.5 自定义数学函数

37.5.1 定义 lua 数学函数

例如

0.041075725336862

```
\directlua{
  function mycalc (x,y)
    return math.sin(x) + math.cos(y)
  end
}
\directlua{tex.print(mycalc(5,0))}
```

37.5.2 定义 luamath 库的数学函数

参考《functions.lua》和《pgflibraryluamath.code.tex》的做法。

0.041076

```
\directlua{
  function pgfluamathfunctions.mytricalc (x,y)
    return math.sin(x) + math.cos(y)
  end
}
\makeatletter
\pgfluamath@install@function\pgfmathmytricalc@=\pgfluamathmytricalc@%
\def\pgfluamathmytricalc@#1#2{%
  \pgfluamathgetresult{pgfluamathfunctions.mytricalc(#1,#2)}}%
\makeatother
\pgfluamathparse{mytricalc(5,0)}\pgfmathresult
```

第三十八章　图柄库

TikZ Library plothandlers

```
\usepgflibrary{plothandlers} % LaTeX and plain TeX and pure pgf
\usepgflibrary[plothandlers] % ConTeXt and pure pgf
\usetikzlibrary{plothandlers} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[plothandlers] % ConTeXt when using TikZ
```

这个库提供一些图柄（plot handlers），另外可以参考基本层的命令 \pgfplothandlerlineto→P. 494 等。

TikZ 会自动加载这个库。

用 \pgfsetmovetofirstplotpoint→P. 494 或 \pgfsetlinetofirstplotpoint→P. 494 可以改变画线图柄对图流的第一个点的处理。

先定义 3 个图流，以便在后文的例子中引用。

```
\pgfplothandlerrecord{\lizione}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{-1cm}}
  \pgfplotstreampoint{\pgfpoint{2.5cm}{-.5cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{-1cm}}
  \pgfplotstreamend
```

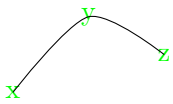
```
\pgfplothandlerrecord{\lizitwo}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
```

```
\pgfplothandlerrecord{\lizithree}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreampoint{\pgfpoint{4cm}{0.7cm}}
  \pgfplotstreampoint{\pgfpoint{5cm}{0.5cm}}
  \pgfplotstreampoint{\pgfpoint{6cm}{1cm}}
  \pgfplotstreamend
```

38.1　曲线图柄

\pgfplothandlercurveto

这个图柄把命令 \pgfpathcurveto 用于图流的点（图流的第一个点除外）。这个图柄的定义见文件《pgflibraryplothandlers.code.tex》，参考命令 \pgfdeclareplothandler→P. 497.

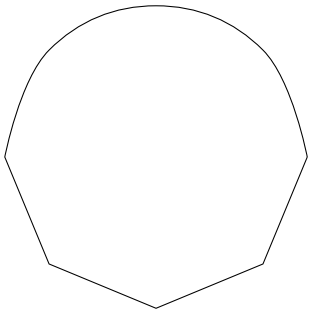



```
\begin{tikzpicture}
  \draw[green] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlercurveto
  \lizitwo
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfsetplottension{⟨*value*⟩}

本命令保存一个数值,默认保存值是 0.5. 命令 \pgfplothandlercurveto, \pgfplothandlerclosedcurve 会利用本命令保存的数值来做计算。若有 4 个样本点均匀分布于一个圆上且张力值 ⟨*value*⟩ 是 1，则绘制的曲线是个圆。

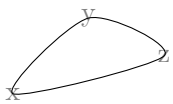
当用命令 \pgfplothandlercurveto 处理图流时，可以在图流中插入本命令来改变张力数值。



```
\tikz{
  \pgfplothandlercurveto
  \pgfsetplottension{0.7}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpointpolar{0}{2cm}}
  \pgfplotstreampoint{\pgfpointpolar{45}{2cm}}
  \pgfplotstreampoint{\pgfpointpolar{90}{2cm}}
  \pgfplotstreampoint{\pgfpointpolar{135}{2cm}}
  \pgfplotstreampoint{\pgfpointpolar{180}{2cm}}
  \pgfsetplottension{0}
  \pgfplotstreampoint{\pgfpointpolar{225}{2cm}}
  \pgfplotstreampoint{\pgfpointpolar{270}{2cm}}
  \pgfplotstreampoint{\pgfpointpolar{315}{2cm}}
  \pgfplotstreampoint{\pgfpointpolar{360}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
}
```

\pgfplothandlerclosedcurve

这个图柄类似 \pgfplothandlercurveto，只是这个图柄会利用图流创建闭合路径。

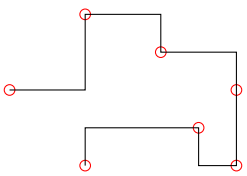



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerclosedcurve
  \lizitwo
  \pgfusepath{stroke}
\end{tikzpicture}
```

38.2 Constant 图柄

\pgfplothandlerconstantlineto

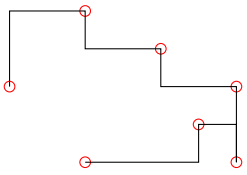
这个图柄的作用参考 plot[const plot].



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizione
  \pgfplothandlerconstantlineto
  \lizione
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerconstantlinetomarkright

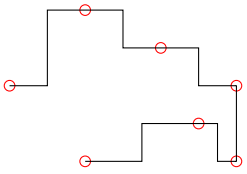
这个图柄的作用参考 plot[const plot mark right].



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizione
  \pgfplothandlerconstantlinetomarkright
  \lizione
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerconstantlinetomarkmid

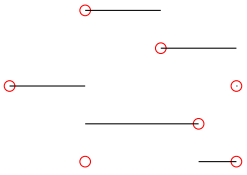
这个图柄的作用参考 plot[const plot mark mid].



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizione
  \pgfplothandlerconstantlinetomarkmid
  \lizione
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerjumpmarkleft

这个图柄的作用参考 plot[jump mark left].



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizione
  \pgfplothandlerjumpmarkleft
  \lizione
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerjumpmarkright

这个图柄的作用参考 plot[jump mark right].

\pgfplothandlerjumpmarkmid

这个图柄的作用参考 plot[jump mark mid].

38.3 Comb 图柄

\pgfplothandlerxcomb

这个图柄的作用参考 plot[xcomb].

\pgfplothandlerycomb

这个图柄的作用参考 plot[ycomb].

\pgfplothandlerpolarcomb

这个图柄的作用参考 plot[polar comb].

\pgfplotxzerolevelstreamconstant{⟨*dimension*⟩}

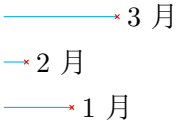
这个命令只对 xcomb 或 xbar 这两种图形有效，对其它图形无效。本命令使得图形中的“细棒”或者 bar 的起点平移到直线 $y = \langle dimension \rangle$ 上。

假设某个部门在 1 月，2 月，3 月的盈利情况如下表：

各月盈利		
1 月	2 月	3 月
15	5	25

各月相对 1 月的盈利		
1 月	2 月	3 月
0	-10	10

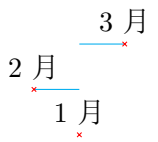
用下图表示上面第一个表格：




```
\begin{tikzpicture}[scale=0.6,x=0.1*1cm]
  \draw [cyan] plot[xcomb,mark=x,mark options={color=red}]
    coordinates {(15,1)(5,2)(25,3)};
  \foreach \Yue/\yue in {(15,1)/1,(5,2)/2,(25,3)/3}
    \node [right] at \Yue {\yue 月};
\end{tikzpicture}
```

这个图形大体上可以比较各月的相对盈利程度。

为了用类似的图形表示上面第二个表格，可以修改上面图形中的点的坐标，不过也可以使用命令 \pgfplotxzerolevelstreamconstant：




```
\begin{tikzpicture}[scale=0.6,x=0.1*1cm]
  \pgfplotxzerolevelstreamconstant{1.5cm}
  \draw [cyan] plot[xcomb,mark=x,mark options={color=red}]
    coordinates {(15,1)(5,2)(25,3)};
  \foreach \Yue/\yue in {(15,1)/1,(5,2)/2,(25,3)/3}
    \node [above] at \Yue {\yue 月};
\end{tikzpicture}
```

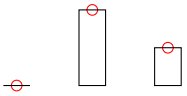
\pgfplotyzerolevelstreamconstant{⟨*dimension*⟩}

类似 \pgfplotxzerolevelstreamconstant，本命令只对 ycomb 或 ybar 这两种图形有效，对其它图形无效。

38.4 Bar 图柄

\pgfplothandlerybar

这个图柄的作用参考 plot[ybar].



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizitwo
\pgfplotxzerolevelstreamconstant{1cm}
  \pgfplothandlerybar
  \lizitwo
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerxbar

这个图柄的作用参考 plot[xbar].

对于 Bar 类型的柱状图，有以下选项可以调节其外观。

/pgf/bar width={⟨*dimension*⟩} (no default, initially 10pt)

/tikz/bar width

这个选项设置柱状图的各个 bar 的宽度。对于 ybar 图形，bar 的宽度指的是其 x 轴方向的尺寸；对于 xbar 图形，bar 的宽度指的是其 y 轴方向的尺寸。⟨*dimension*⟩ 可以是带有长度单位的表达式，会被数学解析器解析。

/pgf/bar shift={⟨dimension⟩} (no default, initially 0pt)

/tikz/bar shift

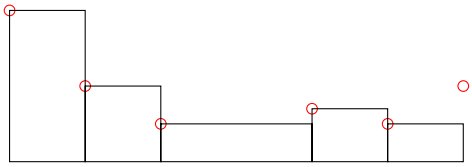
这个选项的作用类似平移选项 xshift，yshift，但只对柱状图中的 bar 有效。对于 ybar 图形，bar shift 指的是各个 bar 在 x 轴方向的平移；对于 xbar 图形，bar shift 指的是各个 bar 在 y 轴方向的平移。⟨dimension⟩ 可以是带有长度单位的表达式，会被数学解析器解析。

\pgfplotbarwidth

这个宏的展开值是选项 /pgf/bar width 的值。

\pgfplothandlerybarinterval

这个图柄的作用参考 plot[ybar interval].



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizithree
\pgfplotxzerolevelstreamconstant{1cm}
  \pgfplothandlerybarinterval
  \lizithree
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerxbarinterval

这个图柄的作用参考 plot[xbar interval].

对于 bar interval 类型的柱状图，有以下选项可以调整其外观。

/pgf/bar interval shift=⟨factor⟩ (no default, initially 0.5)

/tikz/bar interval shift

/pgf/bar interval width=⟨scale⟩ (no default, initially 1)

/tikz/bar interval width

注意以上这 4 个选项的值不是尺寸，而是数字或者运算结果为数字的表达式，⟨factor⟩ 与 ⟨scale⟩ 都会被数学解析器解析。

对于 ybar interval 类型的柱状图，其绘制方式如下：设 (x_i, y_i) 与 (x_{i+1}, y_{i+1}) 是图流中前后相继的两个点，图柄会在这两个点之间构造一个矩形；矩形中心点的横坐标是 $x_i + ⟨factor⟩ \cdot (x_{i+1} - x_i)$；矩形宽度（水平方向的尺寸）是 $⟨scale⟩ \cdot (x_{i+1} - x_i)$；矩形的“身高”是 $\|y_i\|$；图流的最后一个点“落单”。

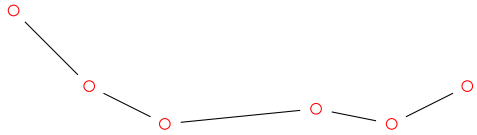
对于 xbar interval 类型的柱状图，其绘制方式如下：设 (x_i, y_i) 与 (x_{i+1}, y_{i+1}) 是图流中前后相继的两个点，图柄会在这两个点之间构造一个矩形；矩形中心点的纵坐标是 $y_i + ⟨factor⟩ \cdot (y_{i+1} - x_i)$；矩形宽度（竖直方向的尺寸）是 $⟨scale⟩ \cdot (y_{i+1} - y_i)$；矩形的“水平长度”是 $\|x_i\|$；图流的最后一个点“落单”。

38.5 Gapped 图柄

\pgfplothandlergaplineto

这个图柄会在图流的点之间画直线段，但每个直线段的起点和终点并非恰好落在图流的点上，而是距离图流的点还有一段距离，从而造成“缺口”效果。这段距离由下面的选项指定：

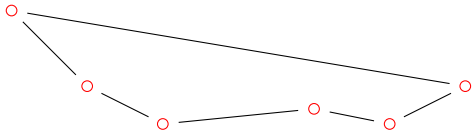
/pgf/gap around stream point=⟨*dimension*⟩ (no default, initially 1.5pt)



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizithree
  \pgfplotxzerolevelstreamconstant{1cm}
  \pgfplothandlergaplineto
  \pgfkeys{/pgf/gap around stream point=6pt}
  \lizithree
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlergapcycle

这个图柄会在图流的点之间画直线段，并把路径作成闭合的多边形，但多边形每个边的起点和终点并非恰好落在顶点上，而是距离顶点还有一段距离，从而造成“缺口”效果。这段距离也是由上面的选项 /pgf/gap around stream point 指定。



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizithree
  \pgfplotxzerolevelstreamconstant{1cm}
  \pgfplothandlergapcycle
  \pgfkeys{/pgf/gap around stream point=6pt}
  \lizithree
  \pgfusepath{stroke}
\end{tikzpicture}
```

38.6 Mark 图柄

\pgfplothandlermark{⟨*mark code*⟩}

这个命令用于自定义一种点标记，⟨*mark code*⟩ 是绘制点标记的命令。有两种编写 ⟨*mark code*⟩ 的思路：

1. 使用已定义的标记，例如可以使用库 plotmarks 定义的标记，前文的例子中有如下代码：

   ```
   \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
   ```

   其中用命令 \pgfuseplotmark{o} 调用了标记类型“o”.

2. 用绘图命令自定义一种标记。首先你要假设有一个“标记坐标系”，在这个坐标系中用绘图命令画出标记。当用自定义标记来标记某个点时，自定义标记的“标记坐标系”的原点会放在“被标记”的点上。


```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{
    \pgfpathcircle{\pgfpointorigin}{4pt}
    \pgfusepath{stroke}}
  \lizitwo
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfsetplotmarkrepeat{⟨repeat⟩}

这个命令的效果参考 plot[mark repeat=⟨r⟩].

\pgfsetplotmarkphase{⟨phase⟩}

这个命令的效果参考 plot[mark phase=⟨p⟩].

\pgfplothandlermarklisted{⟨mark code⟩}{⟨index list⟩}

这个命令的效果参考 plot[mark indices=⟨list⟩]. ⟨mark code⟩ 规定或定义一种标记，⟨index list⟩ 用来指定被标记的点。


```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermarklisted{
    \pgfpathcircle{\pgfpointorigin}{4pt}
    \pgfusepath{stroke}}
  {1,3}
  \lizitwo
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfuseplotmark{⟨plot mark name⟩}

⟨plot mark name⟩ 是某个已定义的标记，本命令调用这个类型的标记。

\pgfdeclareplotmark{⟨plot mark name⟩}{⟨code⟩}

本命令定义一种名称为 ⟨plot mark name⟩ 的点标记；绘图代码 ⟨code⟩ 是对点标记的具体定义；在本命令之后可以用命令 \pgfuseplotmark{⟨plot mark name⟩} 引用自定义的点标记。在编写 ⟨code⟩ 时，你也要假设有一个“标记坐标系”，在这个坐标系中用绘图命令画出标记。当用自定义标记来标记某个点时，自定义标记的“标记坐标系”的原点会放在“被标记”的点上。


```
\pgfdeclareplotmark{my plot mark}{
  \pgfpathcircle{\pgfpoint{0cm}{1ex}}{1ex}
  \pgfusepathqstroke}
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfuseplotmark{my plot mark}}
  \lizitwo
  \pgfusepath{stroke}
\end{tikzpicture}
```

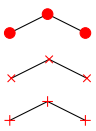
\pgfsetplotmarksize{⟨dimension⟩}

这个命令的作用参考 plot[mark size=⟨dimension⟩]. 本命令将 TEX 的尺寸宏 \pgfplotmarksize 的值设为 ⟨dimension⟩，用以规定点标记的“半径”。这个值是个“推荐值”，在某些情况下这个值会被忽略。所有预定义的点标记都使用本命令的参数 ⟨dimension⟩.

\pgfplotmarksize

这是个 TEX 的尺寸宏，它的值是标记的“推荐值”。

PGF 预定义 3 种类型的 plot 标记，其“名称”分别是：*,x, +，这是 3 个符号，分别对应小圆圈、叉号、加号。



```
\tikz \draw plot[mark=*,mark options={color=red}]
  coordinates{(0,0)(0.5,0.25)(1,0)};\\
\tikz \draw plot[mark=x,mark options={color=red}]
  coordinates{(0,0)(0.5,0.25)(1,0)};\\
\tikz \draw plot[mark=+,mark options={color=red}]
  coordinates{(0,0)(0.5,0.25)(1,0)};
```

第六部分

TikZ

第三十九章　TikZ 环境

39.1　载入宏包和程序库

```
\usepackage{tikz} % LaTeX
\input tikz.tex % plain TeX
\usemodule[tikz] % ConTeXt
```

这个宏包没有选项. 调用 TikZ 宏包时会自动调用 PGF, pgffor 宏包. 当使用 LaTeX 格式时, PGF 几乎总能够自动判断用户使用的驱动. 但在 plain TeX 或 ConTeXt 格式下使用 dvipdfm 驱动时, PGF 不能自己识别你所用的驱动. 在此情况下, 你需要在载入 TikZ 宏包前定义命令

```
\def\pgfsysdriver{pgfsys-dvipdfm.def}
```

这个载入命令会载入路径 ...tex/latex/pgf/frontendlayer 下的文件《tikz.sty》, 这又会导致文件其他被载入。

\usetikzlibrary{⟨*list of libraries*⟩}

当载入 TikZ 后，就可以用这个命令载入各种库 (libraries). 列出的库名称之间用逗号分隔。本命令中的花括号可以换成方括号。

```
\usetikzlibrary[⟨list of libraries⟩]
```

例如

```
\usetikzlibrary{arrows.meta,animations}
```

对于 ⟨*list of libraries*⟩ 中的每个 library 名称，本命令会载入文件 tikzlibrary⟨*library*⟩.code.tex；如果这个文件不存在，就载入文件 pgflibrary⟨*library*⟩.code.tex，如果这个文件也不存在，就给出错误信息。如果你自己编写 library 的话，文件名称要恰当 (让 PGF 能识别它)，文件位置也要恰当 (让 TeX 能找到它)。

39.2　创建一个 picture

39.2.1　{tikzpicture} 环境

TikZ 的最外层绘图区域是 {tikzpicture} 环境，所有绘图命令都要放在环境内。环境内的选项仅在环境内有效。该环境可以用于大多数 LaTeX 模式、环境、命令内，例如用于页眉、页脚命令内，数学模式内。

\begin{tikzpicture}⟨*animations spec*⟩[⟨*options*⟩]
⟨*environment content*⟩
\end{tikzpicture}

- 多数 TikZ 命令，例如 \path，只能用在这个环境里。命令 \tikzset 可以不用在此环境内。很多底层的图形命令如 \pgfpathmoveto 也可以用在这个环境里。

- ⟨*options*⟩ 是环境选项列表，其中的选项会被用于整个环境 (picture)。
- 载入 animations 库后，可以在选项前面指定动画命令 (animation command)。
- 环境的内容 ⟨*environment content*⟩ 被处理，图形命令会被放入盒子中。在环境结束时，PGF 会估计整个图形的边界盒子 (bounding box) 的尺寸，然后调整图形盒子 (picture box) 到达这个尺寸，再确定盒子的基线，然后将盒子放到 TEX 的处理流程中。每遇到一个坐标时 PGF 都会刷新 bounding box 的尺寸，直到 bounding box 包含所有坐标。有时这样估计出来的 bounding box 尺寸不够准确，例如，倾斜线条的线宽 (线条的粗细尺寸) 不被精确计算，曲线的控制点有时会距离曲线较远，以至于 bounding box 过大。此时你可以用选项 `/tikz/use as bounding box` →P. 777 来调整边界盒子。
- 环境中不属于图形命令的文字会被临时转为 `\nullfont` 字体，从而被抑制。
- 各种非 PGF 的命令也不会输出到图形中，因为这会扰乱驱动对位置的计算。
- 注意以下两个选项的执行次序：

```
\tikzset{every scope/.try}%
\tikzset{every picture,⟨options⟩}%
```

```
\tikzpicture[⟨options⟩]
⟨environment contents⟩
\endtikzpicture
```

这是图形环境在 plain TeX 中的版本。

```
\starttikzpicture[⟨options⟩]
⟨environment contents⟩
\stoptikzpicture
```

这是图形环境在 ConTeXt 中的版本。

39.2.1.1 命令 \begin{tikzpicture}

命令 `\begin{tikzpicture}[⟨options⟩]` 导致 `\tikzpicture[⟨options⟩]`.

`\tikzpicture⟨animations spec⟩[⟨options⟩]`

这个命令的定义是：

```
\def\tikzpicture{%
  \begingroup%
    \tikz@startup@env%
    \tikz@collect@scope@anims\tikz@picture}%
\def\tikz@picture[#1]{%
  %\tikz@check@inside@picture%
  \pgfpicture%
  \let\tikz@atbegin@picture=\pgfutil@empty%
  \let\tikz@atend@picture=\pgfutil@empty%
  \let\tikz@transform=\relax%
  \def\tikz@time{.5}%
  \tikz@installcommands%
  \scope[every picture,#1]%
  \iftikz@handle@active@code%
    \tikz@switchoff@shorthands%
  \fi%
  \expandafter\tikz@atbegin@picture%
  \tikz@lib@scope@check%
}%
```

注释：

- 先用 \begingroup 开启一个组。
- 命令 \tikz@startup@env 对一些特殊符号“;”，“;”，“:”，“|”，“,”，“<”，“>”，“"”，“'”，“-”，“=”，“.”，“$”作处理：第一，如果这些符号是活动符，就把它们作为活动符的定义另外保存起来；第二，将这些符号转换成类代码为 12 的普通符号。
- 命令 \tikz@collect@scope@anims

\tikz@collect@scope@anims⟨\macro⟩

此命令的定义是：

```
\def\tikz@collect@scope@anims#1{%
  \pgfutil@ifnextchar[#1{#1[]}%]
}%
```

如果载入 animations 库，那么库文件《tikzlibraryanimations.code.tex》会重定义这个命令，用于收集动画命令。

- 定义 \def\tikz@time{.5} 一般决定自动放置 node 的位置。
- 命令 \tikz@installcommands 会定义 \path, \draw, \scope 等命令，其定义是：

```
\def\tikz@installcommands{%
  \let\tikz@origscope=\scope%
  \let\tikz@origscoped=\scoped%
  \let\tikz@origendscope=\endscope%
  \let\tikz@origstartscope=\startscope%
  \let\tikz@origstopscope=\stopscope%
  \let\tikz@origpath=\path%
  \let\tikz@origagainpath=\agains path%
  \let\tikz@origdraw=\draw%
  \let\tikz@origpattern=\pattern%
  \let\tikz@origfill=\fill%
  \let\tikz@origfilldraw=\filldraw%
  \let\tikz@origshade=\shade%
  \let\tikz@origshadedraw=\shadedraw%
  \let\tikz@origclip=\clip%
  \let\tikz@origuseasboundingbox=\useasboundingbox%
  \let\tikz@orignode=\node%
  \let\tikz@origpic=\pic%
  \let\tikz@origcoordinate=\coordinate%
  \let\tikz@origmatrix=\matrix%
  \let\tikz@origcalendar=\calendar%
  \let\tikz@origdv=\datavisualization%
  \let\tikz@origgraph=\graph%
  %
  \let\scope=\tikz@scope@env%
  \let\scoped=\tikz@scoped%
  \let\endscope=\endtikz@scope@env%
  \let\startscope=\scope%
  \let\stopscope=\endscope%
  \let\path=\tikz@command@path%
  \let\againpath=\tikz@command@againpath%
  %
  \def\draw{\path[draw]}%
  \def\pattern{\path[pattern]}%
  \def\fill{\path[fill]}%
  \def\filldraw{\path[fill,draw]}%
  \def\shade{\path[shade]}%
```

```
  \def\shadedraw{\path[shade,draw]}%
  \def\clip{\path[clip]}%
  \def\graph{\path graph}%
  \def\useasboundingbox{\path[use as bounding box]}%
  \def\node{\tikz@path@overlay{node}}%
  \def\pic{\tikz@path@overlay{pic}}%
  \def\coordinate{\tikz@path@overlay{coordinate}}%
  \def\matrix{\tikz@path@overlay{node[matrix]}}%
  \def\calendar{\tikz@lib@cal@calendar}%
  \def\datavisualization{\tikz@lib@datavisualization}%
}%
```

- 执行命令 \tikz@installcommands 后，命令 \scope 等于 \tikz@scope@env, 是对应 {scope} 环境的命令。参考 \begin{scope} 命令。
- 按 \begin{scope} 的定义 (见后文)，选项的执行次序是：

```
\tikzset{every scope/.try}%
\tikzset{every picture,⟨options⟩}%
```

- 命令 \tikz@switchoff@shorthands 还是处理特殊符号与活动符的定义。
- 宏 \tikz@atbegin@picture 与选项 /tikz/execute at begin picture→P. 670 对应，其定义是：

```
\tikzoption{execute at begin picture}{
↪   \expandafter\def\expandafter\tikz@atbegin@picture\expandafter{
↪   \tikz@atbegin@picture#1}}%
%......
\let\tikz@atbegin@picture=\pgfutil@empty
```

  宏 \tikz@atbegin@picture 保存选项 execute at begin picture 的值。如果多次使用这个选项，那么这些选项值会依次保存到这个宏中。
- 宏 \tikz@lib@scope@check 由文件《tikzlibraryscopes.code.tex》定义，但也可能被其他命令重定义。

39.2.1.2 命令 \end{tikzpicture}

命令 \end{tikzpicture} 导致执行命令 \endtikzpicture.

\endtikzpicture

这个命令的定义是：

```
\def\endtikzpicture{%
    \tikz@atend@picture%
    \global\let\pgf@shift@baseline@smuggle=\pgf@baseline%
    \global\let\pgf@trimleft@final@smuggle=\pgf@trimleft%
    \global\let\pgf@trimright@final@smuggle=\pgf@trimright%
    \global\let\pgf@remember@smuggle=\ifpgfrememberpicturepositiononpage%
    \pgf@remember@layerlist@globally
    \endscope%
    \let\pgf@baseline=\pgf@shift@baseline@smuggle%
    \let\pgf@trimleft=\pgf@trimleft@final@smuggle%
    \let\pgf@trimright=\pgf@trimright@final@smuggle%
    \let\ifpgfrememberpicturepositiononpage=\pgf@remember@smuggle%
    \pgf@restore@layerlist@from@global
  \endpgfpicture\endgroup}%
```

注释：

- 命令 \tikz@atend@picture 与选项 /tikz/execute at end picture→P. 671 对应，其定义是：

```
\tikzoption{execute at end picture}{
↪  \expandafter\def\expandafter\tikz@atend@picture\expandafter{
↪  \tikz@atend@picture#1}}%
%......
\let\tikz@atend@picture=\pgfutil@empty
```

宏 \tikz@atend@picture 保存选项 execute at end picture 的值。如果多次使用这个选项，那么这些选项值会依次保存到这个宏中。

- 在

```
\global\let\pgf@shift@baseline@smuggle=\pgf@baseline%
\global\let\pgf@trimleft@final@smuggle=\pgf@trimleft%
\global\let\pgf@trimright@final@smuggle=\pgf@trimright%
\global\let\pgf@remember@smuggle=\ifpgfrememberpicturepositiononpage%
\pgf@remember@layerlist@globally
```

中的命令参考文件《pgfcorescopes.code.tex》；命令 \pgf@remember@layerlist@globally 会调用文件《pgfcorelayers.code.tex》的命令 \pgf@layerlist，参考 \pgfsetlayers→P. 358.

- 执行 \endscope，结束由命令 \tikzpicture 引入的 \scope. 在执行 \tikz@installcommands 后，\endscope 等于 \endtikz@scope@env. 参考命令 \end{scope}.
- 命令 \pgf@restore@layerlist@from@global 参考文件《pgfcorescopes.code.tex》。
- 执行 \endpgfpicture，结束由命令 \tikzpicture 开启的 \pgfpicture. 整个图形的边界、基线位置在此时计算出来，然后把各个装有图层的盒子依次放到页面上。
- 执行 \endgroup，结束由命令 \tikzpicture 开启的 \begingroup.

39.2.1.3 几个选项

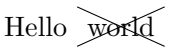
/tikz/baseline=⟨*dimension or coordinate or* ***default***⟩ (default 0pt)

在初始之下，图形的最下端会被放在图形周围文字的基线上。

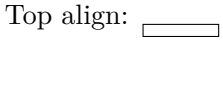
```
xx\tikz{
  \draw[line width=6pt](0,0)circle(10pt);
}xx
```

使用本选项可以把图形中的水平直线 $y = \langle value \rangle$ 作为图形的基线，放在图形周围文字的基线上（⟨*value*⟩是本选项的值）。选项值可以是：

- ⟨*dimension*⟩，带有长度单位的尺寸。
- ⟨*coordinate*⟩，坐标，如果 ⟨*coordinate*⟩ 中有逗号则需要用花括号把 ⟨*coordinate*⟩ 括起来，把图形中通过这个点的水平直线作为图形的基线。在 picture 环境的末尾处才会计算基线位置，所以 ⟨*coordinate*⟩ 可以是图形中 node 的坐标。



```
Hello
\tikz[baseline=(X.base)]
  \node [cross out,draw] (X) {world};
```



```
Top align:
\tikz[baseline=(current bounding box.north)]
  \draw (0,0) rectangle (1cm,1ex);
```

用 baseline=default 恢复本选项的初始设置，即图形的最下端被放在基线上。

此选项的定义是：

```
\tikzoption{baseline}[0pt]{%
  \pgfutil@ifnextchar(%)
```

```
    {\tikz@baseline@coordinate}{\tikz@baseline@simple}#1\@nil}%
\def\tikz@baseline@simple#1\@nil{\pgfsetbaseline{#1}}%
\def\tikz@baseline@coordinate#1\@nil{%
  \pgfsetbaselinepointlater{\tikz@scan@one@point\pgfutil@firstofone#1}}%
```

这个选项使用基本层的命令 \pgfsetbaseline→P. 239, \pgfsetbaselinepointlater→P. 240,

```
\def\pgfsetbaselinepointlater#1{\def\pgf@baseline{#1}}
%...
\def\pgf@default@text{default}%
\def\pgfsetbaseline#1{%
  \def\pgf@temp{#1}%
  \ifx\pgf@temp\pgf@default@text
    \pgfsetbaseline{\pgf@picminy}%
  \else
    \pgfsetbaselinepointlater{\pgfpoint{0pt}{#1}}%
  \fi
}
\pgfsetbaseline{\pgf@picminy}
```

可见，对于 baseline=⟨*argu*⟩:

- 如果参数 ⟨*argu*⟩ 不以开圆括号“(”开头，那么就定义

  ```
  \def\pgf@baseline{\pgfpoint{0pt}{⟨argu⟩}}
  ```

  按 \pgfpoint→P. 250 的定义，参数 ⟨*argu*⟩ 会被 \pgfmathsetlength 处理。
- 如果参数 ⟨*argu*⟩ 以开圆括号“(”开头，那么就定义

  ```
  \def\pgf@baseline{\tikz@scan@one@point\pgfutil@firstofone⟨argu⟩}
  ```

  按 \tikz@scan@one@point→P. 710 的定义，⟨*argu*⟩ 中的坐标表达式会被 \pgfmathparse 解析。注意，命令 \tikz@scan@one@point 在解析 TikZ 坐标时，一般不会考虑坐标变换矩阵。

在 \end{tikzpicture}(即 \endpgfpicture) 那里会执行

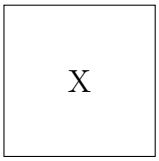
```
\pgf@process{\pgf@baseline}%
\xdef\pgf@shift@baseline{\the\pgf@y}%
```

把基线尺寸全局地保存到 \pgf@shift@baseline, 最后 \pgfsys@typesetpictureboxbox→P. 204 会根据这个基线尺寸决定盒子 \pgfpic (内含整个 picture) 的基线。

由于基线是在 \endpgfpicture 那里计算的，所以参数 ⟨*argu*⟩ 可以是环境中的任何一个点。

/tikz/execute at begin picture=⟨*code*⟩ (no default)

这个选项使得 ⟨*code*⟩ 在图形开始的时候被执行，本选项用作环境 {tikzpicture} 的选项才有效。如果多次使用本选项，则其作用会被累计，即提供的各 ⟨*code*⟩ 会被依次执行。




```
\begin{tikzpicture}[execute at begin picture=%
  { \draw (0,0) rectangle (2,2); }]
  \node at (1,1) {\large X};
\end{tikzpicture}
```

本选项的定义是：

```
\tikzoption{execute at begin picture}{
↪  \expandafter\def\expandafter\tikz@atbegin@picture\expandafter{
↪  \tikz@atbegin@picture#1}}%
%...
\let\tikz@atbegin@picture=\pgfutil@empty
```

本选项重定义宏 \tikz@atbegin@picture.

/tikz/execute at end picture=⟨code⟩ (no default)

这个选项使得 ⟨code⟩ 在图形结束的时候被执行，本选项用作环境 {tikzpicture} 的选项才有效。如果多次使用本选项，则其作用会被累计，即提供的各 ⟨code⟩ 会被依次执行。


```
\begin{tikzpicture}[execute at end picture=%
  {
    \begin{pgfonlayer}{background}
      \path[fill=red!30,rounded corners]
        (current bounding box.south west) rectangle
        (current bounding box.north east);
    \end{pgfonlayer}
  }]
  \node at (0,0) {X};
  \node at (2,1) {Y};
\end{tikzpicture}
```

/tikz/every picture (style, initially empty)

这个样式 (style) 选项会被放到每个 tikzpicture 环境的开始处，也就是说，先执行这个样式，再执行其他环境选项。如果要用这个选项设置某些东西，可以使用 every picture/.style 这种格式，例如：

```
\tikzset{every picture/.style={draw=red,semithick,execute at end picture={...}}}
```

39.2.2 用命令创建一个 picture

\tikz⟨animations spec⟩[⟨options⟩]{⟨path command⟩}

这个命令创建一个 {tikzpicture} 环境，并把 {⟨path command⟩} 放到该环境中。{⟨path command⟩} 中可以包含段落，脆弱命令 (如抄录命令)。如果只有一个路径命令，那么也可以不用花括号；如果有数个路径命令，那么就必须用花括号把它们括起来。

39.2.3 处理类代码，babel 宏包

在 TikZ 图形中，多数代码符号的类别是 12，即普通的文字符号，这有利于解析器正常工作。但是如果载入了某些宏包，例如 babel，符号类别会被强行修改。为了解决这个问题，TikZ 提供了一个小型的库 babel，这个库可以与那些 (能全局地改变符号类别的) 宏包一起使用。babel 库的工作方式是在 {tikzpicture} 环境的开头重设符号类别，并在 node 的开头处重载之前的符号类别设置。

39.2.4 Adding a Background

在默认下图形没有背景 (background)，如果要添加背景，可参考 backgrounds 库。

39.3 使用 scope

在 {tikzpicture} 环境里可以用 {scope} 环境创建 scope。环境 {scope} 只能用在 {tikzpicture} 环境里。当提到 scope 这个词时，一般指的是“环境”，或者 TEX 组。

39.3.1 scope 环境

\begin{scope}⟨animations spec⟩[⟨options⟩]
  ⟨environment content⟩
\end{scope}

此环境的选项 ⟨*options*⟩ 只针对这个环境里的 ⟨*environment contents*⟩。本环境内的剪切路径 (clipping path) 的剪切范围也限于这个环境内。在 [⟨*options*⟩] 中可以使用以下选项。

/tikz/name=⟨*scope name*⟩ (no default)

给 {scope} 环境命名，以便于在动画 (animations) 中引用。这个名称是 high-level 的，驱动并不直接处理这个名称，因此可以在名称中使用空格、数字、字母等符号，但不能使用逗号、点号、冒号等标点符号。

此选项的定义是：

```
\tikzset{
  name/.code={\edef\tikz@fig@name{\tikz@pp@name{#1}}\let\tikz@id@name
  ↪  \tikz@fig@name},%
  name prefix/.initial=,%
  name suffix/.initial=%
}%
\def\tikz@pp@name#1{\csname pgfk@/tikz/name prefix\endcsname#1\csname
↪  pgfk@/tikz/name suffix\endcsname}%
```

此选项也用于给 node 命名。

/tikz/every scope (style, initially empty)

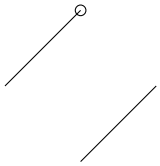
如果这个样式作为 {tikzpicture} 环境的选项，则它会添加到这个 {tikzpicture} 环境内的每个 {scope} 环境的开头；如果在 {tikzpicture} 环境之外使用

```
\tikzset{every scope/.style={...}}
```

那么这个样式对之后的各个 {tikzpicture} 环境和 {scope} 环境都有效。

/tikz/execute at begin scope=⟨*code*⟩ (no default)

⟨*code*⟩ 会在 {scope} 环境开始时被执行，这个选项只能针对 {scope} 环境来使用，并且它只对当前层次的 {scope} 环境有效，对套嵌在当前 {scope} 环境内的子环境无效。如果多次使用本选项则其作用累计。



```
\begin{tikzpicture}
  \begin{scope}[execute at begin scope={%
      \def\aaaa{2pt}
      \draw (1,1) circle (\aaaa);
      \edef\aaaa{\aaaa + 2pt}
    }]
    \draw (0,0)--(1,1);
    \begin{scope}[shift={(0,-1)}]
      \draw (1,0)--(2,1);
    \end{scope}
  \end{scope}
\end{tikzpicture}
```

/tikz/execute at end scope=⟨*code*⟩ (no default)

⟨*code*⟩ 会在 {scope} 环境结尾时被执行，这个选项只能针对 {scope} 环境来使用，并且它只对当前层次的 {scope} 环境有效，对套嵌在当前 {scope} 环境内的子环境无效。如果多次使用本选项则其作用累计。

\scope⟨*animations spec*⟩[⟨*options*⟩]
⟨*environment contents*⟩
\endscope

Plain TeX 中的 scope 环境版本。

\startscope⟨animations spec⟩[⟨options⟩]
⟨environment contents⟩
\stopscope

ConTeXt 中的 scope 环境版本。

39.3.1.1 命令 \begin{scope}

命令 \begin{scope} 导致执行 \scope.

\scope⟨animations spec⟩[⟨options⟩]

在执行 \tikz@installcommands 后，\scope 就等于 \tikz@scope@env, 此命令的定义是：

```
\def\tikz@scope@env{%
  \pgfscope%
  \begingroup%
  \let\tikz@atbegin@scope=\pgfutil@empty%
  \let\tikz@atend@scope=\pgfutil@empty%
  \let\tikz@options=\pgfutil@empty%
  \tikz@clear@rdf@options%
  \let\tikz@mode=\pgfutil@empty%
  \let\tikz@id@name=\pgfutil@empty%
  \tikz@transparency@groupfalse%
  \tikzset{every scope/.try}%
  \tikz@collect@scope@anims\tikz@scope@opt%
}%
\def\tikz@scope@opt[#1]{%
  \tikzset{#1}%
  \tikz@options%
  \tikz@do@rdf@pre@options%
  \iftikz@transparency@group\expandafter\pgftransparencygroup\expandafter
  ↪  [\tikz@transparency@group@options]\tikz@blend@group\fi%
  \tikz@is@nodefalse%
  \tikz@call@id@hook%
  \pgfidscope%
    \tikz@do@rdf@post@options%
    \begingroup%
      \let\tikz@id@name\pgfutil@empty%
      \expandafter\tikz@atbegin@scope%
      \expandafter\pgfclearid%
      \tikz@lib@scope@check%
}%
```

注释：

- \tikz@atbegin@scope 对应选项 /tikz/execute at begin scope→P. 672, 此选项设置的代码会被保存到这个宏中，多次使用这个选项设置代码时，这些代码会被依次保存到这个宏中。
- \tikz@atend@scope 对应选项 /tikz/execute at end scope→P. 672, 此选项设置的代码会被保存到这个宏中，多次使用这个选项设置代码时，这些代码会被依次保存到这个宏中。
- \tikz@options 是由命令 \tikz@addoption→P. 676 定义的宏。
- 关于 \tikz@transparency@groupfalse, 参考选项 /tikz/transparency group→P. 913, /tikz/blend group→P. 907
- 注意 \tikzset{every scope/.try}, 这实际上是执行样式 every scope 中保存的选项，也就是说，如果此前规定了样式

  every scope/.style={⟨key-value list⟩}

那么 ⟨*key-value list*⟩ 此时就会被执行，对此后 (环境内) 的所有路径都有效。

- 命令 \tikz@collect@scope@anims →P. 667 收集动画选项。
- 注意选项的执行次序是：

```
\tikzset{every scope/.try}%
\tikzset{⟨options⟩}%
```

39.3.1.2 命令 \end{scope}

命令 \end{scope} 导致执行 \endscope,

\endscope

在执行 \tikz@installcommands 后，\endscope 就等于 \endtikz@scope@env, 其定义是：

```
\def\endtikz@scope@env{%
      \tikz@atend@scope%
    \endgroup%
  \endpgfidscope%
  \iftikz@transparency@group\endpgftransparencygroup\fi%
  \endgroup%
  \endpgfscope%
  \tikz@lib@scope@check%
}%
```

39.3.2 scope 环境的简写形式—scopes 库

TikZ Library scopes

```
\usetikzlibrary{scopes} % LaTeX and plain TeX
\usetikzlibrary[scopes] % ConTeXt
```

这个库定义一种 scope →P. 671 环境的简写形式。

载入这个库后，你可以在 tikz picture 的某些地方 (不是任意的地方) 使用较为简捷的形式创建一个 {scope} 环境，其形式为 “开花括号–开方括号–选项–闭方括号–图形命令–闭花括号”：

```
{[⟨options⟩]
  ⟨environment contents⟩
}
```

这种简写形式可以套嵌使用。

```
\begin{tikzpicture}
  { [ultra thick]
    { [red] \draw (0mm,9mm) -- (10mm,9mm); }
    \draw (0mm,6mm) -- (10mm,6mm);
  }
  {[green]
    \draw (0mm,3mm) -- (10mm,3mm);
    \draw [blue] (0mm,0mm) -- (10mm,0mm);
  }
\end{tikzpicture}
```

这种简写形式通常用在 5 种地方：

- \begin{tikzpicture} 或 \begin{tikzpicture}[⟨*options*⟩] 之后
- \begin{scope} 或 \begin{scope}[⟨*options*⟩] 之后
- \end{scope} 之后

- 路径结束符号——分号——之后
- 简写的 {scope} 之后

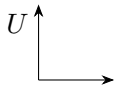
以上几个位置是命令 \tikz@lib@scope@check 的检查点，这个命令检查这种简写的 scope 句法。注意在 \tikz 之后或 \scoped 之后没有这个命令。这个命令是个循环操作，其作用是：

```
{...} 等价于 \scope{...}\endscope\tikz@lib@scope@check
{[⟨options⟩]...} 等价于 \scope[⟨options⟩]{...}\endscope\tikz@lib@scope@check
```

观察下面的例子：

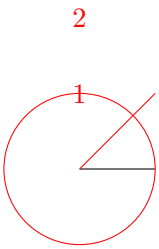
```
\tikz{{[draw=red,dashed] \draw (0,0)--(1,1); }}
```

上面这个简写的 {scope} 环境无效 (在 \tikz 之后)。




```
\begin{tikzpicture}[baseline=0.5cm]
  {[>=Stealth]
   [cyan,x={(60:1cm)},y={(150:1cm)}]
    \draw [->](0,0)--(1,0);
    \draw [->](0,0)--(0,1)node[below left]{$U$};
  }
\end{tikzpicture}
```

上面的选项 [cyan,x={(60:1cm)},y={(150:1cm)}] 无效。




```
\begin{tikzpicture}
  \draw (0,0)--(1,0);
  {[red]
    \draw (0,0) circle (1cm);
    \foreach \i in {1,2} \node at (0,\i) {\i};
    {[green,yshift=0.5cm] \draw (0,0)--(1,1); }
  }
\end{tikzpicture}
```

上面例子中在 \foreach 语句之后的简写 {scope} 环境无效。

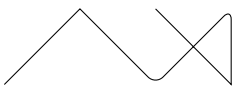
39.3.3 scoped 命令

\scoped⟨*animations spec*⟩[⟨*options*⟩]⟨*path command*⟩

本命令的用法类似 \tikz，它必须用在 {tikzpicture} 环境内。本命令创建一个 {scope} 环境，[⟨*options*⟩] 是该环境的选项，⟨*path command*⟩ 是该环境内的路径命令。如果在 ⟨*path command*⟩ 中有多条路径命令，那么这些命令必须用花括号括起来。

39.3.4 在路径之内插入 scopes

一个路径被包裹在一个组中，在一个路径内部也可以创建 scope:



```
\tikz \draw (0,0) -- (1,1)
  {[rounded corners, red] -- (2,0) -- (3,1)}
  -- (3,0) -- (2,1);
```

上面例子中花括号创建的 scope 不是前文说的“简写 scope”，因为它不被命令 \tikz@lib@scope@check 检查。例子中的选项 rounded corners 的作用范围受到 scope 的限制，并且颜色选项 red 没有起到作用，这是因为 \draw 的默认颜色是 draw=black，颜色 black 把 red 覆盖了。还要注意开启 scope 的符号组合“{[...]”要放在坐标点之后、“--”之前。

对于上面例子的情况，当 Ti*k*Z 在路径内部遇到 { 时就会执行命令 \tikz@beginscope, 遇到 } 时就会执行命令 \tikz@endscope.

39.4 使用图形选项

39.4.1 如何处理图形选项

很多 TikZ 的命令、环境都接受选项 (options)。选项被称为 key, 选项 (及其值) 的列表被称为 key-value lists. 处理选项的命令是 \tikzset，如果你直接使用这个命令，那么最好确认它的有效范围。

\tikzset{⟨*options*⟩}

这个命令会使用 \pgfkeys 来处理 ⟨*options*⟩。⟨*options*⟩ 中的选项是用逗号分隔的“键值对”(⟨*key*⟩=⟨*value*⟩)，使用 pgfkeys 的机制能获得丰富的效果。

在处理一个键值对 (⟨*key*⟩=⟨*value*⟩) 时会有以下动作：

1. 如果 ⟨*key*⟩ 是个完整的 key (以斜线 / 开头)，则直接处理它。
2. 否则，检查 /tikz/⟨*key*⟩ 是不是一个 key，如果是则执行它。
3. 否则，检查 /pgf/⟨*key*⟩ 是不是一个 key，如果是则执行它。
4. 否则，检查 ⟨*key*⟩ 是不是一个颜色名称，如果是则执行 color=⟨*key*⟩。
5. 否则，检查 ⟨*key*⟩ 是不是包含一个连字符 (dash)，如果是则执行 arrows=⟨*key*⟩。
6. 否则，检查 ⟨*key*⟩ 是不是一个 shape 名称，如果是则执行 shape=⟨*key*⟩。
7. 否则，打印一个错误信息。

可见，以 /tikz 或 /pgf 开头的选项都可以用在 \tikzset 中。

\tikzoption{⟨*name*⟩}[⟨*default value*⟩]{⟨*code*⟩}

本命令用于定义 TikZ 选项，即前缀路径为 /tikz 的选项。

⟨*name*⟩ 是被定义选项的名称 (通常是一串字母，可以含有空格)。

⟨*default value*⟩ 是被定义选项的默认值，可以没有 [⟨*default value*⟩] 这一部分。

⟨*code*⟩ 是被定义选项所保存的代码。

当执行选项 ⟨*name*⟩=⟨*value*⟩ 时，把 ⟨*value*⟩ 作为参数传递给 ⟨*code*⟩ 并执行。

这个命令是较旧的命令，今后推荐使用 \tikzset. 见文件 tikz.code.tex. 本命令实际导致以下处理：

```
% 对于给出 [⟨default value⟩] 的情况是:
\pgfkeysdef{/tikz/⟨name⟩}{⟨code⟩}%
\pgfkeyssetvalue{/tikz/⟨name⟩/.@def}{⟨default value⟩}
% 对于不给出 [⟨default value⟩] 的情况是:
\pgfkeysdef{/tikz/⟨name⟩}{⟨code⟩}%
\pgfkeyssetvalue{/tikz/⟨name⟩/.@def}{\pgfkeysvaluerequired}
```

参考 \pgfkeysdef→P. 44, \pgfkeyssetvalue→P. 43.

注意：

- \pgfkeysdef→P. 44 只是定义了控制序列:
  - \csname pgfk@⟨*full key name*⟩/.@cmd\endcsname
  - \csname pgfk@⟨*full key name*⟩/.@body\endcsname

  所以，在执行“\tikzoption{⟨*name*⟩}...”后，并不能用 \pgfkeysvalueof→P. 44{⟨*full key name*⟩} 得到 ⟨*name*⟩ 的值，因为 \pgfkeysvalueof 返回的是 \csname pgfk@⟨*full key name*⟩\endcsname.
- \pgfkeysdef→P. 44 在定义键时，不会将 ⟨*code*⟩ 中的宏展开，在套嵌使用多个 group 的情况下，一不小心就可能会导致“宏没有定义”的错误。

\tikz@addoption{⟨*code*⟩}

此命令重定义宏 \tikz@options, 向宏 \tikz@options 中添加 {⟨*code*⟩}.

```
\def\tikz@addoption#1{%
  \expandafter\def\expandafter\tikz@options\expandafter{\tikz@options#1}}%
```

有的选项会调用命令 \tikz@addoption, 例如选项 draw opacity 的定义是：

```
\tikzoption{draw opacity}{\tikz@addoption{\pgfsetstrokeopacity{#1}}}%
```

当执行选项 draw opacity=0.5 时，PGF 的命令 \pgfsetstrokeopacity{0.5} 就被添加到宏 \tikz@options 所保存的内容中。

\tikz@options

如上，这个宏由命令 \tikz@addoption 定义，注意其中保存的是“命令”，而不是字符串形式的“选项”。

39.4.2 使用 style

参考本手册的 pgfkeys 宏包。

一个 style (样式) 是一组选项组合成的选项，使用 style 能让这些选项只针对某个或某一类图形要素起作用。有许多预定义的样式，例如样式 help lines 规定了颜色、线宽：

```
\tikz[help lines]{
  \draw (0,0)--(2,0);
  \fill (0,0.5) rectangle (2,1);
}
```

在文件《tikz.code.tex》中定义 help lines 样式的代码如下：

```
\tikzset{help lines/.style={color=gray,line width=0.2pt}}
```

可见定义一个 style 的办法就是使用命令 \tikzset (或者 \pgfkeys)，使用手柄 (handler) /.style:

⟨key⟩/.style={⟨values⟩}

因为命令、环境的选项会被 \tikzset 处理，所以也可以在命令、环境的选项列表中定义样式。

```
\tikz[my style/.style={rounded corners, draw=red, fill=green}]{
  \filldraw [my style] (0,0) rectangle (1,1);
}
```

上面例子中在环境选项里定义了样式 my style，所以这个样式只在这个环境范围内可用。

```
\tikz{
  \filldraw [my style/.style={rounded corners, draw=red, fill=green!20}]
  [my style] (0,0) rectangle (1,1);
}
```

上面例子中在命令选项里定义了样式 my style，所以这个样式只在这个命令范围内可用。

用手柄 /.style 可以重定义一个样式，也就是说，如果两次用这个手柄定义同一个样式名称，那么后一次覆盖前一次的定义。如果不想抛弃原来的样式，但还想在原来样式的基础上追加某些选项或操作，则可以用手柄 /.append style 来定义“附加样式”。

```
\tikz[my style/.style={rounded corners, draw=red, fill=green}]{
  \filldraw [my style/.append style={line width=4pt, fill=none}, my style]
    (0,0) rectangle (1,1);
}
```

/.append style 中的选项会附加到 /.style 选项之后起作用，因此可以改写 /.style 中的选项。上面例子中命令 \filldraw 的选项就是

```
rounded corners, draw=red, fill=green, line width=4pt, fill=none
```

其中的填充色被取消了。

手柄 /.prefix style 定义的选项会附加到 /.style 选项之前起作用。

在 /.style={⟨value⟩} 的 ⟨value⟩ 中可以使用一个参数 #1，如果要使用更多参数可以参考本手册的 pgfkeys 宏包。

当定义了一个 style 后，可以用手柄 /.default 为它设置默认值。

/tikz/style={⟨key-value list⟩}

执行 \tikzset{style={⟨key-value list⟩}} 等效于执行 \tikzset{⟨key-value list⟩}.


```
\tikz\node[style={draw=red,fill=green,line width=1mm}]{A};
```

/tikz/set style=⟨something follows \tikzstyle⟩

```
\tikzoption{set style}{\tikzstyle#1}%
```

\tikzstyle{⟨style name⟩}[⟨default style set⟩]⟨something may containing '+'⟩=⟨something will be eaten⟩[⟨style set⟩]

本命令定义样式 ⟨style name⟩.

- 方括号参数 [⟨style set⟩] 是由方括号括起来的一些选项，是必须给出的。注意，如果 ⟨style set⟩ 中含有方括号，则必须用花括号将 ⟨style set⟩ 括起来。
- 参数 [⟨default style set⟩] 是可选的，如果给出, 就作为 ⟨style name⟩ 的默认值：

  ```
  \pgfkeys{/tikz/⟨style name⟩/.default={⟨default style set⟩}}%
  ```

  注意，如果 ⟨default style set⟩ 中含有等号 =, 则必须用花括号将 ⟨default style set⟩ 括起来。
- 可选参数 ⟨something may containing '+'⟩ 是一些代码，其中可能包含字符'+'.
- 如果给出 ⟨something may containing '+'⟩ 且其中包含字符'+', 就把 ⟨style set⟩ 附加到 ⟨style name⟩ 中：

  ```
  \pgfkeys{/tikz/⟨style name⟩/.append style={⟨style set⟩}}%
  ```

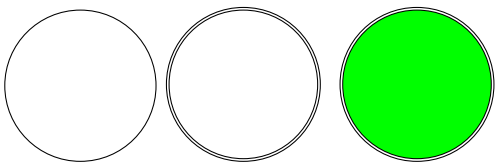
- 如果不给出 ⟨something containing '+'⟩, 或者给出的 ⟨something containing '+'⟩ 中不包含字符'+', 就把 ⟨style set⟩ 保存到 ⟨style name⟩ 中：

  ```
  \pgfkeys{/tikz/⟨style name⟩/.style={⟨style set⟩}}%
  ```

- 可选参数 ⟨something will be eaten⟩ 处于等号 = 与 [⟨style set⟩] 之间，会被吃掉。注意，如果 ⟨something will be eaten⟩ 中含有方括号，则必须用花括号将 ⟨something will be eaten⟩ 括起来。
- 实际上参数 [⟨default style set⟩] 的用处不大，因为 [⟨style set⟩] 是必须给出的，因此总会执行

  ```
  \pgfkeys{/tikz/⟨style name⟩/.append style={⟨style set⟩}}%
  或者
  \pgfkeys{/tikz/⟨style name⟩/.style={⟨style set⟩}}%
  ```

  所以样式 ⟨style name⟩ 总是会被定义的，即使 ⟨style set⟩ 为空。
- 如果不给出 [⟨style set⟩], 那么被吃掉的 ⟨something will be eaten⟩ 将可能包含过多内容，直到遇到方括号（或者其他导致错误的代码）为止。



```
\tikzstyle{temp test}[{draw=red}]=[]
\tikz\draw [temp test] (0,0) circle [radius=1cm];% 默认值无用
\tikzstyle{temp test}=[double]
\tikz\draw [temp test] (0,0) circle [radius=1cm];
\tikzstyle{temp test}A+B=\usepackage{xxx}[fill=green]
\tikz\filldraw [temp test] (0,0) circle [radius=1cm];
```

39.5 环境总结

环境

```
\tikzpicture[⟨options⟩]
    ⟨environment contents⟩
\endtikzpicture
```

的实际结构大致是：

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 以下 tikzpicture 环境开始 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\begingroup%.....................................................................1
  \tikz@startup@env%
  \tikz@collect@scope@anims% 收集动画选项
  %%%%%%%%%%%%%%%%%%%%%%%%% 以下 pgfpicture 环境开始 %%%%%%%%%%%%%%%%%%%%%%%%%
  % 开始 \pgfpicture
  \begingroup%...................................................................2
  \pgfpicturetrue
  \global\advance\pgf@picture@serial@count by1\relax%
  \edef\pgfpictureid{pgfid\the\pgf@picture@serial@count}%
  \let\pgf@nodecallback=\pgfutil@gobble%
  ⟨ 设置寄存器 \pgf@picmaxx, \pgf@pathmaxx 等的初始值⟩
  ⟨ 设置 \ifpgf@relevantforpicturesize 的真值⟩
  % 执行 \pgf@picture
  \setbox\pgfpic\hbox to0pt\bgroup% 定义名称为 \pgfpic 的盒子
    \begingroup%.................................................................3
      \pgfsys@beginpicture%
      \pgfsys@beginscope%
        \begingroup%.............................................................4
          % 初始化线宽、颜色、单位矩阵，清空软路径
          \pgfsetcolor{.}%
          \pgfsetlinewidth{0.4pt}%
          \pgftransformreset%
          \pgfsyssoftpath@setcurrentpath\pgfutil@empty%
          \begingroup%...........................................................5
            \let\pgf@setlengthorig=\setlength%
            \let\pgf@addtolengthorig=\addtolength%
            \let\pgf@selectfontorig=\selectfont%
            \let\setlength=\pgf@setlength%
            \let\addtolength=\pgf@addtolength%
            \let\selectfont=\pgf@selectfont%
            \nullfont\spaceskip0pt\xspaceskip0pt%
            \setbox\pgf@layerbox@main\hbox to0pt\bgroup
            ↪  % 定义名称为 \pgf@layerbox@main 的盒子
              \begingroup%...................................................6
            % 以上 \pgfpicture
                \let\tikz@atbegin@picture=\pgfutil@empty%
                \let\tikz@atend@picture=\pgfutil@empty%
                \let\tikz@transform=\relax%
                \def\tikz@time{.5}%
                \tikz@installcommands%
                %%%%%%%%%%%%%%%% 以下环境 scope 开始 %%%%%%%%%%%%%%%%
                % 开始 \scope[every picture,#1]
                \pgfscope%
                  \begingroup%...........................................7
                    \let\tikz@atbegin@scope=\pgfutil@empty%
                    \let\tikz@atend@scope=\pgfutil@empty%
```

```
      \let\tikz@options=\pgfutil@empty%
      \tikz@clear@rdf@options%
      \let\tikz@mode=\pgfutil@empty%
      \let\tikz@id@name=\pgfutil@empty%
      \tikz@transparency@groupfalse%
      \tikzset{every scope/.try}%
      \tikz@collect@scope@anims% 收集动画选项
      % 执行 \tikz@scope@opt
      \tikzset{every picture,⟨options⟩}
      \tikz@options
      \tikz@do@rdf@pre@options%
      \iftikz@transparency@group\expandafter\pgftransparencygroup
      ↪  \expandafter[\tikz@transparency@group@options]
      ↪  \tikz@blend@group\fi%
      \tikz@is@nodefalse%
      \tikz@call@id@hook%
      \pgfidscope%
      \tikz@do@rdf@post@options%
      \begingroup%.....................................................8
        \let\tikz@id@name\pgfutil@empty%
        \expandafter\tikz@atbegin@scope%
        \expandafter\pgfclearid\tikz@lib@scope@check%
        % 以上 \scope
        \iftikz@handle@active@code%
          \tikz@switchoff@shorthands%
        \fi%
        \expandafter\tikz@atbegin@picture\tikz@lib@scope@check%
          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
          % tikzpicture 环境的内容在这里 !!!
          ⟨environment contents⟩
          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        \tikz@atend@picture%
        \global\let\pgf@shift@baseline@smuggle=\pgf@baseline%
        \global\let\pgf@trimleft@final@smuggle=\pgf@trimleft%
        \global\let\pgf@trimright@final@smuggle=\pgf@trimright%
        \global\let\pgf@remember@smuggle=
        ↪  \ifpgfrememberpicturepositiononpage%
        \pgf@remember@layerlist@globally
        % 执行 \endscope
        \tikz@atend@scope
      \endgroup%.....................................................8
    \endpgfidscope%
    \iftikz@transparency@group\endpgftransparencygroup\fi%
    \endgroup%.....................................................7
  \endpgfscope%
  \tikz@lib@scope@check
  % 以上 \endscope
  %%%%%%%%%%%%%%%% 以上环境 scope 结束 %%%%%%%%%%%%%%%
  \let\pgf@baseline=\pgf@shift@baseline@smuggle%
  \let\pgf@trimleft=\pgf@trimleft@final@smuggle%
  \let\pgf@trimright=\pgf@trimright@final@smuggle%
  \let\ifpgfrememberpicturepositiononpage=\pgf@remember@smuggle%
  \pgf@restore@layerlist@from@global
  % 以下执行 \endpgfpicture
  \ifpgfrememberpicturepositiononpage%
  \hbox toOpt{\pgfsys@markposition{\pgfpictureid}}%
```

```
                \fi%
                % ok, now let's position the box
                \ifdim\pgf@picmaxx=-16000pt\relax%
                % empty picture. make size 0.
                \global\pgf@picmaxx=0pt\relax%
                \global\pgf@picminx=0pt\relax%
                \global\pgf@picmaxy=0pt\relax%
                \global\pgf@picminy=0pt\relax%
                \fi%
                % Shift baseline outside:
                \pgf@relevantforpicturesizefalse%
                \pgf@process{\pgf@baseline}%
                \xdef\pgf@shift@baseline{\the\pgf@y}%
                %
                \pgf@process{\pgf@trimleft}%
                \global\advance\pgf@x by-\pgf@picminx
                \xdef\pgf@trimleft@final{-\the\pgf@x}%
                %
                \pgf@process{\pgf@trimright}%
                \global\advance\pgf@x by-\pgf@picmaxx
                \xdef\pgf@trimright@final{\the\pgf@x}%
                %
                \pgf@remember@layerlist@globally
              \endgroup%.....................................................6
              \hss%
            \egroup% 结束盒子 \pgf@layerbox@main
            \pgf@restore@layerlist@from@global
            \pgf@insertlayers%
          \endgroup%.....................................................5
          \pgfsys@discardpath%
        \endgroup%.....................................................4
      \pgfsys@endscope%
      \pgfsys@endpicture%
    \endgroup%.....................................................3
    \hss%
    \egroup% 结束盒子 \pgfpic
    % 确定盒子 \pgfpic 的基线、边界，将这个盒子插入到文档的当前位置
    \pgfsys@typesetpicturebox\pgfpic%
  \endgroup%.....................................................2
  % 以上 \endpgfpicture
  %%%%%%%%%%%%%%%%%%%%%%%%%% 以上 pgfpicture 环境结束 %%%%%%%%%%%%%%%%%%%%%%%%%%
\endgroup%.....................................................1
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 以上 tikzpicture 环境结束 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

注意这个结构中的组比较多，在注释中标注了 8 个组。

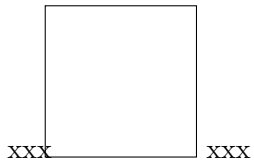
39.5.1 根据环境结构分析一个错误

用选项 /tikz/trim left→P. 779 调整图形的边界时，下面代码会导致错误：

```
\begin{tikzpicture}
    \pgfmathsetmacro{\abcd}{1+2+3}%
    \tikzset{trim left=\abcd pt}
    \draw (0,0) rectangle (2,2);
\end{tikzpicture}
```

错误信息是 `! Undefined control sequence. <argument> \abcd`

下面的代码正常：




```
xxx
\pgfmathsetmacro{\abcd}{1+2+3}%
\begin{tikzpicture}
    \tikzset{trim left=\abcd pt}
    \draw (0,0) rectangle (2,2);
\end{tikzpicture}
xxx
```

选项 /tikz/trim left 的定义是 (见《tikz.code.tex》)：

```
\tikzoption{trim left}[0pt]{\pgfutil@ifnextchar({\tikz@trim@coordinate{left}}{
↪  \tikz@trim@simple{left}}#1\@nil}%)%
\def\tikz@trim@simple#1#2\@nil{\csname pgfsettrim#1\endcsname{#2}}%
```

所以 \tikzset{trim left=\abcd pt} 导致

```
\pgfsettrimleft{\abcd pt}
```

在《pgfcorescopes.code.tex》中有：

```
\def\pgfsettrimleftpointlater#1{\def\pgf@trimleft{#1}}
\def\pgfsettrimleft#1{%
  \def\pgf@temp{#1}%
  \ifx\pgf@temp\pgf@default@text
    \pgfsettrimleft{\pgf@picminx}
  \else
    \pgfsettrimleftpointlater{\pgfpoint{#1}{0pt}}%
  \fi
}
```

所以 \pgfsettrimleft{\abcd pt} 导致

```
\def\pgf@trimleft{\pgfpoint{\abcd pt}{0pt}}
```

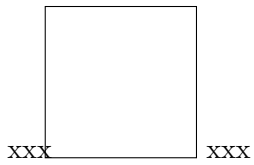
在 \endpgfpicture 的展开过程中，处理 \pgf@trimleft 的命令是：

```
\pgf@process{\pgf@trimleft}%
\global\advance\pgf@x by-\pgf@picminx
\xdef\pgf@trimleft@final{-\the\pgf@x}%
```

其中的 \pgf@process{\pgf@trimleft} 展开为

```
{\pgfpoint{\abcd pt}{0pt}\global\pgf@x=\pgf@x\global\pgf@y=\pgf@y}
```

其中的 \pgfpoint{\abcd pt}{0pt} 展开时会出错，因为定义 \abcd 的命令 \pgfmathsetmacro 被限制在 (前面环境结构概略中的)8 号组内，此命令对 \abcd 的定义也不是全局地，而实际计算左侧边界的命令还在 7 号组之外，此处的 \abcd 是无定义的。如果在使用选项 /tikz/trim left 时就让 \abcd 展开，可以避免这种问题，下面使用手柄 /.expanded:




```
xxx
\begin{tikzpicture}
    \pgfmathsetmacro{\abcd}{1+2+3}%
    \tikzset{trim left/.expanded=\abcd pt}
    \draw (0,0) rectangle (2,2);
\end{tikzpicture}
xxx
```

第四十章　设置坐标

40.1　Overview

一个坐标对应图形画布 (canvas) 上的一个点。Ti*k*Z 使用自己的句法来指定坐标。通常坐标数据都要放在圆括号里，当 Ti*k*Z 遇到开圆括号“(”时就倾向于认为遇到了一个坐标。指定坐标的一般句法是：

(`[`⟨*options*⟩`]`⟨*coordinate specification*⟩)

此句法中的可选项 ⟨*options*⟩ 是针对坐标的选项，一般是变换选项。⟨*coordinate specification*⟩ 利用某种坐标系统 (coordinate system) 来指定一个坐标点。

对于几何图形来说，坐标系是比较直观的，但这里说的“坐标系统”是 Ti*k*Z 层面的计算体系，是从“用户给的坐标数据”到“基本层坐标”之间的对应。Ti*k*Z 会用命令 `\tikz@scan@one@point` 解析用户给出的坐标，得到基本层命令表达的坐标 (或者是为尺寸寄存器 `\pgf@x`, `\pgf@y` 赋值)，然后把这个基本层坐标看作是底层画布坐标系统中的坐标，从而确定页面上的一个位置点。有的坐标系统专门计算 node 边界上的点，有的专门计算“切点”。你也可以自己定义新的坐标系统。

有两种选定坐标系统的方式：

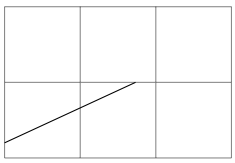
Explicitly 你可以“显式地”指定坐标系统，所谓“显式地”指的是你不仅要给出坐标数据，还要给出坐标系统的名称，其一般格式是

(⟨*coordinate system*⟩ `cs:`⟨*list of key-value pairs specific to the coordinate system*⟩)

这个格式的特点是带有符号组合“`cs:`”，命令 `\tikz@scan@one@point` 会检查被解析的坐标数据中是否含有这个符号组合，如果有，就认为是“显示的”。

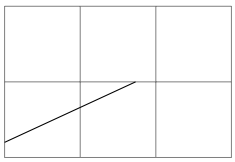
Implicitly 显式地指定坐标的句法格式有点繁琐，“隐式地”格式会简洁一些。“隐式地”指的是你不需要给出坐标系统的名称，Ti*k*Z 会根据你的输入格式自动确定所对应的坐标系统。例如 `(0,0)` 对应笛卡尔坐标，`(30:2)` 对应极坐标 (其中 30 代表角度)。

对比下面两个例子：



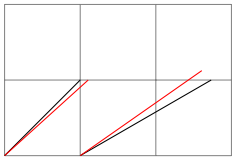
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (canvas cs:x=0cm,y=2mm)
    -- (canvas polar cs:radius=2cm,angle=30);
\end{tikzpicture}
```

上面这个例子“显式地”指定坐标，下面的例子“隐式地”指定坐标：

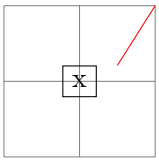


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0cm,2mm) -- (30:2cm);
\end{tikzpicture}
```

可以给单个坐标使用选项 `[`⟨*options*⟩`]`，但是选项 ⟨*options*⟩ 仅限于变换选项。例如：



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw              (0,0) -- (1,1);
  \draw [red]        (0,0) -- ([xshift=3pt] 1,1);
  \draw              (1,0) -- +(30:2cm);
  \draw [red]        (1,0) -- +([shift=(135:5pt)] 30:2cm);
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \draw [help lines] (-1,-1) grid (1,1);
  \node [draw]       (x) {x};
  \draw [red]        (1,1) -- ([xshift=5mm] x.north);
\end{tikzpicture}
```

TikZ 最常用的坐标系统是 canvas coordinate system, xyz coordinate system, canvas polar coordinate system, xyz polar coordinate system 这 4 种，它们的区别是：

- 带有 canvas 一词的坐标系统：
  - canvas coordinate system 利用命令 \pgfpoint →P. 250 来指定坐标点
  - canvas polar coordinate system 利用命令 \pgfpointpolar →P. 251 来指定坐标点
- 带有 xyz 一词的坐标系统：
  - xyz coordinate system 利用命令 \pgfpointxyz →P. 254 来指定坐标点
  - xyz polar coordinate system 利用命令 \pgfpointpolarxy →P. 254 来指定坐标点

命令 \pgfpointxyz, \pgfpointpolarxy 的计算过程会考虑 \pgfsetxvec →P. 253, \pgfsetyvec →P. 253, \pgfsetzvec →P. 2 的设置 (这 3 个命令指定 xyz coordinate system 的 3 个基向量)，而命令 \pgfpoint, \pgfpointpolar 则不会考虑。

坐标系统 canvas coordinate system, canvas polar coordinate system, xyz polar coordinate system 都处理二维坐标，而 xyz coordinate system 可以处理三维坐标，即用二维来模拟三维，“模拟”的意思是，选定平面上的 3 个具有相同起点 O 的定点向量 v_x, v_y, v_z 充当 3 维仿射标架，通过这个标架将三维坐标与平面上的点对应起来：坐标 (x, y, z) 对应的点是 $xv_x + yv_y + zv_z + O$. 当需要指定一个三维点时，最好使用 xyz coordinate system.

40.2 坐标系统

坐标系统使用的选项：

```
\tikzset{cs/x/.store in=\tikz@cs@x}%
\tikzset{cs/y/.store in=\tikz@cs@y}%
\tikzset{cs/z/.store in=\tikz@cs@z}%
\tikzset{cs/angle/.store in=\tikz@cs@angle}%
\tikzset{cs/x radius/.store in=\tikz@cs@xradius}%
\tikzset{cs/y radius/.store in=\tikz@cs@yradius}%
\tikzset{cs/radius/.style={/tikz/cs/x radius=#1,/tikz/cs/y radius=#1}}%
\tikzset{cs/name/.store in=\tikz@cs@node}%
\tikzset{cs/anchor/.store in=\tikz@cs@anchor}%
```

40.2.1 Canvas, XYZ, and Polar Coordinate Systems

40.2.1.1 Coordinate system canvas

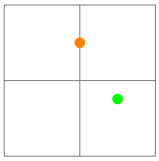
画布坐标系统，默认其 x 轴正向向右，y 轴的正向向上，使用选项 x=d_x 和 y=d_y 设置坐标数据，数据是带长度单位的尺寸，也可以是 (展开为长度的) 宏。使用变换选项可以修改画布坐标系统坐标轴的单位向量。

/tikz/cs/x=⟨*dimension*⟩ (no default, initially 0pt)

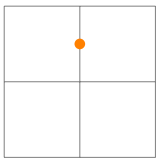
用于确定 canvas 坐标系统下坐标的 x 分量，⟨*dimension*⟩ 一般是带单位的尺寸，也可以不带长度单位，也可以是宏，也可以是比较复杂的算式，⟨*dimension*⟩ 会被数学引擎的命令 \pgfmathparse 处理。如果 ⟨*dimension*⟩ 不带长度单位，例如 /tikz/cs/x=2+3, 那么它相当于 /tikz/cs/x=2pt+3pt, 即会被看作是单位为 pt 的式子。

/tikz/cs/y=⟨*dimension*⟩ (no default, initially 0pt)

与上一选项类似，本选项用于确定 canvas 坐标系统下坐标的 y 分量。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \fill [orange] (canvas cs:x=1cm,y=1.5cm) circle (2pt);
  \fill [green] (canvas cs:x=1.5cm,y=0.03*\textheight) circle (2pt);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \fill [orange] (canvas cs:x=28.45274,y=1.5cm) circle (2pt);
\end{tikzpicture}
```

指定画布坐标的“隐式”形式是，如 (2cm,30pt) 这种带单位的形式，或者 (2cm,\textheight)。

canvas coordinate system 的定义是：

```
\tikzdeclarecoordinatesystem{canvas}
{%
  \tikzset{cs/.cd,x=0pt,y=0pt,#1}%
  \pgfpoint{\tikz@cs@x}{\tikz@cs@y}%
}%
```

按这个定义，canvas coordinate system 只适合处理二维点，因为命令 \pgfpoint 只处理 2 个参数。

40.2.1.2 Coordinate system xyz

在 xyz 坐标系统下你可以指定二维坐标、三维坐标，坐标数据用选项 x=⟨*factor*⟩，y=⟨*factor*⟩，z=⟨*factor*⟩ 给出。默认 x 轴，y 轴，z 轴的单位向量分别是 $(1\text{cm}, 0)$, $(0, 1\text{cm})$, $(-3.85\text{mm}, -3.85\text{mm})$, 坐标轴的单位向量可以用选项 /tikz/x → P. 859, /tikz/y → P. 861, /tikz/z → P. 861 来设置，例如 x=-2cm 就把 $(-2\text{cm}, 0)$ 作为 x 轴的单位向量，参考“坐标变换”的内容。

/tikz/cs/x=⟨*factor*⟩ (no default, initially 0)

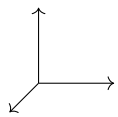
这里的 ⟨*factor*⟩ 一般是不带长度单位的数值，把 ⟨*factor*⟩ 与 x 轴的单位向量相乘就得到本选项所指定的位置。⟨*factor*⟩ 也可以是宏，也可以是比较复杂的算式，会被命令 \pgfmathparse 处理。如果 ⟨*factor*⟩ 带上长度单位，例如 /tikz/cs/x=1cm, 那么它相当于 /tikz/cs/x=28.45274, 因为有转换关系 1cm = 28.45274pt, 命令 \pgfmathparse 会自动做好这种转换。

/tikz/cs/y=⟨*factor*⟩ (no default, initially 0)

类似上一选项。

/tikz/cs/z=⟨*factor*⟩ (no default, initially 0)

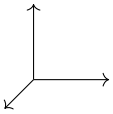
类似上一选项。



```
\begin{tikzpicture}[->]
  \draw (0,0) -- (xyz cs:x=1);
  \draw (0,0) -- (xyz cs:y=1);
  \draw (0,0) -- (xyz cs:z=1);
\end{tikzpicture}
```

```
\begin{tikzpicture}[->]
  \draw (0,0) -- (xyz cs:x=0.1cm);
  \draw (0,0) -- (xyz cs:y=1);
  \draw (0,0) -- (xyz cs:z=1);
\end{tikzpicture}
```

这种坐标的隐式形式是，例如 `(xyz cs:x=a,y=b)` 等价于 `(a,b)`，`(xyz cs:x=a,y=b,z=c)` 等价于 `(a,b,c)`，其中没有长度单位。



```
\begin{tikzpicture}[->]
  \draw (0,0) -- (1,0);
  \draw (0,0) -- (0,1,0);
  \draw (0,0) -- (0,0,1);
\end{tikzpicture}
```

对于给出的隐式坐标 (x_1, x_2), 其坐标数据 x_i 会被 `\pgfmathparse` 解析，

- 如果 x_i 中含有长度单位，则将这个坐标解释到 `canvas` 坐标系统中；x_i 中不带单位的数值被默认具有单位 `pt`;
- 如果 x_i 不含有长度单位，则将其解释到 `xyz` 坐标系统中，
- 例如：
  - `(2pt,3)` 等效于 `(2pt,0pt)+(0,3)`, 也就是把 `2pt` 解释为 `canvas` 坐标系统的 x 轴中的数据，把 `3` 解释为 `xyz` 坐标系统的 y 轴中的数据；
  - `(3+2mm,4)` 等效于 `(3pt+2mm,0pt)+(0,4)`, 这里在 `3` 后面添加长度单位 `pt`
  - `(3+2pt,4+5pt)` 等效于 `(3pt+2pt,0pt)+(0pt,4pt+5pt)`.

参考命令 `\tikz@scan@one@point`.

xyz coordinate system 的定义是：

```
\tikzdeclarecoordinatesystem{xyz}
{%
  \tikzset{cs/.cd,x=0,y=0,z=0,#1}%
  \pgfpointxyz{\tikz@cs@x}{\tikz@cs@y}{\tikz@cs@z}%
}%
```

按这个定义，xyz coordinate system 可以处理三维点。

40.2.1.3 Coordinate system `canvas polar`

在 `canvas polar` 坐标系统下，可以使用选项 `angle=` 和 `radius=` 来指定坐标，这两个选项会被翻译到 `canvas` 坐标系统中来确定一个点。

`/tikz/cs/angle=`⟨*degrees*⟩ (no default)

本选项指定坐标点的角度，⟨*degrees*⟩ 是角度制下的数据，限制在 −360 到 720 之间 (?)，会被命令 `\pgfmathparse` 处理。

`/tikz/cs/radius=`⟨*dimension*⟩ (no default)

本选项同时指定 `x radius=`⟨*dimension*⟩, `y radius=`⟨*dimension*⟩, 参数 ⟨*dimension*⟩ 一般是带长度单位的尺寸，也可以是宏，也可以是比较复杂的算式，它会被命令 `\pgfmathparse` 处理。如果 ⟨*dimension*⟩ 不带长度单位，就认为它的单位是 `pt`.

`/tikz/cs/x radius=`⟨*dimension*⟩ (no default)

`/tikz/cs/y radius=`⟨*dimension*⟩ (no default)

假设 x radius=d_x, y=d_y, angle=θ, 则确定的点是 $(\cos(\theta)d_x, \sin(\theta)d_y)$, 也就是：在横半轴长度是 d_x，纵半轴长度是 d_y 的椭圆上找到方向角为 θ 的点。

```
\tikz \draw (0,0) -- (canvas polar cs:angle=960,radius=1cm);
```

canvas polar 坐标系统下的隐式坐标形式是，例如 (30:1pt)，(30:1mm and 2pt)，角度默认角度制，长度数据都带单位。在极坐标中，可以使用单词 up, down, left, right, north, south, west, east, north east, north west, south east, south west 来代表角度，例如

```
\tikz\draw (0,0) -- (up:1)--++(right:1cm)--cycle;
```

相关定义是：

```
\def\tikz@polar@dir@up{90}%
\def\tikz@polar@dir@down{-90}%
\def\tikz@polar@dir@left{180}%
\def\tikz@polar@dir@right{0}%
\def\tikz@polar@dir@north{90}%
\def\tikz@polar@dir@south{-90}%
\def\tikz@polar@dir@east{0}%
\def\tikz@polar@dir@west{180}%
\expandafter\def\csname tikz@polar@dir@north east\endcsname{45}%
\expandafter\def\csname tikz@polar@dir@north west\endcsname{135}%
\expandafter\def\csname tikz@polar@dir@south east\endcsname{-45}%
\expandafter\def\csname tikz@polar@dir@south west\endcsname{-135}%
```

canvas polar coordinate system 的定义是：

```
\tikzdeclarecoordinatesystem{canvas polar}
{%
  \tikzset{cs/.cd,angle=0,radius=0cm,#1}%
  \pgfpointpolar{\tikz@cs@angle}{\tikz@cs@xradius and \tikz@cs@yradius}%
}%
```

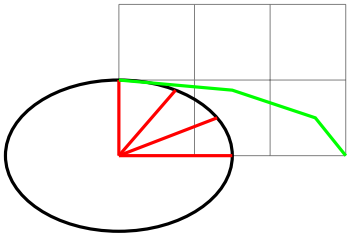
40.2.1.4 Coordinate system xyz polar, xy polar

使用选项 angle，radius，x radius，y radius 提供坐标数据，然后翻译到 xyz 坐标系统中。极坐标系统只用于二维情况，它确定的点位于 xyz 坐标系统的 xy 平面上，它按如下方式确定一个点：

假设 xyz 坐标系统的 x 轴的单位向量是 v_x，y 轴的单位向量是 v_y。在默认下 $v_x = (1\text{cm}, 0\text{cm})$，$v_y = (0\text{cm}, 1\text{cm})$(可以通过变换选项修改)。给出选项 angle=$\theta$，x radius=$f_x$，y radius=$f_y$，这里 θ，f_x，f_y 都是数值，所确定的点是：

$$\cos(\theta)f_x \cdot v_x + \sin(\theta)f_y \cdot v_y$$

也就是：先以 v_x 为横半轴，v_y 为纵半轴确定一个椭圆 E，记 E 的中心点是 O。在 E 上找一个点 P，使得从 v_x 到 $\overrightarrow{OP}$ 的角度是 θ。然后变换 E, 使得横半轴变为 $f_x \cdot v_x$，纵半轴变为 $f_y \cdot v_y$，得到椭圆 E', 从而把点 P 变成点 P'，那么点 P' 就是最终确定的点。注意此时从 v_x 到 $\overrightarrow{OP'}$ 的角度未必还是 θ。



```
\begin{tikzpicture}[x=1.5cm,y=1cm,very thick]
  \draw[help lines] (0cm,0cm) grid (3cm,2cm);
  \draw circle (1);
  {[red]
    \draw (0,0) -- (xyz polar cs:angle=0,radius=1);
    \draw (0,0) -- (xyz polar cs:angle=30,radius=1);
    \draw (0,0) -- (xyz polar cs:angle=60,radius=1);
    \draw (0,0) -- (xyz polar cs:angle=90,radius=1);
  }
  {[green]
    \draw (xyz polar cs:angle=0,x radius=2,y radius=1)
      -- (xyz polar cs:angle=30,x radius=2,y radius=1)
      -- (xyz polar cs:angle=60,x radius=2,y radius=1)
      -- (xyz polar cs:angle=90,x radius=2,y radius=1);
  }
\end{tikzpicture}
```

上面这个例子中，x 轴的单位向量是 $v_x = (1.5\text{cm}, 0\text{cm})$，$y$ 轴的单位向量是 $v_y = (0\text{cm}, 1\text{cm})$，所以画出的圆形实际是椭圆。

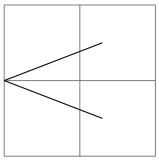
`/tikz/cs/angle=`⟨*degrees*⟩ (no default)

`/tikz/cs/radius=`⟨*factor*⟩ (no default)

本选项同时指定 `x radius=`⟨*factor*⟩，`y radius=`⟨*factor*⟩.

`/tikz/cs/x radius=`⟨*factor*⟩ (no default)

`/tikz/cs/y radius=`⟨*factor*⟩ (no default)



```
\begin{tikzpicture}[x=1.5cm,y=1cm]
\draw[help lines] (0cm,-1cm) grid (2cm,1cm);
\draw (0,0)-- (xyz polar cs:angle=30, radius=1);
\draw (0,0)-- (xyz polar cs:angle=-30, x radius=1, y radius=1pt);
\end{tikzpicture}
```

对应这个坐标系统的隐式坐标形式是，例如 `(30:1)`，`(30:1 and 2)`，角度默认角度制，默认长度单位是 cm，注意没有 `(30:1mm and 2)` 这种带单位混合不带单位的情况。

xyz polar coordinate system 的定义是：

```
\tikzdeclarecoordinatesystem{xyz polar}
{%
  \tikzset{cs/.cd,angle=0,radius=0,#1}%
  \pgfpointpolarxy{\tikz@cs@angle}{\tikz@cs@xradius and \tikz@cs@yradius}%
}%
\tikzaliascoordinatesystem{xy polar}{xyz polar}%
```

可见极坐标系统只用于二维情况，而 `xy polar` 是 `xyz polar` 的别名。

40.2.2 质心坐标系统 barycentric

给定 n 个向量 v_1，$\cdots$，v_n，n 个数值 a_1，$\cdots$，a_n，它们决定一个坐标

$$\frac{a_1 v_1 + \cdots + a_n v_n}{a_1 + \cdots + a_2},$$

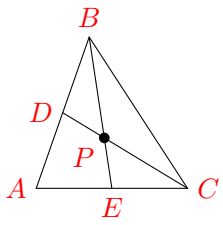
如果把点 v_i 看作质点，把数值 a_i 看作 v_i 的质量，上式就是这个质点组的质心。

与 `barycentric` coordinate system 这个坐标系统对应的坐标格式是：

```
([⟨options⟩]barycentric cs:⟨node name⟩=⟨number⟩,⟨node name⟩=⟨number⟩...)
```

注意其中 ⟨*node name*⟩ 的前后不能随意加空格。⟨*node name*⟩ 是 node(或 coordinate) 的名称，代表 node 的 `center` 点 (看作是“质点”)。目前只能使用 node 的名称，不能使用诸如 `node.center`，`node.north` 等形式，如果你需要用这样的一个点就得先把这个点保存到另一个坐标名称中，然后利用该坐标名称 (例如`\coordinate(save point)at(node.north);`)。

参数 ⟨*number*⟩ 可以是纯数值 (可以是负值)，也可以是其他能被命令 `\pgfmathparse` 解析的表达式，解析结果看作是相应质点的“质量”，注意解析结果的符号可以是正、负、零。




```
\begin{tikzpicture}[every node/.style={red}]
  \coordinate [label=180:$A$] (A) at (0,0);
  \coordinate [label=90:$B$] (B) at (0.7,2);
  \coordinate [label=0:$C$] (C) at (2,0);
  \coordinate [label=180:$D$] (D) at ($(A)!0.5!(B)$);
  \coordinate [label=270:$E$] (E) at ($(A)!0.5!(C)$);
  \coordinate (P) at (barycentric cs:A=-1,B=-1,C=-1);
  \draw (A) -- (B) -- (C) -- cycle;
  \draw (B) -- (E) (C) -- (D);
  \fill (P) circle (2pt) node [below left] {$P$};
\end{tikzpicture}
```

40.2.3 node 坐标系统

创建一个 node 后，就会有与该 node 相关的各种坐标位置，其形式是

```
([⟨options⟩]node cs:⟨key-value list⟩)
```

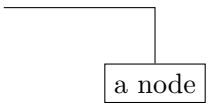
在 ⟨*key-value list*⟩ 中可以使用以下选项。

/tikz/cs/name=⟨*node name*⟩ (no default)

指定 node 的名称，利用该 node 的坐标系统。

/tikz/cs/anchor=⟨*anchor*⟩ (no default)

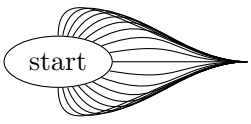
指定 node 的 ⟨*anchor*⟩ 位置点。




```
\begin{tikzpicture}
  \node (circle) at (2,0) [draw] {a node};
  \draw (node cs:name=circle,anchor=north) |- (0,1);
\end{tikzpicture}
```

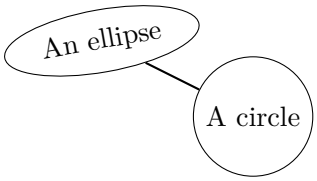
/tikz/cs/angle=⟨*degrees*⟩ (no default)

从 node 的中心点做方向角为 ⟨*degrees*⟩ 的射线，与 node 边界线的交点就是本选项确定的点。⟨*degrees*⟩ 的值会被赋予 TEX 计数器，所以 ⟨*degrees*⟩ 的小数部分会被忽略。




```
\begin{tikzpicture}
  \node (start) [draw,shape=ellipse] {start};
  \foreach \angle in {-90, -80, ..., 90}
  \draw (node cs:name=start,angle=\angle)
    .. controls +(\angle:1cm) and +(-1,0) .. (2.5,0);
\end{tikzpicture}
```

在用线条连接 node 时，或在两个 node 之间画线时，可以用选项 `anchor=` 或 `angle=` 指定的点作为线条的端点；如果不使用这两个选项，而是仅仅给出 node 的名称，那么 Ti*k*Z 会自动计算出 node 边界上的某个“恰当的”点并用线连接起来；如果 Ti*k*Z 不能确定“恰当的”点，就会把 node 的中心点作为线条的端点，不过此时处于 node 内部的那一部分线条会被裁掉。




```
\begin{tikzpicture}
  \path (0,0) node(a) [ellipse,rotate=10,draw] {An ellipse}
        (2,-1) node(b) [circle,draw] {A circle};
  \draw[thick] (node cs:name=a) -- (node cs:name=b);
\end{tikzpicture}
```

在用线条连接两个 node 时，如果仅仅给出 node 的名称，那么对于“line-to”操作“--”，纵横线操作“|-”和“-|”，“curve-to”操作“..”，会自动计算所需的 anchor 位置点作为线条的端点；对于其它操作，例如，parabola 或 plot，直接把 node 的中心点作为线条的端点，处于 node 内部的那一部分线条会被裁掉。

因为在用线条连接 node 时，处于 node 内部的那一部分线条会被裁掉，所以，例如


```
\begin{tikzpicture}
  \node (A) {$A$};
  \draw (-1,0) -- (node cs:name=A) -- (1,0);
\end{tikzpicture}
```

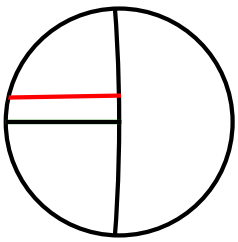
其中的 node 起到了“截断”作用，线条被截成两段，是不连续的，两段之间以 move-to 方式联系。

如果给出 node 名称，然后给出一个相对坐标 (relative coordinates)，那么相对坐标就相对于 node 的中心位置来确定。


```
\tikz{
  \node (x) [draw] {\Large Text};
  \fill [green](node cs:name=x) +(1,0) circle (2pt);
  \fill [red](node cs:name=x,anchor=north) +(1,0) circle (2pt);
}
```

隐式地指定一个 node 坐标的方法很简单，例如，用 node 名称 (a)，用 node 的锚位置 (a.north)，用 node 的角度位置 (a.30)。

注意在隐式格式 (a.⟨*angle*⟩) 中，对于多数预定义的 node 形状 (如 rectangle, circle) 来说，⟨*angle*⟩ 的有效形式是整数形式，或者展开为整数的宏。⟨*angle*⟩ 会被赋予一个整数寄存器 (计数器) 来参与计算，计数器只保存整数，所以 ⟨*angle*⟩ 的小数部分会被忽略。



```
\begin{tikzpicture}[ultra thick]
  \draw [clip] (20,0) circle [radius=1.5cm];
  \node (a)[draw,circle,minimum size=40cm]{};
  \draw [red](0,0)--(a.1);
  \draw [green](0,0)--(a.0.5);
  \draw (0,0)--(a.0);
\end{tikzpicture}
```

上图中没有绿色线，因为 (a.0.5) 被当成 (a.0), 然后绿色线被黑色线覆盖了。

node 坐标系统的定义是：

```
\tikzdeclarecoordinatesystem{node}
{%
  \tikzset{cs/.cd,name=,anchor=none,angle=none,#1}%
  \ifx\tikz@cs@anchor\tikz@nonetext%
    \ifx\tikz@cs@angle\tikz@nonetext%
      \expandafter\ifx\csname pgf@sh@ns@\tikz@cs@node\endcsname\tikz@coordinate@text%
      \else
        \aftergroup\tikz@shapebordertrue%
        \edef\tikz@shapeborder@name{\tikz@pp@name{\tikz@cs@node}}%
      \fi%
      \pgfpointanchor{\tikz@pp@name{\tikz@cs@node}}{center}%
    \else%
      \pgfpointanchor{\tikz@pp@name{\tikz@cs@node}}{\tikz@cs@angle}%
```

```
    \fi%
  \else%
    \pgfpointanchor{\tikz@pp@name{\tikz@cs@node}}{\tikz@cs@anchor}%
  \fi%
}%
```

40.2.4 tangent 坐标系统

这个坐标系统是在文件《tikzlibrarycalc.code.tex》中定义的，调用 TikZ 的库 `calc` 后才能使用 `tangent` 坐标系统。每个 node 都有自己的形状 (shape)，shape 实际上是个路径。假设 ⟨*node*⟩ 的形状是 ⟨*shape*⟩，有一个点 ⟨*point*⟩，过点 ⟨*point*⟩ 做 ⟨*shape*⟩ 的切线，切线可能有数条，每条切线对应一个切点，这些切点就 `tangent` 坐标系统所要确定的点；而且这些切点会被自动编号，可以利用编号来引用切点。

这个坐标系统的坐标没有隐式形式。

这个坐标系统需要用到以下选项。

`/tikz/cs/node`=⟨*node name*⟩ (no default)

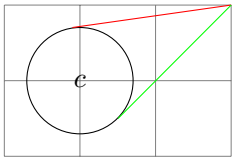
指定所需的 node。

`/tikz/cs/point`=⟨*point*⟩ (no default)

指定点。

`/tikz/cs/solution`=⟨*number*⟩ (no default)

指定切点的编号 ⟨*number*⟩ 来引用相应的切点。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \coordinate (a) at (3,2);
  \node [circle,draw] (c) at (1,1) [minimum size=40pt] {$c$};
  \draw [red] (a)--(tangent cs:node=c,point={(a)},solution=1);
  \draw [green] (a)--(tangent cs:node=c,point={(a)},solution=2);
\end{tikzpicture}
```

这个坐标系统没有隐式的坐标格式。目前，只能针对形状为

- `coordinate`
- `circle`

的 node 计算切点。

40.2.5 自定义坐标系

`\tikzdeclarecoordinatesystem`{⟨*name*⟩}{⟨*code*⟩}

这个命令声明一个新的坐标系统，⟨*name*⟩ 是该系统的名称。该系统对应的坐标形式是

```
(⟨name⟩ cs:⟨arguments⟩)
```

在 ⟨*code*⟩ 中应当包含参数符号 `#1`, 这个参数符号代表的是 ⟨*arguments*⟩. 执行 ⟨*code*⟩ 的应当实现一个计算过程，这个计算过程的结尾处要直接或间接地对 TEX 尺寸寄存器 `\pgf@x`, `\pgf@y` 赋值，这两个寄存器分别作为 x 分量和 y 分量一起确定一个 `canvas` 坐标系统中的点——记为 ⟨*target*⟩——这样就由 ⟨*arguments*⟩ 得到了 ⟨*target*⟩.

通常，⟨*arguments*⟩ 是一个键值列表，在 ⟨*code*⟩ 中通过 key 机制处理 ⟨*arguments*⟩, 也可以用其它方式处理。

例如，xyz coordinate system 的定义是：

```
\tikzdeclarecoordinatesystem{xyz}
{%
  \tikzset{cs/.cd,x=0,y=0,z=0,#1}%
  \pgfpointxyz{\tikz@cs@x}{\tikz@cs@y}{\tikz@cs@z}%
}%
```

当解析 `(xyz cs:x=1,y=2,z=3)` 时，会执行：

```
\tikzset{cs/.cd,x=0,y=0,z=0,x=1,y=2,z=3}%
\pgfpointxyz{\tikz@cs@x}{\tikz@cs@y}{\tikz@cs@z}%
```

得到点 `\pgfpointxyz{1}{2}{3}`，命令 `\pgfpointxyz` → P. 254 会“全局地”为 `\pgf@x`, `\pgf@y` 赋值。本命令的定义是：

```
\def\tikzdeclarecoordinatesystem#1#2{%
  \expandafter\def\csname tikz@parse@cs@#1\endcsname(##1){%
    \pgf@process{%
      #2%
      \global\let\tikz@smubble@b=\tikz@shapeborder@name%
    }%
    \let\tikz@shapeborder@name=\tikz@smubble@b%
    \edef\tikz@return@coordinate{\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}}%
}%
```

`\tikzaliascoordinatesystem{`⟨*new name*⟩`}{`⟨*old name*⟩`}`

这个命令给已存在的坐标系统 ⟨*old name*⟩ 另起一个新名称 ⟨*new name*⟩，两个名称都可用。

40.3 交点坐标

40.3.1 水平线与竖直线的交点：perpendicular 坐标系统

给出点 P 和 Q，`perpendicular` 坐标系统能够计算过点 P 的水平线与过点 Q 的竖直线的交点。

`/tikz/cs/horizontal line through=`{⟨*coordinate*⟩} (no default)

指定水平线所通过的点。

`/tikz/cs/vertical line through=`{⟨*coordinate*⟩} (no default)

指定竖直线所通过的点。

这个坐标系统有隐式的坐标格式：

```
(⟨p⟩ |- ⟨q⟩)
(⟨q⟩ -| ⟨p⟩)
```

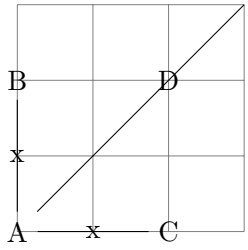
例如 `(2,1 |- 3,4)` 与 `(3,4 -| 2,1)` 表示同一个点，这里注意：

(a) 其中只有一对圆括号，⟨*p*⟩ 和 ⟨*q*⟩ 都不带圆括号；

(b) 点 ⟨*p*⟩ 和 ⟨*q*⟩ 可以是各种坐标系统的格式；

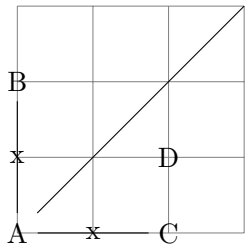
(c) ⟨*p*⟩ 或 ⟨*q*⟩ 可以是运算式，如果算式比较复杂或者算式中含有圆括号，那最好把整个算式用花括号括起来。

(d) `-|` 或 `|-` 可以连续使用，例如 `(0,0 -| 1,1 |- 0,2)` 等于 `(1,0 |- 0,2)`，即按照从左到右的次序依次处理。如果这个序列中包含复杂算式，那么可能需要用花括号来“定界”。




```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,3);
  \node (A) at (0,0) {A};
  \node (B) at (0,2) {B};
  \node (C) at (A -| 2,2) {C};
  \draw (A) -- (B) node [midway] {x};
  \draw (A) -- (C) node [midway] {x};
  \node at ({{$(A)!.5!(B)$} -| {$(A)!.5!(C)$}} |- B -| C) {D};
  \draw (A)--(A |- 1,3 -| {2,0 -| 3,0});
\end{tikzpicture}
```

如果把上面例子中 `{{{$(A)!.5!(B)$} -| {$(A)!.5!(C)$}}` 的外层花括号去掉就是另外一种结果：




```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,3);
  \node (A) at (0,0) {A};
  \node (B) at (0,2) {B};
  \node (C) at (A -| 2,2) {C};
  \draw (A) -- (B) node [midway] {x};
  \draw (A) -- (C) node [midway] {x};
  \node at ({$(A)!.5!(B)$} -| {$(A)!.5!(C)$} |- B -| C) {D};
  \draw (A)--(A |- 1,3 -| 2,0 -| 3,0);
\end{tikzpicture}
```

可见其中的 `|- B` 没有起到作用。

40.3.2 任意路径的交点

TikZ Library intersections

```
\usetikzlibrary{intersections} % LaTeX and plain TeX
\usetikzlibrary[intersections] % ConTeXt
```

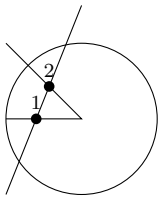
这个库能计算任意两个路径的交点。由于 TEX 的计算精度所限，所处理的路径不能太复杂。特别地，如果一个路径由很多小的线段构成 (如装饰路径)，那最好不要计算它与别的路径的交点。

计算两个路径交点的一般步骤是：在创建这两个路径时用选项 `name path` 给路径命名，然后再开启一个路径或环境并使用选项 `name intersectios` 计算交点。注意，计算交点时一定要确保两个路径有交点，否则结果可能会很意外。

`/tikz/name path`=⟨*name*⟩ (no default)

`/tikz/name path global`=⟨*name*⟩ (no default)

这两个选项的作用是，在路径创建后、使用前，给路径配备名称 ⟨*name*⟩。选项 `name path` 所做的命名只在当前的 scope 内有效。选项 `name path global` 所做的命名全局有效。当给一个路径命名后，路径名称所指的内容就只有该路径本身，不包括添加到此路径上的 node。




```
\begin{tikzpicture}
  \draw [name path=aaa] (0,0)--
    (1,0) node[name path=aaa,circle,minimum size=2cm,draw]{}
    --(0,1);
  \draw [name path=bbb] (0,-1)--(1,1.5);
  \fill [name intersections={of=aaa and bbb,name=i,total=\t}]
    \foreach \s in {1,...,\t}
    {(i-\s) circle (2pt) node [above] {\footnotesize\s}};
\end{tikzpicture}
```

下面的交点都能得到：

```
\begin{tikzpicture}
    \draw [name path=a] (0,0) -- (2,3) node  [ name path=b, ...]{};
    \fill [name intersections={of=a and b,...}] ...;
```

```
\end{tikzpicture}
\begin{tikzpicture}
    \draw (0,0) -- (2,3) node  [name path=a, ...]{};
    \draw [name path=b] (2,3) -- (3,0);
    \fill [name intersections={of=a and b,...}] ...;
\end{tikzpicture}
```

在文件《tikzlibraryintersections.code.tex》中，选项 name path 的定义是：

```
\pgfkeys{%
  /tikz/name path/.code={%
    % hm. Do we need this "reset old option" as in 'name path global'
    % for this case as well?
    \tikz@key@name@path{#1}{\def}%
  },
}%
```

执行选项 name path=⟨*name*⟩ 导致执行

```
\tikz@key@name@path{⟨name⟩}{\def}
```

命令 \tikz@key@name@path 的定义是：

```
\def\tikz@key@name@path#1#2{%
    \tikz@addmode{%
      \pgfsyssoftpath@getcurrentpath\tikz@intersect@temppath@round%
      \pgfprocessround\tikz@intersect@temppath@round\tikz@intersect@temppath%
      \ifx\tikz@intersect@namedpaths\pgfutil@empty%
      \else%
        \tikz@intersect@namedpaths%
      \fi%
      \tikz@intersect@addto@path@names{#1}{#2}%
    }%
}%
```

此命令会用 \tikz@addmode → P. 765 向 \tikz@mode 中添加一些代码，如果执行这些代码，那么其作用是：

1. 执行 \pgfsyssoftpath@getcurrentpath, 将当前路径的软路径形式保存到宏 \tikz@intersect@temppath@round 中，
2. 执行 \pgfprocessround, 检查是否需要将 \tikz@intersect@temppath@round 中的软路径的尖角变成圆角，并做相应的尖角、圆角变化，再把改变后 (或者没有改变) 的软路径保存到宏 \tikz@intersect@temppath 中
3. 检查 \tikz@intersect@namedpaths 是否等于 \pgfutil@empty, 如果不是，则执行 \tikz@intersect@namedpaths(见下文). 文件《tikzlibraryintersections.code.tex》规定

   ```
   \let\tikz@intersect@namedpaths=\pgfutil@empty
   ```

4. 执行 \tikz@intersect@addto@path@names{⟨*name*⟩}\def

命令 \tikz@intersect@addto@path@names 的定义是：

```
\def\tikz@intersect@addto@path@names#1#2{%
  \edef\tikz@marshal{#2\expandafter\noexpand\csname tikz@intersect@path@name@#1
  ↪  \endcsname}%
  \expandafter\expandafter\expandafter\def%
  \expandafter\expandafter\expandafter\tikz@marshal%
    \expandafter\expandafter\expandafter{\expandafter\tikz@marshal\expandafter{
    ↪  \tikz@intersect@temppath}}%
```

```
  \expandafter\pgfutil@g@addto@macro\expandafter\tikz@intersect@namedpaths
  ↪  \expandafter{\tikz@marshal}%
}%
```

所以 \tikz@intersect@addto@path@names{⟨name⟩}\def 导致

```
\edef\tikz@marshal{\def\expandafter\noexpand\csname tikz@intersect@path@name@
↪  ⟨name⟩\endcsname}
\def\tikz@marshal{% 重定义 \tikz@marshal
  \def\csname tikz@intersect@path@name@⟨name⟩\endcsname{%
    ⟨ 展开的 \tikz@intersect@temppath, 即当前路径的软路径形式⟩
  }%
}%
\pgfutil@g@addto@macro\tikz@intersect@namedpaths{展开的 \tikz@marshal}
```

\pgfutil@g@addto@macro 全局地定义宏 \tikz@intersect@namedpaths, 这个宏里面保存的是一系列定义

```
\expandafter\def\csname tikz@intersect@path@name@⟨name 1⟩\endcsname{软路径 1}%
\expandafter\def\csname tikz@intersect@path@name@⟨name 2⟩\endcsname{软路径 2}%
\expandafter\def\csname tikz@intersect@path@name@⟨name 3⟩\endcsname{软路径 3}%
%......
```

所以展开宏 \tikz@intersect@namedpaths 的结果就是这些定义，有：

```
% at the end of evey \path command ...
\let\tikz@finish@orig=\tikz@finish
\def\tikz@finish{%
  \tikz@finish@orig%
  \tikz@intersect@finish%
}%

% ... make the named path variables available
\def\tikz@intersect@finish{%
  \tikz@intersect@namedpaths%
}%
```

所以在命令 \tikz@finish → P. 759 的处理过程的末尾处，宏 \tikz@intersect@namedpaths 将被展开。

/tikz/name intersections={⟨options⟩} (no default)

这里 ⟨options⟩ 中的选项必须是以 /tikz/intersection 为前缀的选项 (在下文列出)，因为这个 key 会自动把 ⟨options⟩ 中的选项前缀改成 /tikz/intersection，然后处理之。两个路径每相交一次就创建一个交点坐标，第一个被创建的交点坐标会被自动命名为 intersection-1，第二个被创建的交点坐标会被自动命名为 intersection-2，依次类推。可以用选项 /tikz/intersection/name 修改交点名称，交点坐标的总数目也可以保存下来。

/tikz/intersection/of=⟨name path 1⟩ and ⟨name path 2⟩ (no default)

这个选项指定针对哪两条路径计算其交点。

执行这个选项导致执行

```
\tikz@intersect@path@names@parse⟨name path 1⟩ and ⟨name path 2⟩\tikz@stop
导致
\def\tikz@intersect@path@a{⟨name path 1⟩}%
\def\tikz@intersect@path@b{⟨name path 2⟩}%
```

可见执行这个选项就定义了宏 \tikz@intersect@path@a, \tikz@intersect@path@b, 这两个宏的初始值被 let 为 \pgfutil@empty:

```
\let\tikz@intersect@path@a=\pgfutil@empty
\let\tikz@intersect@path@b=\pgfutil@empty
```

/tikz/intersection/name=⟨*prefix*⟩ (no default, initially intersection)

执行这个选项导致执行

```
\def\tikz@intersect@name{⟨prefix⟩}
```

这个选项会修改宏 \tikz@intersect@name 的值，这个宏的初始值被 let 为 \pgfutil@empty:

```
\let\tikz@intersect@name=\pgfutil@empty
```

如果这个选项的值是空的，那么宏 \tikz@intersect@name 就等于 \pgfutil@empty.

选项 name intersections 会检查宏 \tikz@intersect@name 是否等于 \pgfutil@empty:

- 如果等于 \pgfutil@empty, 就

  ```
  \def\tikz@intersect@@name{intersection}%
  ```

  这会使得各个交点的名称是 intersection-1, intersection-2, ...
- 如果不等于 \pgfutil@empty, 就

  ```
  \let\tikz@intersect@@name=\tikz@intersect@name%
  ```

  这会使得各个交点的名称是 ⟨*prefix*⟩-1, ⟨*prefix*⟩-2, ...

宏 \tikz@intersect@@name 保存交点名称的前缀。

/tikz/intersection/total=⟨*macro*⟩ (no default)

执行这个选项导致执行

```
\def\tikz@intersect@total{⟨macro⟩}
```

这个选项会修改宏 \tikz@intersect@total 的值，这个宏的初始值被 let 为 \pgfutil@empty:

```
\let\tikz@intersect@total=\pgfutil@empty
```

如果这个选项的值是空的，那么宏 \tikz@intersect@total 就等于 \pgfutil@empty.

选项 name intersections 会检查宏 \tikz@intersect@total 是否等于 \pgfutil@empty:

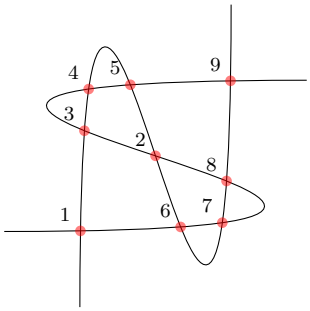
- 如果等于 \pgfutil@empty, 则什么也不做
- 如果不等于 \pgfutil@empty, 就

  ```
  \expandafter\let\tikz@intersect@total=\pgfintersectionsolutions%
  ```

  也就是

  ```
  \let⟨macro⟩=\pgfintersectionsolutions%
  ```

  使得 ⟨*macro*⟩ 保存交点总数。




```
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1]
    (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw [name path=curve 2]
    (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);
  \fill [name intersections={of=curve 1 and curve 2, name=i, total=\t}]
    [red, opacity=0.5, every node/.style={above left, black, opacity=1
    ↪  }]
    \foreach \s in {1,...,\t}
      {(i-\s) circle (2pt) node {\footnotesize\s}};
\end{tikzpicture}
```

/tikz/intersection/by={⟨*comma-separated list*⟩} (no default)

这个 key 的作用是给交点起“别名”，或者对交点执行某些操作。⟨*comma-separated list*⟩ 是一个用逗号分隔的列表，这个列表会被 `\foreach` 语句处理，所以列表的形式是 `\foreach` 语句所允许的形式。⟨*comma-separated list*⟩ 的格式是

[⟨*options 1*⟩]⟨*name 1*⟩,[⟨*options 2*⟩]⟨*name 2*⟩...

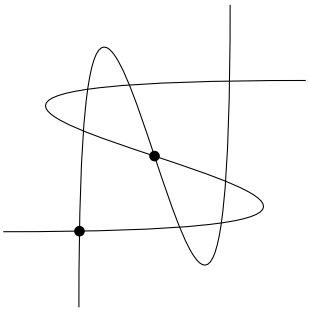
- ⟨*comma-separated list*⟩ 中列举项目的个数不能大于 (可以小于) 交点的总个数，其中第一个列举项目对应第一个交点，第二个列举项目对应第二个交点，以此类推。如果在列表中使用 `\foreach` 语句所允许的省略号，则自动把列举项目的个数变成交点个数，使二者 (交点个数与列举项目个数) 自动匹配。
- ⟨*name i*⟩ 是可选的，代表第 ⟨*i*⟩ 个交点的别名；
- [⟨*options i*⟩] 是可选的，它是一组针对第 ⟨*i*⟩ 个交点的选项设置。注意 [⟨*options i*⟩] 带方括号。

实际上对于每一个列举项目 [⟨*options i*⟩]⟨*name i*⟩, 都会执行：

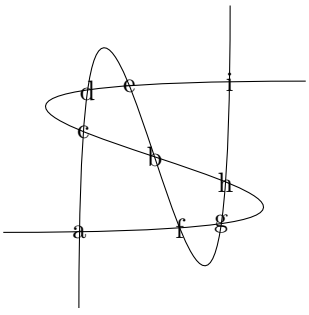
```
\coordinate [⟨options i⟩] (⟨name i⟩) at (\tikz@intersect@@name-
↪  \tikz@intersection@count);
```

也就是在第 ⟨*i*⟩ 个交点位置上另外创建一个名称为 ⟨*name i*⟩ 的新坐标点，同时为这个新坐标点执行选项 [⟨*options i*⟩].
执行这个选项导致执行

```
\def\tikz@intersect@by{⟨comma-separated list⟩}
```



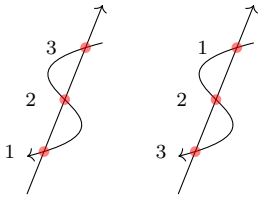
```
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1)
  ↪  .. (2,1);
  \draw [name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8)
  ↪  .. (1,2);
  \fill [name intersections={of=curve 1 and curve 2, by={a,b}}]
    (a) circle (2pt)
    (b) circle (2pt);
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1)
  ↪  .. (2,1);
  \draw [name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8)
  ↪  .. (1,2);
  \fill [name intersections={
    of=curve 1 and curve 2,
    by={[label=center:a],[label=center:...],[label=center:i]}}];
\end{tikzpicture}
```

/tikz/intersection/sort by=⟨*path name*⟩ (no default)

在默认下，按照寻找交点的算法依次给交点命名，但这个算法规则并不直观。这个选项按路径 ⟨*path name*⟩ 的走向将交点排序，顺次命名 (安排交点的编号)，名称仍然是 ⟨*prefix*⟩-⟨*number*⟩ 的形式。




```
\begin{tikzpicture}
  \clip (-0.5,-0.75) rectangle (3.25,2.25);
  \foreach \pathname/\shift in {line/0cm, curve/2cm}{
  \tikzset{xshift=\shift}
  \draw [->, name path=curve] (1,1.5) .. controls (-1,1) and
  ↪  (2,0.5) .. (0,0);
  \draw [->, name path=line] (0,-.5) -- (1,2) ;
  \fill [name intersections={of=line and curve,sort by=\pathname,
  ↪  name=i}]
    [red, opacity=0.5, every node/.style={left=.25cm, black,
    ↪  opacity=1}]
  \foreach \s in {1,2,3}{(i-\s) circle (2pt) node {\footnotesize\s
  ↪  }};
  }
\end{tikzpicture}
```

这个选项导致

```
\edef\tikz@intersect@sort@by{⟨path name⟩}\tikz@intersect@check@sort@by%
```

命令 \tikz@intersect@check@sort@by 的定义是：

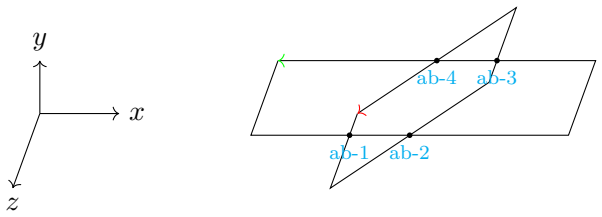
```
\def\tikz@intersect@check@sort@by{%
    \ifx\tikz@intersect@sort@by\tikz@intersect@path@a%
        \pgfintersectionsortbyfirstpath%
    \else%
        \ifx\tikz@intersect@sort@by\tikz@intersect@path@b%
            \pgfintersectionsortbysecondpath%
        \else%
            \pgf@intersect@sortfalse%
        \fi%
    \fi%
}%
```

以上代码中的 \pgfintersectionsortbyfirstpath, \pgfintersectionsortbysecondpath 见文件《pgflibraryintersections.code.tex》：

```
\newif\ifpgf@intersect@sort
\newif\ifpgf@intersect@sort@by@second@path
\def\pgfintersectionsortbyfirstpath{%
  \pgf@intersect@sorttrue%
  \pgf@intersect@sort@by@second@pathfalse%
}%
\def\pgfintersectionsortbysecondpath{%
  \pgf@intersect@sorttrue%
  \pgf@intersect@sort@by@second@pathtrue%
}%
```

可见，选项 sort by=⟨*path name*⟩ 应该写在选项 of=⟨*name path 1*⟩ and ⟨*name path 2*⟩ 的后面，也就是说，应该先处理选项 of，再处理选项 sort by，这是因为选项 sort by 的处理用到 \tikz@intersect@path@a, \tikz@intersect@path@b. 如果选项 sort by 写在了选项 of 的前面，不会报错，但可能得不到预期的结果。如果 ⟨*path name*⟩ 是个无效名称，也不会导致报错。

假设 ⟨*name path 1*⟩ 与 ⟨*name path 2*⟩ 都是用 3 维坐标创建的路径，那么它们的交点也是可以计算的，不过会把它们作为 2 维路径来计算交点，得到交点也是 2 维点，即只有两个坐标分量。这是因为只是针对软路径计算交点，而各种路径的软路径都是平面上的路径。例如：



(ab-1)= $(-4.24121pt, -11.65254pt)$

```
\tikz[z={(-110:1.5cm)},x={(-0:1.5cm)},scale=0.7]{
  \begin{scope}[shift={(-4,0)}]
    \draw [->](0,0,0)--(1,0,0) node[right]{$x$};
    \draw [->](0,0,0)--(0,1,0) node[above]{$y$};
    \draw [->](0,0,0)--(0,0,1) node[below]{$z$};
  \end{scope}

  \draw [-{>[red]},name path=a,]
    (0,0,0)--++(0,0,1)--++(2,2,0)--++(0,0,-1)--(0,0,0);
  \draw [-{>[green]},name path=b,]
    (-1,1,0)--++(0,0,1)--++(4,0,0)--++(0,0,-1)--(-1,1,0);
  {[font=\footnotesize,text=cyan]
    \fill [name intersections={of=a and b,name=ab,sort by=a,total=\t}]
      \foreach \i in {1,...,\t} {(ab-\i) circle(1.5pt) node[below]{ab-\i}};
  }
  \path let \p1=(ab-1) in node at(6,0){\texttt{(ab-1)}$=(\x1, \y1)$};
}
```

上面例子中，如果路径 a，b 是 3 维路径，那么交点 (ab-1)，(ab-3) 都是不该有的。

40.3.2.1 选项 /tikz/name intersections 的处理

执行

\pgfkeys{/tikz/name intersections={⟨*options*⟩}}

导致的处理过程大致如下：

1. 执行选项 by= 清空此选项保存的内容。
2. 执行 ⟨*options*⟩ 中列出的选项。
3. 检查命令 \tikz@intersect@path@name@\tikz@intersect@path@a 是否已定义，
   - 如果已定义，则
     (a) 检查命令 \tikz@intersect@path@name@\tikz@intersect@path@a 是否已定义，
     – 如果已定义，则
     i. 执行

```
\pgfintersectionofpaths%
{%
  \expandafter\pgfsetpath\csname tikz@intersect@path@name@
  ↪  \tikz@intersect@path@a\endcsname%
}%
{%
  \expandafter\pgfsetpath\csname tikz@intersect@path@name@
  ↪  \tikz@intersect@path@b\endcsname%
}%
```

计算两个路径的交点。其中的命令 \pgfintersectionofpaths 是文件《pgflibraryintersections.code.tex》定义的。

ii. 如果 ⟨*options*⟩ 中使用了选项 total=⟨*macro*⟩，就把交点总数保存到 ⟨*macro*⟩ 中；如果没有使用这个选项，那么可以用 \pgfintersectionsolutions 得到交点总数。

iii. 如果 ⟨*options*⟩ 中使用了选项 name=⟨*prefix*⟩，就有定义

```
\def\tikz@intersect@name{⟨prefix⟩}%
\let\tikz@intersect@@name=\tikz@intersect@name%
```

否则定义

```
\def\tikz@intersect@@name{intersection}%
```

也就是说，可以用 intersection-1, intersection-2, ... 来引用交点。

iv. 用循环 \pgfmathloop ... \repeatpgfmathloop 执行

```
\path[reset cm]\pgfextra{\pgftransformshift{
↪  \pgfpointintersectionsolution{\pgfmathcounter}}}%
  coordinate (\tikz@intersect@@name-\pgfmathcounter);
```

其中的 \pgfmathcounter 的值从 1 递增到 \pgfintersectionsolutions. 这实际是在各个交点位置上创建新的命名坐标。注意其中使用了选项 reset cm, 以及命令 \pgftransformshift→P. 264.

注意，当 \pgfintersectionsolutions 不大于 0（没有交点）时，上述循环不会创建坐标名称。

v. 如果 ⟨*options*⟩ 中使用了选项 by=⟨*comma-separated list*⟩, 则把 ⟨*comma-separated list*⟩ 保存在宏 \tikz@intersect@by 中，执行 foerach 循环操作：

```
\foreach\tikz@intersect@name[count=\tikz@intersection@count] in
↪  \tikz@intersect@by%
{\expandafter\tikz@intersect@name@parse\tikz@intersect@name\pgf@stop}
↪  %
```

\tikz@intersect@name@parse 的定义是：

```
\def\tikz@intersect@name@parse{\pgfutil@ifnextchar[
↪  \tikz@intersect@name@parse@opt{\tikz@intersect@name@parse@opt[]}}
↪  %}
\def\tikz@intersect@name@parse@opt[#1]#2\pgf@stop{%
  \coordinate [#1] (#2) at (\tikz@intersect@@name-
  ↪  \tikz@intersection@count);
}%
```

– 如果未定义，则执行

```
\tikzerror{I do not know the path named `\tikz@intersect@path@b'.
↪  Perhaps you misspelt it}}%
```

给出错误提示。

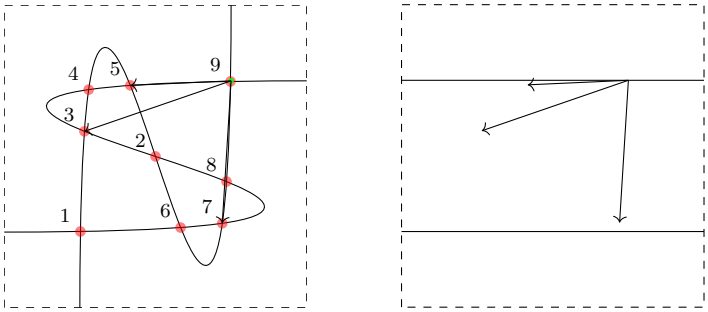
- 如果未定义，则执行

```
\tikzerror{I do not know the path named `\tikz@intersect@path@a'. Perhaps you
↪  misspelt it}}%
```

给出错误提示。

40.3.2.2 注意的问题

第一类问题：交点是全局定义的 假如有前后两个 {tikzpicture} 环境，前面的 {tikzpicture} 环境使用选项 name intersections 计算出了交点 (i-1), (i-2), (i-3), 后面的 {tikzpicture} 环境只是计算出了交点 (i-1), 那么前面环境得到的交点 (i-2), (i-3) 仍然可以用在后面的环境中，例如：



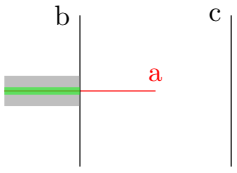

```
\begin{tikzpicture}
    \clip (-2,-2) rectangle (2,2);
    \draw [name path=curve 1](-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
    \draw [name path=curve 2](-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);
    \fill [name intersections={of=curve 1 and curve 2, name=i, total=\t,}]
          [red, opacity=0.5, every node/.style={above left, black, opacity=1}]
        \foreach \s in {1,...,\t} {(i-\s) circle [radius=2pt] node {\footnotesize\s}};
    \fill [green] (1,1) circle [radius=1pt];
    \draw [->](1,1)--(i-3);
    \draw [->](1,1)--(i-5);
    \draw [->](1,1)--(i-7);
    \coordinate (a) at (current bounding box.south west);
    \coordinate (b) at (current bounding box.north east);
    \draw [dashed](a) rectangle (b);
\end{tikzpicture}
\hspace{1cm}
\begin{tikzpicture}
    \draw [name path=A] (-2,1)--(2,1);
    \draw [name path=B](-2,-1)--(2,-1);
    \fill [name intersections={of=A and B, name=i}];
    \draw [->](1,1)--(i-3);
    \draw [->](1,1)--(i-5);
    \draw [->](1,1)--(i-7);
    \draw [dashed](a) rectangle (b);
\end{tikzpicture}
```

出现这一情况的原因是，选项 `name intersections` 在执行循环 `\pgfmathloop ... \repeatpgfmathloop` 时，并不会清除编号过大的坐标名称，而坐标名称 (也就是 node 名称) 是全局定义的。此时需要配合选项 `/tikz/intersection/total`, 或者宏 `\pgfintersectionsolutions` 来使用交点，以防错误地引用交点。

第二类问题：选项 by 容易出现的问题 库文件《pgflibraryintersections.code.tex》可能有以下处理：

- 如果两个线段没有交点，那么什么都不做，并不会给出错误信息；
- 如果一个线段与一个曲线、一个曲线与一个曲线没有交点，那么就返回原点，

对于以上两个情况，要注意计算结果是否所期望的交点。




```
\begin{tikzpicture}
  \draw [name path=a,red] (0,0)--(2,0) node[above]{a};
  \draw [name path=b] (1,-1)--(1,1) node[left]{b};
  \draw [name path=c] (3,-1)--(3,1) node[left]{c};
  \path [name intersections={of=a and b, by=m}];
  \path [name intersections={of=a and c, by=n}];
  \draw [line width=4mm,gray,opacity=0.5](0,0)--(m);
  \draw [line width=1mm,green,opacity=0.5](0,0)--(n);
\end{tikzpicture}
```

上面例子中，第一次计算交点时选项 `by=m` 给交点起一个别名 `(m)`, 这是正确的；但第二次计算交点时，两个线段没有交点，库《pgflibraryintersections.code.tex》什么都不做，选项 `by=n` 给交点起的别名 `(n)` 与 `(m)` 实际上是同一个点。这是因为选项 `by=n` 只是为点 ⟨*name*⟩-⟨*count*⟩（此处是 `intersection-1`）起一个别名；而点 ⟨*name*⟩-⟨*count*⟩ 在第一次计算交点时已经被全局地定义。

40.4 相对坐标，增量坐标

40.4.1 指定相对坐标

通常，在构建路径的过程中，“当前点”指的是，在构建路径的某个时刻，刚刚被构建路径的操作 (move-to, line-to, curve-to, close) 处理过的、被正式纳入路径中的坐标点。例如对于

```
\draw (0,0) (1,1);
```

其中在 (0,0) 与 (1,1) 之间有一个 move-to 操作。在执行这个 move-to 操作时，点 (0,0) 是这个 move-to 操作的“生长点”，即当前点。当执行完毕这个 move-to 操作，把点 (1,1) 纳入路径中后，点 (1,1) 就成为当前点。这个意思对“--”操作也是一样的，而“..”操作就稍微复杂一些。有时候需要经过计算才能得到点的坐标，在这个计算过程中涉及的坐标数据、以及计算结果都不被直接纳入路径，只有被构建路径的操作处理过的计算结果才被 (正式) 纳入路径。

“参照点”指的是在计算某个坐标位置时所参照的、相对的点。

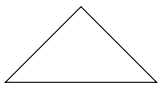
对于有两个加号的相对坐标 如 (1,1)--++(30:2) 中的“++”的效果是：计算 (1,1)+(30:2) 的和, 然后，在“--” (line-to) 操作中，把当前点变成 (1,1)+(30:2), 也把当前参照点变成 (1,1)+(30:2).

```
\tikz \draw (0,0) -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
```

上例中，当前点从 (0,0) 逐步变到 (0,1)，图形等价于

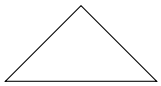
```
\tikz \draw (0,0) --  (1,0) --  (1,1) --  (0,1) -- cycle;
```

对于有一个加号的相对坐标 如 (1,1)--+(30:2) 中的“+”的效果是：计算 (1,1)+(30:2), 在“--” (line-to) 操作中，当前点变成 (1,1)+(30:2), 但保持当前参照点不变。



```
\tikz \draw (0,0) -- +(1,0) -- +(0,1) -- +(-1,0) -- cycle;
```

上例中的参照点始终是“(0,0)”，上例等价于

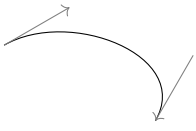


```
\tikz \draw (0,0) --  (1,0) --  (0,1) --  (-1,0) -- cycle;
```

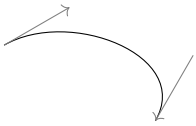
对 (A)--+(B)--(C)--+(D) 中的 +(D) 来说，其参照点是 (C)，例如

```
\tikz \draw (0,0)--+(1,0)--(1,1)--+(-1,0);
```

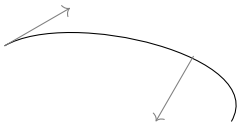
如果将相对坐标用作 Bézier 曲线的控制点，那么其中的相对参照规则是：对于 (A)..controls +(B) and +(c)..+(D)，则 +(B) 相对于 (A)，而 +(C) 相对于 +(D)，+(D) 相对于 (A)。这种相对参照规则有利于掌握控制曲线在起点、终点处的切线方向。



```
\begin{tikzpicture}
  \draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. (3,-1);
  \draw[gray,->] (1,0) -- +(30:1cm);
  \draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. +(2,-1);
  \draw[gray,->] (1,0) -- +(30:1cm);
  \draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. +(3,-1);
  \draw[gray,->] (1,0) -- +(30:1cm);
  \draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```

40.4.2 旋转的相对坐标——曲线上一点处的坐标系

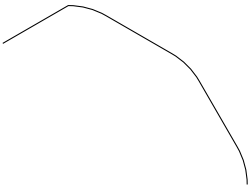
在构建路径的时候，如果想利用路径在当前点的切线方向就可以使用 turn 坐标系统。设点 P 是曲线 s 上的一点，曲线 s 在点 P 处的单位切向量是 $\mathbf{x}_P$，与 $\mathbf{x}_P$ 成右手直角系的单位向量记为 $\mathbf{y}_P$，那么以点 P 为原点，以 $\mathbf{x}_P$ 为 x 轴的单位向量，以 $\mathbf{y}_P$ 为 y 轴的单位向量构成的坐标系就是 turn 坐标系统采用的参照系。

/tikz/turn (no value)

这个 key 用作坐标的选项，它会局部地平移、旋转坐标系得到它需要的参照系。

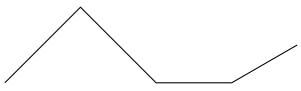
```
\tikz \draw (0,0) -- (1,1) -- ([turn]-45:1cm) -- ([turn]-30:1cm);
```

上面例子中第一个 [turn] 的作用是：第一，确定路径在当前点 (即 (1,1) 点) 的 turn 坐标系，这个坐标系是 {(1,1);(45:1cm),(135:1cm)}，即以 (1,1) 为原点，以 (45:1cm) 为 x 轴的单位向量，以 (135:1cm) 为 y 轴的单位向量的坐标系，这是通过局部地平移、旋转原来的坐标系得到的；第二，在这个 turn 坐标系内找出坐标为 (-45:1cm) 的点，使之成为当前点。



```
\tikz [delta angle=30, radius=1cm]
  \draw (0,0) arc [start angle=0] -- ([turn]0:1cm)
    arc [start angle=30] -- ([turn]0:1cm)
    arc [start angle=60] -- ([turn]30:1cm);
```

```
\tikz \draw (0,0) to [bend left] (2,1) -- ([turn]0:1cm);
```

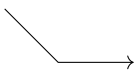


```
\tikz \draw plot coordinates {(0,0) (1,1) (2,0) (3,0) } --
↪  ([turn]30:1cm);
```

Mark

用 turn 坐标系统时要注意下面的情况。观察下面的例子：

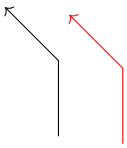
```
\tikz \draw [->](0,0)-- ([turn]1,0);
```



```
\tikz \draw [->] (0,0)--([turn]45:1)--([turn]45:1);
```

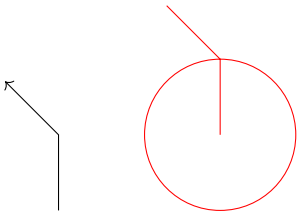
上面两个例子说明，如果路径以 (0,0)--([turn]...) 开头，则这个 turn 坐标系的 x 轴的正方向指向下方。

```
\tikz \draw [->] (0,0)(1,0)--([turn]45:1)--([turn]45:1);
```



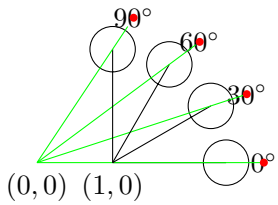
```
\tikz \draw [->] (0,0)(0.1,0.1)--([turn]45:1)--([turn]45:1);
\tikz \draw [->,red] (1,1)--([turn]45:1)--([turn]45:1);
```

在画圆的操作 `circle` 后面使用 `[turn]` 时也要注意。观察下图：



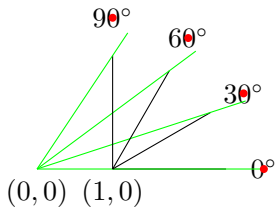
```
\tikz \draw [->] (1,1)--([turn]45:1)--([turn]45:1);\hspace{1cm}
\tikz \draw [->,red] (1,1)
↪  circle(1)--([turn]45:1)--([turn]45:1);
```

上面例子中，点 `(1,1)` 是画圆操作 `circle` 的当前点，画完圆后的当前点又返回到 `(1,1)`，所以两个 `\draw` 命令的 `[turn]` 方向其实是一样的。




```
\begin{tikzpicture}
\foreach  \ang in {0,30,...,90}
{ \draw (1,0)-- +(\ang:1.5) coordinate (d\ang) circle (0.3)
    ([turn] 0.5,0) node (node\ang){$\ang^\circ$};
  \draw [green] (0,0) -- ($(0,0)!1.2!(d\ang)$);
  \fill [red](node\ang.center) circle (1.5pt);}
  \node [below] at (1,0) {$(1,0)$};
  \node [below] at (0,0) {$(0,0)$};
\end{tikzpicture}
```

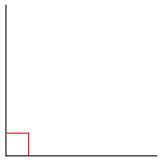
上面例子中，`(1,0)-- +(\ang:1.5)` 是黑色的射线，按原来的期望，坐标点 `([turn] 0.5,0)` 应当位于黑色射线上，也就是说，坐标点 `([turn] 0.5,0)` 的标签 (角度标签) 的中心点应当位于黑色射线上，但却跑到了绿色射线上。将上面代码中的 `circle (0.3)` 删除重画：




```
\begin{tikzpicture}
\foreach  \ang in {0,30,...,90}
{ \draw (1,0)-- +(\ang:1.5) coordinate (d\ang)
    ([turn] 0.5,0) node (node\ang){$\ang^\circ$};
  \draw [green] (0,0) -- ($(0,0)!1.2!(d\ang)$);
  \fill [red](node\ang.center) circle (1.5pt);}
  \node [below] at (1,0) {$(1,0)$};
  \node [below] at (0,0) {$(0,0)$};
\end{tikzpicture}
```

这样坐标点 `([turn] 0.5,0)` 标签的位置就符合原来的期望了。

使用选项 `/tikz/turn` 可以画直角标记，例如



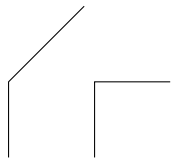
```
\begin{tikzpicture}
  \draw (0,0) -| (-2,2);
  \path (0,0)--(-2,0)--([turn]-90:3mm)coordinate(a)
    --([turn]-90:3mm)coordinate(b)--([turn]-90:3mm)coordinate(c);
  \draw [red](a)--(b)--(c);
\end{tikzpicture}
```

40.4.3 相对坐标的参照点的局部化

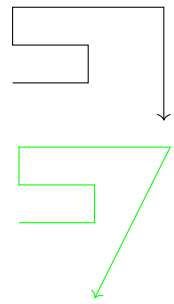
可以在路径内部使用花括号创建一个 scope, 通常，路径内的这种 scope 只能限制某些选项的作用范围，并不能将相对坐标的参照点也限制在内。下面的选项可以将相对坐标的参照点局部化。

`/tikz/current point is local=`⟨*boolean*⟩ (no default, initially `false`)

本选项一般用在“路径内的 scope”中，当本选项的值是 `false` 时，相对坐标的参照点的变化如常。在路径内部创建一个 scop，假设这个 scop 开始时 (读取开花括号 `{` 时) 的当前点是 p(这是 scope 之前的当前点)；在 scop 内部，当前点会不断变化，当这个 scop 结束时 (读取闭花括号 `}` 时)，命令 `\path` 会检查本选项的值是否为 `true`，如果是就把当前点设置为 p, 即恢复到 scope 之前的那个当前点。

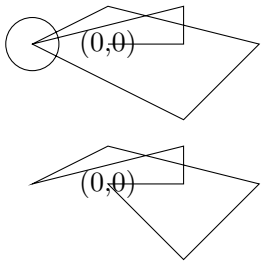


```
\tikz\draw (0,0) -- ++(0,1) -- ++(1,1);
\tikz\draw (0,0) {[current point is local]-- ++(0,1)} -- ++(1,1);
```



```
\tikz \draw[->] (0,0) -- ++(1,0) -- ++(0,0.5)
    -- ++(-1,0) -- ++(0,0.5) -- ++(2,0)
    -- ++(0,-1.5); \\[3mm]
\tikz \draw[->,green] (0,0) -- ++(1,0) -- ++(0,0.5)
    {[current point is local]-- ++(-1,0) -- ++(0,0.5) -- ++(2,0) }
    -- ++(0,-1.5);
```

但是当 scope 内含有 circle 路径时，情况会变得复杂，例如




```
\tikz \draw (0,0) node{(0,0)} -- ++(1,0) -- ++(0,0.5)
  {[current point is local]-- (-1,0) circle (10pt)--(0,0.5)--(2,0)}
  -- ++(0,-1.5) -- cycle; \\[3mm]
\tikz \draw (0,0) node{(0,0)} -- ++(1,0) -- ++(0,0.5)
  {[current point is local]-- (-1,0) -- (0,0.5) -- (2,0) }
  -- ++(0,-1.5) -- cycle;
```

这是因为 circle 会用一个 move-to 操作将当前点变到圆心位置；另外 cycle 导致的 close 操作只把当前点和“最近出现的 move-to 的落脚点”连起来。

40.5 坐标计算

TikZ Library `calc`

```
\usetikzlibrary{calc} % LaTeX and plain TeX
\usetikzlibrary[calc] % ConTeXt
```

本节介绍的坐标计算功能需要库 `calc` 的支持。

40.5.1 一般句法

`([`⟨*options*⟩`]$`⟨*coordinate computation*⟩`$)`

例如 `($(1,2)-(2,3)$)` 计算两个向量的差。注意句式中的两个 `$` 表示 TikZ 的坐标计算，不代表 TEX 的数学模式。在 TikZ 用命令 `\tikz@scan@one@point` → P. 710 解析坐标的过程中，符号 `$` 会导致执行命令 `\tikz@parse@calculator`, 解析坐标算式。

注意，两个坐标 (向量) 之间只能做“加、减”运算。

40.5.2 数乘坐标 (向量)

当坐标 (向量) 有系数时，基本句法是：

$$\langle factor\rangle\texttt{*}\langle coordinate\rangle$$

或者

$$\langle factor\ 1\rangle\texttt{*}\langle coordinate\ 1\rangle \pm \langle factor\ 2\rangle\texttt{*}\langle coordinate\ 2\rangle$$

当按照这个格式写出一串符号后，需要判断这串符号中的哪一部分属于坐标 (向量) 的系数，为了避免无法判断的错误，注意以下几点：

- $\langle factor\rangle$* 这一部分是可选的，如果没有 $\langle factor\rangle$ 那就不要有 *。
- 如果 Ti*k*Z 认为你提供了 $\langle factor\rangle$，那么在第一个符号组合 *(之前的那些符号就是 $\langle factor\rangle$。因此，如果你提供的 $\langle factor\rangle$ 中含有符号组合 *(，那就应当用花括号把整个 $\langle factor\rangle$ 括起来，以避免歧义。例如 ($sqrt(5)*(1/3)*(2,3)$) 会导致错误，而 (${sqrt(5)*(1/3)}*(2,3)$) 则可以接受。
- 对于 $\langle factor\rangle$ 之后、$\langle coordinate\rangle$ 之前的乘积符号 * 来说，如果这个 * 的两侧 (或某一侧) 是圆括号，那么这个 * 与圆括号之间不能有空格。但在 $\langle factor\rangle$ 内部的 * 两侧可以有空格。
- 像 ($+(1,2)$), ($-(1,2)$) 这样只是强调一个“正、负向量”的表达式会导致错误，而 ($+1*(1,2)$), ($-1*(1,2)$) 则可以接受。
- 当 $\langle factor\rangle$ 以“(”开头时，最好用花括号把 $\langle factor\rangle$ 包裹起来，因为此时 Ti*k*Z 倾向于认为遇到了一个坐标 (向量)。例如像 ($(2)*(1,2)$), ($(-1)*(1,2)$) 这样的表达式会导致错误，而 (${(2)}*(1,2)$), (${(-1)}*(1,2)$), ($2*(1,2)$), ($-1*(1,2)$) 则可以接受。
- 提供 $\langle factor\rangle$ 后，宏 \pgfmathparse 会解析 $\langle factor\rangle$，所以 $\langle factor\rangle$ 可以是复杂的算式，但最好使用花括号包裹 $\langle factor\rangle$，例如

  ($\{atan(sqrt(5) *(2/3)) / 100 * exp(1/5 * ln(7))\}*(2,3)$)

  其中 {atan(sqrt(5) *(2/3)) / 100 * exp(1/5 * ln(7))} 是 $\langle factor\rangle$，整个坐标就是

  $$\frac{\arctan\left(\sqrt{5}\cdot\frac{2}{3}\right)}{100}\cdot\sqrt[5]{7}\cdot\begin{pmatrix}2\\3\end{pmatrix}.$$

- 像 ($2*((1,2)-(2,3))$) 这种圆括号直接套嵌圆括号的形式会导致错误，此时可以套嵌使用 $...$，例如 ($2*($(1,2)-(2,3)$)$) 是可以接受的。
- 像 ($cos(30)*({cos(10)},{sin(10)})+({cos(20)},{sin(20)})$) 这种形式是可以接受的，注意其中使用花括号来避免出现“圆括号直接套嵌圆括号”的情况。
- 如果在 perpendicular 坐标系统中使用坐标计算式，可能需要将 $...$ 之外层的圆括号换成花括号，例如：

```
\fill ({$(a.south)+(0,-40pt)$} -| {$(a.east)+(40pt,0)$}) circle(2pt);
```

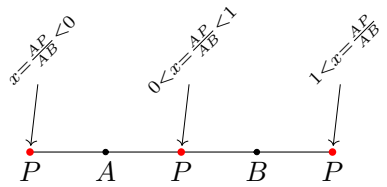
40.5.3 比例—角度定点句法

⟨*coordinate*⟩!⟨*number*⟩!⟨*angle*⟩:⟨*second coordinate*⟩

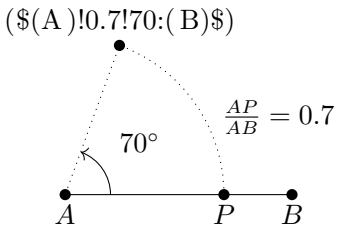
其中 ⟨*number*⟩ 是任意实数，⟨*angle*⟩ 是角度制下的数值，⟨*angle*⟩: 是可选的。⟨*number*⟩ 会被 `\pgfmathparse` 解析，所以 ⟨*number*⟩ 可以是比较复杂的算式。

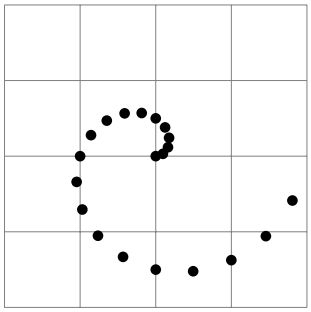
例如 `($(A)!x!(B)$)` 等价于 `(${(1-x)}*(A)+x*(B)$)`. 记 `($(A)!x!(B)$)` 为 P，则 x 与 P 的对应关系如下图所示

$P =$`($(A)!x!(B)$) = (${(1-x)}*(A)+x*(B)$)`



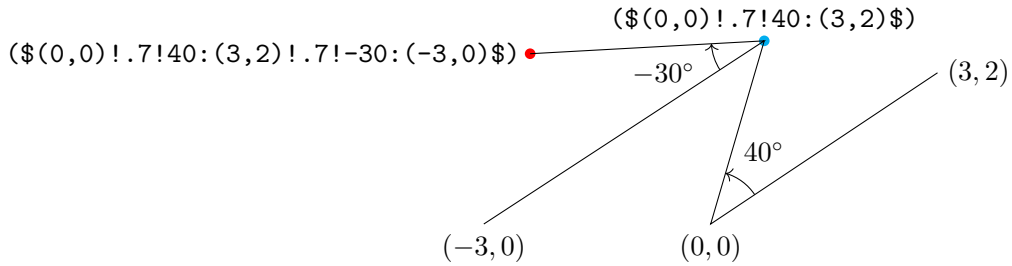
记 `($(A)!x!`⟨*angle*⟩`:(B)$)` 为点 Q，Q 这样得到：先将点 `(B)` 绕 `(A)` 旋转角度 ⟨*angle*⟩, 得到点 `(B')`, 然后再计算 `($(A)!x!`⟨*angle*⟩`:(B')$)`; 这相当于先计算 `($(A)!x!(B)$)` (记为 P)，然后将 P 绕点 A 旋转角度 ⟨*angle*⟩，得到 Q. 例如，`($(A)!0.7!70:(B)$)` 如下图所示





```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (4,4);
  \foreach \i in {0,0.1,...,2}
  \fill ($(2,2) !\i! \i*180:(3,2)$) circle (2pt);
\end{tikzpicture}
```

这种句式可以套嵌叠加使用，例如，如果设置 `\coordinate(x)at($(A)!0.7!70:(B)$);` 那么 `($(A)!0.7!70:(B)!0.5!20:(C)$)` 就等价于 `($(x)!0.5!20:(C)$)`.




```
\begin{tikzpicture}
  \coordinate (o) at (0,0);
  \coordinate (b) at (3,2);
  \coordinate (a) at ($(o)!.7!40:(b)$);
  \coordinate (d) at (-3,0);
  \coordinate (c) at ($(a)!.7!-30:(d)$);

  \fill [cyan]  (a) circle (2pt);
```

```
  \draw  (b) node [right] {$(3,2)$}
      -- (o) node [below] {$(0,0)$}
      -- (a) node [above] {\ttt{(\$(0,0)!.7!40:(3,2)\$)}}
      pic [draw, ->, "$40^\circ$", angle radius=7mm, angle eccentricity=1.7] {angle = b--o--a};
  \fill [red]  (c)  circle (2pt);
  \draw (d) node [below] {$(-3,0)$} -- (a)
      -- (c) node [left] {\ttt{(\$(0,0)!.7!40:(3,2)!.7!-30:(-3,0)\$)}}
      pic [draw, <-,  "$-30^\circ$", angle radius=7mm, angle eccentricity=2] {angle =c--a--d};
\end{tikzpicture}
```

注意的问题 在 ⟨*coordinate A*⟩`!`⟨*expression*⟩`!`⟨*angle*⟩`:`⟨*coordinate B*⟩ 这个句式中：

- 当 ⟨*expression*⟩ 以“`(`”开头时，最好用花括号把 ⟨*expression*⟩ 包裹起来，因为此时 Ti*k*Z 倾向于认为遇到了一个坐标 (向量)。例如 `($(1,3)!(1+2)*3!(1,1)$)` 会导致错误。

```
\begin{tikzpicture}
    \draw [->](0,0)--($(0,1)!{(1-0.6)*3}!(2,1)$);
\end{tikzpicture}
```

  按文件《tikzlibrarycalc.code.tex》,当用花括号包裹 ⟨*expression*⟩ 时,⟨*expression*⟩ 会成为 `\tikz@cc@mid@num` 的参数，被 `\pgfmathparse` 处理。
- ⟨*expression*⟩ 会被命令 `\pgfmathparse`→P. 110 解析，假设解析的结果是 x(不带长度单位的数值)。如果 ⟨*expression*⟩ 中含有长度单位，那么 `\ifpgfmathunitsdeclared` 的真值是 true，此时这个句式等价于

$$\langle \textit{coordinate A} \rangle \texttt{!}x\texttt{pt!} \langle \textit{angle} \rangle \texttt{:} \langle \textit{coordinate B} \rangle$$

  见下文，参考 `\pgfmathscalar`→P. 114。
- 对 ⟨*angle*⟩ 的处理是:`\pgftransformrotate`→P. 265`{`⟨*angle*⟩`}`, 所以 ⟨*angle*⟩ 也会被命令 `\pgfmathparse` 解析。

40.5.4 距离—角度定点句法

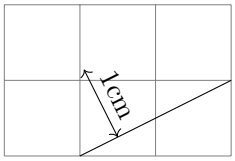
⟨*coordinate*⟩`!`⟨*dimension*⟩`!`⟨*angle*⟩`:`⟨*second coordinate*⟩

其中 ⟨*dimension*⟩ 是带单位的长度的表达式，将被命令 `\pgfmathparse`→P. 110 解析，解析结果可以是负值。⟨*angle*⟩`:` 是可选的，会被 `\pgfmathparse` 解析。。

`($(A)!`⟨*dimension*⟩`!(B)$)` 是以点 `(A)` 为起点，沿着向量 $\overrightarrow{\texttt{(A)(B)}}$ 的方向，移动 ⟨*dimension*⟩ 所确定的点，⟨*dimension*⟩ 可以是负值尺寸。

`($(A)!`⟨*dimension*⟩`!`⟨*angle*⟩`:(B)$)` 这样计算:先将点 `(B)` 绕 `(A)` 旋转角度 ⟨*angle*⟩, 得到点 `(B')`, 然后再计算 `($(A)!`⟨*dimension*⟩`!(B')$)`; 这相当于是将 `($(A)!`⟨*dimension*⟩`!(B)$)` 绕点 `(A)` 旋转角度 ⟨*angle*⟩ 确定的点。

注意，当 ⟨*dimension*⟩ 以“`(`”开头时，最好用花括号把 ⟨*dimension*⟩ 包裹起来，因为此时 Ti*k*Z 倾向于认为遇到了一个坐标 (向量)。

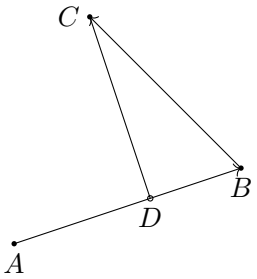



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \coordinate (a) at (1,0);
  \coordinate (b) at (3,1);
  \draw (a) -- (b);
  \coordinate (c) at ($ (a)!.25!(b) $);
  \coordinate (d) at ($ (c)!1cm!90:(b) $);
  \draw [<->] (c) -- (d) node [sloped,midway,above] {1cm};
\end{tikzpicture}
```

40.5.5 正射影—角度定点句法

⟨*coordinate*⟩!⟨*projection coordinate*⟩!⟨*angle*⟩:⟨*second coordinate*⟩

`($(A)!(C)!(B)$)` 表示的是点 `(C)` 在直线 `(A)(B)` 上的垂足，计算的是点 $\left(\overrightarrow{BC}\cdot\frac{\overrightarrow{AB}}{|AB|}\right)\cdot\frac{\overrightarrow{AB}}{|AB|}+$`(B)`，如下图：



点 `(A)`，`(B)`，`(C)` 会被 `\tikz@scan@one@point` 解析，⟨*angle*⟩ 会被 `\pgfmathparse` 解析。

`($(A)!(C)!`⟨*angle*⟩`:(B)$)` 的主要处理是：
假设点 `(A)` 的坐标是 (x_A,y_A)`pt`，点 `(B)` 的坐标是 (x_B,y_B)`pt`，点 `(C)` 的坐标是 (x_C,y_C)`pt`，

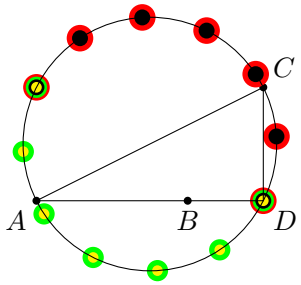
1. 用 `\begingroup` 开启一个组
2. 某些操作……(略)
3. 设置一个花括号组，在这个组内执行以下步骤：
   (a) 执行 `\pgftransformreset`
   (b) 执行平移命令 `\pgftransformshift{\pgfqpoint{`x_A`pt}{`y_A`pt}}`
   (c) 执行旋转命令 `\pgftransformrotate{`⟨*angle*⟩`}`
   (d) 执行平移命令 `\pgftransformshift{\pgfqpoint{`$-x_A$`pt}{`$-y_A$`pt}}`
   (e) 执行以上变换命令后，把用于描点的绘图标架 (坐标系) 记为 $x'O'y'$.
   (f) 对 `(B)` 的坐标做变换 `\pgfpointtransformed{\pgfqpoint{`x_B`pt}{`y_B`pt}}`，也就是在标架 $x'O'y'$ 中找出坐标为 (x_B,y_B)`pt` 的点，记之为 B'，其坐标记为 (x,y)`pt`.
   这个花括号组的作用是限制组内的变换命令 (标架 $x'O'y'$ 只在这个组内有效)，以及局部地赋值计算，计算的结果 (x,y)`pt` 则会被推到这个花括号组之后，做进一步的计算。
4. 计算：

$$\begin{aligned}
&B'-A=(x-x_A,y-y_A)\texttt{pt}=(\theta:r\texttt{pt})=\overrightarrow{AB'},\\
&C-A=(x_C-x_A,y_C-y_A)\texttt{pt}=(x'_C,y'_C)\texttt{pt}=\overrightarrow{AC},\\
&w=x'_C\cos(\theta)+y'_C\sin(\theta),\\
&D=(x_A+w\cos(\theta),y_A+w\sin(\theta))\texttt{pt},
\end{aligned}$$

   也就是计算点 $D=\left(\overrightarrow{AC}\cdot\frac{\overrightarrow{AB'}}{|AB'|}\right)\cdot\frac{\overrightarrow{AB'}}{|AB'|}+$`(A)`.
5. 用 `\endgroup` 结束之前设置的组
6. 把点 D 的坐标全局地保存到 `\pgf@x`, `\pgf@y` 中。

简言之，`($(A)!(C)!`⟨*angle*⟩`:(B)$)` 的处理相当于：将点 `(B)` 绕点 `(A)` 旋转角度 ⟨*angle*⟩，得到点 `(B')`，然后计算 `($(A)!(C)!(B')$)`，所得到的点位于以 `(A)(C)` 为直径的圆上。

观察下面的图形：




```
\begin{tikzpicture}
  \coordinate (A) at (0,0);
  \coordinate (B) at (2,0);
  \coordinate (C) at (3,1.5);
  \coordinate (D) at ($(A)!(C)!(B)$); % (C) 在直线 (A)(B) 上的垂足
  \coordinate (O) at ($(A)!0.5!(C)$); % (A)(C) 的中点

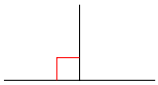
  \fill (A) node [below left] {$A$} circle (1.5pt)
        (B) node [below] {$B$} circle (1.5pt)
        (C) node [above right] {$C$} circle (1.5pt)
        (D) node [below right] {$D$} circle (1.5pt);

  \foreach \ang in {0,15,...,90}
    \fill [red]    ($(A)!(C)!\ang:(B)$) circle (5pt) ;
  \foreach \ang in {90,105,...,180}
    \fill [green]  ($(A)!(C)!\ang:(B)$) circle (4pt) ;
  \foreach \ang in {180,195,...,270}
    \fill [black]  ($(A)!(C)!\ang:(B)$) circle (3pt);
  \foreach \ang in {270,285,...,360}
    \fill [yellow] ($(A)!(C)!\ang:(B)$) circle (2pt);

  \node [draw,circle through={(D)}] at(O){};
  \draw (A)--(D)--(C)--cycle;
\end{tikzpicture}
```

上图中的圆以 AC 为直径，从 `($(A)!(C)!0:(B)$)` 到 `($(A)!(C)!180:(B)$)` 恰好绕圆一周，从 `($(A) ! (C) ! 180 : (B)$)` 到 `($(A)!(C)!360:(B)$)` 也是恰好绕圆一周。

利用以上两种句法可以画直角标记，例如：



```
\begin{tikzpicture}
  \draw (1,1)--(1,0) (0,0)--(2,0);
  \coordinate (a) at ($(0,0)!(1,1)!(2,0)$);
  \draw [red]($(a)!3mm!(1,1)$) -- ($(a)!3mm!(1,1)!3mm!90:(1,1)$) --
  ↪  ([turn]90:3mm);
\end{tikzpicture}
```

40.6 TikZ 解析坐标的命令

在文件《tikz.code.tex》中的命令 `\tikz@scan@one@point` 解析各种 TikZ 坐标形式，如，`(1,1)`，`+(1,1)`，`++(30:2)`，`($2*(1,2)$)`，`(a-|b)`，`(xyz cs:x=1)` 等，解析的结果 (多数情况下) 是基本层的命令规定的一个点，如解析 `(1cm,1cm)` 的结果是 `\pgfpoint{1cm}{1cm}`，其他的 TikZ 坐标形式会被解析为其他的基本层命令，如 `\pgfqpoint`，`\pgfpointpolar`，`\pgfpointanchor`，`\pgfpointxy`，`\pgfpointpolarxy` 等基本层命令规定的点。执行这些基本层命令的主要结果是为尺寸寄存器 `\pgf@x`，`\pgf@y` 赋值，其赋值可能是全局的，也可能是局部的，参考基本层部分。

例如，一般情况下：

- 像 `(1cm,1cm)` 这种带有长度单位的、普通坐标形式的解析结果是 `\pgfpoint` →P. 250 命令，此命令对 `\pgf@x`，`\pgf@y` 的赋值不是全局地。
- 像 `(1,1)` 这种不带长度单位的、普通坐标形式的解析结果是 `\pgfqpoint` →P. 251 命令，此命令对 `\pgf@x`，`\pgf@y` 的赋值是全局地。
- 对于 coordinate 名称，以及 node 名称，调用命令 `\pgfpointanchor` →P. 469 计算出相应的 `center` 位置，其解析结果是对 `\pgf@x`，`\pgf@y` 做全局赋值。
- 对于 node 的 anchor 位置、node 边界上的点，也是调用命令 `\pgfpointanchor` →P. 469 计算出相应的坐标，其解析结果是对 `\pgf@x`，`\pgf@y` 做全局赋值。

`\tikz@scan@one@point`⟨*command*⟩⟨*TikZ coordinate*⟩

此命令的使用格式是

\tikz@scan@one@point⟨\macro⟩⟨TikZ point⟩

其中的 ⟨TikZ point⟩ 是将要被解析的 TikZ 的坐标形式，而 ⟨\macro⟩ 应该是这样定义的宏：

```
\def⟨\macro⟩#1{%
    #1%
    % 处理 \pgf@x, \pgf@y 的代码
}
```

也就是说解析 ⟨TikZ point⟩ 的结果 (基本层命令表达的点) 恰是 ⟨\macro⟩ 的参数，所以，⟨\macro⟩ 可以是，例如 \pgfpathmoveto, 而

\tikz@scan@one@point\pgfpathmoveto(30:2)

的结果是

\pgfpathmoveto{\pgfpointpolarxy {30}{2 and 2}}

在解析 ⟨TikZ point⟩ 时，不考虑之前的变换矩阵。

10.0pt,, 15.0pt

```
\makeatletter
\def\aaa#1#2{%
    \tikz@scan@one@point\pgf@process#2%
    \pgf@x=#1\pgf@x%
    \pgf@y=#1\pgf@y%
}
\aaa{5}{(2pt,3pt)}%
\the\pgf@x,, \the\pgf@y
\makeatother
```

上面例子用 5 来乘向量 (2pt,3pt).

2.0pt,, 2.0pt

```
\makeatletter
\def\aaa#1{#1}%
\tikz@scan@one@point\aaa(2pt,2pt)%
{\tikz@scan@one@point\aaa(1pt,1pt)}
\the\pgf@x,, \the\pgf@y
\makeatother
```

上面例子表明命令 \pgfpoint → P. 250 对 \pgf@x, \pgf@y 的赋值不是全局地。

命令 \tikz@scan@one@point 能解析各种形式的坐标，所以它的完整定义很长，在解析坐标表达式时，它还会使用 calc 库的命令。

40.6.1 参数 ⟨TikZ coordinate⟩ 的形式

在《tikz.code.tex》中有以下定义：

```
\def\tikz@scan@one@point#1{%
  \let\tikz@to@use@whom=\tikz@to@use@last@coordinate%
  \tikz@shapeborderfalse%
  \pgfutil@ifnextchar+{\tikz@scan@relative#1}{\tikz@scan@absolute#1}}%
\def\tikz@scan@absolute#1{%
  \pgfutil@ifnextchar({\tikz@scan@@absolute#1}%)
  {%
    \advance\tikz@expandcount by -1
    \ifnum\tikz@expandcount<0\relax%
      \let\pgfutil@next=\tikz@@scangiveup%
    \else%
      \let\pgfutil@next=\tikz@@scanexpand%
    \fi%
    \pgfutil@next{#1}%
  }%
```

```
}%
\def\tikz@@scanexpand#1{\expandafter\tikz@scan@one@point\expandafter#1}%
\def\tikz@@scangiveup#1{\tikzerror{Cannot parse this coordinate}#1{\pgfpointorigin}}%
```

以上定义对参数 ⟨*TikZ coordinate*⟩ 的形式作出了限制。在简单情况下，它是以 + 或 (开头的 TikZ 坐标。它也可以是一个保存 TikZ 坐标的宏，此时 TikZ 会设置一个花括号组，在这个组内用 \tikz@@scanexpand 展开这个宏。有可能需要多次展开 (套嵌多个花括号组)，直到得到 TikZ 的坐标。每展开一次，计数器 \tikz@expandcount 的值就被减 1, 当这个计数器的值小于 0 时，就报错。

参数 ⟨*TikZ coordinate*⟩ 不能是用花括号包裹的坐标，因为 \expandafter 对开花括号 { 无效。例如下面代码

```
\makeatletter
\tikz@scan@one@point\pgf@process{(0,0)}
\makeatother
```

会导致错误。

40.6.2 解析 coordinate 名称, node 名称, node 的 anchor 位置，node 边界上的点

当 \tikz@scan@one@point 认为被解析的坐标是 coordinate 名称, node 名称, node 的 anchor 位置、node 边界 (即 node 的 shape 路径) 上的点时，就调用命令 \tikz@parse@node.

说明一 对于 \tikz@parse@node\⟨*macro*⟩(⟨坐标名⟩) 来说：

- 当 ⟨坐标名⟩ 是个 coordinate 名称，node 名称时，命令 \tikz@parse@node 就执行

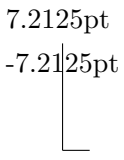
  ```
  \tikz@calc@anchor⟨ 坐标名⟩.center\tikz@stop
  ```

  计算其 center 位置点的坐标。
- 当 ⟨坐标名⟩ 代表 node 的 anchor 位置、边界上的点时，命令 \tikz@parse@node 就执行，例如

  ```
  \tikz@calc@anchor⟨ 坐标名⟩.north\tikz@stop
  或者
  \tikz@calc@anchor⟨ 坐标名⟩.60\tikz@stop
  ```

  计算 node 边界上的点坐标。

上述的命令 \tikz@calc@anchor 会调用 \pgfpointanchor→P. 469 计算相应的坐标，注意这个命令会在当前变换矩阵的逆矩阵下计算相应点的坐标，参考 \pgfpointanchor→P. 469, 例如：




```
\begin{tikzpicture}
    \node (a)[minimum size=20pt]{};
    \makeatletter
    \def\testtransform#1{%
        #1%
        \edef\aaaa{\the\pgf@x}%
        \edef\bbbb{\the\pgf@y}%
    }
    \begin{scope}[rotate=45]
        \tikz@scan@one@point\testtransform(a.0)%
        \draw (0,0)--(1,1) node [align=left]{\aaaa\\ \bbbb};
        \draw (0,0)--(a.0);
    \end{scope}
    \makeatother
\end{tikzpicture}
```

上面例子中，第一个 \node 命令面临的变换矩阵是单位矩阵，它创建 (a) 后，其边界上的点 (a.0) 的坐标是 (10.2pt,0.0pt)(其中的 ".2pt" 是半个线宽，一个线宽默认是 0.4pt)；环境 scope 的选项 rotate=45 改变了变换矩阵；宏 \aaaa, \bbbb 是在 rotate=45 的矩阵的逆矩阵下计算出来的 (a.0) 的坐标，即

(7.2125pt,-7.2125pt); 然后,第二个 \draw 命令又在 rotate=45 的矩阵下对坐标 (7.2125pt,-7.2125pt) 做变换，恰好得到 (10.2pt,0.0pt).

说明二 对于 \tikz@parse@node\⟨*macro*⟩(⟨*node* 的本名⟩) 来说：

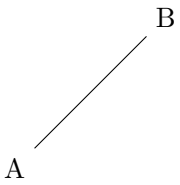
- 当用命令 \pgfmultipartnode 创建一个 node 时，需要为 node 指定一个 shape 名称。当用命令 \node 创建一个 node 时，也可以为 node 指定一个 shape (见 /tikz/shape→P. 805), 如果不指定，一般就默认为 rectangle.
  命令 \tikz@parse@node 会检查 \csname pgf@sh@ns@⟨*node* 的全名⟩\endcsname 是否已定义，这个控制序列保存的是 ⟨*node* 的全名⟩ 具有的 shape 名称，也就是检查这个 shape 是否已定义；这里的 ⟨*node* 的全名⟩ 指的是“前缀—本名—后缀”这种名称，“前缀”由选项 /tikz/name prefix→P. 806 指定，“后缀”由选项 /tikz/name suffix→P. 807 指定，注意“前缀”、“后缀”都是局部有效的：
  - 如果已定义，并且 shape 名称不是单词“coordinate”, 命令 \tikz@parse@node 会设置真值 \tikz@shapebordertrue, 并把 ⟨*node* 的全名⟩ 保存到 \tikz@shapeborder@name 中：

    ```
    \tikz@shapebordertrue%
    \def\tikz@shapeborder@name{\tikz@pp@name{⟨node 的本名⟩}}%
    ```

  - 如果未定义，就再检查 \csname pgf@sh@ns@⟨*node* 的本名⟩\endcsname 是否已定义，也就是检查 ⟨*node* 的本名⟩ 具有的 shape 是否已定义：
    * 如果已定义，并且 shape 名称不是单词“coordinate”, 命令 \tikz@parse@node 会设置真值 \tikz@shapebordertrue, 并把 ⟨*node* 的本名⟩ 保存到 \tikz@shapeborder@name 中：

      ```
      \tikz@shapebordertrue%
      \def\tikz@shapeborder@name{⟨node 的本名⟩}%
      ```

  规定真值 \tikz@shapebordertrue 以及宏 \tikz@shapeborder@name 是为了与 line-to, curve-to 操作配合。例如在下面的图形中：


  ```
  \tikz{
      \node (a) at (0,0) {A};
      \node (b) at (2,2) {B};
      \draw (a)--(b);
  }
  ```

  连接 (a) 与 (b) 的线段的起点在 (a) 的边界上，终点在 (b) 的边界上。在 \draw 命令中，解析 (a) 与 (b) 时会设置真值 \tikz@shapebordertrue, 这导致 line-to 操作“--”启用计算边界点的步骤，参考 \tikz@lineto 的定义。
- 如果 ⟨*node* 的全名⟩ 或者 ⟨*node* 的本名⟩ 是已定义的，那它一定是全局定义的，与这个 node 有关的信息 (它的位置坐标、尺寸、形状等) 也是全局定义的，这导致下面的现象：


  ```
  \begin{tikzpicture}
  \draw [->] (0,0)--(1,1) node [right] (A) {$A$};
  \draw [->] (2,0)--(A);
  \end{tikzpicture}
  \hspace{1cm}%%%
  \begin{tikzpicture}
  \draw (A.south west) rectangle (A.north east);
  \draw [->] (2,0)--(A);
  \end{tikzpicture}
  ```

  上面例子中，第一个 {tikzpicture} 环境里与 (A) 有关的信息仍然可以用在第二个 {tikzpicture}

环境里。

40.6.3 长度单位的影响

举例来说，当命令 \tikz@scan@one@point 解析 (⟨x⟩,⟨y⟩) 或者 (30:⟨r⟩) 时，其中的 ⟨x⟩, ⟨y⟩, ⟨r⟩ 都会被 \pgfmathparse→P. 110 解析。如果 ⟨x⟩, ⟨y⟩, ⟨r⟩ 中出现了长度单位，那么 \ifpgfmathunitsdeclared→P. 113 的真值就是 true, 命令 \tikz@checkunit 引用这个真值，这个命令的定义是：

```
\newif\iftikz@isdimension
\def\tikz@checkunit#1{%
  \pgfmathparse{#1}%
  \let\iftikz@isdimension=\ifpgfmathunitsdeclared%
}%
```

各个情况是：

- 对于 (30:⟨r⟩)
  - 如果 ⟨r⟩ 有长度单位，就得到坐标

    ```
    \pgfpointpolar{30}{⟨r⟩ and ⟨r⟩}
    ```

  - 如果 ⟨r⟩ 没有长度单位，就得到坐标

    ```
    \pgfpointpolarxy{30}{⟨r⟩ and ⟨r⟩}
    ```

- 对于 (⟨x⟩,⟨y⟩)
  - 如果 ⟨x⟩ 有长度单位，⟨y⟩ 有长度单位，就得到坐标

    ```
    \pgfpoint{⟨x⟩}{⟨y⟩}
    ```

  - 如果 ⟨x⟩ 有长度单位，⟨y⟩ 没有长度单位，就得到坐标

    ```
    \pgfpointadd{\pgfpoint{x}{0pt}}{\pgfpointxy{0}{y}}
    ```

  - 如果 ⟨x⟩ 没有长度单位，⟨y⟩ 有长度单位，就得到坐标

    ```
    \pgfpointadd{\pgfpoint{0pt}{y}}{\pgfpointxy{x}{0}}
    ```

  - 如果 ⟨x⟩ 没有长度单位，⟨y⟩ 没有长度单位，就得到坐标

    ```
    \pgfpointxy{⟨x⟩}{⟨y⟩}
    ```

按命令 \pgfpointxy→P. 253, \pgfpointpolarxy→P. 254 的定义，二者都受到 \pgfsetxvec→P. 253 的影响，而 \pgfpoint→P. 250, \pgfpointpolar→P. 251 不会受到 \pgfsetxvec 的影响。

40.6.4 解析 Coordinate System

如果被解析的坐标形式中含有符号“cs:”，例如 (xyz cs:x=1,y=2), 那么 \tikz@scan@one@point 就调用命令 \tikz@parse@coordinatesystem, 其定义是：

```
\def\tikz@parse@coordinatesystem#1(#2 cs:#3){%
  \let\tikz@return@coordinate=\pgfpointorigin%
  \pgfutil@ifundefined{tikz@parse@cs@#2}
  {\tikzerror{Unknown coordinate system '#2'}}
  {\csname tikz@parse@cs@#2\endcsname(#3)}%
  \expandafter#1\expandafter{\tikz@return@coordinate}%
}%
```

可见本命令会检查控制序列 \csname tikz@parse@cs@⟨*cs name*⟩\endcsname 是否已定义，如果已定义，就执行

\csname tikz@parse@cs@⟨*cs name*⟩\endcsname(⟨*arguments*⟩)

这会得到一个基本层的坐标命令，并将其保存到 \tikz@return@coordinate 中。

参考 \tikzdeclarecoordinatesystem →P. 691.

40.6.5 解析三维坐标

对于三维点 (⟨*x*⟩,⟨*y*⟩,⟨*z*⟩)，无论 ⟨*x*⟩, ⟨*y*⟩, ⟨*z*⟩ 是否含有长度单位，都会被转为

\pgfpointxyz{⟨*x*⟩}{⟨*y*⟩}{⟨*z*⟩}

所以在 ⟨*x*⟩, ⟨*y*⟩, ⟨*z*⟩ 中最好不要出现长度单位。

40.6.6 解析空坐标

例如

\tikz@scan@one@point\xxxx(,2)

会转变为

\xxxx{\pgfpointxyz{}{2}{}}

按 \pgfpointxyz 的定义，\pgfpointxyz{}{2}{} 得到这样一个点：这个点的 x 坐标是 xyz 坐标系统的 x 轴的单位坐标；这个点的 y 坐标是 xyz 坐标系统的 y 轴的单位坐标的 2 倍；这个点的 z 坐标是 xyz 坐标系统的 z 轴的单位坐标。

40.7 路径的当前点

从 Ti*k*Z 的处理过程看，“当前点”这个说法比较模糊。

40.7.1 第一种当前点

在 Ti*k*Z 的处理一个点坐标的过程中，命令 \tikz@scan@one@point 的解析结果——记为 {⟨*last point*⟩}——会被 \tikz@moveto, \tikz@lineto, \tikz@curveA, \tikz@curveC 等命令处理，对于这些命令来说，“当前点”——如果可以的话——应该指的是 \tikz@last@position 保存的点，这个点的坐标分量由尺寸寄存器 \tikz@lastx, \tikz@lasty 临时、局部的保存。

在路径命令 \path 开始的时候，寄存器 \tikz@lastx, \tikz@lasty 的值就被初始化为 0pt, 每遇到新的点时，这两个寄存器的值通常会被更新。但是这两个寄存器可能只是临时地保存当前点的坐标，也可能不保存当前点的坐标。这两个寄存器的值可能频繁变化，其主要用处是用来做计算。修改这两个寄存器值的命令主要是：

\tikz@make@last@position{⟨*last point*⟩}

此命令的定义是：

```
% Make given point the last position visited
\def\tikz@make@last@position#1{%
  \pgf@process{#1}%
  \tikz@lastx=\pgf@x\relax%
  \tikz@lasty=\pgf@y\relax%
  \iftikz@updatecurrent%
    \tikz@lastxsaved=\pgf@x\relax%
    \tikz@lastysaved=\pgf@y\relax%
  \fi%
```

```
  \tikz@updatecurrenttrue%
}%
```

本命令

1. 先用 \pgf@process →P. 250 处理点 ⟨*last point*⟩，使得这个点的坐标值，也就是寄存器 \pgf@x, \pgf@y 的值被全局化。
2. 为寄存器 \tikz@lastx, \tikz@lasty 局部地赋值——当前点的坐标
3. 若 \iftikz@updatecurrent 的判断为真，则为寄存器 \tikz@lastxsaved, \tikz@lastysaved 局部地赋值，这两个寄存器也用作某个点——姑且称之为 saved 点——的坐标。
4. 设置真值 \tikz@updatecurrenttrue, 这个真值是一般设置，所以当前点一般与 saved 点具有相同的坐标，也就是说，寄存器 \tikz@lastxsaved, \tikz@lastysaved 与寄存器 \tikz@lastx, \tikz@lasty 一般会被同时更新，其值分别相等。

\tikz@last@position

此命令的定义是：

```
\def\tikz@last@position{\pgfqpoint{\tikz@lastx}{\tikz@lasty}}%
```

40.7.2 第二种当前点

TikZ 的相对坐标的参照点也有“当前性”。相对坐标的参照点由 \tikz@last@position@saved 保存，其坐标分量由寄存器 \tikz@lastxsaved, \tikz@lastysaved 临时、局部地保存，所以相对坐标的参照点就是前文说的 saved 点。

\tikz@last@position@saved

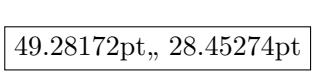
此命令的定义是：

```
\def\tikz@last@position@saved{\pgfqpoint{\tikz@lastxsaved}{\tikz@lastysaved}}%
```

当 \tikz@scan@one@point 解析 +(1,0), ++(0,1) 之类的相对坐标时，会调用命令 \tikz@scan@relative 来解析。对 +(1,0) 与 ++(0,1) 的处理是类似地，区别在于：+(1,0) 会导致 \tikz@updatecurrentfalse, 这个真值又会导致 saved 点保持不变。

40.7.3 第三种当前点

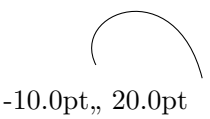
前端的 TikZ 路径构造命令会转变为 PGF 基本层的路径构造命令，再进一步转变为 PGF 系统层的软路径构造命令，所以，当说到“路径的当前点”时，应该指的是软路径的当前点，而不是对应寄存器 \pgf@x, \pgf@y 的点。例如：




```
\begin{tikzpicture}
    \path (30:2);
    \pgfgetlastxy{\macrox}{\macroy}
    \node [draw](a) at(0,0) {\macrox,, \macroy};% 显示 (30:2) 的坐标
    \path (0,0)--(a.60);
    \pgfgetlastxy{\macrox}{\macroy}
    \node (b) at(0,1) {\macrox,, \macroy};% 显示 (a.60) 的坐标
\end{tikzpicture}
```

参考 \pgfgetlastxy →P. 260.

而 curve-to 操作，即“..controls ⟨*A*⟩ and ⟨*B*⟩..⟨*C*⟩”对点的处理相对地复杂一些：



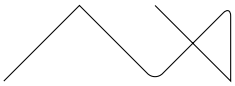

```
\begin{tikzpicture}
    \draw (0,0)..controls (-10pt,20pt) and (30pt,35pt)..(40pt,-5pt);
    \pgfgetlastxy{\macrox}{\macroy}
    \node (a) at(0,-0.5) {\macrox,, \macroy};% 显示第一支持点的坐标
\end{tikzpicture}
```

第四十一章　设置路径的语句

\path⟨*specification*⟩

这个命令只能用在 {tikzpicyure} 环境里。⟨*specification*⟩ 是一些列路径操作 (path operations，例如“--(0,0)”)，用于确定路径是如何构成的。在任何路径操作之后都可以给出图形选项 (graphic options)，即写在方括号里的选项。选项的作用的情况各有不同：

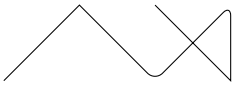
1. 有的选项有这样的特点：当遇到该选项时它会立即起效，并且只对它之后的那一部分路径有作用，对它之前的那一部分路径没有作用。例如 rounded corners，sharp corners 是这样的选项。



```
\tikz \draw (0,0) -- (1,1)
  [rounded corners] -- (2,0) -- (3,1)
  [sharp corners] -- (3,0) -- (2,1);
```

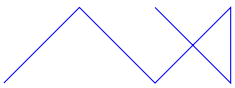
变换选项也属于这种类型。

2. 前述的那些能够立即起效的选项可以被“scoped”，



```
\tikz \draw (0,0) -- (1,1)
  {[rounded corners] -- (2,0) -- (3,1)}
   -- (3,0) -- (2,1);
```

3. 有的选项，无论把它放在路径的哪个位置，总是对整个路径起作用。例如颜色选项 color= 就是这样的，如果给一个路径使用两个 color=，那么后给出的颜色选项有效。多数选项属于这种类型。



```
\tikz \draw (0,0) -- (1,1)
  [color=red] -- (2,0) -- (3,1)
  [color=blue] -- (3,0) -- (2,1);
```

通常，命令 \path 只是创建路径，不画出路径，其作用可能仅仅是使得图形的尺寸变大。如果要让路径具有其它特征，例如线条颜色、填充色、线宽等，就得使用相应的选项。可以给路径添加 node，但 node 不属于路径本身。只有路径操作构建的点才属于路径，其它 (例如颜色、线宽) 不属于路径。

/tikz/name=⟨*path name*⟩　(no default)

给路径命名，然后可以用名称 ⟨*path name*⟩ 引用该路径，例如可以在创建动画 (animations) 时引用。在计算路径交点时用到的选项 /tikz/name path →P. 693 与这个选项并不相同。名称是“high-level”的，驱动并不识别这个名称，所以在 ⟨*path name*⟩ 中可以使用空格、数字、字母或其它符号，但不能使用逗号、点号、冒号等标点符号。

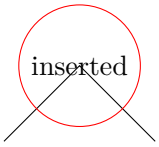
下面的 style 可以影响整个 scope：

/tikz/every path　(style, initially empty)

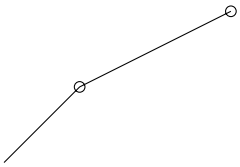
这个 style 会被添加到每个路径的开始处。

/tikz/insert path=⟨*path*⟩ (no default)

这里的 ⟨*path*⟩ 指的不是路径命令 \path，而是那些可以用在“\path”之后的、构成 (描绘) 路径的东西，例如“[fill=red]”，“--(1,1)”，“node[draw]{}”，“{[red]...}”等。这个选项会把 ⟨*path*⟩ 插入到当前位置。




```
\tikz \draw (0,0) --(1,1)
  [insert path={node[draw=red,circle]{inserted}}]
  --(2,0);
```

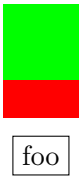


```
\tikz [c/.style={insert path={circle[radius=2pt]}}]
  \draw (0,0) -- (1,1) [c] -- (3,2) [c];
```

但是这个选项不能用作 node 的选项。

/tikz/append after command=⟨*path code*⟩ (no default)

这里的 ⟨*path code*⟩ 也是那些可以用在“\path”之后的、构成路径的东西。当某个路径带有这个选项后，在该路径的结尾处插入 ⟨*path code*⟩。这个选项可以用作 node 的选项。如果多次使用这个选项，那么它们提供的 ⟨*path code*⟩ 会依次被执行。




```
\tikz \node (foo) [draw,
  append after command={node [fill=red,minimum size=1cm]at(0,1){}},
  append after command={node [fill=green,minimum size=1cm]at(0,1.5){}}
  ]{foo};
```

上面例子中，“添加红色 node 的选项”位于“添加绿色 node 的选项”之前，所以先画出红色 node，再画出绿色 node。两个 node 有重叠部分，所以绿色 node 遮挡了红色 node。

此选项的定义是：

```
\tikzset{
  append after command/.code=\expandafter\def\expandafter\tikz@after@path
  ↪  \expandafter{\tikz@after@path#1},
}%
\let\tikz@after@path\pgfutil@empty
```

可见本选项提供的 ⟨*path code*⟩ 会被依次保存到宏 \tikz@after@path 中。

/tikz/prefix after command=⟨*path code*⟩ (no default)

本选项的定义是：

```
\tikzset{
  prefix after command/.code={%
    \def\tikz@temp{#1}%
    \expandafter\expandafter\expandafter\def%
    \expandafter\expandafter\expandafter\tikz@after@path%
    \expandafter\expandafter\expandafter{%
      \expandafter\tikz@temp\tikz@after@path}%
  },
}%
\let\tikz@after@path\pgfutil@empty
```

可见本选项重定义宏 \tikz@after@path. 假如使用选项

```
prefix after command=⟨path p1⟩,
prefix after command=⟨path p2⟩,
prefix after command=⟨path p3⟩,
append after command=⟨path a1⟩,
append after command=⟨path a2⟩,
append after command=⟨path a3⟩,
```

那么保存在宏 `\tikz@after@path` 中的内容就是

⟨*path p3*⟩⟨*path p2*⟩⟨*path p1*⟩⟨*path a1*⟩⟨*path a2*⟩⟨*path a3*⟩

也就是说，如果多次使用这个选项，那么它们的执行次序是：后给出选项先执行，先给出的选项后执行。

```
\tikz \node (foo) [draw,
  prefix after command={node [fill=red,minimum size=1cm]at(0,1){}},
  prefix after command={node [fill=green,minimum size=1cm]at(0,1.5){}}
  ]{foo};
```

foo

上面例子中，先给出的红色 node 遮挡了后给出的绿色 node。

41.1 Move-To 操作

`\path`...⟨*coordinate*⟩...

move-to 操作 (operation) 通常出现在两种地方，例如


```
\begin{tikzpicture}
  \draw (0,0) --(2,0) (0,1) --(2,1);
\end{tikzpicture}
```

其中在 (0,0) 和 (0,1) 这两个坐标的前面有 move-to 操作。在 (0,0) 前面的 move-to 操作处于路径的开端，表示路径的开始；在 (0,1) 前面的 move-to 操作使得路径产生“跳跃”，这种“跳跃”把路径截断成两部分 (两个“子路径”) 或者说两个片段 (segment)，使得路径不再连续。注意这种截断并不是“视觉上的截断”，下面例子

```
\begin{tikzpicture}
  \draw [line width=10pt] (0,0) --(2,0) (2,0) --(2,1);
\end{tikzpicture}
```

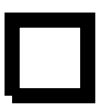
上图在视觉上好像是连续路径，但它并不是连续的 (拐角处有缺口)，因为在 (2,0) (2,0) 中间有一个 move-to 操作，将路径截成两段。


```
\tikz \draw (0,0)--(1,0) (1,1)--(2,1) (3,0);
```

上面图形中的路径中有三部分：线段、线段、孤立点，其中点 (0,0)，(1,1) 是 move-to 操作的“落脚点”。move-to 操作在点 (0,0) 开启路径，在点 (1,1) 处开启一个子路径。而点 (3,0) 是个孤立点，包含在当前路径的边界盒子中。

预定义的坐标点 (current subpath start)

这是个预定义的 node, 其形状是 `coordinate`, 当在路径中写出这个坐标名称后，这个名称代表的是它之前的、最近出现的 move-to 操作的“落脚点”。



```
\tikz [line width=2mm]
  \draw (0,0) -- (1,0) -- (1,1)
    -- (0,1) -- (current subpath start);
```

见文件《pgfmoduleshapes.code.tex》。

41.1.1 Move-To 操作的定义

move-to 操作的定义是：

```
\newif\iftikz@shapeborder


% Syntax for moveto:
% <point>
\def\tikz@movetoabs{\tikz@moveto(}%
\def\tikz@movetorel{\tikz@moveto+}%
\def\tikz@moveto{%
  \tikz@scan@one@point{\tikz@@moveto}}%
\def\tikz@@moveto#1{%
    \tikz@make@last@position{#1}%
  \iftikz@shapeborder%
    % ok, the moveto will have to wait. flag that we have a moveto in
    % waiting:
    \edef\tikz@moveto@waiting{\tikz@shapeborder@name}%
  \else%
    \tikz@@movetosave{\tikz@last@position}%
    \let\tikz@moveto@waiting=\relax%
  \fi%
  \tikz@scan@next@command%
}%
```

对于 `\tikz@moveto`⟨*point*⟩ 来说，命令 `\tikz@scan@one@point` 解析 ⟨*point*⟩ 的结果 ⟨*parsed point*⟩ 作为 `\tikz@@moveto` 的参数。如果 ⟨*point*⟩ 是 node 名称，那么 `\iftikz@shapeborder` 的真值就是 true, 否则真值是 false.

`\tikz@@moveto`{⟨*parsed point*⟩}

此命令的定义如上。命令 `\tikz@make@last@position` 使得 ⟨*parsed point*⟩ 成为当前点。此命令会重定义 `\tikz@moveto@waiting`.

```

  % Wrapper around \pgfpathmoveto that adds a save
\def\tikz@@movetosave#1{%
  {\pgftransformreset
    \pgf@process{#1}%
    \xdef\tikz@marshal{%
      \tikz@lastmovetox=\the\pgf@x\relax%
      \tikz@lastmovetoy=\the\pgf@y\relax%
    }%
  }%
  \tikz@marshal
  \pgfpathmoveto{#1}%
}%
```

\tikz@@movetosave{⟨*parsed point*⟩}

此命令的定义如上。参数 ⟨*parsed point*⟩ 通常是 \tikz@scan@one@point 的解析结果，此命令将点 ⟨*parsed point*⟩ 的坐标分量 (局部地) 保存到寄存器 \tikz@lastmovetox, \tikz@lastmovetoy, 并执行

\pgfpathmoveto{⟨*parsed point*⟩}

```
\let\tikz@moveto@waiting=\relax % normally, nothing is waiting...
```

规定 \tikz@moveto@waiting 的初始值。

```
\def\tikz@flush@moveto{%
  \ifx\tikz@moveto@waiting\relax%
  \else%
    \tikz@@movetosave{\tikz@last@position}%
  \fi%
  \let\tikz@moveto@waiting=\relax%
}%
```

\tikz@flush@moveto

此命令的定义如上，它常用在某些构造路径的操作的开头，例如在 rectangle, circle 操作的开头。

```
\def\tikz@flush@moveto@toward#1#2#3{%
  % #1 = a point towards which the last moveto should be corrected
  % #2 = a dimension to which the corrected x-coordinate should be stored
  % #3 = a dimension for the corrected y-coordinate
  \ifx\tikz@moveto@waiting\relax%
    % do nothing
  \else%
    \pgf@process{\pgfpointshapeborder{\tikz@moveto@waiting}{#1}}%
    #2=\pgf@x%
    #3=\pgf@y%
    \edef\tikz@timer@start{\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}%
    \tikz@@movetosave{\pgfqpoint{\pgf@x}{\pgf@y}}%
  \fi%
  \let\tikz@moveto@waiting=\relax%
}%
```

\tikz@flush@moveto@toward{⟨*a parsed point*⟩}{⟨*register a*⟩}{⟨*register b*⟩}

此命令的定义如上。参数 ⟨*a parsed point*⟩ 通常是 \tikz@scan@one@point 的解析结果，或者是一个基本层的坐标。参数 ⟨*register a*⟩, ⟨*register b*⟩ 是两个尺寸寄存器。

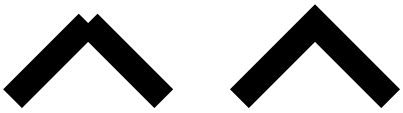
此命令一般与 \tikz@moveto@waiting 配合使用。

41.2 Line-To 操作

41.2.1 线段

\path...--⟨*coordinate or cycle*⟩...;

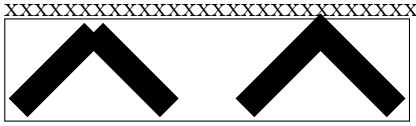
两个连字符号“--”代表 line-to 操作，它以当前点为起点创建一段线段来延伸当前路径。当前点就是符号“--”前面的点。线段终点是 ⟨*coordinate*⟩ 或者是由“闭合操作”cycle 确定的点。line-to 操作是一种“连续地”延伸当前路径的方式，move-to 操作则是“不连续地”延伸当前路径的方式，当线条的线宽较大时可以明显地看出两种操作之间的区别，如下



```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) --(1,1) (1,1) --(2,0);
  \draw (3,0) -- (4,1) -- (5,0);
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

上面例子中的 move-to 操作使得线条交角处有缺口，而 line-to 操作则不产生这种缺口。如果把上例中的命令 \useasboundingbox 去掉就是下面的效果：

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx



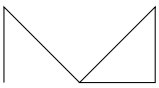
```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx\\
\begin{tikzpicture}
  \scoped[line width=10pt]{
    \draw (0,0) --(1,1) (1,1) --(2,0);
    \draw (3,0) -- (4,1) -- (5,0);}
  \draw (current bounding box.south west) rectangle (current bounding box.north east);
\end{tikzpicture}
```

如果一个封闭曲线 (多边形) 的起止点相同，为了不在该点出现缺口，使用“闭合操作”cycle，它使得曲线 (多边形) 变成闭合的。例如

```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) -- (1,0) -- (0,0)
        (2,0) -- (3,1) -- (3,0) -- (2,0);
  \draw (5,0) -- (6,1) -- (6,0) -- cycle
        (7,0) -- (8,1) -- (8,0) -- cycle;
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

上面例子看出，“闭合操作”cycle 的作用范围受到 move-to 操作的限制。如果使用 move-to 操作把路径截断为数段子路径，那么闭合操作 cycle 只对当前的连续子路径有效，它实际上返回点 (current subpath start)。闭合操作 cycle 不仅把当前位置返回到当前连续子路径的起点，还防止线条交角处出现“缺口”。

```
\begin{tikzpicture}
  \draw (0,0)--(1,0)--(1,1)--cycle--(0,1);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw (0,0)--(1,0)--(1,1)--cycle--+(-1,1)--(-1,0);
\end{tikzpicture}
```

闭合操作 cycle 可以用于“--”，“..”，“sin”，“grid”之后，但不能用于 graph 或 plot 之后。注意，对于 --(0,0) 来说，可以把 -- 看作是命令，把 (0,0) 看作是 -- 的参数，不能在 -- 与 (0,0) 之间随意添加代码。

41.2.1.1 line-to 的定义

当 \path 命令遇到“--”中的第一个“-”时，就执行 \tikz@lineto, 其定义是：

```
% Syntax for lineto:
% -- <point>

\def\tikz@lineto{%
  \pgfutil@ifnextchar |%
  {\expandafter\tikz@hv@lineto\pgfutil@gobble}%
  {\expandafter\pgfutil@ifnextchar\tikz@activebar{\expandafter\tikz@hv@lineto\
      pgfutil@gobble}%
    {\expandafter\tikz@lineto@mid\pgfutil@gobble}}}%
\def\tikz@lineto@mid{%
  \pgfutil@ifnextchar n{\tikz@collect@label@onpath\tikz@lineto@mid}%
  {%
    \pgfutil@ifnextchar c{\tikz@close}{%
      \pgfutil@ifnextchar p{\tikz@lineto@plot@or@pic}{\tikz@scan@one@point{\tikz@@lineto
          }}}}}%
\def\tikz@lineto@plot@or@pic p{%
  \pgfutil@ifnextchar i{\tikz@collect@pic@onpath\tikz@lineto@mid p}{%
    \pgfsetlinetofirstplotpoint\tikz@plot}%
}%
```

可见 \tikz@lineto 会识别“-|”，“--node”，“--cycle”，“--plot”，“--pic”这些符号组合。在“--”中的第二个“-”，以及“-|”中的“|”会被 \pgfutil@gobble 吃掉。

对于“--⟨*TikZ point*⟩”这个情况，坐标 ⟨*TikZ point*⟩ 会被命令 \tikz@scan@one@point 解析，解析结果作为命令 \tikz@@lineto 的参数。

```
\def\tikz@@lineto#1{%
  % Record the starting point for later labels on the path:
  \edef\tikz@timer@start{\noexpand\pgfqpoint{\the\tikz@lastx}{\the\tikz@lasty}}
  \iftikz@shapeborder%
    % ok, target is a shape. recalculate end
    \pgf@process{\pgfpointshapeborder{\tikz@shapeborder@name}{\tikz@last@position}}%
    \tikz@make@last@position{\pgfqpoint{\pgf@x}{\pgf@y}}%
    \tikz@flush@moveto@toward{\tikz@last@position}\pgf@x\pgf@y%
    \tikz@path@lineto{\tikz@last@position}%
```

```
    \edef\tikz@timer@end{\noexpand\pgfqpoint{\the\tikz@lastx}{\the\tikz@lasty}}%
    \tikz@make@last@position{#1}%
    \edef\tikz@moveto@waiting{\tikz@shapeborder@name}%
  \else%
    % target is a reasonable point...
    % Record the starting point for later labels on the path:
    \tikz@make@last@position{#1}%
    \tikz@flush@moveto@toward{\tikz@last@position}\pgf@x\pgf@y%
    \tikz@path@lineto{\tikz@last@position}%
    \edef\tikz@timer@end{\noexpand\pgfqpoint{\the\tikz@lastx}{\the\tikz@lasty}}%
  \fi%
  \let\tikz@timer=\tikz@timer@line%
  \let\tikz@tangent\tikz@timer@start%
  \tikz@scan@next@command%
}%
```

`\tikz@@lineto`{⟨*parsed point B*⟩}

此命令的定义如上。对于“⟨*point A*⟩--⟨*point B*⟩”这个情况，假设坐标 ⟨*point A*⟩, ⟨*point B*⟩ 分别被解析为 ⟨*parsed point A*⟩, ⟨*parsed point B*⟩, 此命令的处理是：

1. 定义宏 `\tikz@timer@start`, 保存 ⟨*parsed point A*⟩.
2. 如果 ⟨*point B*⟩ 是个 node 名称，那么此时 `\iftikz@shapeborder` 的真值是 true.
   (a) 此时 `\tikz@last@position` 保存的是 ⟨*parsed point A*⟩, 执行

   ```
   \pgfpointshapeborder{⟨point B⟩}{⟨parsed point A⟩}
   ```

   得到 ⟨*point B*⟩ 的边界上的一个点，记为 ⟨*point B b*⟩.
   (b) 执行 `\tikz@make@last@position` 将 ⟨*point B b*⟩ 保存到 `\tikz@last@position`.
   (c) 执行 `\tikz@flush@moveto@toward`→P. 722, 如果 ⟨*point A*⟩ 是个 node 名称，那么这个命令将得到 ⟨*point A*⟩ 的边界上的一个点，如果 ⟨*point A*⟩ 不是个 node 名称，那么这个命令什么也不做。
   (d) 执行 `\tikz@path@lineto`, 也就是 `\pgfpathlineto`→P. 277{⟨*point B b*⟩}.
   (e) 定义宏 `\tikz@timer@end`, 保存 ⟨*point B b*⟩.
   (f) 执行 `\tikz@make@last@position` 将 ⟨*parsed point B*⟩ 保存到 `\tikz@last@position`.
   (g) 将 ⟨*point B*⟩ 的全名保存到 `\tikz@moveto@waiting`.
3. 如果 ⟨*point B*⟩ 不是个 node 名称，那么此时 `\iftikz@shapeborder` 的真值是 false.
   (a) 执行 `\tikz@make@last@position` 将 ⟨*parsed point B*⟩ 保存到 `\tikz@last@position`.
   (b) 执行 `\tikz@flush@moveto@toward`→P. 722, 如果 ⟨*point A*⟩ 是个 node 名称，那么这个命令将得到 ⟨*point A*⟩ 的边界上的一个点，如果 ⟨*point A*⟩ 不是个 node 名称，那么这个命令什么也不做。
   (c) 执行 `\tikz@path@lineto`, 也就是 `\pgfpathlineto`→P. 277{⟨*parsed point B*⟩}.
   (d) 定义宏 `\tikz@timer@end`, 保存 ⟨*parsed point B*⟩.
4. `\tikz@timer@line` 的定义是：

```
\def\tikz@timer@line{%
  \pgftransformlineattime{\tikz@time}{\tikz@timer@start}{\tikz@timer@end}%
}%
```

5. 执行 \tikz@scan@next@command, 解析下一个命令。

```
% snake or lineto?
\def\tikz@path@lineto#1{%
  \iftikz@snaked%
    {
      \pgfsyssoftpathmovetorelevantfalse%
      \pgfpathsnakesto{\tikz@presnake,{\tikz@snake}{\tikz@mainsnakelength}{\noexpand\
          tikz@snake@install@trans}{},\tikz@postsnake}{#1}%
    }
  \else%
    \pgfpathlineto{#1}%
  \fi%
}%
```

\tikz@path@lineto{⟨*a parsed point*⟩}

此命令的定义如上。参数 ⟨*a parsed point*⟩ 通常是 \tikz@scan@one@point 的解析结果，或者一个基本层坐标。多数情况下，此命令等效于

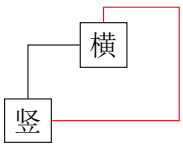
```
\pgfpathlineto{{⟨a parsed point⟩}}
```

41.2.2 横线和竖线

\path...-|⟨*coordinate or cycle*⟩...;

\path...|-⟨*coordinate or cycle*⟩...;

“-|”或者“|-”用来创建水平线和竖直线。要想用水平线和竖直线把两个点连接起来，就可以在这两个点之间使用“-|”或者“|-”操作。




```
\begin{tikzpicture}
  \draw (0,0) node(竖) [draw] {竖} (1,1) node(横) [draw] {横};
  \draw (竖) |- (横);
  \draw[color=red] (竖) -| (2,1.5) -| (横);
\end{tikzpicture}
```

```
\begin{tikzpicture}[ultra thick]
  \draw (0,0) -- (1,1) -| cycle;
\end{tikzpicture}
```

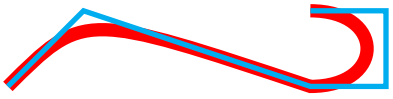
41.3 Curve-To 操作

curve-to 操作创建 Bézier 曲线。

\path⟨···⟩..controls⟨*c*⟩and⟨*d*⟩..⟨*y or cycle*⟩⟨···⟩;

这个操作以当前点为起点创建一段 3 次 Bézier 曲线来延伸当前路径。假设当前点是 ⟨*x*⟩，则 ⟨*x*⟩ 是起点，⟨*c*⟩ 是第二点，⟨*d*⟩ 是第三点，⟨*y*⟩ 是终点。

句法中的 and⟨*d*⟩ 是可选的，如果没有这一部分，就默认 ⟨*c*⟩=⟨*d*⟩。



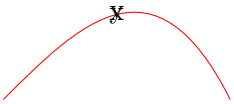
```
\begin{tikzpicture}
  \draw[color=red,line width=5pt] (0,0) .. controls (1,1) .. (4,0)
                                  .. controls (5,0) and (5,1) .. (4,1);
  \draw[color=cyan,line width=2pt] (0,0) -- (1,1) -- (4,0) -- (5,0) -- (5,1) -- (4,1);
\end{tikzpicture}
```

```
\begin{tikzpicture}
  \draw[line width=10pt] (0,0) -- (2,0) .. controls (1,1) .. cycle;
\end{tikzpicture}
```

当线段与控制曲线有公共点时，在公共点处也有是否连续的问题，如果二者是 move-to 连接方式，连接部分可能有缺口。

```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) (1,1) .. controls (1,0) and (2,0) .. (2,0);
  \draw [yshift=-1.5cm]
        (0,0) -- (1,1)       .. controls (1,0) and (2,0) .. (2,0);
\end{tikzpicture}
```

当需要给控制曲线加 node 或 pic 时，可以加在“..”的后面，例如：




```
\begin{tikzpicture}
  \draw[draw=red]
        (0,0) .. node {x} controls (1,1) and (2,2).. node {y} (3,0);
\end{tikzpicture}
```

41.4 矩形操作

\path...rectangle⟨*corner or cycle*⟩...;

这个操作以当前点和 ⟨*corner or cycle*⟩ 所指定的点为对角线端点来创建一个矩形，矩形的四边是横平竖直的。

rectangle 的定义是：

```
% Syntax for rectangles:
% rectangle <corner point>
\def\tikz@rect ectangle{%
  \tikz@flush@moveto%
  \edef\tikz@timer@start{\noexpand\pgfqpoint{\the\tikz@lastx}{\the\tikz@lasty}}%
  \tikz@@rect}%
```

对于 ⟨*form A*⟩ rectangle ⟨*form B*⟩ 来说，假设解析 ⟨*form A*⟩ 与 ⟨*form B*⟩ 的结果分别是 ⟨*form A'*⟩ 与 ⟨*form B'*⟩, 宏 \tikz@timer@start 保存的是 ⟨*form A'*⟩.

```
\def\tikz@@rect{%
  \pgfutil@ifnextchar n{\tikz@collect@label@onpath\tikz@@rect}{%
  \pgfutil@ifnextchar p{\tikz@collect@pic@onpath\tikz@@rect}%
  {\pgfutil@ifnextchar c{\tikz@collect@coordinate@onpath\tikz@@rect}%
```

```
    {
      \pgf@xa=\tikz@lastx\relax%
      \pgf@ya=\tikz@lasty\relax%
      \tikz@scan@one@point\tikz@rectB}}}}%
```

命令 \tikz@@rect 会识别“rectangle node”，“rectangle pic”，“rectangle coordinate”这些句子，如果不是这些句子，会把 ⟨*form A'*⟩ 的坐标临时保存到寄存器 \pgf@xa, \pgf@ya, 然后解析 ⟨*form B*⟩, 解析结果 ⟨*form B'*⟩ 作为 \tikz@rectB 的参数。

```
\def\tikz@rectB#1{%
  \tikz@make@last@position{#1}%
  \edef\tikz@timer@end{\noexpand\pgfqpoint{\the\tikz@lastx}{\the\tikz@lasty}}%
  \let\tikz@timer=\tikz@timer@line%
        \tikz@@movetosave{\pgfqpoint{\pgf@xa}{\pgf@ya}}%
  \tikz@path@lineto{\pgfqpoint{\pgf@xa}{\tikz@lasty}}%
  \tikz@path@lineto{\pgfqpoint{\tikz@lastx}{\tikz@lasty}}%
  \tikz@path@lineto{\pgfqpoint{\tikz@lastx}{\pgf@ya}}%
  \iftikz@snaked%
    \tikz@path@lineto{\pgfqpoint{\pgf@xa}{\pgf@ya}}%
  \fi%
  \pgfpathclose%
        \tikz@@movetosave{\pgfqpoint{\tikz@lastx}{\tikz@lasty}}%
  \def\pgfstrokehook{}%
  \let\tikz@tangent\relax%
  \tikz@scan@next@command%
}%
```

执行 \tikz@rectB{⟨*form B'*⟩} 会：

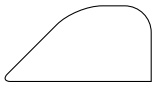
- 定义宏 \tikz@timer@end 保存 ⟨*form B'*⟩
- 先 move-to 到点 ⟨*form A'*⟩, 然后用 line-to 操作按顺时针方向画 4 个线段，然后执行 \pgfpathclose 闭合 4 个线段，再 move-to 到点 ⟨*form B'*⟩. 注意，这与 \pgfpathrectangle →P. 287 不一样。

41.5 Rounding Corners

前述几种创建路径的操作都受到选项 rounded corners 的影响。

/tikz/rounded corners=⟨*inset*⟩ (default 4pt)

这个选项是可以被 scoped 的，并且它只对它后面的、由前述几种操作创建的那一部分路径有效果。它把连续路径上的尖角改为圆弧。⟨*inset*⟩ 是带长度单位的尺寸，用于指定圆弧的半径。注意 ⟨*inset*⟩ 不会受到变换选项 scale= 的影响。这个选项应该放在坐标点之后、创建路径的操作符号之前。



```
\begin{tikzpicture}
\draw (0,0) [rounded corners=10pt] -- (1,1) -- (2,1)
    [sharp corners] -- (2,0)
    [rounded corners] -- cycle;
\end{tikzpicture}
```

```
\tikz \draw[rounded corners=1ex] (0,0) rectangle (20pt,2ex);
```

当路径上的尖角是 90° 角时，本选项创建的圆弧角才是“圆弧”。如果路径上的尖角是由很短的线段构成的，那么使用本选项的效果可能不好，此时使用尖角 (sharp corners) 会比较好。

/tikz/sharp corners (no value)

本选项关闭圆角功能，使用尖角。

41.6 创建圆、椭圆

\path...circle[⟨options⟩]...;

这个命令在路径上添加一个圆或者椭圆，圆的圆心是当前点，或者可以在 ⟨options⟩ 中使用选项 at=⟨coordinate⟩ 来指定圆心位置。在创建圆后，当前点返回到圆的圆心。本命令调用 \pgfpathellipse → P. 286 来工作。针对这个操作的选项如下。

/tikz/x radius=⟨value⟩ (no default)

本选项设置圆的 horizontal radius，键值 ⟨value⟩ 可以是带长度单位的尺寸，也可以是纯数值。如果 ⟨value⟩ 是纯数值，则该值会被解释到 xy 坐标系统中。参数 ⟨value⟩ 会被 \pgfmathparse → P. 110 解析。

/tikz/y radius=⟨value⟩ (no default)

类似 x radius。

/tikz/radius=⟨value⟩ (no default)

将 x radius 和 y radius 都设为 ⟨value⟩。

以上选项的定义是：

```
% Radii and arc options
\tikzset{x radius/.initial=0pt}%
\tikzset{y radius/.initial=0pt}%
\tikzset{%
  radius/.code={%
    \pgfmathparse{#1}%
    \ifpgfmathunitsdeclared
      \pgfkeyssetevalue{/tikz/x radius}{\pgfmathresult pt}%
      \pgfkeyssetevalue{/tikz/y radius}{\pgfmathresult pt}%
    \else
      \pgfkeyssetevalue{/tikz/x radius}{\pgfmathresult}%
      \pgfkeyssetevalue{/tikz/y radius}{\pgfmathresult}%
    \fi
  }%
}%
```

/tikz/at=⟨coordinate⟩ (no default)

本选项可以设置圆的圆心位置。

```
\tikzoption{at}{\tikz@scan@one@point\tikz@set@at#1}%
\def\tikz@set@at#1{\def\tikz@node@at{#1}}%
```

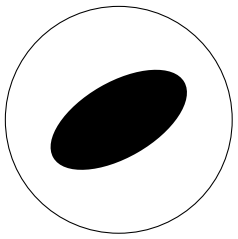
/tikz/every circle (style, no value)

在这个 style 的有效范围内，它针对每个圆。

circle 操作对选项 ⟨options⟩ 的处理是

```
\tikzset{every circle/.try,⟨options⟩}%
```

所以在 ⟨*options*⟩ 中可以使用任何前缀为 /tikz/ 的选项，例如 rotate，scale 等选项，这些选项只对此圆有效。



```
\begin{tikzpicture}
  \draw (1,0) circle [radius=1.5];
  \fill (1,0) circle [x radius=1cm, y radius=5mm, rotate=30];
\end{tikzpicture}
```

如果你觉得选项 radius 或 x radius 写起来过长，不方便，可以自定义 key 来代替它们：

```
\tikzset{r/.style={radius=#1},rx/.style={x radius=#1},ry/.style={y radius=#1}}
```

这样定义后，就可以使用 circle [r=1cm] 或 circle [rx=1,ry=1.5] 这样的简写格式了。

\path...ellipse[⟨*options*⟩]...;

与 circle 操作类似。也有个较旧的句法：ellipse(⟨*x radius*⟩ and ⟨*y radius*⟩) 来指定椭圆的横半轴和纵半轴。

41.6.1 指定圆半径的旧句法

有一个较旧的句法来指定圆的半径,例如 circle(2pt)，circle(0.5)，circle(1 and 0.5)，circle(1cm and 0.5cm)(数字单位默认为 cm)，把半径直接写在圆括号里。但是不能写 circle(1 and 0.5cm) 这样一个数字不带长度单位、一个数字带长度单位的格式。

- 对于 circle(⟨*expression*⟩) 来说，参数 ⟨*expression*⟩ 会被 \pgfmathparse→P. 110 解析，解析结果保存在 \tikz@ellipse@x 中。如果 ⟨*expression*⟩ 中含有长度单位，则 \ifpgfmathunitsdeclared→P. 113 的真值会被设为 true.
  - 如果 ⟨*expression*⟩ 中含有长度单位，即有 \pgfmathunitsdeclaredtrue, 则执行

```
\pgfpathellipse{\pgfpointorigin}%
  {\pgfqpoint{\tikz@ellipse@x pt}{0pt}}%
  {\pgfpoint{0pt}{\tikz@ellipse@x pt}}%
```

  - 如果 ⟨*expression*⟩ 中不含有长度单位，即有 \pgfmathunitsdeclaredfalse, 则执行

```
\pgfpathellipse{\pgfpointorigin}%
  {\pgfpointxy{\tikz@ellipse@x}{0}}%
  {\pgfpointxy{0}{\tikz@ellipse@x}}%
```

- 对于 circle(⟨*expression 1*⟩ and ⟨*expression 2*⟩) 来说，参数 ⟨*expression 1*⟩ 和 ⟨*expression 2*⟩ 都会被 \pgfmathparse→P. 110 解析，解析结果保存在 \tikz@ellipse@x, \tikz@ellipse@y 中。
  - 如果 ⟨*expression 1*⟩ 和 ⟨*expression 2*⟩ 都含有长度单位，则执行

```
\pgfpathellipse{\pgfpointorigin}{%
  \pgfqpoint{\tikz@ellipse@x pt}{0pt}}{\pgfpoint{0pt}{\tikz@ellipse@y pt}}%
```

  - 如果 ⟨*expression 1*⟩ 和 ⟨*expression 2*⟩ 都不含有长度单位，则执行

```
\pgfpathellipse{\pgfpointorigin}{%
  \pgfpointxy{\tikz@ellipse@x}{0}}{\pgfpointxy{0}{\tikz@ellipse@y}}%
```

  - 如果 ⟨*expression 1*⟩ 和 ⟨*expression 2*⟩ 二者中的一个含有长度单位、另一个不含有长度单位，则执行

```
\tikzerror{You cannot mix dimensions and dimensionless values in an ellipse}%
```

41.6.2 circle 操作的定义

在文件《tikz.code.tex》中有定义：

```
% Syntax for circles:
% circle [options] % where options should set, at least, radius
% circle (radius) % deprecated
%
% Syntax for ellipses:
% ellipse [options] % identical to circle.
% ellipse (x-radius and y-radius) % deprecated
%
% radii can be dimensionless, then they are in the xy-system
\def\tikz@circle ircle{\tikz@flush@moveto\tikz@@circle}%
\def\tikz@ellipse llipse{\tikz@flush@moveto\tikz@@circle}%
\def\tikz@@circle{%
  \let\tikz@tangent\relax%
  \pgfutil@ifnextchar(\tikz@@@circle
  {\pgfutil@ifnextchar[\tikz@circle@opt{%])
    \advance\tikz@expandcount by -10\relax% go down quickly
    \ifnum\tikz@expandcount<0\relax%
      \let\pgfutil@next=\tikz@@circle@normal%
    \else%
      \let\pgfutil@next=\tikz@@circle@scanexpand%
    \fi%
    \pgfutil@next%
  }}%
}%
\def\tikz@@circle@scanexpand{\expandafter\tikz@@circle}%
\def\tikz@@circle@normal{\tikz@circle@opt[]}%
```

命令 \tikz@circle ircle 解析 circle 句子。

从 \tikz@@circle 的定义看，可以使用“circle ⟨*a token*⟩”这样的句子，其中的记号 ⟨*a token*⟩ 不是“(”也不是“[”，当遇到 ⟨*a token*⟩ 时，计数器 \tikz@expandcount 的值会被减去 10，此时，

- 如果这个计数器的值 < 0，则执行 \tikz@circle@opt[]，这相当于句子“circle[]”，


```
\tikz[every circle/.style={radius=5mm}]{
    \draw (0,0)circle node{A};
}
```

- 如果这个计数器的值 $\geqslant 0$，则把 ⟨*a token*⟩ 展开，再重复执行 \tikz@@circle

```

\def\tikz@circle@opt[#1]{%
  {%
    \def\tikz@node@at{\tikz@last@position}%
    \let\tikz@transform=\pgfutil@empty%
    \tikzset{every circle/.try,#1}%
```

```
    \pgftransformshift{\tikz@node@at}%
    \tikz@transform%
    \tikz@do@ellipse{\pgfkeysvalueof{/tikz/x radius}}{\pgfkeysvalueof{/tikz/y radius}}
  }%
  \tikz@scan@next@command%
}%

\def\tikz@@@circle(#1){%
  {%
    \pgftransformshift{\tikz@last@position}%
    \pgfutil@in@{ and }{#1}%
    \ifpgfutil@in@%
      \tikz@@ellipseB(#1)%
    \else%
      \tikz@do@circle{#1}%
    \fi%
  }%
  \tikz@scan@next@command%
}%
\def\tikz@@ellipseB(#1 and #2){%
  \tikz@do@ellipse{#1}{#2}%
}%
\def\tikz@do@circle#1{%
  \pgfmathparse{#1}%
  \let\tikz@ellipse@x=\pgfmathresult
  \ifpgfmathunitsdeclared
    \pgfpathellipse{\pgfpointorigin}%
      {\pgfqpoint{\tikz@ellipse@x pt}{0pt}}%
      {\pgfpoint{0pt}{\tikz@ellipse@x pt}}%
  \else
    \pgfpathellipse{\pgfpointorigin}%
      {\pgfpointxy{\tikz@ellipse@x}{0}}%
      {\pgfpointxy{0}{\tikz@ellipse@x}}%
  \fi
}
\def\tikz@do@ellipse#1#2{
  \pgfmathparse{#1}%
  \let\tikz@ellipse@x=\pgfmathresult%
  \ifpgfmathunitsdeclared%
    \pgfmathparse{#2}%
    \let\tikz@ellipse@y=\pgfmathresult%
    \ifpgfmathunitsdeclared%
      \pgfpathellipse{\pgfpointorigin}{%
```

```
      \pgfqpoint{\tikz@ellipse@x pt}{0pt}}{\pgfpoint{0pt}{\tikz@ellipse@y pt}}%
    \else%
      \tikzerror{You cannot mix dimensions and dimensionless values in an ellipse}%
    \fi%
  \else%
    \pgfmathparse{#2}%
    \let\tikz@ellipse@y=\pgfmathresult%
    \ifpgfmathunitsdeclared%
      \tikzerror{You cannot mix dimensions and dimensionless values in an ellipse}%
    \else%
      \pgfpathellipse{\pgfpointorigin}{%
        \pgfpointxy{\tikz@ellipse@x}{0}}{\pgfpointxy{0}{\tikz@ellipse@y}}%
    \fi%
  \fi%
}%
```

注释：

- 命令 `\tikz@circle@opt` 设置一个花括号组，在组内处理选项、执行 `\tikz@do@ellipse`。
- 对于 `circle[`⟨*options*⟩`]` 这种句子，在 ⟨*options*⟩ 中的选项 `x radius=`⟨*value x*⟩, `y radius=`⟨*value y*⟩, 其参数 ⟨*value x*⟩, ⟨*value y*⟩ 会被命令 `\pgfmathparse` 解析，解析结果分别保存在 `\tikz@ellipse@x`, `\tikz@ellipse@y` 中，参数 ⟨*value x*⟩ 与 ⟨*value y*⟩ 是否含有长度单位对处理过程有影响。

41.7 Arc 操作

`\path...arc[`⟨*options*⟩`]...;`

假设 ⟨x⟩ 是当前点，`arc` 操作以 ⟨x⟩ 为起点创建一段圆弧 (椭圆弧)。圆弧的形态可以这样设想：假设一个圆以坐标系原点 O 为圆心，以 r 为半径，在圆上有两个点 A，B；$\overrightarrow{OA}$ 的方向角是 θ_A，$\overrightarrow{OB}$ 的方向角是 θ_B；从点 A 开始沿着圆周向点 B 运动，运动的方向这样规定：若 $\theta_A < \theta_B$ 则逆时针运动，若 $\theta_A > \theta_B$ 则顺时针运动；这个运动过程所走过的圆弧记为 ⟨AB⟩，称 θ_A 为“起始角度”(start angle)，称 θ_B 为“终止角度”(end angle)，称 $\delta = \theta_B - \theta_A$ 为“角度差”(delta angle)；平移圆弧 ⟨AB⟩ 使得点 A 与当前点 ⟨x⟩ 重合，这样就把一段圆弧添加到了路径上，路径得以延伸。影响圆弧形态的参数有半径、起始角度、终止角度、角度差，而半径又包括横向半径和纵向半径。这些参数对应下面的选项。

`/tikz/x radius=`⟨*value*⟩ (no default)

见 `/tikz/x radius` → P. 729.

`/tikz/y radius=`⟨*value*⟩ (no default)

见 `/tikz/y radius` → P. 729.

`/tikz/start angle=`⟨*degrees*⟩ (no default)

设置起始角度。

`/tikz/end angle=`⟨*degrees*⟩ (no default)

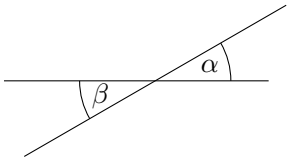
设置终止角度。

`/tikz/delta angle=`⟨*degrees*⟩ (no default)

设置角度差。

以上选项的定义是：

```
\tikzset{start angle/.initial=}%
\tikzset{end angle/.initial=}%
\tikzset{delta angle/.initial=}%
```




```
\begin{tikzpicture}[radius=1cm,delta angle=30]
  \draw (-1,0) -- +(3.5,0);
  \draw (1,0) ++(210:2cm) -- +(30:4cm);
  \draw (1,0) +(0:1cm) arc [start angle=0];
  \draw (1,0) +(180:1cm) arc [start angle=180];
  \path (1,0) ++(15:.75cm) node{$\alpha$};
  \path (1,0) ++(15:-.75cm) node{$\beta$};
\end{tikzpicture}
```

也有一个较简捷的句法来指定圆弧：

arc(⟨*start angle*⟩:⟨*end angle*⟩:⟨*radius*⟩)

或者

arc(⟨*start angle*⟩:⟨*end angle*⟩:⟨*x radius*⟩ and ⟨*y radius*⟩)

41.8 Grid 操作

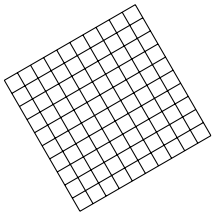
\path...grid[⟨*options*⟩]⟨*corner or cycle*⟩...;

假设 ⟨*x*⟩ 是当前点，grid 操作创建网格，网格以 ⟨*x*⟩ 和 ⟨*corner or cycle*⟩ 为对角线端点。注意，网格总是以原点为一个格点。在 ⟨*options*⟩ 中可以使用选项调整网格的步长。

/tikz/step=⟨*number or dimension or coordinate*⟩ (no default, initially 1cm)

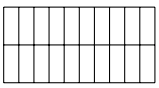
设置网格在 x 轴方向和 y 轴方向的步长。

- 如果本选项的值是带长度单位的尺寸，则在 canvas 坐标系统中解释这个尺寸，直接以这个尺寸为步长。
- 如果本选项的值是纯数值，这个数值会被解释到 xy 坐标系统中，再计算出步长。

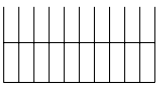


```
\tikz[rotate=30] \draw[step=0.2] (0,0) grid (2,2);
```

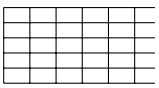
- 如果本选项的值是坐标 (a,b)，假设这个坐标被解析为基本层的坐标 (a',b')，则网格在 x 轴方向的步长是 a'，在 y 轴方向的步长是 b'，也就是说一个网格单元 (一个网眼) 的对角线向量就是 (a',b').



```
\tikz \draw[step={(0.2,0.5)}] (0,0) grid (2,1);
```



```
\tikz \draw[step={(0.2,15pt)}] (0,0) grid (2,1);
```



```
\tikz \draw[step={(30:0.4)}] (0,0) grid (2,1);
```

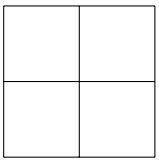
- 最终确定步长后 (最终的步长是尺寸)，仅当 x 轴或 y 轴的步长 > 0.01pt 时，才会在这个轴上画出网格。

```
\tikz \draw[step={(0,0.5)}] (0,0) grid (2,1);
```

上面例子中，x 轴上的网格步长是 0, 所以 x 轴上没有网格线。

```
\tikz \draw[x=-1cm] (0,0) grid[step=1] (2,2);
```

上面例子中，由于选项 x=-1cm 将 xyz 坐标系统的 x 轴的单位长度设为负值，使得 step=1 决定的 x 轴方向的步长是负值，所以没有 x 轴的网格线，对比下面的：



```
\tikz \draw[x=-1cm] (0,0) grid[xstep=-1] (2,2);
```

本选项的定义是：

```
\def\tikz@handle@vec#1#2{\pgfutil@ifnextchar({\tikz@handle@coordinate#1}{
↪  \tikz@handle@single#2}}%
\def\tikz@handle@coordinate#1{\tikz@scan@one@point#1}%
\def\tikz@handle@single#1#2\relax{#1{#2}}%
%......
% Grid options
\tikzoption{xstep}{\def\tikz@grid@x{#1}}%
\tikzoption{ystep}{\def\tikz@grid@y{#1}}%
\tikzoption{step}{\tikz@handle@vec{\tikz@step@point}{\tikz@step@single}#1\relax
↪  }%
\def\tikz@step@single#1{\def\tikz@grid@x{#1}\def\tikz@grid@y{#1}}%
\def\tikz@step@point#1{\pgf@process{#1}\edef\tikz@grid@x{\the\pgf@x}
↪  \edef\tikz@grid@y{\the\pgf@y}}%

\def\tikz@grid@x{1cm}%
\def\tikz@grid@y{1cm}%
```

可见对于 step=⟨*value*⟩ 来说，

- 如果 ⟨*value*⟩ 以开圆括号“(”开头，则导致

  ```
  \tikz@handle@coordinate\tikz@step@point⟨value⟩\relax
  导致
  \tikz@scan@one@point\tikz@step@point⟨value⟩\relax
  ```

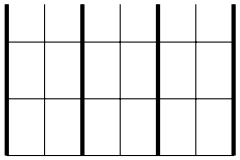
  此时 TikZ 会把 ⟨*value*⟩ 当作一个坐标并解析它，解析结果通常是一个基本层的点，这个点的坐标尺寸会被分别保存到宏 \tikz@grid@x, \tikz@grid@y 中。
- 如果 ⟨*value*⟩ 不以开圆括号“(”开头，则导致

  ```
  \tikz@handle@single\tikz@step@single{⟨value⟩}\relax
  导致
  \def\tikz@grid@x{⟨value⟩}\def\tikz@grid@y{⟨value⟩}
  ```

  此时直接把 ⟨*value*⟩ 保存到宏 \tikz@grid@x, \tikz@grid@y 中。
- 注意，如果 ⟨*value*⟩ 是包含逗号的坐标点，那么应该把坐标 ⟨*value*⟩ 用花括号括起来，因为逗号会被命令 \pgfkeys 作为前后两个选项的分界标志。

/tikz/xstep=⟨*dimension or number*⟩ (no default, initially 1cm)

设置网格在 x 轴方向的步长。



```
\begin{tikzpicture}
  \draw (0,0) grid [xstep=.5,ystep=.75] (3,2);
  \draw[ultra thick] (0,0) grid [ystep=0] (3,2);
\end{tikzpicture}
```

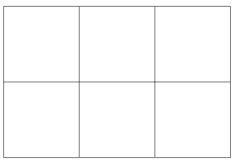
/tikz/ystep=⟨*dimension or number*⟩ (no default, initially 1cm)

设置网格在 y 轴方向的步长。

由于数据计算时有误差，网格最外层本该有的网格线可能被忽略，此时需要手工添加。

/tikz/help lines (style, initially line width=0.2pt,gray)

这个 style 通常用于画网格。



```
\tikz \draw[help lines] (0,0) grid (3,2);
```

41.8.1 Grid 操作的定义

Grid 操作的定义是：

```
% Syntax for grids:
% grid <corner point>
\def\tikz@grid id{%
  \tikz@flush@moveto%
  \pgf@xa=\tikz@lastx\relax%
  \pgf@ya=\tikz@lasty\relax%
  \pgfutil@ifnextchar[{\tikz@gridA}{\tikz@gridA[]}}%}%
\def\tikz@gridA[#1]{%
  \def\tikz@grid@options{#1}%
  \tikz@cycle@expander{\tikz@scan@one@point\tikz@gridB}}%
\def\tikz@gridB#1{%
  \tikz@make@last@position{#1}%
  \let\tikz@tangent\relax%
  {%
    \let\tikz@after@path\pgfutil@empty%
    \expandafter\tikzset\expandafter{\tikz@grid@options}
    \tikz@checkunit{\tikz@grid@x}%
    \iftikz@isdimension%
      \pgf@process{\pgfpoint{\tikz@grid@x}{0pt}}%
    \else%
      \pgf@process{\pgfpointxy{\tikz@grid@x}{0}}%
    \fi%
    \pgf@xb=\pgf@x%
    \pgf@yb=\pgf@y%
    \tikz@checkunit{\tikz@grid@y}%
```

```
    \iftikz@isdimension%
      \pgf@process{\pgfpoint{0pt}{\tikz@grid@y}}%
    \else%
      \pgf@process{\pgfpointxy{0}{\tikz@grid@y}}%
    \fi%
    \advance\pgf@xb by\pgf@x%
    \advance\pgf@yb by\pgf@y%
    \pgfpathgrid[stepx=\pgf@xb,stepy=\pgf@yb]%
      {\pgfqpoint{\pgf@xa}{\pgf@ya}}{\pgfqpoint{\tikz@lastx}{\tikz@lasty}}%
  \expandafter}%
  \expandafter\tikz@scan@next@command\tikz@after@path%
}%
```

对于 ⟨*point A*⟩ `grid` [⟨*options*⟩] ⟨*point B*⟩ 来说，假设坐标 ⟨*point A*⟩, ⟨*point B*⟩ 被解析为 ⟨*point A'*⟩, ⟨*point B'*⟩,

1. 如果 ⟨*point A*⟩ 不是 node 名称，则会 move-to 到 ⟨*point A'*⟩.
2. ⟨*point B'*⟩ 用作 `\tikz@gridB` 的参数。命令 `\tikz@gridB` 会设置一个花括号组，在组内执行

   ```
   \tikzset{⟨options⟩}
   ```

   在 ⟨*options*⟩ 中可以使用任何前缀为 `/tikz/` 的选项。命令 `\tikz@gridB` 也在这个组内执行 `\pgfpathgrid` 画出网格。命令 `\pgfpathgrid` 会检查 x 轴或 y 轴的步长是否 > 0.01pt, 只有在“是”的情况下，才会在这个轴上画出网格。
3. 宏 `\tikz@grid@x`, `\tikz@grid@y` 分别保存 x 轴向，y 轴向的步长。Ti*k*Z 会检查 `\tikz@grid@x`, `\tikz@grid@y` 中是否含有长度单位，如果没有长度单位，就按 xyz 坐标系统来解释；如果有长度单位，就按 canvas 坐标系统来解释。
4. 宏 `\tikz@grid@x`, `\tikz@grid@y` 都会被 `\pgfmathparse` 解析，所以它们可以是复杂的算式。

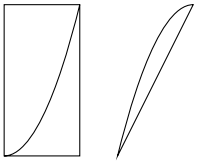
41.9 Parabola 操作

`parabola` 操作创建一段或两段抛物线 (parabola) 来延伸当前路径，抛物线都是 $f(x) = ax^2 + bx + c$ 这样的，只需要确定其顶点和另外一个点就可以确定其形态。

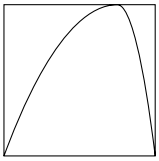
`\path...parabola`[⟨*options*⟩]`bend`⟨*bend coordinate*⟩⟨*coordiante or cycle*⟩

以当前点为起点，`parabola` 操作创建一段抛物线 (或两段相连的抛物线)，抛物线的终点是 ⟨*coordiante or cycle*⟩。其中的 `bend`⟨*bend coordinate*⟩ 是可选的，用于指定抛物线的顶点。如果不给出 `bend`⟨*bend coordinate*⟩，那就只能创建一段抛物线 (默认其起点为顶点)。当给出 `bend`⟨*bend coordinate*⟩ 时，可以创建两段相连的抛物线，且这两段抛物线都以 ⟨*bend coordinate*⟩ 为顶点。

如果给出 `bend +(1,1)` 这种相对坐标的形式，则这个相对坐标实际所指的点另有规定，参考下面的选项 `bend pos`。



```
\begin{tikzpicture}
  \draw (0,0) rectangle (1,2)
        (0,0) parabola (1,2);
  \draw[xshift=1.5cm] (0,0) -- (1,2) parabola cycle;
\end{tikzpicture}
```



```
\tikz \draw (0,0) rectangle (2,2)
  (0,0) parabola bend(1.5,2) (2,0); % 两段抛物线，都以 (1.5,2) 为顶点
```

上面例子中有两段抛物线，其顶点都是 (1.5,2)。其中 bend(1.5,2) 把抛物线的顶点指定为 (1.5,2)，但这样的抛物线不能同时经过点 (0,0) 和 (1,1)，所以是两段抛物线。

/tikz/bend=⟨*coordinate*⟩ (no default)

等效于 bend⟨*bend coordinate*⟩。

/tikz/bend pos=⟨*fraction*⟩ (no default)

这个选项用于确定像 bend +(1,1) 这种相对坐标形式的抛物线顶点。下面代码

```
\tikz \draw ⟨P1⟩ parabola [bend pos=⟨f⟩] bend +⟨P2⟩ ⟨P3⟩;
```

所确定的抛物线顶点是

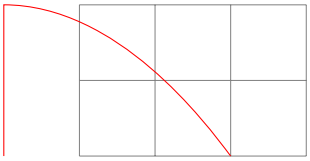
$$\langle P1\rangle+\langle f\rangle\times(\langle P3\rangle-\langle P1\rangle)+\langle P2\rangle$$

也就是说，先在直线 ⟨*P1*⟩⟨*P3*⟩ 上确定一个点，再以向量 ⟨*P2*⟩ 为平移向量来平移该点，即得到抛物线的顶点。

下面代码

```
\tikz \draw ⟨P1⟩ parabola [bend pos=⟨f⟩] bend +⟨P2⟩ +⟨P3⟩;
```

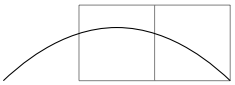
中的 +⟨*P3*⟩ 相对于 ⟨*P1*⟩, 即 +⟨*P3*⟩ 确定的点是 ⟨*P1*⟩+⟨*P3*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw[red] (-1,0) parabola bend +(0,2) +(3,0);
\end{tikzpicture}
```

/tikz/parabola height=⟨*dimension*⟩ (no default)

这个选项等价于 bend pos=0.5,bend={+(0pt,⟨*dimension*⟩)}，注意 ⟨*dimension*⟩ 要带上长度单位，否则默认长度单位是 pt.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,1);
  \draw (-1,0) parabola[parabola height=20] +(3,0);
\end{tikzpicture}
```

/tikz/bend at start (style, no value)

等效于选项 bend pos=0,bend={+(0,0)}.

/tikz/bend at end (style, no value)

等效于选项 bend pos=1,bend={+(0,0)}.

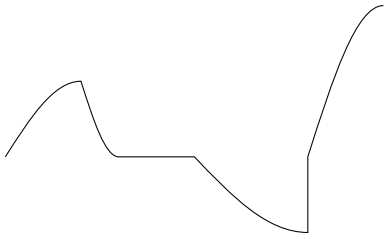
41.10 Sine 和 Cosine 操作

\path...sin⟨*coordinate or cycle*⟩...;

以当前点为起点，sin 操作画正弦曲线，以 ⟨*coordinate or cycle*⟩ 为曲线终点。起点与终点之间的曲线是将正弦函数 $\sin(x)$ 在区间 $[0,\frac{\pi}{2}]$ 上的图像作某种变换后得到的图形。

- 如果起点与终点在同一水平线或竖直线上，则 sin 操作画一个线段。
- 如果起点在终点下方，则 sin 操作画的曲线形态类似函数 $\sin(x)$ 在区间 $[0,\frac{\pi}{2}]$ 上的图像。

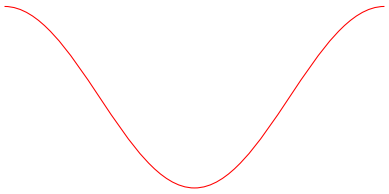
- 如果起点在终点上方，则 sin 操作画的曲线形态类似函数 $\sin(x)$ 在区间 $[\pi, \frac{3\pi}{2}]$ 上的图像。



```
\begin{tikzpicture}
\draw (0,0) sin (1,1) sin (1.5,0) sin (2.5,0)
      sin (4,-1) sin (4,0) sin (5,2);
\end{tikzpicture}
```

\path...cos⟨*coordinate or cycle*⟩...;

cos 操作与 sin 操作类似，在起点与终点之间的曲线是将余弦函数 $\cos(x)$ 在区间 $[0, \frac{\pi}{2}]$ 上的图像作某种变换后得到的图形。



```
\begin{tikzpicture}[xscale=1.57,scale=0.8]
\draw[color=red] (0,1.5) cos (1,0) sin (2,-1.5)
    cos (3,0) sin (4,1.5);
\end{tikzpicture}
```

41.11 SVG 操作

41.12 Plot 操作

41.13 To Path 操作

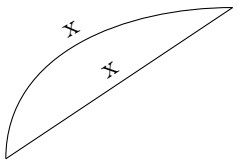
To Path 操作会在其位置上插入一个花括号分组，并在这个花括号组内插入某些代码来延伸当前路径，这与 edge 操作 不一样，edge 操作会中断当前路径。

\path...to[⟨*options*⟩]⟨*node*⟩⟨*coordinate or cycle*⟩...;

to 操作可以用一段路径把两个点连起来。在默认下，to 操作用线段连接两个点，可以在 ⟨*options*⟩ 中使用下面所说的选项、宏自己定义 (调整) to 操作所画的路径。

起点与目标点 to 操作之前的点是起点 (start coordinate)，to 操作之后的 ⟨*coordinate or cycle*⟩ 指定目标点 (target coordinate)。目标点保存在宏 \tikztotarget 中，起点保存在宏 \tikztostart 中，注意保存在这两个宏中的是坐标分量值，不包括圆括号。

to path 上的 node 在 to 操作之后可以使用 node，例如 (a) to node {x} (b)，给 to path 添加 node。所添加的 node 句子保存在宏 \tikztonodes 中。

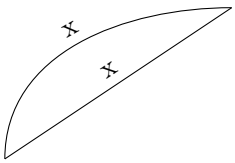



```
\begin{tikzpicture}
  \draw (0,0) to node [sloped,above] {x} (3,2);
  \draw (0,0) to[out=90,in=180] node [sloped,above] {x} (3,2);
\end{tikzpicture}
```

可以在 ⟨*options*⟩ 中使用下面的选项给 to path 添加 node：

/tikz/edge node={⟨*node specification*⟩} (no default)

这个选项给 to path 添加 node 语句。




```
\begin{tikzpicture}
  \draw (0,0) to [edge node={node [sloped,above] {x}}] (3,2);
  \draw (0,0) to [out=90,in=180,
        edge node={node [sloped,above] {x}}] (3,2);
\end{tikzpicture}
```

此选项的定义是：

```
\tikzset{
  edge node/.code={
    \expandafter\def\expandafter\tikz@tonodes\expandafter{\tikz@tonodes #1}
  },
  edge label/.style={/tikz/edge node={node[auto]{#1}}},
  edge label'/.style={/tikz/edge node={node[auto,swap]{#1}}},
}%
```

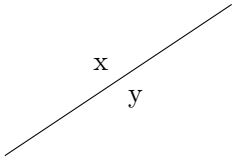
可见当执行 edge node={⟨*node specification*⟩} 后，宏 \tikz@tonodes 保存的内容就会被扩展；如果多次使用本选项，那么其作用会被累计，会依次扩展宏 \tikz@tonodes 的内容。按后文的分析,宏 \tikz@tonodes 的内容会被复制到宏 \tikztonodes 中;而宏 \tikztonodes 可用于构造 \tikz@to@path, 参考选项 /tikz/to path.

/tikz/edge label=⟨*text*⟩ (no default)

这是 edge node={node[auto]{⟨*text*⟩}} 的简写。

/tikz/edge label'=⟨*text*⟩ (no default)

这是 edge node={node[auto,swap]{⟨*text*⟩}} 的简写。

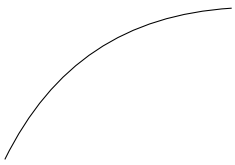



```
\tikz \draw (0,0) to [edge label=x, edge label'=y] (3,2);
```

载入 quotes 库后，可以用这个库提供的办法给路径加 node 标签，参考 §17.12.2.

to path 的样式 下面的 style 会添加到 to path 的开头：

/tikz/every to (style, initially empty)



```
\tikz[every to/.style={bend left}]
  \draw (0,0) to (3,2);
```

选项 选项 to path 用于定义 to 操作所画的路径。在默认下，to 路径被定义为线段，可以改成曲线。

/tikz/to path=⟨*path*⟩ (no default)

按后文的分析，命令

\path[⟨*options 1*⟩] ... ⟨*start*⟩ to [⟨*options 2*⟩] ⟨*node sepci*⟩ ⟨*target*⟩ ...;

等价于

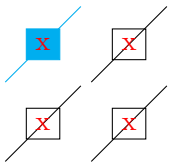
\path[⟨*options 1*⟩]...⟨*start*⟩{ [every to/.try][⟨*options 2*⟩] \tikz@to@path }...;

其中 ⟨*options 2*⟩ 是 to 后面方括号里的选项。而 /tikz/to path 的定义是：

```
\tikzoption{to path}{\def\tikz@to@path{#1}}%
\def\tikz@to@path{-- (\tikztotarget) \tikztonodes}%
```

所以宏 \tikz@to@path 的内容就是本选项的参数 ⟨*path*⟩.

注意 to 操作在当前路径上插入一个花括号 scope, 有的 to 选项会受到这个 scope 的限制，观察下面的例子:



```
\tikz \draw (0,0) to[red,dashed] node[draw,fill] {x} (1,1)[cyan];
\tikz \draw (0,0) {[red,dashed]--node[draw]{x} (1,1)};\\
\tikz[to path={[red,dashed] -- (\tikztotarget) \tikztonodes}]
  \draw (0,0) to node[draw]{x} (1,1);
\tikz [every to/.style={red,dashed}]
  \draw (0,0) to node[draw] {x} (1,1);
```

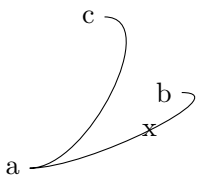
上面例子中，处于 to 操作辖制范围内的选项 red 只对 node 中的文字有效，对 node 的背景线颜色、填充色，对 scope 内的子路径颜色都无效；node 的背景线颜色、填充色、scope 内的子路径颜色使用的是“主路径”的设置 (被主路径的颜色设置覆盖了)。而选项 dashed 则没有任何作用。

在 ⟨*path*⟩ 中可以使用下面的宏：

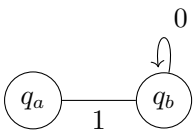
- \tikztostart,这个宏将展开为起点坐标的分量值,即不带圆括号的数据,例如“1pt,2pt”，如果给宏加圆括号 (\tikztostart)，则展开为“(1pt,2pt)”.
- \tikztotarget，这个宏将展开为目标点坐标的分量值，不带圆括号。
- \tikztonodes，这个宏将展开为 to 路径的 node 标签 (edge 后面的 node 语句)，默认标签位置在路径中间 (类似时间点的“中间”pos=0.5)。

⟨*path*⟩ 的默认设置是 -- (\tikztotarget) \tikztonodes

注意，像 --\tikztonodes (\tikztotarget) 这样，在 -- 后面直接使用 \tikztonodes 的做法是不合适的，Ti*k*Z 不允许把展开值为 node 的宏放在 -- 的后面。




```
\begin{tikzpicture}[to path={
    .. controls +(1,0) and +(1,0)
    .. (\tikztotarget) \tikztonodes}]
  \node (a) at (0,0) {a};
  \node (b) at (2,1) {b};
  \node (c) at (1,2) {c};
  \draw (a) to node {x} (b)
        (a) to (c);
\end{tikzpicture}
```




```
\tikzset{
  my loop/.style={to path={
    .. controls +(80:1) and +(100:1)
    ..(\tikztotarget) \tikztonodes}},
  my state/.style={circle,draw}}
\begin{tikzpicture}[shorten >=2pt]
\node [my state] (a) at (210:1) {$q_a$};
\node [my state] (b) at (330:1) {$q_b$};
\draw[->] (a) to node[below] {1} (b)
    to [my loop] node[above right] {0} (b);
\end{tikzpicture}
```

/tikz/execute at begin to=⟨*code*⟩ (no default)

代码 ⟨*code*⟩ 会在插入 to 路径前被执行，可以用来添加其它路径或者做某些计算。

本选项的定义是：

```
\tikzoption{execute at begin to}{\expandafter\def\expandafter
↪  \tikz@atbegin@to\expandafter{\tikz@atbegin@to#1}}%
\tikzoption{execute at end to}{\expandafter\def\expandafter\tikz@atend@to
↪  \expandafter{\tikz@atend@to#1}}%
```

可见执行 execute at begin to=⟨*code*⟩ 后，⟨*code*⟩ 就成为宏 \tikz@atbegin@to 的定义内容。

/tikz/execute at end to=⟨*code*⟩ (no default)

代码 ⟨*code*⟩ 会在插入 to 路径后被执行。参照上一选项，执行 execute at end to=⟨*code*⟩ 后，⟨*code*⟩ 就成为宏 \tikz@atend@to 的定义内容。

```
\tikz[every to/.style={draw,dashed}]
  \draw (0,0) to (2,0);\\
\tikz[every to/.style={append after command={[draw,dashed]}}]
  \draw (0,0) to (2,0);
```

```
\tikz[execute at end to={[draw,dashed]}]
  \draw (0,0) to (2,0);
\tikz[execute at end to={\path[draw,dashed];}]
  \draw (0,0) to (2,0);
```

在 topaths 库中定义了几种类型的 to path.

41.13.1 关于 to 操作

在文件《tikz.code.tex》中，to 操作主要由命令

- \tikz@to o
- \tikz@to@use@last@coordinate
- \tikz@to@or@edge
- \tikz@to@or@edge@option
- \tikz@@to@collect
- \tikz@@to@or@edge@coordinate
- \tikz@@to@or@edge@math
- \tikz@@to@or@edge@@coordinate
- \tikz@do@to

实现，其主要操作过程是：

1. 执行 \tikz@to@use@last@coordinate 将当前点坐标保存在宏 \tikztostart 中，作为起点

```
\def\tikz@to@use@last@coordinate{%
  \iftikz@shapeborder%
    \edef\tikztostart{\tikz@shapeborder@name}%
  \else%
    \edef\tikztostart{\the\tikz@lastx,\the\tikz@lasty}%
  \fi%
}%
```

   按 \iftikz@shapeborder 的真值，宏 \tikztostart 保存的可能是：
   - 一个 node 名称，即 \tikz@shapeborder@name, 这是包含“前缀”、“后缀”的完整名称
   - 带有长度单位的坐标尺寸，即寄存器 \tikz@lastx, \tikz@lasty 的当前值
2. 清空两个 (局部的、临时的) 保存选项、标签的宏：

```
\let\tikz@@to@local@options\pgfutil@empty%
\let\tikz@collected@onpath=\pgfutil@empty%
```

3. 执行 \tikz@to@or@edge, 把 to 操作的选项 (包括动画选项) 保存到 \tikz@@to@local@options 中
4. 执行 \tikz@@to@collect, 识别以下符号组合：
   - “to (”,如 to (1,1), 坐标会被 \tikz@scan@one@point 解析,解析结果用作 \tikz@@to@or@edge@math 的参数

- “to n”，即 to node, 如果是这个情况，则执行 \tikz@collect@label@onpath 来收集 node 句子，再重复执行 \tikz@@to@collect
- “to p”，即 to pic, 如果是这个情况，则执行 \tikz@collect@pic@onpath 来收集 node 句子，再重复执行 \tikz@@to@collect
- “to c”，即 to coordinate, 如果是这个情况，则执行 \tikz@collect@coordinate@onpath 来收集 node 句子，再重复执行 \tikz@@to@collect
- “to +”，如 to +(1,1), to ++(1,1), 坐标会被 \tikz@scan@one@point 解析，解析结果用作 \tikz@@to@or@edge@math 的参数

5. 执行 \tikz@@to@or@edge@math, 将 to 路径的目标点坐标 (\pgf@x, \pgf@y 的值) 保存到 \tikztotarget
6. 执行 \tikz@do@to, 此命令的定义是：

```
\def\tikz@do@to{%
  \let\tikz@tonodes=\tikz@collected@onpath%
  \def\tikztonodes{{\pgfextra{\tikz@node@is@a@labeltrue}\tikz@tonodes}}%
  \let\tikz@collected@onpath=\pgfutil@empty%
  \tikz@scan@next@command%
  {%
    \pgfextra{\let\tikz@after@path\pgfutil@empty}%
    \pgfextra{\tikz@atbegin@to}%
    \pgfextra{\tikz@enable@edge@quotes}%
    [style=every to]\expandafter[\tikz@@to@local@options]\tikz@to@path%
    \pgfextra{\tikz@atend@to}%
    \pgf@stop%
    \expandafter\tikz@scan@next@command\expandafter%
  }\tikz@after@path%
}%
```

其中：

- 宏 \tikz@atbegin@to 保存选项 execute at begin to 所提供的代码，此选项的效果是累计的
- \tikz@enable@edge@quotes 的初始值是

  ```
  \let\tikz@enable@edge@quotes\relax
  ```

  如果载入 quotes 库，那么这个命令会被重定义
- 宏 \tikz@atend@to 保存选项 execute at end to 所提供的代码，此选项的效果是累计的
- 还定义了宏 \tikztonodes,

  \tikztonodes

  这个宏保存一个花括号组，组内保存真值 \tikz@node@is@a@labeltrue, 以及 to 操作的标签 (to 后面的 node 语句)。真值 \tikz@node@is@a@labeltrue 的作用参考 \tikz@node@transformatio
  这个真值导致把 to 路径的“0.5 时刻”点作为 node 标签的锚定点 (如果不修改其锚定点的话)。
  在执行 \tikz@to@path 时，用到这个宏。

- \tikz@to@path, 参考选项 /tikz/to path → P. 740
- 当 \tikz@scan@next@command 扫描到记号 \pgf@stop 时，标志着 to 路径结束
- \tikz@after@path 与选项 /tikz/append after command → P. 719, /tikz/prefix after command → P. 719 有关，但也可能会被其他内部命令修改。按定义，这个 \tikz@after@path 保存的内容仅仅属于 to 操作的范围。

注意命令 \tikz@do@to 插入一个花括号组,是路径内部的 scope, 开花括号 { 导致 \tikz@beginscope, 闭花括号 } 导致 \tikz@endscope, 在这个 scope 内放置构造 to 路径的代码：

```
[style=every to]\expandafter[\tikz@@to@local@options]\tikz@to@path%
```

在保存 to 选项的宏 \tikz@@to@local@options 被展开之前，样式 every to 已经被执行。

\expandafter 会在执行 \tikz@endscope 前先展开 \tikz@after@path.

按上面的分析，命令

\path[⟨*options 1*⟩] ... ⟨*start*⟩ to [⟨*options 2*⟩] ⟨*node sepci*⟩ ⟨*target*⟩ ...;

等价于

\path[⟨*options 1*⟩]...⟨*start*⟩{\tikz@atbegin@to[every to/.try][⟨*options 2*⟩]\tikz@to@path\tikz@atend@to}⟨展开的 *\tikz@after@path*⟩...;

所以 to 操作延伸当前路径。

41.14 edge 路径

edge 操作的用法类似 to 操作和 node 操作，但与 to 操作不同，在 Ti*k*Z 的初始设置下，edge 操作会暂时中断当前主路径的构建过程，将一个新的路径 p 保存到盒子 \tikz@whichbox 中，在画出主路径后，再释放这个盒子，画出这个新路径，所以 (在初始之下) 由 edge 操作创建的路径跟 node 一样，都是主路径的附加，不属于主路径。

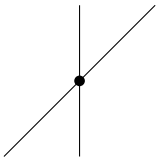
如果希望 edge 操作创建的路径属于主路径，那么可以利用 /tikz/edge macro →P. 746 重新定义 edge 操作使用的绘制代码。

edge 操作可以带有诸多选项来设定其所创建的路径外观。

下面代码

```
\tikz{
  \draw [name path=A] (0,0) edge (2,2);
  \draw [name path=B](1,0)--(1,2);
  \fill [name intersections={of=A and B,by=x}](x) circle [radius=2pt];
}
```

会导致错误，因为其中的 edge 操作创建的路径 (线段 (0,0) -- (2,2)) 不属于命令 \draw 创建的路径 A，当命令 \fill 计算交点时，导致计算失败。下面的代码则有效：



```
\tikz{
  \draw (0,0) edge [name path=A] (2,2);
  \draw [name path=B](1,0)--(1,2);
  \fill [name intersections={of=A and B,by=x}](x) circle [radius=2pt];
}
```

因为其中的选项 name path=A 是 edge 操作的选项，即线段 (0,0) -- (2,2) 的名称。

edge 操作的基本句法是：

\path...edge[⟨*options*⟩]⟨*node spec*⟩(⟨*coordinate*⟩)...;

edge 操作的作用是在创建主路径后，把下面的路径添加到图形中：

\path[every edge,⟨*options*⟩] (\tikztostart) ⟨*path*⟩;

其中的 ⟨*path*⟩ 是选项 /tikz/to path →P. 740 规定的路径，在此选项的默认下，edge 操作创建的路径是：

\path[every edge,⟨*options*⟩] (\tikztostart) -- (\tikztotarget) \tikztonodes;

例如：

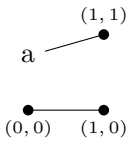
```
\tikz\draw (0,0) edge(1,1) --(1,0);
```

其中的主路径是 (0,0)--(1,0)，点 (0,0) 是 edge 操作的当前点（即 (\tikztostart)）；主路径的

构建过程会在当前点 (0,0) 处暂时中断，然后执行 edge 操作，构建一个 to path 路径 (默认情况下是线段 (0,0)--(1,1))，在 edge 操作 (即保存一个 \path 命令) 完成后，继续主路径的构建过程；画出主路径后，再画出 edge 操作保存的 to path 路径。

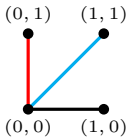
edge 操作与 to 操作相比较：

区别之一 如果紧挨着 edge 操作之前有 node 操作，例如 (0,0) node(a)[above]{a} edge (1,1)，那么这个名称为 (a) 的 node 才是 edge 操作的“起点”(start coordinate，即 (\tikztostart))，这一点与 to 操作、node 操作很不一样。例如：



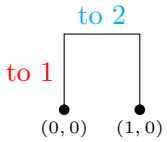

```
\tikz{
  \draw (0,0) node(a)[above=5mm]{a} edge (1,1) -- (1,0);
  \fill [font=\scriptsize]
    (0,0) circle (2pt) node [below] {$(0,0)$}
    (1,1) circle (2pt) node [above] {$(1,1)$}
    (1,0) circle (2pt) node [below] {$(1,0)$};
}
```

区别之二 如果连续使用数个 edge 操作，那么这些 edge 操作的起点(start coordinate，即 (\tikztostart))相同，这一点与 node 操作类似，而与 to 操作不同。也就是说，edge 操作不改变主路径的当前点。例如：

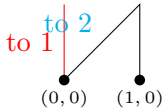



```
\tikz{
  \draw [very thick] (0,0) edge[red](0,1) edge[cyan](1,1)--(1,0);
  \fill [font=\scriptsize]
    (0,0) circle (2pt) node [below] {$(0,0)$}
    (1,1) circle (2pt) node [above] {$(1,1)$}
    (1,0) circle (2pt) node [below] {$(1,0)$}
    (0,1) circle (2pt) node [above] {$(0,1)$};
}
```

对上面的 \draw 命令来说，主路径在当前点 (0,0) 处中断，两个 edge 操作的起点都是点 (0,0)；执行完这两个 edge 操作后，继续构建主路径，此时当前点还是点 (0,0). 对比下面 to 操作的例子：

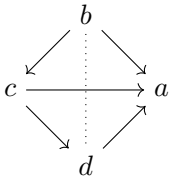



```
\tikz{
  \draw (0,0) to[red,"to 1"left](0,1) to[cyan,"to 2"](1,1)--(1,0);
  \fill [font=\scriptsize]
    (0,0) circle (2pt) node [below] {$(0,0)$}
    (1,0) circle (2pt) node [below] {$(1,0)$};
}
```



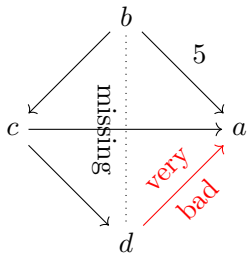

```
\tikz{
  \draw (0,0) edge[red,"to 1"left](0,1) to[cyan,"to
  ↪  2"](1,1)--(1,0);
  \fill [font=\scriptsize]
    (0,0) circle (2pt) node [below] {$(0,0)$}
    (1,0) circle (2pt) node [below] {$(1,0)$};
}
```

区别之三 edge 操作能继承主路径的选项，这与 to 操作一样：




```
\begin{tikzpicture}
  \node foreach \name/\angle in {a/0,b/90,c/180,d/270}
    (\name) at (\angle:1) {$\name$};
  \path[->] (b) edge (a)
                edge (c)
                edge [-,dotted] (d)
            (c) edge (a)
                edge (d)
            (d) edge (a);
\end{tikzpicture}
```

不过 edge 操作自己的选项对它自己创建的路径有效，这一点与 to 操作不同，例如：




```
\begin{tikzpicture}
  \node foreach \name/\angle in {a/0,b/90,c/180,d/270}
    (\name) at (\angle:1.5) {$\name$};
  \path[->] (b) edge node[above right] {$5$} (a)
                edge (c)
                edge [-,dotted] node[below,sloped] {missing} (d)
            (c) edge (a)
                edge (d)
            (d) edge [red] node[above,sloped] {very}
                           node[below,sloped] {bad} (a);
\end{tikzpicture}
```

对比下面 to 操作的例子：


```
\tikz\draw (0,0) to [dotted,red] node[draw] {to} (1,1);
```

可见 to 操作后的选项对它自己创建的路径根本没起作用，只是颜色选项对 to 后的 node 的文字内容有作用。

/tikz/every edge (style, initially draw)

此样式针对每个 edge 操作的样式。按前文，此样式起作用的位置是

```
\path[every edge, ...] ...;
```

注意，下面的

```
\path [every edge/.style={...}]
  (0,0) edge (2,2)
        edge (2,0);
```

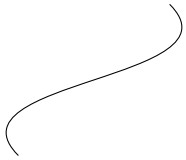
样式 every edge 对两个 edge 都有效，但下面的

```
\path
  (0,0) edge[every edge/.style={...}] (2,2)
        edge (2,0);
```

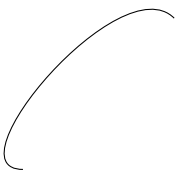
样式 every edge 对两个 edge 都无效 (因为只是定义样式，尚未使用样式)。

41.14.1 Edges 路径创建边的命令

在 TikZ 的初始设置下，edge 操作调用 to 操作创建边的命令，所以可以修改 to path 从而修改 edge 所创建的路径，例如：



```
\begin{tikzpicture}
  [to path={.. controls +(-1,1) and
    +(1,-1) .. (\tikztotarget) \tikztonodes}]
  \path (0,0) edge (2,2);
\end{tikzpicture}
```



```
\begin{tikzpicture}[control/.style 2 args={to path=
  {.. controls +(#1) and +(#2) ..
    (\tikztotarget) \tikztonodes}}]
  \path (0,0) edge[control={-1,0}{135:1}] (2,2);
\end{tikzpicture}
```

/tikz/edge macro={⟨*code*⟩} (initially empty)

使用这个选项后，edge 路径将由代码 ⟨*code*⟩ 画出。此选项的作用见后文。

本选项的定义是：

```
\tikzset{edge macro/.store in=\tikz@edge@macro}%
\let\tikz@edge@macro\pgfutil@empty
```

⟨*code*⟩ 会被保存到 `\tikz@edge@macro` 中。

⟨*code*⟩ 应当是能处理 2 个参数的命令 \⟨*cmd*⟩, TikZ 会以如下的形式：

\⟨*cmd*⟩{⟨*edge options*⟩}{⟨*node spec*⟩}

调用 \⟨*cmd*⟩, 其中的参数 ⟨*edge options*⟩ 与 ⟨*node spec*⟩ 就是 `edge` 操作中的参数：

`edge` [⟨*edge options*⟩] ⟨*node spec*⟩

注意，当 ⟨*code*⟩ 被执行时，还是处于主路径的创建过程中的，要考虑是否让 \⟨*cmd*⟩ 具有暂时中断主路径的作用；而且，TikZ 也不会用 `\tikz@scan@next@command` 解析⟨*code*⟩, 所以在 ⟨*code*⟩ 中直接使用 `--` 或者 `node`，`to` 之类的操作是不能被解析的；如果要在 ⟨*code*⟩ 中使用命令 `\tikz@scan@next@command`, 那么要注意：

- ⟨*code*⟩ 是被放在一个 `\begingroup` 与 `\endgroup` 的组合中来执行的，而 `\tikz@scan@next@command` 无法识别 `\endgroup`.
- 在 ⟨*code*⟩ 被执行完毕，`\endgroup` 结束组后，TikZ 还会再次执行 `\tikz@scan@next@command`.
- `\tikz@scan@next@command` 解析分号 `;` 后，才会停止解析，并调用 `\tikz@finish` →P. 759.

41.14.2 Edges 路径上的标签：Quotes Syntax

给 `edge` 路径加标签的方法有：

- 在 `edge` 之后加 node 语句。
- 给 `edge` 加 `/tikz/edge node` →P. 739, `/tikz/edge label` →P. 740 选项，但不能直接使用 `label`，`pin` 选项。
- 调用 quotes 程序，使用引用句法选项。

当在 `edge` 的选项中写出引用句法选项 `"`⟨*text*⟩`"'`⟨*options*⟩ 后，这个引用句法选项会被转变为：

`edge node=node [every edge quotes`⟨*options*⟩`]{`⟨*text*⟩`}`

其中的 `every edge quotes` 是个样式。

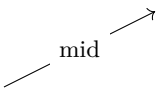
`/tikz/every edge quotes` (style, initially `auto`)

这个选项的初始值是 `auto`, 它规定引用句法创建的标签位于 `edge` 路径的左侧。可以用 `auto=right` 或用带撇号的引用句法来转换标签位置。


```
\tikz \draw (0,0) edge ["left", ->] (2,0);
```


```
\tikz [every edge quotes/.style={auto=right}]
  \draw (0,0) edge ["right", ->] (2,0);
```

如果重定义这个样式，那么默认的 `auto` 设置会被覆盖：




```
\tikz [every edge quotes/.style={fill=white,font=\small}]
  \draw (0,0) edge ["mid", ->] (2,1);
```

如果想在一个 `edge` 路径的左侧、右侧都放置标签，可以连续使用多个引用句法选项，并使用撇号“`'`”（等效于选项 `swap`）：

start left
right end

```
\tikz
  \draw (0,0) edge ["left", "right"',
                    "start" near start,
                    "end"' near end] (4,0);
```

start left
right

```
\tikz [tight/.style={inner sep=1pt}, loose/.style={inner sep=.7em}]
  \draw (0,0) edge ["left" tight,
                    "right"' loose,
                    "start" near start] (4,0);
```

41.14.3 关于 edge 操作

与 edge 相关的选项有：

- /tikz/edge node→P. 739, 设置 edge 的 node 标签 (edge 后的 node 语句)
- /tikz/edge label→P. 740, 设置 edge 的 node 标签
- /tikz/edge label'→P. 740, 设置 edge 的 node 标签
- /tikz/to path→P. 740, 设置 edge 操作的画边的代码
- /tikz/execute at begin to→P. 741
- /tikz/execute at end to→P. 742
- /tikz/every edge→P. 746, 样式

在文件《tikz.code.tex》中，edges 操作主要由命令

- \tikz@@e@char dge
- \tikz@edge@plain
- \tikz@to@or@edge
- \tikz@to@or@edge@option
- \tikz@@to@collect
- \tikz@@to@or@edge@coordinate
- \tikz@@to@or@edge@math
- \tikz@@to@or@edge@@coordinate
- \tikz@do@edge

实现，其主要操作过程是：

1. 用 \begingroup 开启一个组。
2. 执行 \tikz@to@use@whom, 此时有 3 个情况：
   (a) 如果 \tikz@to@use@whom 没有定义，则继续后面的步骤。
   (b) 如果 edge 之前是一个 node 语句,那么此时的 \tikz@to@use@whom 等于 \tikz@to@use@last@fig@name, 这导致定义 \tikztostart, 它保存这个 node 语句所创建的 node 的名称。
   在 \tikz@do@after@path@smuggle→P. 845 的定义中有：
   ```
   \let\tikz@to@use@whom=\tikz@to@use@last@fig@name%
   ```
   (c) 如果 edge 之前是一个 TikZ 坐标 (包括 node 名称的情况)，那么此时的 \tikz@to@use@whom 等于 \tikz@to@use@last@coordinate. 此时，这个 TikZ 坐标被 \tikz@scan@one@point→P. 710 解析后，就保存到 \tikztostart 中。
   在 \tikz@scan@one@point→P. 710 的定义中有：
   ```
   \let\tikz@to@use@whom=\tikz@to@use@last@coordinate%
   ```
3. 清空两个 (局部的、临时的) 保存选项、标签的宏：

```
\let\tikz@@to@local@options\pgfutil@empty%
\let\tikz@collected@onpath=\pgfutil@empty%
```

4. 收集 edge 操作的选项、标签、终点
   - 把 edge 操作的选项保存到 \tikz@@to@local@options 中
   - 把 edge 操作的标签 (edge 后面的 node 语句) 保存到 \tikz@collect@label@onpath 中
   - 把 edge 操作的终点保存到 \tikztotarget 中
5. 检查 \ifx\tikz@edge@macro\pgfutil@empty, 宏 \tikz@edge@macro 与 /tikz/edge macro →P. 746 有关：
   - 如果检查结果是 true,
     (a) 设置盒子：

```
\setbox\tikz@whichbox=\hbox\bgroup%
  \unhbox\tikz@whichbox%
  \hbox\bgroup
    \bgroup%
      \pgfinterruptpath%
        \pgfscope%
        %
        ⟨box content⟩
        %
        \endpgfscope%
      \endpgfinterruptpath%
    \egroup
  \egroup%
\egroup%
```

上面的盒子代码中有 \unhbox\tikz@whichbox, 所以实际是个递归定义，当连续使用 edge 操作时 (edge [...] ... edge [...] ...), 各个 edge 操作的内容会被依次保存到盒子 \tikz@whichbox 中。盒子 \tikz@whichbox 的最初定义是：

```
\newbox\tikz@figbox
%......
\def\tikz@whichbox{\tikz@figbox}%
```

所以对 \tikz@whichbox 的操作就是对 \tikz@figbox 的操作。

在 ⟨box content⟩ 中保存：

i. 一些有“清空”作用的代码，这些代码将清空某些宏的内容，例如用于保存 TikZ 变换矩阵、选项的内部宏，以及宏 \tikz@after@path (这个宏保存选项 append after command, prefix after command 提供的代码)

ii. 保存

```
\let\tikz@tonodes=\tikz@collected@onpath%
\def\tikztonodes{{\pgfextra{\tikz@node@is@a@labeltrue}\tikz@tonodes}}%
```

如果执行以上代码,就会把 edge 操作的标签 (edge 后面的 node 语句) 保存到宏 \tikztonodes 里面

iii. \pgfinterruptpath，将用于中断当前的主路径

iv. \pgfscope, 将用于开启一个 scope 绘图环境

v. \tikz@atbegin@to, 这个宏保存选项 execute at begin to 所提供的代码，注意此选项的效果是累计的

vi. \tikz@enable@edge@quotes, 这是 quotes 库提供的命令

vii. 保存

```
\path[style=every edge]\expandafter[\tikz@@to@local@options](
↪  \tikztostart)\tikz@to@path
\pgfextra{\global\let\tikz@after@path@smuggle=\tikz@after@path};%
```

这是绘制 edge 路径的代码，宏 \tikz@to@path 的定义是：

```
\tikzoption{to path}{\def\tikz@to@path{#1}}%
\def\tikz@to@path{-- (\tikztotarget) \tikztonodes}%
```

参考选项 /tikz/to path→P. 740.

注意，在保存 edge 选项的宏 \tikz@@to@local@options 被展开之前，样式 every edge 已经被展开。

viii. \tikz@atend@to, 这个宏保存选项 execute at end to 所提供的代码，注意此选项的效果是累计的

ix. \endpgfscope, 将用于结束 scope 绘图环境

x. \endpgfinterruptpath，将用于恢复当前的主路径

(b) 执行

```
\global\setbox\tikz@tempbox=\box\tikz@whichbox%
\expandafter\endgroup%
\expandafter\setbox\tikz@whichbox=\box\tikz@tempbox%
```

其中的 \endgroup 结束之前由 \begingroup 开始的组，而 2 个 \expandafter 将盒子 \tikz@whichbox 的定义放在这个组外，留待之后使用这个盒子。

- 如果检查结果是 false, 也就是说，之前使用了选项 edge macro=⟨*code*⟩, 那么执行

```
  \expandafter\expandafter\expandafter\tikz@edge@macro%
  \expandafter\expandafter\expandafter{\expandafter\tikz@@to@local@options
  ↪  \expandafter}\expandafter{\tikz@collected@onpath}%
\endgroup%
\let\tikz@after@path@smuggle=\pgfutil@empty%
```

有定义

```
\tikzset{edge macro/.store in=\tikz@edge@macro}%
\let\tikz@edge@macro\pgfutil@empty
```

宏 \tikz@edge@macro (即 ⟨*code*⟩) 应该能处理两个参数：

– 第一个参数是 \tikz@@to@local@options 的展开值，即 edge 操作的 (edge 后面放在方括号里的那些) 选项

– 第二个参数是 \tikz@collected@onpath 的展开值，即 edge 操作的 (edge 后面的) node 语句

但用手柄 /.store in 定义的宏 \tikz@edge@macro 是不带参数的，所以 ⟨*code*⟩ 应该是 (或等价于) 下面的形式：

```
\def\aaaa#1#2{...}%
\aaaa
```

6. 执行 \tikz@scan@next@command 处理 \tikz@after@path@smuggle 的展开值。
按前面代码,在 \tikz@edge@macro 等于 empty 的情况下,\tikz@after@path@smuggle 等于 \tikz@after@path,
参考 /tikz/append after command→P. 719, /tikz/prefix after command→P. 719

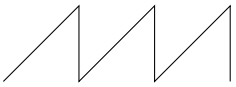
在主路径结束处,即遇到主路径末尾的分号 ; 时,执行命令 \tikz@finish, 此命令会引入盒子 \tikz@figbox 的内容，也就是 \tikz@whichbox 中保存的 edge 路径，因为有定义

```
\def\tikz@whichbox{\tikz@figbox}
```

41.15 Foreach 操作

\path...foreach⟨*variables*⟩[⟨*options*⟩]in{⟨*path commands*⟩}...;

当 \tikz@scan@next@command 扫描到符号“f”时，命令 \tikz@handle@more 会导出 \tikz@fchar; 扫描到记号 \foreach 时，命令 \tikz@handle@more 会导出 \tikz@foreach.



```
\tikz \draw (0,0) foreach \x in {1,...,3} { -- (\x,1) -- (\x,0) };
```

41.16 Let 操作

先看一个例子：

```
\tikz {\draw [red,line width=4pt]
  let
    \n1={sqrt(2)+0.5},
    \n2=1
  in
    (0,0)--(\n1,\n2);}
```

这个例子中，let 引起对宏 \n1，\n2 的赋值，然后在 in 引起的操作中使用这些宏。这就是 let ... in 操作的基本思路。

注意，let ... in 操作需要 Ti*k*Z 库 calc 的支持；单词 let 与 in 也必须匹配使用。

\path...let⟨*assignment*⟩,⟨*assignment*⟩,⟨*assignment*⟩...in...;

被赋值的宏有 3 种类型：

1. 数值 (number)：用宏 \n⟨*number register*⟩ 来存储数值，注意这里必须用字母“n”，而且这个宏必须带后缀。例如可以带数字后缀，\n1，\n2；可以带文字字符 (包括空格) 后缀，此时需要给文字字符加花括号，如 \n{text}，\n{string} 等等。如果数字后缀的序号超过“9”，则需要用花括号把数字后缀括起来，例如 \n{12}，\n{123} 等。
2. 坐标点 (point)：用宏 \p⟨*number register*⟩ 来存储坐标，注意这里必须用字母“p”，而且这个宏也必须带后缀，后缀格式与 \n⟨*number register*⟩ 类似。
3. 坐标分量：设置坐标点宏后，可以用坐标分量宏来引用坐标分量，例如，设置 let \p3=(a,b) 后，则宏 \x3 的值就是分量 a，宏 \y3 的值就是分量 b；再如，设置 let \p{text}=(a,b) 后，宏 \x{text} 的值就是分量 a，宏 \y{text} 的值就是分量 b. 也就是说，坐标分量宏与坐标点宏是通过后缀对应起来的，注意这里坐标分量宏的名称必须使用字母“x，y”.

   对坐标分量宏的赋值由 Ti*k*Z 自动完成。

以上宏的值存储在专门的寄存器 (register) 中，这是专属于 Ti*k*Z 的寄存器。

\n⟨*number register*⟩={⟨*formula*⟩}
\n⟨*number register*⟩

如果 \n⟨*number register*⟩ 位于等号“=”的左侧，则是赋值格式，其中的 ⟨*formula*⟩ 会被 \pgfmathparse 解析，运算结果保存到寄存器 ⟨*number register*⟩ 中，运算结果的长度单位会被转换为 pt。如果 \n⟨*number register*⟩ 位于等号“=”的右侧，或者位于其它地方时，它会被展开，展开值是相应寄存器 ⟨*number register*⟩ 中的值。

\p⟨*point register*⟩={⟨*formula*⟩}
\p⟨*point register*⟩

这个宏所存储的坐标数据不带圆括号，并且所有数据都转为以 pt 为单位的长度储存起来。设置 let \p1=(1pt,1pt+2pt) 后，\p1 展开为不带圆括号的数据“1pt,3pt”，而 (\p1) 才是坐标形

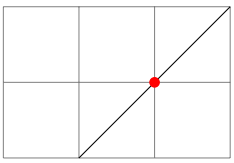
式 (1pt,3pt)，所以在引用坐标宏绘图时应该给宏加上圆括号。

```
\x{⟨point register⟩}
\y{⟨point register⟩}
```

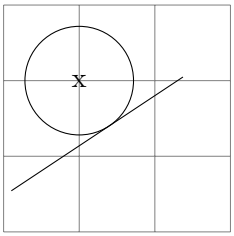
这两个宏分别引用后缀为 ⟨*point register*⟩ 的坐标宏的第一、第二个分量，它的值是一个以 pt 为单位的长度。按照坐标处理规则，(1pt+2,3) 等于 (3pt,3cm)，(1pt+2,3+4pt) 等于 (3pt,7pt)，在使用 \x⟨*point register*⟩ 和 \y⟨*point register*⟩ 做计算时要注意长度单位。

注意，这两个宏只在当前命令之内有效。

以上各个宏只能用在 let 操作中，不过 let ... in 操作内所规定的 coordinate 在此后都是有效的 (限于当前 {tikzpicture} 环境)，如下例：

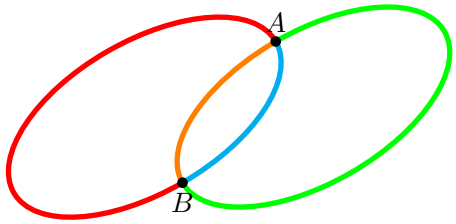


```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \path
    let
      \p1 = (1,0),
      \p2 = (3,2),
      \p{center} = ($ (\p1) !.5! (\p2) $)
    in
      coordinate (p1) at (\p1)
      coordinate (p2) at (\p2)
      coordinate (center) at (\p{center});
  \draw (p1) -- (p2);
  \fill[red] (center) circle [radius=2pt];
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,3);
  \coordinate (a) at (rnd,rnd);
  \coordinate (b) at (3-rnd,3-rnd);
  \draw (a) -- (b);
  \node (c) at (1,2) {x};
  \draw let \p1 = ($ (a)!(c)!(b) - (c) $),
            \n1 = {veclen(\x1,\y1)}
        in circle [at=(c), radius=\n1];
\end{tikzpicture}
```

下面是个稍微复杂一些的例子：




```
\begin{tikzpicture}[rotate=30]
  \draw [name path=p1] (0,0) ellipse (2 and 1);
  \draw [name path=p2] (2,-1) ellipse (2 and 1);
  \path [name intersections={of=p1 and p2,by={A,B}}];
  \coordinate (A') at($(A)-(2,-1)$);
  \coordinate (B') at($(B)-(2,-1)$);

  \draw [red,line width=2pt]
    let \p1=(A), \p2=(B)
    in  (A) arc [start angle={atan(\y1/\x1)}, end angle={atan(\y2/\x2)+180},
                 x radius=2cm, y radius=1cm];
  \draw [cyan,line width=2pt]
    let \p1=(A), \p2=(B)
    in  (A) arc [start angle={atan(\y1/\x1)}, end angle={-atan(\y2/\x2)},
                 x radius=2cm, y radius=1cm];
```

```
  \draw [green,line width=2pt]
    let \p1=(A'), \p2=(B')
    in  (A) arc [start angle={atan2(\y1,\x1)}, end angle={-atan2(\y2,\x2)},
                 x radius=2cm, y radius=1cm];
  \draw [orange,line width=2pt]
    let \p1=(A'), \p2=(B')
    in  (A) arc [start angle={atan2(\y1,\x1)}, end angle={atan2(\y2,\x2)},
                 x radius=2cm, y radius=1cm];
  \fill (A) circle (2pt) node [above] {$A$}
        (B) circle (2pt) node [below] {$B$};
\end{tikzpicture}
```

注意，下面的用法

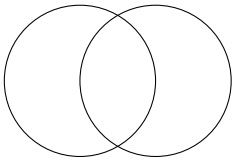
```
\tikz\node let \p1=(1,1) in at(\x1,\y1) {A};
```

会导致错误，而

```
\tikz\path let \p1=(1,1) in node at(\x1,\y1) {A};
```

则正常。

在当前路径命令之内使用 let 操作，以及命令 \pgfextra→P. 755, \endpgfextra, 可以将某些需要的点坐标转存到其他的宏中，例如



14.26349pt, 24.64009pt

```
\begin{tikzpicture}
  \draw [name path=a] circle (1cm);
  \draw [name path=b] (1,0) circle (1cm);
  \path [name intersections={of=a and b,by={A,B}}]
    let \p1=(A),\p2=(B)  in
    \pgfextra
      \xdef\uuuu{\x1} \xdef\vvvv{\y1}
    \endpgfextra
  ;
\end{tikzpicture}\par \uuuu, \vvvv
```

使用命令 \pgfgetlastxy→P. 260 也能达到类似的效果。

41.17 Scoping 操作

在一个路径内部，当 TikZ 遇到“{”时就会执行命令 \tikz@beginscope, 遇到“}”时就会执行命令 \tikz@endscope, 二者构成一个 scope. 选项 /tikz/current point is local→P. 705 与这种 scope 相关，此选项的定义是：

```
% Current point updates
\newif\iftikz@current@point@local
\tikzset{current point is local/.is if=tikz@current@point@local}%
```

本选项对应的 \iftikz@current@point@local 对命令 \tikz@@endscope 有影响。

41.18 Node and Edge 操作

41.19 Graph 操作

41.20 Pic 操作

41.21 Attribute Animation 操作

\path...:⟨*animation attribute*⟩={⟨*options*⟩}...;

这个路径操作等效于

[animate = { myself:⟨*animate attribute*⟩={⟨*options*⟩}}]

这会在当前路径上添加一个动画 (animation)。

41.22 PGF-Extra 操作

每个绘图命令，例如 \draw，都有自己的句法，不能向命令中随意添加 (句法不能接受的) 代码。例如，下面的代码会导致错误：

```
\tikz\draw [red] \tikzmath{\a=sqrt(3);}(0,0)--(\a,1);
```

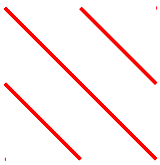
这个代码试图在命令 \draw 的内部使用数学程序库的命令做计算，然后用计算结果来构建路径，这会导致错误信息：

```
! Package tikz Error: Giving up on this path. Did you forget a semicolon?.
```

下面的代码：

```
\tikz \draw \def\aaa{0,0} (\aaa)--(1,1);
```

在路径中直接使用宏 \def 来做定义，也会导致错误。不过在路径中的适当位置插入 \foreach 命令则是可以接受的：



```
\tikz
  \draw [red,very thick]
    \foreach \i in {0,1,2}
      \foreach \j in {0,1,2}
      {(\j,\i)--(\i,\j)};
```

再如下面的方式也可以：

AAAA

```
\def\nnn{node}
\def\nnnn{\nnn}
\tikz\path\nnnn{AAAA};
```

在构建 TikZ 路径的过程中，有时候需要中断路径构建过程，执行某些代码，然后再继续构建路径，这就用到下面的命令。

\pgfextra{⟨*code*⟩}

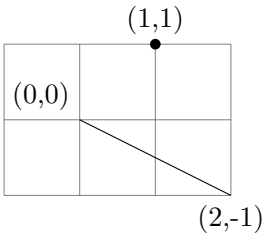
这个命令只能用在 TikZ 路径的构建过程中，注意一定要用花括号把 ⟨*code*⟩ 包裹起来。

实际上在《tikz.code.tex》中有定义：

```
\def\tikz@extra{\pgfutil@ifnextchar\bgroup\tikz@@extra\relax}%
\long\def\tikz@@extra#1{#1\tikz@scan@next@command}%
\let\endpgfextra=\tikz@scan@next@command
```

```
\def\pgfextra{pgfextra}%
```

可见宏 \pgfextra 只是保存一串字母，它本身并不需要参数，所以包裹 ⟨*code*⟩ 的这一对花括号不是 \pgfextra 的参数定界标志。某些内部命令在遇到 (作为 token 的) \pgfextra 时会有所反应，反应的结果之一是把记号 \pgfextra 吃掉，并且执行命令 \tikz@extra, 此时包裹 ⟨*code*⟩ 的这一对花括号实际上是命令 \tikz@@extra 的参数定界标志。在实际执行 ⟨*code*⟩ 时，这一对花括号并不是限制 ⟨*code*⟩ 的 TEX 组。




```
\newdimen\mydima
\newdimen\mydimb
\begin{tikzpicture}
  \mydima=1cm
  \mydimb=1cm
  \draw [help lines] (-1,-1) grid (2,1);
  \draw (0,0) node[above left]{(0,0)}
    \pgfextra{\mydima=2cm \mydimb=-1cm}
      -- (\mydima,\mydimb) node[below]{(2,-1)};
  \fill (\mydima,\mydimb) node[above]{(1,1)}
    circle (2pt);
\end{tikzpicture}
```

\pgfextra⟨*code*⟩\endpgfextra

若 \pgfextra 之后的 ⟨*code*⟩ 不处于花括号括内，则要用 \endpgfextra 来配合 \pgfextra. 注意 \pgfextra 和 \endpgfextra 配合起来并不引入额外的 TEX 分组来限制 ⟨*code*⟩.

41.23 在 TikZ 中使用软路径

关于软路径可参考 §121.

TikZ 的路径构造命令会转化为 PGF 基本层的路径构造命令，在用变换矩阵对坐标做变换后，再进一步转化为 PGF 系统层的软路径命令，所以软路径是很“底层”的内容。

/tikz/save path=⟨*macro*⟩ (no default)

将当前的软路径全局地保存在宏 ⟨*macro*⟩ 中。

此选项的定义是：

```
\tikzoption{save path}{\tikz@addmode{\pgfsyssoftpath@getcurrentpath#1
↪  \global\let#1=#1}}%
```

/tikz/use path=⟨*macro*⟩ (no default)

将保存在宏 ⟨*macro*⟩ 中的软路径调出作为当前路径。

```
\tikzoption{use path}{\tikz@addmode{\pgfsyssoftpath@setcurrentpath#1}}%
```

```
\begin{tikzpicture}
  \path[save path=\pathA,name path=A] (0,1) to [bend left] (1,0);
  \path[save path=\pathB,name path=B]
    (0,0) .. controls (.33,.1) and (.66,.9) .. (1,1);
  \fill[name intersections={of=A and B}] (intersection-1) circle (2pt);
  \draw[orange][use path=\pathA];
  \draw[red]    [use path=\pathB];
\end{tikzpicture}
```

41.24 构建路径的大致过程

41.24.1 作为框架的命令

在命令 \tikzpicture 的展开过程中 (在 tikzpicture 环境的开头) 会执行命令 \tikz@installcommands, 此命令定义一些 TikZ 命令，如命令 \path, \draw 的定义是：

```
% Install the abbreviated commands
%
\def\tikz@installcommands{%
  %......
  \let\path=\tikz@command@path%
  \let\tikz@origdraw=\draw%
  \def\draw{\path[draw]}%
  %......
}
%......
\def\tikz@uninstallcommands{%
  %......
  \let\path=\tikz@origpath%
  \let\draw=\tikz@origdraw%
  %......
}
```

所以，当 TikZ 遇到 \path 时就执行 \tikz@command@path; 当遇到 \draw 时就执行 \tikz@command@path[draw].

在《tikz.code.tex》中，构建路径的大致过程体现在以下数个命令中：

- \tikz@command@path
- \tikz@@command@path
- \tikz@scan@next@command
- \tikz@handle
- \tikz@handle@more
- \tikz@finish

\tikz@command@path

此命令的处理是：

1. 执行一些代码，以兼容 beamer 文类的命令
2. 执行 \tikz@@command@path

\tikz@@command@path

此命令的处理是：

1. 保存线宽

   ```
   \edef\tikzscope@linewidth{\the\pgflinewidth}
   ```

2. 用 \begingroup 开启一个组
3. 定义盒子

   ```
   \setbox\tikz@figbox=\box\pgfutil@voidb@x%
   \setbox\tikz@figbox@bg=\box\pgfutil@voidb@x%
   ```

   盒子 \tikz@figbox, \tikz@figbox@bg 主要用来盛放 node, edge 等附加物。
4. 将一些宏、寄存器的值初始化，

   ```
   \let\tikz@path@do@at@end=\tikz@lib@scope@check%
   \let\tikz@options=\pgfutil@empty%
   ```

```
\tikz@clear@rdf@options%
\let\tikz@mode=\pgfutil@empty%
\let\tikz@moveto@waiting=\relax%
\let\tikz@timer=\relax%
\let\tikz@tangent=\relax%
\let\tikz@collected@onpath=\pgfutil@empty%
\let\tikz@preactions=\pgfutil@empty%
\let\tikz@postactions=\pgfutil@empty%
\tikz@snakedfalse%
\tikz@decoratepathfalse%
\tikz@node@is@a@labelfalse%
\tikz@expandcount=100\relax%
\pgf@path@lastx=0pt%
\pgf@path@lasty=0pt%
\tikz@lastx=0pt%
\tikz@lasty=0pt%
\tikz@lastxsaved=0pt%
\tikz@lastysaved=0pt%
```

上面的 \tikz@expandcount 是个计数器

\tikz@expandcount

这是个计数器

```
\newcount\tikz@expandcount
```

这个计数器的值在此被设置为 100, 之后在 \tikz@expand →P. 758, \tikz@handle@more →P. 759 那里可能被修改。也有别的命令会利用这个计数器来限制展开次数，例如 \tikz@@circle.

宏 \tikz@options 保存的内容被清空，但在 tikzpicture(scope) 环境的开头处，这个宏已经展开过了，其中的命令 (环境选项) 已经被执行了。

5. 执行 \tikzset{every path/.try}, 样式 /tikz/every path →P. 718 在此处起作用。
6. 执行 \tikz@scan@next@command

\tikz@scan@next@command⟨*tokens*⟩

此命令的处理是：

1. 执行

```
\ifx\tikz@collected@onpath\pgfutil@empty%
\else%
  \tikz@invoke@collected@onpath%
\fi%
```

上面代码中，宏 \tikz@collected@onpath 是处理路径的附加内容的，如 node 标签。

2. 执行

```
\afterassignment\tikz@handle\let\pgf@let@token=⟨tokens⟩
```

意思是，先让 \pgf@let@token 等于 ⟨*tokens*⟩ 的第一个记号——记为 t_0, 此时 t_0 会被吃掉，从而把 ⟨*tokens*⟩ 变成 ⟨*tokens 1*⟩, 然后再执行

```
\tikz@handle⟨tokens 1⟩
```

\tikz@handle

此命令会对之前定义的 \pgf@let@token 进行识别 (识别的时候也会调用 \tikz@handle@more →P. 759 来帮忙), 根据识别情况来决定下一步如何进行。

此命令的处理是：

1. 执行 \let\pgfutil@next=\tikz@expand

\tikz@expand

这个命令的定义是：

```
\def\tikz@expand{%
  \advance\tikz@expandcount by -1%
  \ifnum\tikz@expandcount<0\relax%
    \tikzerror{Giving up on this path. Did you forget a semicolon?}%
    \let\pgfutil@next=\tikz@finish%
  \else%
    \let\pgfutil@next=\tikz@@expand
  \fi%
  \pgfutil@next}%

\def\tikz@@expand{%
  \expandafter\tikz@scan@next@command\pgf@let@token}%
```

\tikz@expand 的作用是，如果命令 \tikz@handle 不能识别之前定义的 \pgf@let@token，那么就：

(a) 让计数器 \tikz@expandcount 的值减去 1

(b) 用 \expandafter 将 \pgf@let@token 展开——展开结果记为 $\backslash$pgf@let@token$_1$

(c) 用命令 \tikz@scan@next@command → P. 757 处理 $\backslash$pgf@let@token$_1$，

(d) 将 $\backslash$pgf@let@token$_1$ 的第一个记号记为 t_1，再用命令 \tikz@handle 识别 t_1

(e) 如果还是不能识别 t_1，就再执行 \tikz@expand 来重复之前的“减去 1—展开—识别”操作

(f) 在重复之前操作的过程中，若计数器 \tikz@expandcount 的值小于 0，就执行 \tikzerror 报错。因为命令 \tikz@@command@path → P. 756 将计数器 \tikz@expandcount 的值初始化为 100，所以至多有 100 次的机会来展开 \pgf@let@token

AAAA

```
\def\n{node}
\def\nn{\n}
\def\nnn{\nn}
\def\nnnn{\nnn}
\tikz\path\nnnn{AAAA};
```

上面这个例子中，命令 \tikz@handle 不能识别 \nnnn，于是执行 \tikz@expand 将其展开，总共需要展开 4 次，直到命令 \tikz@handle 能识别“node”的第一个字母“n”.

2. 检查 \pgf@let@token 的当前值 (识别它)。检查过程是个较长的 \ifx ... \else ... \fi 句子

```
\ifx\pgf@let@token(%)
  \let\pgfutil@next=\tikz@movetoabs%
\else%
  \ifx\pgf@let@token+%
    \let\pgfutil@next=\tikz@movetorel%
  \else%
%......
```

检查过程是，将 \pgf@let@token 与“(”，“+”，“-”，“.”，“r”，“n”，“[”，“c”，“\bgroup”，“\egroup”，“;”逐个比较，

- 如果 \pgf@let@token 是 “(”，则令 \pgfutil@next 等于 \tikz@movetoabs
- 如果 \pgf@let@token 是 “+”，则令 \pgfutil@next 等于 \tikz@movetorel
- 如果 \pgf@let@token 是 “-”，则令 \pgfutil@next 等于 \tikz@lineto
- 如果 \pgf@let@token 是 “.”，则令 \pgfutil@next 等于 \tikz@dot

- 如果 \pgf@let@token 是 “r”, 则令 \pgfutil@next 等于 \tikz@rect
- 如果 \pgf@let@token 是 “n”, 则令 \pgfutil@next 等于 \tikz@fig
- 如果 \pgf@let@token 是 “[”, 则令 \pgfutil@next 等于 \tikz@parse@options
- 如果 \pgf@let@token 是 “c”, 则令 \pgfutil@next 等于 \tikz@cchar
- 如果 \pgf@let@token 是 “\bgroup”, 则令 \pgfutil@next 等于 \tikz@beginscope
- 如果 \pgf@let@token 是 “\egroup”, 则令 \pgfutil@next 等于 \tikz@endscope
- 如果 \pgf@let@token 是 “;”, 则令 \pgfutil@next 等于 \tikz@finish
- 如果 \pgf@let@token 不是以上符号，则令 \pgfutil@next 等于 \tikz@handle@more

3. 执行 \pgfutil@next, 即

   \pgfutil@next⟨*tokens 1*⟩

   若 \pgf@let@token 不是以上符号，就执行 \pgfutil@next (等于 \tikz@handle@more) 继续进行识别。

\tikz@handle@more

此命令：

1. 用 \ifx ... \else ... \fi 句子检查 \pgf@let@token 的当前值，
   - 如果 \pgf@let@token 是 “a”, 则令 \pgfutil@next 等于 \tikz@a@char
   - 如果 \pgf@let@token 是 “e”, 则令 \pgfutil@next 等于 \tikz@e@char
   - 如果 \pgf@let@token 是 “g”, 则令 \pgfutil@next 等于 \tikz@g@char
   - 如果 \pgf@let@token 是 “s”, 则令 \pgfutil@next 等于 \tikz@schar
   - 如果 \pgf@let@token 是 “|”, 则令 \pgfutil@next 等于 \tikz@vh@lineto
   - 如果 \pgf@let@token 是 “p”, 则令 \pgfutil@next 等于 \tikz@pchar, 再执行

     \pgfsetmovetofirstplotpoint →P. 494

     这是与图柄、图流有关的命令，也就说，这个 “p” 可能是“plot”的第一个字母，当然也可能是“pic”的第一个字母
   - 如果 \pgf@let@token 是 “t”, 则令 \pgfutil@next 等于 \tikz@to
   - 如果 \pgf@let@token 是 “\pgfextra”, 则令 \pgfutil@next 等于 \tikz@extra
   - 如果 \pgf@let@token 是 “\foreach”, 则令 \pgfutil@next 等于 \tikz@fchar
   - 如果 \pgf@let@token 是 “f”, 则令 \pgfutil@next 等于 \tikz@fchar
   - 如果 \pgf@let@token 是 “\pgf@stop”, 则令 \pgfutil@next 等于 \relax, 路径的构造至此结束
   - 如果 \pgf@let@token 是 “\par”, 则令 \pgfutil@next 等于 \tikz@scan@next@command
   - 如果 \pgf@let@token 是 “d”, 则令 \pgfutil@next 等于 \tikz@decoration
   - 如果 \pgf@let@token 是 “l”, 则令 \pgfutil@next 等于 \tikz@l@char
   - 如果 \pgf@let@token 是 “:”, 则令 \pgfutil@next 等于 \tikz@colon@char
   - 如果 \pgf@let@token 不是以上符号，则令 \pgfutil@next 等于 \tikz@expand →P. 758(见前文)
2. 如果 \pgf@let@token 不是 \tikz@expand, 即若能成功识别 \pgf@let@token, 则设置计数器值 \tikz@expandcount=100.
3. 执行 \pgfutil@next, 即

   \pgfutil@next⟨剩下的未被吃掉的记号⟩

\tikz@finish

此命令是路径的结束阶段，它很长……此命令之前的路径命令都是“准备工作”——例如准备好坐标，

设置外观选项，设置 node 标签，创建软路径等等——也就是“设计路径”，而此命令才是把“设计”变成实现 (画出来) 的步骤。

本命令的展开步骤是：

1. 执行 \box\tikz@figbox@bg, 插入 (释放) 盒子 \tikz@figbox@bg 里的内容
2. 执行装饰操作

```
\iftikz@decoratepath%
    \tikz@lib@dec@decorate@path%
\fi%
```

3. 执行选项 /tikz/preaction→P. 781 设置的操作

```
\pgfsyssoftpath@getcurrentpath\tikz@actions@path%
\edef\tikz@restorepathsize{%
  \global\pgf@pathmaxx=\the\pgf@pathmaxx%
  \global\pgf@pathmaxy=\the\pgf@pathmaxy%
  \global\pgf@pathminx=\the\pgf@pathminx%
  \global\pgf@pathminy=\the\pgf@pathminy%
}%
\tikz@preactions%
```

这个步骤：

(a) 先把当前的软路径保存到 \tikz@actions@path

(b) 保存当前的路径边界坐标

(c) 执行 \tikz@preactions, 参考 /tikz/preaction.

4. 重置模式

```
\let\tikz@path@picture=\pgfutil@empty%
\tikz@mode@fillfalse%
\tikz@mode@drawfalse%
\tikz@mode@doublefalse%
\tikz@mode@clipfalse%
\tikz@mode@boundaryfalse%
\tikz@mode@fade@pathfalse%
\tikz@mode@fade@scopefalse%
```

从以上代码看，所谓“模式”指的是 draw, fill, double, clip, fade, shade 等外观模式。

5. 定义

```
\edef\tikz@pathextend{%
    {\noexpand\pgfqpoint{\the\pgf@pathminx}{\the\pgf@pathminy}}%
    {\noexpand\pgfqpoint{\the\pgf@pathmaxx}{\the\pgf@pathmaxy}}%
}%
```

保存路径的边界尺寸坐标

6. 执行 \tikz@mode→P. 766, 将保存在 \tikz@mode 中的模式调出来。
7. 检查 fade 模式

```
% Path fading counts as an option:
\iftikz@mode@fade@path%
    \tikz@addoption{%
        \iftikz@fade@adjust%
            \iftikz@mode@draw%
                \pgfsetfadingforcurrentpathstroked{\tikz@path@fading}{
                ↪  \tikz@do@fade@transform}%
            \else%
```

```
                \pgfsetfadingforcurrentpath{\tikz@path@fading}{
                ↪  \tikz@do@fade@transform}%
            \fi%
        \else%
            \pgfsetfading{\tikz@path@fading}{\tikz@do@fade@transform}%
        \fi%
        \tikz@mode@fade@pathfalse% no more fading...
    }%
\fi%
```

8. 检查 scope fading

```
\iftikz@mode@fade@scope%
  \iftikz@fade@adjust%
    \iftikz@mode@draw%
      \pgfsetfadingforcurrentpathstroked{\tikz@scope@fading}{
      ↪  \tikz@do@fade@transform}%
    \else%
      \pgfsetfadingforcurrentpath{\tikz@scope@fading}{\tikz@do@fade@transform}
      ↪  %
    \fi%
  \else%
    \pgfsetfading{\tikz@scope@fading}{\tikz@do@fade@transform}%
  \fi%
  \tikz@mode@fade@scopefalse%
\fi%
```

9. 然后情况就有点复杂了，之后的步骤包含数个基本的操作，这些基本操作又可以按一定次序组合成数种情况。基本的操作是：

   操作 1 由检查 \tikz@options 引起的操作

```
\ifx\tikz@options\pgfutil@empty%
\else%
    \pgfsys@beginscope%
        \let\pgfscope@stroke@color=\pgf@strokecolor@global%
        \let\pgfscope@fill@color=\pgf@fillcolor@global%
        \begingroup%
            \tikz@options%
            〈 内容〉
        \endgroup%
        \global\let\pgf@strokecolor@global=\pgfscope@stroke@color%
        \global\let\pgf@fillcolor@global=\pgfscope@fill@color%
    \pgfsys@endscope%
    \iftikz@mode@clip%
        \tikzerror{Extra options not allowed for clipping path command.}%
    \fi%
\fi
```

   操作 a 必须执行的操作

```
\tikz@do@rdf@pre@options%
\tikz@is@nodefalse%
\tikz@call@id@hook%
```

   操作 2 由 \iftikz@mode@clip 引起的操作

```
\iftikz@mode@clip\else%
    \pgfidscope%
        \tikz@do@rdf@post@options%
```

```
        \begingroup%
            ⟨ 内容⟩
        \endgroup
    \endpgfidscope
\fi%
```

操作 3 由 \iftikz@mode@fill, \iftikz@mode@shade 引起的操作

```
\iftikz@mode@fill%
    \iftikz@mode@shade%
        \pgfsyssoftpath@getcurrentpath\tikz@temppath
        \pgfprocessround{\tikz@temppath}{\tikz@temppath}% change the path
        \pgfsyssoftpath@setcurrentpath\tikz@temppath%
        \pgfsyssoftpath@invokecurrentpath%
        \pgfpushtype%
        \pgfusetype{.path fill}%
        \pgfsys@fill%
        \pgfpoptype%
        \tikz@mode@fillfalse% no more filling...
    \else%
        \ifx\tikz@path@picture\pgfutil@empty%
        \else%
            \pgfsyssoftpath@getcurrentpath\tikz@temppath
            \pgfprocessround{\tikz@temppath}{\tikz@temppath}
            ↪  % change the path
            \pgfsyssoftpath@setcurrentpath\tikz@temppath%
            \pgfsyssoftpath@invokecurrentpath%
            \pgfpushtype%
            \pgfusetype{.path fill}%
            \pgfsys@fill%
            \pgfpoptype%
            \tikz@mode@fillfalse% no more filling...
      \fi%
    \fi%
\fi%
```

操作 4 由 \iftikz@mode@shade 引起的操作

```
\iftikz@mode@shade%
    \pgfsyssoftpath@getcurrentpath\tikz@temppath
    \pgfprocessround{\tikz@temppath}{\tikz@temppath}% change the path
    \pgfsyssoftpath@setcurrentpath\tikz@temppath%
    \pgfpushtype%
    \pgfusetype{.path shade}%
    \pgfshadepath{\tikz@shading}{\tikz@shade@angle}%
    \pgfpoptype%
    \tikz@mode@shadefalse% no more shading...
\fi%
```

操作 5 由选项 /tikz/path picture→P. 775 引起的操作

```
\ifx\tikz@path@picture\pgfutil@empty%
\else%
    \begingroup%
        \pgfusetype{.path picture}%
        \pgfidscope%
            \pgfsys@beginscope%
                \let\tikz@id@name\pgfutil@empty%
```

```
                \pgfclearid%
                \pgfsyssoftpath@getcurrentpath\tikz@temppath
                \pgfprocessround{\tikz@temppath}{\tikz@temppath}
                ↪  % change the path
                \pgfsyssoftpath@setcurrentpath\tikz@temppath%
                \pgfsyssoftpath@invokecurrentpath%
                \pgfsys@clipnext%
                \pgfsys@discardpath%
                \pgf@relevantforpicturesizefalse%
                \expandafter\def\csname pgf@sh@ns@path picture bounding box
                ↪  \endcsname{rectangle}
                \expandafter\edef\csname pgf@sh@np@path picture bounding
                ↪  box\endcsname{%
                    \def\noexpand\southwest{\noexpand\pgfqpoint{\the
                    ↪  \pgf@pathminx}{\the\pgf@pathminy}}%
                    \def\noexpand\northeast{\noexpand\pgfqpoint{\the
                    ↪  \pgf@pathmaxx}{\the\pgf@pathmaxy}}%
                }
                \expandafter\def\csname pgf@sh@nt@path picture bounding box
                ↪  \endcsname{{1}{0}{0}{1}{0pt}{0pt}}
                \expandafter\def\csname pgf@sh@pi@path picture bounding box
                ↪  \endcsname{\pgfpictureid}
                \pgfinterruptpath%
                    \tikz@path@picture%
                \endpgfinterruptpath%
            \pgfsys@endscope%
        \endpgfidscope%
    \endgroup%
    \let\tikz@path@picture=\pgfutil@empty%
\fi%
```

操作 6 由 `\iftikz@mode@draw`, `\iftikz@mode@double` 引起的操作

```
\iftikz@mode@draw%
    \iftikz@mode@double%
      % Change line width
        \begingroup%
            \pgfsys@beginscope%
                \tikz@double@setup%
                ⟨ 内容⟩
                \iftikz@mode@double%
             \pgfsys@endscope%
        \endgroup%
                \fi%
    \fi%
\fi%
```

操作 b 必须执行的操作

```
\pgfpushtype%
\edef\tikz@temp{\noexpand\pgfusepath{%
    \iftikz@mode@fill fill,\fi%
    \iftikz@mode@draw draw,\fi%
    \iftikz@mode@clip clip\fi%
}}%
\pgfusetype{.path}%
\tikz@temp%
\pgfpoptype%
```

```
\tikz@mode@fillfalse% no more filling
```

操作 c 必须执行的操作

```
\tikz@mode@drawfalse% no more stroking
\tikz@postactions%
\box\tikz@figbox%
```

以上的“操作 a”，“操作 b”，“操作 c”都是一定会被执行的；其余操作的有效代码 (按照真值判断) 可能被执行，也可能不被执行；这些操作实际上是按照 if 句子的结构套嵌着的，其套嵌次序是：

```
1 [ a 2 [ 3 4 5 6 [ b ] c] ]
```

其中的方括号表示套嵌在 if 句子的 ⟨内容⟩ 位置的操作，意思是：

- 例如 `6 [ b ]` 的意思是，如果执行 操作 `6` 的有效代码, 那么把 操作 `b` 放在 操作 `6` 的 ⟨内容⟩ 位置处执行；如果不执行 操作 `6` 的有效代码, 那就在这个地方执行 操作 `b`, 注意 操作 `b` 是一定会被执行的。
- 如果执行 操作 `2` 的有效代码, 那么把 `3 4 5 6 [ b ] c` 放在 操作 `2` 的 ⟨内容⟩ 位置处执行；如果不执行 操作 `2` 的有效代码, 那就在这个地方执行 `3 4 5 6 [ b ] c`.
- 如果执行 操作 `1` 的有效代码, 那么把 `a 2 [ 3 4 5 6 [ b ] c]` 放在 操作 `1` 的 ⟨内容⟩ 位置处执行；如果不执行 操作 `1` 的有效代码, 那就在这个地方执行 `a 2 [ 3 4 5 6 [ b ] c]`.

这种套嵌结构是好几个 if 句子做成的。

10. 执行

```
\iftikz@mode@clip%
    \aftergroup\pgf@relevantforpicturesizefalse%
\fi%
```

11. 执行

```
\iftikz@mode@boundary%
    \aftergroup\pgf@relevantforpicturesizefalse%
\fi%
```

12. 执行 `\endgroup`, 结束在 `\tikz@@command@path` P. 756 的开头处开启的组。
13. 执行 `\global\pgflinewidth=\tikzscope@linewidth`, 将线宽改回当前路径之前的值，这个值是在 `\tikz@@command@path` P. 756 的开头处被保存的。
14. 执行 `\tikz@path@do@at@end`, 即选项 `/tikz/postaction` P. 783 设置的操作，结束。

分析一个错误 下面代码会导致错误：

```
\tikz\path[thick,clip];
```

错误信息是：! Package tikz Error: Extra options not allowed for clipping path command..
出错的原因：选项 `thick`, `clip` 的定义是

```
\tikzset{thick/.style={line width=0.8pt}}%
\tikzoption{line width}{\tikz@semiaddlinewidth{#1}}%
\def\tikz@semiaddlinewidth#1{\tikz@addoption{\pgfsetlinewidth{#1}}
↪  \pgfmathsetlength\pgflinewidth{#1}}%
\tikzoption{clip}[]{\tikz@addmode{\tikz@mode@cliptrue}}%
```

- 执行 `\tikzset{clip}` 会把 `\tikz@mode@cliptrue` 添加到 `\tikz@mode` 中；
- 执行 `\tikzset{thick}` 会把 `\pgfsetlinewidth{0.8pt}` 添加到 `\tikz@options` P. 677 中，当然就使得 `\tikz@options` 的定义不同于 `\pgfutil@empty`, 因此前文说的 操作 `1` 的有效代码会被执行；
- 在 操作 `1` 末尾的 if 句子会执行 `\tikzerror` 报错。

下面代码也会出错：

```
\tikz\path[draw=red,clip];
```

而

```
\tikz\path[draw,clip];
```

不会出错，因为有定义：

```
\tikzoption{draw}[]{%
  \edef\tikz@temp{#1}%
  \ifx\tikz@temp\tikz@nonetext%
    \tikz@addmode{\tikz@mode@drawfalse}%
  \else%
    \ifx\tikz@temp\pgfutil@empty%
    \else%
      \tikz@addoption{\pgfsetstrokecolor{#1}}%
      \def\tikz@strokecolor{#1}%
    \fi%
    \tikz@addmode{\tikz@mode@drawtrue}%
  \fi%
}%
```

按选项 `draw` 的定义，`draw=red` 会重定义 `\tikz@options`，使其定义不同于 `\pgfutil@empty` 的定义。而不带参数的 `draw` 则不会这么做。如果需要为剪切路径设置颜色和线宽，可以：

```
\begin{tikzpicture}
    \begin{scope}[draw=red,line width=0.5mm]
        \draw[clip](0,0)rectangle(1,1);
        \begin{scope}[draw=black,line width=0.4pt]
            \node[draw,fill=green,circle]{AAA};
        \end{scope}
\end{scope}
\end{tikzpicture}
```

上面代码中，scope 环境的选项会使得 `\tikz@options` 不是空的，但命令 `\draw` 会先把 `\tikz@options` 做成空的 (见`\tikz@@command@path`→P. 756)，然后再读取选项 `clip`. 这样做虽然能直接给剪切路径设置颜色、线宽，但也会使得剪切路径的半个线宽被覆盖。

41.24.2 路径的模式

文件《tikz.code.tex》中有声明：

```
\newif\iftikz@mode@double
\newif\iftikz@mode@fill
\newif\iftikz@mode@draw
\newif\iftikz@mode@clip
\newif\iftikz@mode@boundary
\newif\iftikz@mode@shade
\newif\iftikz@mode@fade@path
\newif\iftikz@mode@fade@scope
\let\tikz@mode=\pgfutil@empty

\def\tikz@nonetext{none}%
```

路径的模式 (mode) 主要是指 double, fill, draw, clip, boundary, shade, fade 等外观参数，还可能会包括其他操作，例如选项 `/tikz/name path`→P. 693 会向“模式”中添加一些操作。“模式”保存在 `\tikz@mode` 中。模式会在路径结束命令 `\tikz@finish`→P. 759 的展开过程中起作用。

`\tikz@addmode{`⟨*mode code*⟩`}`

此命令的定义是：

```
\def\tikz@addmode#1{%
    \expandafter\def\expandafter\tikz@mode\expandafter{\tikz@mode#1}}%
```

参数 ⟨*mode code*⟩ 可能是 \iftikz@mode@double 等的真值，也可能是其他代码。
本命令重定义宏 \tikz@mode, 将 ⟨*mode code*⟩ 添加到宏 \tikz@mode 的定义内容中。
有的选项会利用这个命令来添加或修改模式，例如 /tikz/shade → P. 776 的定义是：

```
\tikzset{
    shade/.is choice,
    shade/.default=true,
    shade/true/.code=\tikz@addmode{\tikz@mode@shadetrue},
    shade/false/.code=\tikz@addmode{\tikz@mode@shadefalse},
    shade/none/.code=\tikz@addmode{\tikz@mode@shadefalse},
}%
```

当执行 \tikzset{shade} 时，就导致 \tikz@addmode{\tikz@mode@shadetrue}.

\tikz@mode

这个宏用于保存当前路径的模式，它的初始值被 let 为 \pgfutil@empty (空的)。
这个宏会被 \tikz@addmode 重定义。

/tikz/fill=⟨*color*⟩

这个选项的定义是：

```
\tikzoption{fill}[]{%
  \edef\tikz@temp{#1}%
  \ifx\tikz@temp\tikz@nonetext%
    \tikz@addmode{\tikz@mode@fillfalse}%
  \else%
    \ifx\tikz@temp\pgfutil@empty%
    \else%
      \tikz@addoption{\pgfsetfillcolor{#1}}%
      \def\tikz@fillcolor{#1}%
    \fi%
    \tikz@addmode{\tikz@mode@filltrue}%
  \fi%
}%
```

这个定义的意思是：

- 若执行 \tikzset{fill=none}, 则导致 \tikz@addmode{\tikz@mode@fillfalse}
- 若执行 \tikzset{fill=}, 则仅仅导致 \tikz@addmode{\tikz@mode@filltrue}
- 若执行 \tikzset{fill=red}, 则导致

```
\tikz@addoption{\pgfsetfillcolor{red}}%
\def\tikz@fillcolor{red}%
\tikz@addmode{\tikz@mode@filltrue}
```

/tikz/save path

```
\tikzoption{save path}{\tikz@addmode{\pgfsyssoftpath@getcurrentpath#1
↪  \global\let#1=#1}}%
```

41.24.3 收集某些要素的宏

\tikz@collected@onpath

这个宏的初始值被 let 为 \pgfutil@empty:

```
\let\tikz@collected@onpath=\pgfutil@empty%
```

路径内的某些要素，如 node 标签，pic 句子，动画命令，会被临时保存到这个宏中。例如 `--node[draw]A node[pos=0.8]B` 中的 `--` 导致 `\tikz@lineto`, 进而导致

```
\tikz@collect@label@onpath\tikz@lineto@mid node[draw]{A} node[pos=0.8]{B}
```

进而导致把 `node[draw]Anode[pos=0.8]B` 添加到 `\tikz@collected@onpath` 中。在适当的时候，保存在 `\tikz@collected@onpath` 中的内容会被处理。

参考：

- `\tikz@collect@label@onpath`
- `\tikz@collect@pic@onpath`
- `\tikz@collect@label@scan`
- `\tikz@collect@nodes`
- `\tikz@collect@nodes@one`
- `\tikz@collect@nodes@group`
- `\tikz@collect@animation`
- `\tikz@collect@paran`
- `\tikz@collect@options`
- `\tikz@collect@arg`
- `\tikz@invoke@collected@onpath`

\tikz@invoke@collected@onpath

本命令的主要作用是处理保存在 `\tikz@collected@onpath` 中的内容。

```
\def\tikz@invoke@collected@onpath{%
  \tikz@node@is@a@labeltrue%
  \let\tikz@temp=\tikz@collected@onpath%
  \let\tikz@collected@onpath=\pgfutil@empty%
  \expandafter\tikz@scan@next@command\tikz@temp\pgf@stop%
  \tikz@node@is@a@labelfalse%
}%
```

第四十二章　Actions on Paths

42.1　Overview

当一个路径被构造 (constructed) 后，可以对它做很多事情。例如，可以把它“画出”(draw 或说 stroke)，可以用颜色填充它，把它用作剪切路径来剪切别的路径，等等。

`\draw`

用在 `{tikzpicture}` 环境内，是 `\path[draw]` 的缩略。

`\fill`

用在 `{tikzpicture}` 环境内，是 `\path[fill]` 的缩略。

`\filldraw`

用在 `{tikzpicture}` 环境内，是 `\path[fill,draw]` 的缩略。

`\pattern`

用在 `{tikzpicture}` 环境内，是 `\path[pattern]` 的缩略。

`\shade`

用在 `{tikzpicture}` 环境内，是 `\path[shade]` 的缩略。

`\shadedraw`

用在 `{tikzpicture}` 环境内，是 `\path[shadedraw]` 的缩略。

`\clip`

用在 `{tikzpicture}` 环境内，是 `\path[clip]` 的缩略。

`\useasboundingbox`

用在 `{tikzpicture}` 环境内，是 `\path[useasboundingbox]` 的缩略。

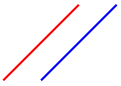
42.2　指定颜色

`/tikz/color=`⟨*color name*⟩　(no default)

为当前的 scope 设置颜色。这个颜色会被用于 `fill`，`draw`，文字颜色。对于 LaTeX 用户来说，⟨*color name*⟩ 可以是“LaTeX- color”，或者是 `xcolor` 宏包允许的带有叹号“`!`”的颜色名称格式。

可以省略“`color=`”，只写出 ⟨*color name*⟩.

注意选项 `draw=`⟨*color name*⟩，`fill=`⟨*color name*⟩，`color=`⟨*color name*⟩ 的作用次序。



```
\tikz[thick]{
  \draw [color=blue,draw=red] (0,0)--(1,1);
  \draw [draw=red,color=blue] (0.5,0)--(1.5,1);
}
```

42.3 画路径

/tikz/draw=⟨*color*⟩ (default is scope's color setting)

这个选项用作路径选项时，使得路径被“画出”(draw, stroke)。画路径的颜色是 ⟨*color*⟩。颜色 ⟨*color*⟩ 是可选的，如果不给出 ⟨*color*⟩，那么就使用 `color=` 值。

如果使用 `draw=none`，则关闭画线功能。

这个选项用作 `{scope}` 或者 `{tikzpicture}` 环境的选项时，并不导致环境内的各个路径被画出，此时只是设置画线时所用的颜色。

42.3.1 Graphic Parameters: Line Width, Line Cap, and Line Join

以下选项仅在启用画线功能 (使用 `draw` 选项) 时有效，它们影响所画出的线条的“外观”。

/tikz/line width=⟨*dimension*⟩ (no default, initially `0.4pt`)

指定线宽。设置线条的线宽，尺寸带单位。如果使用 `\draw[line width=0pt]...` ，那么所画的线条是“最细”的线条，在某些高分辨率的显示器上可能无法直接用眼睛辨认。

/tikz/ultra thin (style, no value)

本选项设置线宽为 0.1pt.

/tikz/very thin (style, no value)

本选项设置线宽为 0.2pt.

/tikz/thin (style, no value)

本选项设置线宽为 0.4pt.

/tikz/semithick (style, no value)

本选项设置线宽为 0.6pt.

/tikz/thick (style, no value)

本选项设置线宽为 0.8pt.

/tikz/very thick (style, no value)

本选项设置线宽为 1.2pt.

/tikz/ultra thick (style, no value)

本选项设置线宽为 1.6pt.

/tikz/line cap=⟨*type*⟩ (no default, initially `butt`)

设置路径线条端点的“帽子”，可选的“帽子”是 `round`(圆形), `rect`(方形), `butt`(没有帽子，光头，头是方的)，观察下面的例子：

```
\begin{tikzpicture}
  \begin{scope}[line width=10pt]
    \draw[line cap=rect] (0,0 ) -- (2,0);
    \draw[line cap=butt] (0,.5) -- (2,.5);
    \draw[line cap=round] (0,1 ) -- (2,1);
  \end{scope}
\end{tikzpicture}
```

可见帽子会凸出路径端点之外，`round`(圆形) 帽子的直径是线宽，`rect`(方形) 帽子的凸出长度是半个线宽。(参考 PDF Reference 的 Line cap styles.)

/tikz/line join=⟨*type*⟩ (no default, initially miter)

设置路径上的角 (不包括路径端点) 的外缘的外观。可选的类型是 round(圆), bevel(平), miter(尖).

```
\begin{tikzpicture}[line width=10pt]
  \draw[line join=round] (0,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[line join=bevel] (1.25,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[line join=miter] (2.5,0) -- ++(.5,1) -- ++(.5,-1);
  \useasboundingbox (0,1.5);
\end{tikzpicture}
```

参考 PDF Reference 的 Line join styles, 形状为 round 的线结合使用圆弧，圆弧的直径是线宽：

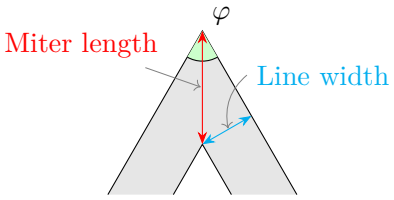
```
\begin{tikzpicture}
  \draw [line join=round,line width=5pt,double distance=0.4pt]
    (0,0)--++(60:1.5)--++(-60:1.5);
  \draw [red](60:1.5) circle (5pt);
\end{tikzpicture}
```

在形状为 bevel 的线结合中，线条端点使用 butt 帽子，但缺口会被填平：

```
\begin{tikzpicture}
  \draw [line join=bevel,line width=5pt,double distance=0.4pt]
    (0,0)--++(60:1.5)--++(-60:1.5);
\end{tikzpicture}
```

/tikz/miter limit=⟨*factor*⟩ (no default, initially 10)

当设置 line join=miter 并且角度非常小时，角的外缘会很尖锐，角尖很长。如果角尖突出角顶点的距离大于 ⟨*factor*⟩ 与线宽的乘积，则自动改为 line join=bevel。如下图所示，如果 ⟨*factor*⟩ $<$ $\frac{\text{Miter length}}{\text{Line width}} = \frac{1}{\sin(\frac{\varphi}{2})}$，那么就自动变成 line join=bevel 形式。参考《PDF Reference》(6 edition, v 1.7) 的 §4.3.2.



```
\begin{tikzpicture}[line width=3pt]
  \draw (0,0) -- +(3,.1) -- +(0,.2);
  \draw[miter limit=50] (0,1.5) -- +(3,.1) -- +(0,.2);
  \useasboundingbox (14,0);
\end{tikzpicture}
```

上面例子中如果去掉 \useasboundingbox 就是下面的效果：

XXX  XXX

```
XXX\begin{tikzpicture}[line width=3pt]
%  \draw (0,0) -- +(3,.1) -- +(0,.2);
  \draw[miter limit=50] (0,1.5) -- +(3,.1) -- +(0,.2);
%  \useasboundingbox (14,0);
\end{tikzpicture}XXX
```

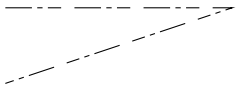
可见 Miter length 的某一部分是不被计入边界盒子内的。

42.3.2 Graphic Parameters: Dash Pattern

/tikz/dash pattern=⟨*dash pattern*⟩ (no default)

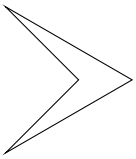
本选项设置线型，线型指的是实线、点线、虚线等。例如，on 2pt off 3pt on 4pt off 4pt，其中 on 2pt 表示移动 2pt 并画线，off 3pt 表示移动 3pt 但不画线。这样就定义了一个线型，它在起止点间重复使用这个线型。

注意，这种定义必须以 on 开头。



```
\begin{tikzpicture}[dash pattern=on 10pt off 2pt on 2pt
    off 2pt on 5pt off 5pt]
  \draw (0pt,0pt) -- (3cm,0pt)--(0,-1);
\end{tikzpicture}
```

如果定义线型 dash pattern=on 0pt off 4cm，实际上就是画线 (并非不画线)，但画笔只是沿着路径走了一遍，没有留下墨迹。



```
\begin{tikzpicture}
\draw [-{Stealth[angle=60:2cm,open,inset=1cm]},
  dash pattern=on 0pt off 4cm ]
  (0,0)--(3,0);
\end{tikzpicture}
```

上面例子中，主路径是 (0,0)--(3,0)，箭头是主路径的“附加”，不属于路径本身。画主路径时没有“墨迹”，所以只有一个箭头可见。

/tikz/dash phase=⟨*dash phase*⟩ (no default, initially 0pt)

这里的 ⟨*dash phase*⟩ 是带长度单位的尺寸。画路径线条时，选项 dash pattern 定义的线型是重复出现的，可以看作是周期性的，因而是有相位的，⟨*dash phase*⟩ 代表相位。

/tikz/dash=⟨*dash pattern*⟩phase⟨*dash phase*⟩ (no default)

同时设置 dash pattern 和 dash phase。

/tikz/solid (style, no value)

实线线型，这是画路径线条时的默认值。

/tikz/dotted (style, no value)

点线线型。此线型的定义是

```
\tikzset{dotted/.style={dash pattern=on \pgflinewidth off 2pt}}%
```

/tikz/densely dotted (style, no value)

密集点线线型。

/tikz/loosely dotted (style, no value)

稀疏点线线型。

/tikz/dashed (style, no value)

由“短线—空白”构成的线型，即常说的虚线。

/tikz/densely dashed (style, no value)

密集虚线线型。

/tikz/loosely dashed (style, no value)

稀疏虚线线型。

/tikz/dash dot (style, no value)

由“短线—空白—点—空白”构成的线型。

/tikz/densely dash dot (style, no value)

密集的短线—点线型。

/tikz/loosely dash dot (style, no value)

稀疏的短线—点线型。

/tikz/dash dot dot (style, no value)

短线—点—点线型。

/tikz/densely dash dot dot (style, no value)

密集的短线—点—点线型。

/tikz/loosely dash dot dot (style, no value)

稀疏的短线—点—点线型。

样式 dotted 的放大效果是：

```
\tikz\draw [dotted,line width=4mm](0,0)--(3,0);
```

可见 dotted 这种线型不适合较大的线宽值。为了得到由“圆点”构成的线型，可以使用下面的办法：


```
\def\aaaa{4pt}
\tikz\draw [
  line width=\aaaa,
  dash pattern=on 0.0001pt off 2*\aaaa,
  line cap=round] (0,0)--(3,0);
```

42.3.3 Graphic Parameters: 线条透明度

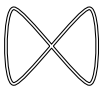
参考选项 /tikz/draw opacity → P. 905.

42.3.4 Graphic Parameters: Double Lines and Bordered Lines

/tikz/double=⟨*core color*⟩ (default white)

这个选项得到“双线”效果，实际上是沿着路径画线条且画两次，例如，先沿着路径画一条线宽是 10pt 的黑色粗线，然后沿着路径一条线宽是 6pt 的白色细线，这样就得到双线效果；单条黑色线的线宽是 2pt，双线间距是 6pt，双线间的颜色是白色。

此时的 thin, thick, line width 指的是整个双线宽度的一半。



```
\tikz \draw[double]
  plot[smooth cycle] coordinates{(0,0) (1,1) (1,0) (0,1)};
```

```
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
  \draw[draw=orange!10,double=red,very thick] (0,1) -- (1,0);
\end{tikzpicture}
```

/tikz/double distance=⟨*dimension*⟩ (no default, initially 0.6pt)

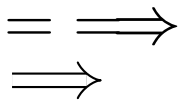
本选项会自动选定 double 选项 (即开启画双线功能)，并设置双线间距为 ⟨*dimension*⟩(带长度单位)，间距以双线的内侧边界为测量界限。

/tikz/double distance between line centers=⟨*dimension*⟩ (no default)

本选项会自动选定 double 选项 (即开启画双线功能)，并设置双线间距为 ⟨*dimension*⟩(带长度单位)，间距以双线线条的中心为测量界限。

/tikz/double equal sign distance (style, no value)

本选项会自动选定 double 选项 (即开启画双线功能)，并根据当前字体尺寸来设置双线间距。在当前字体尺寸下，数学等号“=”的上下横线之间的间距就是本选项设置的双线间距。



```
\Huge \baselineskip=20pt $=\implies$ \newline
\tikz[baseline]
  \draw[double equal sign distance,thick,-implies]
    (0,0.55ex) --++(3ex,0); \hfill~
```

42.4 在路径上添加箭头

42.5 填充路径

“填充路径”指的是将路径围起来的“内部”区域填色。这首先需要路径是封闭的，但如果路径是开的，程序先将其作成闭合的，然后填充。当路径比较复杂时，判断一个点是否属于应该填色的“内部”就不太容易。程序总是用 nonzero rule 或者 even odd rule 来判断一个点是否属于被填充区域，默认用 nonzero rule.

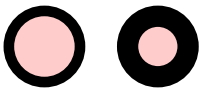
/tikz/fill=⟨*color*⟩ (default is scope’s color setting)

这个选项使得路径被填充。填充色或者是 ⟨*color*⟩，或者是用 color= 设置的颜色。如果 fill=none，则取消填充。

```
\begin{tikzpicture}[fill=yellow!80!black,line width=5pt]
  \filldraw (0,0) -- (1,1) -- (2,1);
  \filldraw (4,0) circle (.5cm) (4.5,0) circle (.5cm);
  \filldraw[even odd rule] (6,0) circle (.5cm) (6.5,0) circle (.5cm);
  \filldraw (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

上面第 1 个和第 4 个路径都是非封闭的，但先把它们作成封闭的然后填充颜色。

如果一个路径带有 fill 和 draw 选项 (如 \filldraw 命令)，则先 fill 再 draw，这样填充色就不会掩盖线宽，比较下面两个圆：

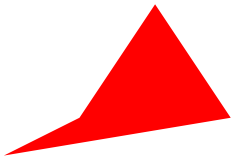


```
\tikz{
  \draw [line width=8pt]                  (0,0)   circle (0.4);
  \fill [fill=red!20]                     (0,0)   circle (0.4);
  \filldraw [line width=8pt,fill=red!20]  (1.5,0) circle (0.4);
}
```

如果填充一段线段会得到下面的效果：

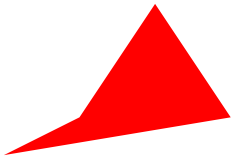
```
\tikz{
  \fill (0,0)--(1,1);
}
```

填充的结果是线段，所以在填充路径时，路径中的 move-to 操作可能带来的意外：



```
\tikz{
  \fill[red] (0,0)--(1,0.5)--(2,2) (2,2)--(3,0.5)--(0,0);
}
```

上面例子中的填充过程会把两段非封闭子路径分别封闭起来，然后判断“内部区域”，与下面例子等效：



```
\tikz{
  \fill[red] (0,0)--(1,0.5)--(2,2)--cycle (2,2)--(3,0.5)--(0,0)--cycle;
}
```

上面例子中,三角形 (0,0)--(1,0.5)--(2,2)--(0,0) 的内部区域不在填充范围内 (参照选项 /tikz/nonzero rule→P. 774), 因为 (0,0)--(1,0.5)--(2,2)--(0,0) 是逆时针方向, 而 (2,2)--(3,0.5)--(0,0)--(2,2) 是顺时针方向; 但线段 (0,0)--(2,2) 却在填充范围内。

注意 PGF 只是输出填充命令,实际执行填充操作的是与输出语言有关的程序。例如在《PDF Reference》中有关于 filling 以及 nonzero, even-odd 规则的解释:"The **f** operator uses the current nonstroking color to paint the entire region enclosed by the current path. If the path consists of several disconnected subpaths, ..." 所以决定最终填充效果的是输出语言, 而不是 PGF 本身。

42.5.1 Graphic Parameters: 填充 Pattern

除了用颜色填充路径外, 也可以用 pattern 来填充路径。这种填充类似贴瓷砖的工作, 设想给一块地面或墙面贴瓷砖, 每块瓷砖的尺寸和图案都是一样的, 一块瓷砖就是一个 tiling; 具有不同尺寸、图案的瓷砖是不同的 tiling pattern。

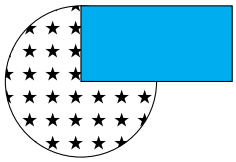
Tiling patterns 分两种: 不变色的 (inherently colored patterns) 和可变色的 (form-only patterns), 两种 pattern 都缺少可变性, 它们不会因放缩、旋转变换而变化。

在库 patterns 中定义了一些 pattern.

/tikz/pattern=⟨*name*⟩ (default is scope's pattern)

这个选项会用名称为 ⟨*name*⟩ 的 pattern 来填充路径。如果不给出 ⟨*name*⟩, 那就采用之前设置的 pattern 类型。设置 pattern=none 取消这种填充。

pattern 和 fill 的功能类似, 当这两个选项并列时, 无论哪个在前, 后面的总会抑制前面的。

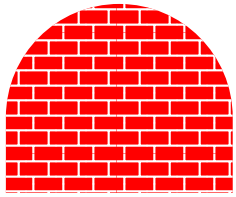


```
\begin{tikzpicture}
  \draw[fill=red,pattern=fivepointed stars,]
    (0,0) circle (1cm);
  \draw[pattern=fivepointed stars,fill=cyan]
    (0,0) rectangle (2,1);
\end{tikzpicture}
```

没有预定义的 pattern, 可以直接使用的 pattern 都在库 patterns 中。

/tikz/pattern color=⟨*color*⟩ (no default)

本选项用于可变色的 pattern, 对不可变色的 pattern 无效。



```
\begin{tikzpicture}
  \def\mypath{(0,0) -- +(0,1) arc (180:0:1.5cm) -- +(0,-1)}
  \fill [red] \mypath;
  \pattern[pattern color=white,pattern=bricks] \mypath;
\end{tikzpicture}
```

42.5.2 Graphic Parameters: 非零规则和奇偶规则

在构建路径时用到一系列的点, 这些点的先后次序决定了路径的"方向"。如果一个路径由数个相互分离的子路径构成, 那么每个子路径都有自己的方向。非零规则和奇偶规则都要参考路径方向, 默认非零规则。

由操作 rectangle, circle 创建的矩形和圆的方向都是逆时针方向。

/tikz/nonzero rule (no value)

这是默认的填充规则; 为了确定点 A 是否属于被填充的区域, 考虑从 A 出发的射线 $\vec{l}$ 并从 0 开始计数。如果 $\vec{l}$ 与区域的边界 (路径) 相交且边界线从射线 $\vec{l}$ 的左侧穿到右侧, 则计数加 1, 否则计数减 1; 若计数结果非 0, 则点 A 属于该区域, 否则点 A 不属于该区域。

/tikz/even odd rule (no value)

与上一规则做法类似，如果 $\vec{l}$ 与区域的边界线相交的次数是奇数，则点 A 属于该区域，否则点 A 不属于该区域。

```
\tikz\filldraw [nonzero rule,fill=red!20](0,0)circle(0.5)(0.5,0)circle(0.5);
\hspace{1cm}
\tikz\filldraw [even odd rule,fill=red!20](0,0)circle(0.5)(0.5,0)circle(0.5);
```

42.5.3 Graphic Parameters: 填充透明度

参考选项 /tikz/fill opacity → P. 905.

42.6 用任意图像填充路径

除了可以用颜色，pattern 填充路径外，还可以用自定义的路径或者从外部插入图形来填充路径，这个效果当然可以用 clip 选项做到，也可以用下面的选项实现。

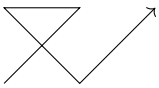
/tikz/path picture=⟨*code*⟩ (no default)

当一个路径带有这个选项后，会发生以下动作：先执行具有填充性质的操作 (如 fill，pattern，shade 等选项)，然后开启一个局部 scope，把路径用作剪切路径，然后执行 ⟨*code*⟩，由 ⟨*code*⟩ 创建的图形会被剪切；然后结束局部 scope；然后画出路径 (如果路径带有 draw 选项的话)。

类似 fill，draw 选项，该选项在一个路径之内有效。每当新路径开始时，path picture 都会被清空。⟨*code*⟩ 可以是 Ti*k*Z 的绘图命令，如由 \draw，\node 等；也可以插入外部图像。如果要插入外部图形，必须把 graphicx 宏包的 \includegraphics 命令放在 \node 命令中。

Predefined node path picture bounding box

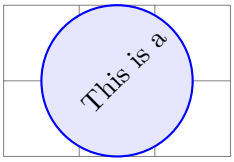
每个路径都有自己的边界盒子 (bounding box)，在默认下这个边界盒子是恰好装下该路径的矩形盒子，并且每当向路径中添加新的坐标点 (路径被延伸) 时，路径的边界盒子都会被刷新。也就是说，在路径创建过程的不同位置 (时间点) 上有不同的边界盒子，PGF 能实时地跟踪路径的边界盒子。当创建一个路径时，PGF 会自动生成一个名称为 current path bounding box 的矩形 node，这个 node 的尺寸、位置与该路径的、当前时间点的 bounding box 相同。也就是说，在路径创建过程的不同位置 (时间点) 上有不同的 current path bounding box。作为一个 node，current path bounding box 有自己的 node 坐标系统。



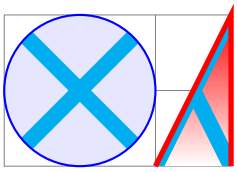
```
\tikz{
  \draw [->] (0,0)--(1,1)--(current path bounding box.north
  ↪  west)--(current path bounding box.south east)--(2,1);
}
```

注意计算 current path bounding box 时不考虑路径的线宽，也不把添加到路径上的 node 计算在内。

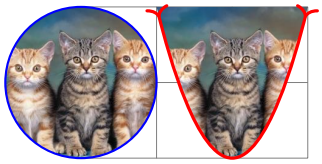
当路径带有 path picture=⟨*code*⟩ 选项时，一个名称为 path picture bounding box 的 node 会被自动创建，它的尺寸和位置与 (执行 ⟨*code*⟩ 之前的，即剪切 ⟨*code*⟩ 路径之前的) current path bounding box 一样。也就是说，在执行 ⟨*code*⟩ 之前就已经创建了 path picture bounding box，所以在 ⟨*code*⟩ 中可以引用 path picture bounding box 的坐标系统中位置点。




```
\begin{tikzpicture}
\draw [help lines] (0,0) grid (3,2);
\filldraw [fill=blue!10,draw=blue,thick] (1.5,1) circle (1)
  [path picture={
    \node [rotate=45,xshift=1cm]
      at (path picture bounding box.center) {
      This is a long text.
    };}
  ];
\end{tikzpicture}
```



```
\begin{tikzpicture}[cross/.style={path picture={
  \draw[cyan,line width=6pt]
    (path picture bounding box.south east) --
    (path picture bounding box.north west)
    (path picture bounding box.south west) --
    (path picture bounding box.north east);
  }}]
\draw [help lines] (0,0) grid (3,2);
\filldraw [cross,fill=blue!10,draw=blue,thick]
  (1,1) circle (1);
\path [cross,top color=red,draw=red,line width=2pt]
  (2,0) -- (3,2) -- (3,0);
\end{tikzpicture}
```



```
\begin{tikzpicture}[path image/.style={
  path picture={
    \node at (path picture bounding box.center) {
      \includegraphics[height=3cm]{#1}
    };}}]
\draw [help lines] (0,0) grid (3,2);
\draw [path image=cats,draw=blue,thick]
  (0,1) circle (1);
\draw [path image=cats,draw=red,very thick,>-<]
  (1,2) parabola[parabola height=-2cm] (3,2);
\end{tikzpicture}
```

这个选项的定义是：

```
\tikzset{path picture/.code=\tikz@addmode{\def\tikz@path@picture{#1}}}%
```

42.7 用颜色渐变填充路径

/tikz/shade (no value)

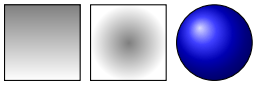
这个选项用颜色渐变填充路径。如果路径还带有 draw 选项，则先填充、后画出。如果同时给出 shade 和 fill 选项，则 fill 选项无效。

```
\tikz{
  \draw [shade,fill] (0,0) rectangle (1,1);
}
```

默认的渐变模式是上部灰色、下部白色。可以使用选项 /tikz/shading 选择颜色渐变的类型。

/tikz/shading=⟨*name*⟩ (no default)

选定名称为 ⟨*name*⟩ 的颜色渐变类型。库 shadings 定义了几种颜色渐变类型。也可以自定义颜色渐变，参考 \pgfdeclarehorizontalshading → P. 366.



```
\tikz \shadedraw [shading=axis] (0,0) rectangle (1,1);
\tikz \shadedraw [shading=radial] (0,0) rectangle (1,1);
\tikz \shadedraw [shading=ball] (0,0) circle (.5cm);
```

/tikz/shading angle=⟨*degrees*⟩ (no default, initially 0)

这个选项设置一个角度方向，使得颜色沿着这个方向渐变。

42.8 调整边界盒子

PGF 不仅能实时跟踪路径的边界盒子 (current path bounding box)，而且每当完成一个路径后，PGF 也会刷新整个图形 (picture) 的边界盒子来容纳新创建的路径，即能实时地跟踪图形的边界盒子。

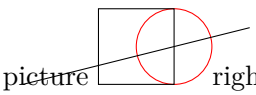
与路径、图形的边界盒子相关的选项、命令、node 有：

- 预定义 node，路径的边界盒子，current path bounding box
- 预定义 node，图形的边界盒子，current bounding box
- /tikz/overlay → P. 833
- /tikz/use as bounding box
- \useasboundingbox → P. 778
- \pgfresetboundingbox → P. 293
- \ifpgf@relevantforpicturesize → P. 294

/tikz/use as bounding box (no value)

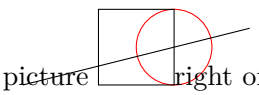
如果一个路径带有这个选项，那么将该路径计入图形的 bounding box 后，就停止刷新图形的 bounding box，即该路径之后的路径都不影响图形的 bounding box.

这个选项功能通常配合命令 \pgfresetboundingbox 一起使用，该命令的作用是重设图形的边界盒子，即把当前的图形边界盒子的尺寸设为 0，从当下开始，重新计算图形的边界盒子.

Left of picture  right of picture.

```
Left of picture
\begin{tikzpicture}
  \draw [red] (3,0.5) circle (0.5cm);
  \draw[use as bounding box] (2,0) rectangle (3,1);
  \draw (1,0) -- (4,.75);
\end{tikzpicture}right of picture.
```

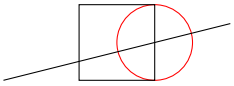
上面例子中，图形的 bounding box 仅由前两个路径决定。

Left of picture  right of picture.

```
Left of picture
\begin{tikzpicture}
\draw [red] (3,0.5) circle (0.5cm);
\pgfresetboundingbox
\draw[use as bounding box] (2,0) rectangle (3,1);
\draw (1,0) -- (4,.75);
\end{tikzpicture}right of picture.
```

上面例子中，命令 \pgfresetboundingbox 和选项 use as bounding box 配合使用，使得图形的 bounding box 仅由第二个路径决定。

但是注意，如果这个选项处于 TEX 分组内，其作用受到分组的限制 (因此也会受到 {scope} 环境的限制，如果在 {scope} 环境内的话)，比较下面的例子：

Left of picture  right of picture.

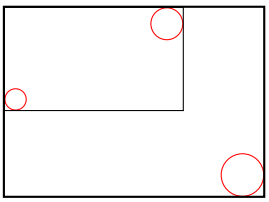
```
Left of picture
\begin{tikzpicture}
  \draw [red] (3,0.5) circle (0.5cm);
  {\draw[use as bounding box] (2,0) rectangle (3,1);}
  \draw (1,0) -- (4,.75);
\end{tikzpicture}
right of picture.
```

\useasboundingbox

这个命令是 \path[use as bounding box] 的缩写，注意其中没有 draw 选项。

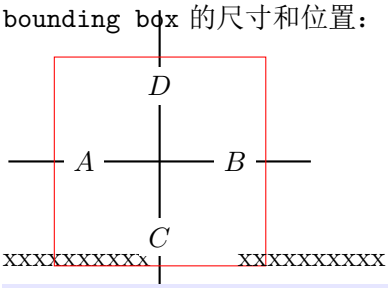
Predefined node current bounding box

这个预定义的 node 代表图形在当前时间点的边界盒子，它有自己的 node 坐标系统。



```
\begin{tikzpicture}
  \draw[red] (0,0) circle (4pt);
  \draw[red] (2,1) circle (6pt);
  \draw (current bounding box.south west) rectangle
        (current bounding box.north east);
  \draw[red] (3,-1) circle (8pt);
  \draw[thick] (current bounding box.south west) rectangle
               (current bounding box.north east);
\end{tikzpicture}
```

若 current bounding box 用作某个环境的选项，则在该环境结束时才会确定这个 current bounding box 的尺寸和位置：




```
xxxxxxxxxx
\begin{tikzpicture}[trim left={(current bounding box.center)},
                    trim right={($(current bounding box.east)+(-.5,0)$)},]
  \draw [thick] (-2,0)--(2,0);
  \draw [thick] (0,-2)--(0,2);
  {[every node/.style={fill=white}]
    \node (a) at (-1,0){$A$};
    \node (b) at (1,0){$B$};
    \node (c) at (0,-1){$C$};
    \node (d) at (0,1){$D$};}
  \pgfresetboundingbox
  \node [draw=red,fit=(a)(b)(c)(d)]{};
\end{tikzpicture}
xxxxxxxxxx
```

对比下面两个例子：

X  X

```
X\tikz{
  \draw [blue,line width=1cm,opacity=0.3] (0,0)--(1,0)
  ↪ node[text=red] at(current bounding box.north){x};
}X
```

X  X

```
X\tikz[baseline=(current bounding box.north)]{
  \draw [blue,line width=1cm,opacity=0.3] (0,0)--(1,0)
  ↪ node[text=red] at(current bounding box.north){x};
}X
```

这表明，当 current bounding box 用在一个路径内时，其计算是不考虑线宽的；当 current bounding box 用作环境选项时，其计算是考虑线宽的。对照 /tikz/miter limit →P. 770 处的例子。

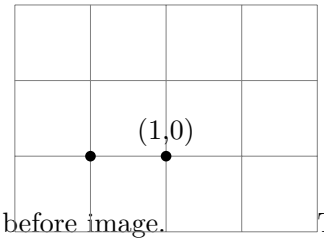
/tikz/trim left=⟨*dimension or coordinate or default*⟩ (default 0pt)

这个选项作为环境选项，设置整个图形的 bounding box 的左侧边界，默认为 0pt，即以坐标原点或说以直线 $x=$0pt 来标示左侧边界。trim left 只是设置图形的左侧边界，它不会清除边界左侧的那一部分图形。

如果设置 trim left=⟨*dimension*⟩，则以直线 $x=$⟨*dimension*⟩ 来标示左侧边界 (这里 ⟨*dimension*⟩ 是带单位的尺寸)。

如果设置 trim left={(a,b)}，则以直线 $x=$a 来标示左侧边界，注意如果这里的坐标含有逗号，则要用花括号将坐标括起来。

trim left=default 用于重设左侧边界。



Text before image. Text after image.

```
Text before image.%
\begin{tikzpicture}[trim left={(1,2)}]
  \draw [help lines](-1,-1) grid (3,2);
  \fill (0,0) circle (2pt)
    (1,0) node[above]{(1,0)} circle (2pt);;
\end{tikzpicture}%
Text after image.
```

/tikz/trim right=⟨*dimension or coordinate or default*⟩ (no default)

这个选项作为环境选项，设置整个图形的 bounding box 的右侧边界。

trim right=default 用于重设右侧边界。

/pgf/trim lowlevel=true|false (no default, initially false)

如果设置 trim lowlevel=false，则 external 库在输出图形时会输出完整的图形。如果设置 trim lowlevel=true，则 external 库在输出图形时只输出 trim left 之右，trim right 之左的部分。

盛放 node 的内容的盒子是个 TEX 盒子，TEX 盒子内有自己的基线位置。当把 Ti*k*Z 的绘图环境作为 node 的内容时，图形环境的基线与 TEX 盒子的基线对齐，此时图形边界盒子的中心未必能与 node 的中心重合，但有时候希望图形边界盒子的中心位于 node 的中心上，用上面的选项可以做到这一点。

```
\begin{tikzpicture}
  \node [draw, text height=0pt, text depth=0pt, text width=0pt,
         inner sep=0pt, minimum size=1cm]
    {\tikz \draw [red] (-1,-1)--(1,1);};
\end{tikzpicture}
```

```
\begin{tikzpicture}
  \node [draw, text height=0pt, text depth=0pt, text width=0pt,
         inner sep=0pt, minimum size=1cm]
    {\tikz[baseline=0cm, trim left={(current bounding box.center)}]
     \draw [red] (-1,-1)--(1,1);};
\end{tikzpicture}
```

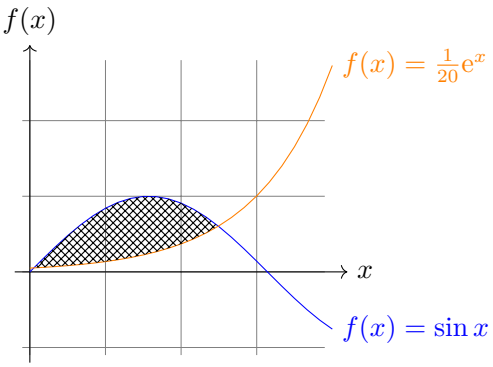
```
\begin{tikzpicture}
  \node [draw, clip, text height=0pt, text depth=0pt, text width=0pt,
         inner sep=0pt, minimum size=1cm]
    {\tikz[baseline=0cm, trim left={(current bounding box.center)}]
     \draw [red] (-1,-1)--(1,1);};
\end{tikzpicture}
```

42.9 剪切

剪切路径指的是起到剪刀作用的路径。被剪切路径指的是“被剪刀裁剪”的路径。

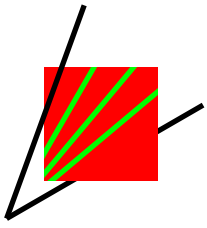
/tikz/clip (no value)

如果一个路径带有 clip 选项，该路径会成为剪切路径，对其后的路径进行剪切。如果该路径是自交的，则使用非零规则 (默认) 或奇偶规则来判断路径的内部和外部。



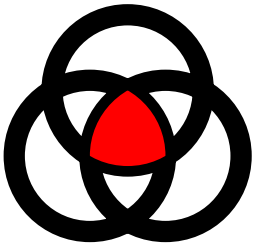

```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,2.8);
  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,3) node[above] {$f(x)$};
  \draw[color=blue] plot (\x,{sin(\x r)}) node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot (\x,{0.05*exp(\x)})
    node[right] {$f(x) = \frac{1}{20} \mathrm e^x$};
  \path [clip] plot (\x,{sin(\x r)});
  \fill[pattern=crosshatch] plot (\x,{0.05*exp(\x)})--(0,3)--cycle;
\end{tikzpicture}
```

clip 是图形状态参数，只在当前环境 (例如 {scope} 环境) 内有效，其效力持续至当前环境结束。一对花括号并不能限制 clip 的作用范围。



```
\begin{tikzpicture}[line width=2pt]
\draw (0,0) -- (30:3cm);
\begin{scope}[fill=red,draw=green]
  \fill[clip] (0.5,0.5) rectangle (2,2);
  \draw (0,0) -- (40:3cm);
  \draw (0,0) -- (50:3cm);
  \draw (0,0) -- (60:3cm);
\end{scope}
\draw (0,0) -- (70:3cm);
\end{tikzpicture}
```

如果一个环境内有多个剪切路径，则剪切效果会被累计。

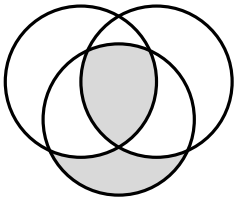


```
\begin{tikzpicture}[line width=8pt]
  \draw (0,0)circle(1cm) (1,0)circle(1cm)
        (60:1)circle(1cm);
  \path[clip] (0,0) circle (1cm);
  \path[clip] (1,0) circle (1cm);
  \fill[red] (60:1)circle(1cm);
\end{tikzpicture}
```

如果一个路径带有 clip 选项，那么就不能再同时带有某些选项，例如颜色、线宽等，这些选项可以在环境选项中列出。注意选项的作用次序依次是 fill, draw, clip.

\clip

这个命令是 \path[clip] 的缩写。



```
\tikz[very thick]{
  \tikzmath{\banjing=1;}
  \fill[fill=gray!30] circle (\banjing);
  \fill[fill=white] (-0.5,0.5) circle (\banjing);
  \fill[fill=white] (0.5,0.5) circle (\banjing);
  {[]
    \clip circle (1);
    \clip (-0.5,0.5) circle (\banjing);
    \fill [fill=gray!30] (0.5,0.5) circle (\banjing);
  }
  \draw circle (\banjing);
  \draw (-0.5,0.5) circle (\banjing);
  \draw (0.5,0.5) circle (\banjing);
}
```

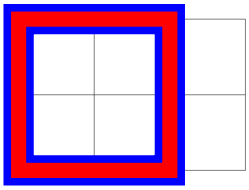
上面例子中，用一个 scope 环境限制剪切范围；为了避免填充色覆盖半个线宽，在最后画出 3 个圆。

42.10 对一个路径执行多重操作

当画一个路径时，可能有多个选项对该路径起作用，这些选项可能来自环境选项设置，也可能来自路径本身的选项设置，这些选项的作用有一定次序。下面的选项可以用于改变选项的作用次序。

/tikz/preaction=⟨*options*⟩ (no default)

这个选项可以用于 \path，\draw 等路径命令，也可以作为 node 操作的选项，但如果用于 {scope} 环境的环境选项则无效。本选项的参数 ⟨*options*⟩ 是那些能用作路径选项 (\path[⟨*options*⟩]) 的选项。当这个选项用作路径命令 \path 的选项时，其作用是：在程序构建好路径、但尚未使用路径 (使用路径指的是画出路径、填充路径等操作) 时，开启一个域，在这个域中利用本选项的 ⟨*options*⟩ 画出路径 (而不是用路径命令的选项画出路径)，然后再使用路径命令的通常选项画出路径，实际上路径被用不同选项画了两次。

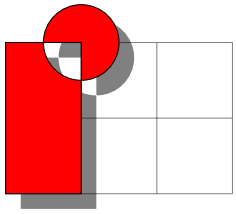


```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw
  [preaction={draw,line width=4mm,blue}]
  [line width=2mm,red]
  (0,0) rectangle (2,2);
\end{tikzpicture}
```

上面例子中，对同一个矩形，先以 4mm 线宽、蓝色画出，然后再以 2mm 线宽、红色画出。

注意，在第一次用 ⟨*options*⟩ 画出路径之前，路径就已被构建了，针对路径的坐标变换选项 (如 rotate) 在 ⟨*options*⟩ 中无效。第二次画路径时所使用的 (由通常方式设置的) 选项中的坐标变换选项对整个路径 (第二次画出的路径) 有效。

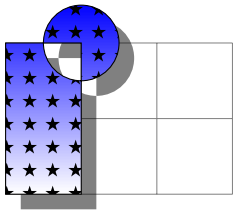
但 preaction 指定的 ⟨*options*⟩ 中可以有画布变换选项。



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw
  [preaction={fill=black,opacity=.5,
   transform canvas={xshift=2mm,yshift=-2mm}}]
  [fill=red]
  (0,0) rectangle (1,2)
  (1,2) circle (5mm);
\end{tikzpicture}
```

用上面例子中的办法可以定义阴影样式 (shadow style)，更多阴影样式参考 shadow library.

可以多次使用 preaction 选项，一个 preaction 选项导致路径被画一次，这些 preaction 选项按次序被执行，相互没有影响 (即各个选项的 ⟨*options*⟩ 各自起作用)，这样就可以自定义多组选项的作用次序。



```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw [pattern=fivepointed stars]
  [preaction={fill=black,opacity=.5,
    transform canvas={xshift=2mm,yshift=-2mm}}]
  [preaction={top color=blue,bottom color=white}]
  (0,0) rectangle (1,2)
  (1,2) circle (5mm);
\end{tikzpicture}
```

本选项的定义是：

```
\tikzset{preaction/.code=\expandafter\def\expandafter\tikz@preactions\expandafter{
↪  \tikz@preactions\tikz@extra@preaction{#1}}}%
\let\tikz@preactions=\pgfutil@empty
%......
\def\tikz@extra@preaction#1{%
  {%
    \pgfsys@beginscope%
      \setbox\tikz@figbox=\box\pgfutil@voidb@x%
      \setbox\tikz@figbox@bg=\box\pgfutil@voidb@x%
      \path[#1];% do extra path
      \pgfsyssoftpath@setcurrentpath\tikz@actions@path% restore
      \tikz@restorepathsize%
    \pgfsys@endscope%
  }%
}%
```

当多次使用本选项时：

```
preaction={⟨options 1⟩,preaction={⟨options 2⟩...
```

在宏 \tikz@preactions 中保存的是

```
\tikz@extra@preaction{⟨options 1⟩}\tikz@extra@preaction{⟨options 2⟩}...
```

当 Ti*k*Z 遇到路径结束符号——分号 ; 时，就执行命令 \tikz@finish, 此命令的展开过程中有

```
\pgfsyssoftpath@getcurrentpath\tikz@actions@path%
\edef\tikz@restorepathsize{%
  \global\pgf@pathmaxx=\the\pgf@pathmaxx%
  \global\pgf@pathmaxy=\the\pgf@pathmaxy%
  \global\pgf@pathminx=\the\pgf@pathminx%
  \global\pgf@pathminy=\the\pgf@pathminy%
}%
\tikz@preactions%
```

此时宏 \tikz@preactions 的作用是：以软路径 \tikz@actions@path 为蓝本，依次执行各个

\tikz@extra@preaction, 按不同的选项，将路径依次画出来。

注意这里处理的是“软路径”，在 preaction={⟨options⟩} 中，只有那些能影响软路径的处理的选项才有用。例如，变换选项 rotate=45 不起作用（因为它在构造软路径之前，对坐标计算起作用），而画布变换 transform canvas 有作用。对软路径的处理是在 \tikz@finish → P. 759 那里进行的，处理方式主要决定于 \tikz@mode → P. 766.

/tikz/postaction=⟨options⟩ (no default)

类似 preaction 选项，只是在用通常设置的路径选项画出路径后，再用 ⟨options⟩ 第二次画路径。本选项的定义是：

```
\tikzset{postaction/.code=\expandafter\def\expandafter\tikz@postactions
↪  \expandafter{\tikz@postactions\tikz@extra@postaction{#1}}}%
\let\tikz@postactions=\pgfutil@empty
%......
\def\tikz@extra@postaction#1{%
  {%
    \pgfsys@beginscope%
      \setbox\tikz@figbox=\box\pgfutil@voidb@x%
      \setbox\tikz@figbox@bg=\box\pgfutil@voidb@x%
      \tikz@restorepathsize%
      \path[#1]\pgfextra{\pgfsyssoftpath@setcurrentpath\tikz@actions@path};
      ↪  % do extra path
      \pgf@resetpathsizes%
    \pgfsys@endscope%
  }%
}%
```

42.11 装饰路径

参考 /pgf/decoration → P. 917.

42.12 在起点或终点处截去一段路径

/tikz/shorten <=⟨dimension⟩ (no default)

/tikz/shorten >=⟨dimension⟩ (no default)

这两个选项通常配合箭头选项一起使用。在文件《tikz.code.tex》中定义了这两个选项：

```
\tikzoption{shorten <}{\pgfsetshortenstart{#1}}
\tikzoption{shorten >}{\pgfsetshortenend{#1}}
```

其中的命令 \pgfsetshortenstart → P. 299 能够在路径的开头处把路径裁掉一段；命令 \pgfsetshortenend → P. 299 能够在路径的结尾处把路径裁掉一段。使用选项 shorten <=⟨dimension⟩ 可以在路径的起点处截掉一段长度为 ⟨dimension⟩ 的路径。使用选项 shorten >=⟨dimension⟩ 可以在路径的终点处截掉一段长度为 ⟨dimension⟩ 的路径。


```
\begin{tikzpicture}
  \node [draw] (p) {x};
  \node [draw] (q) at(2,0) {x};
  \draw [green](p.north east)--(q.north west);
  \draw [red,shorten <=5mm](p.south east)--(q.south west);
\end{tikzpicture}
```

第四十三章　Arrows

43.1　Overview

库 arrows.meta 提供 3.0 版的 Ti*k*Z 的新箭头，库 arrows，arrows.spaced 提供旧的箭头。

Ti*k*Z 允许在线条的端点处画一个或数个箭头，并可以设置每个箭头的方向、颜色等外观样式。有多种预定义的箭头，也可以自定义一种箭头，参考 \pgfdeclarearrow^{→P. 316}.

本节所介绍的箭头特性都属于 Ti*k*Z 的 3.0 版本，旧版本的箭头没有这些特性 (例如 scale=2 对旧版本箭头无效)。为了区别，在 LaTeX 中，新版箭头的名称的首字母都用大写。

旧的库 arrows，arrows.spaced 已经被新程序库 arrows.meta 代替，但是如果你愿意，仍可调用旧的库来使用旧的箭头。

43.2　如何添加箭头

添加箭头时要注意以下事项：

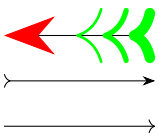
1. 使用选项 arrows 或其简缩形式来为线条添加箭头。
2. 可以为选项 tips 赋值，来决定是否画出箭头 (默认画出箭头)。
3. 箭头选项不能与 clip 选项同时用于同一个路径。
4. 被添加箭头的路径不能是“封闭的”，例如，不能给由 circle，rectangle 操作生成的路径加箭头；不能给使用 cycle 的路径加箭头。

/tikz/arrows=⟨*start arrow specification*⟩-⟨*end arrow specification*⟩ (no default)

这个选项给线条的始端或末端添加箭头，其中短线“-”代表路径。可以省略 arrows=，只写出等号右侧的部分，用小于号“<”和大于号“>”代表箭头，符号组合“->”，“<-”分别指示在路径的末端、始端加箭头并且规定了箭头方向；“<->”，“>-<”，“>>>-<<<”等符号的意思类似；“-Stealth”则规定路径末端采用 Stealth 样式的箭头。

在默认下，符号“>”代表的箭头样式名称是 Computer Modern Rightarrow，而“<”代表的箭头样式是 Computer Modern Rightarrow 的翻转。数学模式中的命令 \to 也默认 Computer Modern Rightarrow 样式。

实际上，表示箭头的符号“<”，“>”以及箭头样式名称“Stealth”都是“操作”，跟 circle，rectangle 一样，可以带有选项。



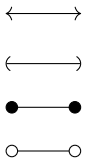
```
\begin{tikzpicture}[scale=2]
  \draw[->] (0,0) -- (1,0);
  \draw[>-Stealth] (0,0.3) -- (1,0.3);
  \draw[{Stealth[scale=4,red]}-{<[scale=4,green,line width=1pt]
                                <[scale=4,green,line width=2pt]
                                <[scale=4,green,line width=4pt]}]
    (0,0.6) -- (1,0.6);
\end{tikzpicture}
```

注意上面的例子表明，环境选项 scale=2 对箭头无效。还要注意，如果箭头带有选项，要把箭头及其

选项用花括号括起来，如上例，这样 TikZ 就不会把属于 Stealth 的 (用于界定选项列表的) 闭方括号“]”与命令 \draw 的相混淆。

箭头操作画出来的其实是个路径，如上例，Stealth 箭头是个四边形 (有一个内凹的顶角) 并且默认是填充颜色的，默认的 Computer Modern Rightarrow 箭头只画出路径线条，没有填充颜色的属性。

这几个符号：“<”，“>”，“(”，“)”，“*”，“o”都是箭头操作，它们画出的箭头形态如下：



```
\tikz\draw[<->](0,0)--(1,0);\\
\tikz\draw[(-)](0,0)--(1,0);\\
\tikz\draw[*-*](0,0)--(1,0);\\
\tikz\draw[o-o](0,0)--(1,0);
```

/pgf/tips=⟨*value*⟩ (default true, initially on draw)

/tikz/tips

这是 /pgf/tips 的别名。这个选项决定在什么情况下给路径添加箭头。其中的 ⟨*value*⟩ 可以是以下取值情况：

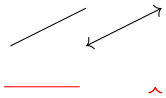
- true，这是默认值，即当只写出 tips 而不用等号“=”给它赋值时，其值就默认为 true.
- proper
- on draw，这是初始值，即当不使用该选项时，程序实际会以 tips=on draw 的设置工作。
- on proper draw
- never 或 false

总结起来，在以下情况下路径中不会出现箭头：

- 没有设置箭头，例如只写出 arrows=- 就没有箭头。
- 使用了 clip 选项。
- 将 tips 选项的值设为 never 或 false.
- 将 tips 选项的值设为 on draw 或 on proper draw，但是没有同时给出 draw 选项。
- 空路径 (不含任何坐标) 不会有箭头。
- 如果一个路径中存在闭合子路径 (例如，由 circle, rectangle, cycle 所引起的路径是闭合的)，那么主路径上没有箭头。

当路径中的子路径都非闭合时，才有可能为这个路径加箭头，并且箭头只可能加在最后一个子路径上。对于给“最后一个子路径”加箭头还有以下几种情况。

1. 如果该子路径是非退化的，即至少包含两个不同坐标点，箭头按正常方式添加。
2. 如果该子路径不包含任何坐标点，则没有箭头。
3. 如果该子路径只包含一个坐标点，那么仅当 tips=true 或 tips=on draw 时，箭头添加到这个点上，并且箭头指向上；但如果 tips=proper 或 tips=on proper draw，则没有箭头。



```
\tikz \draw [<->] (0,0)--(1,0.5) (1,0)--(2,0.5);

\tikz [red,<->] \draw (0,0) -- (1,0) (2,0);
```

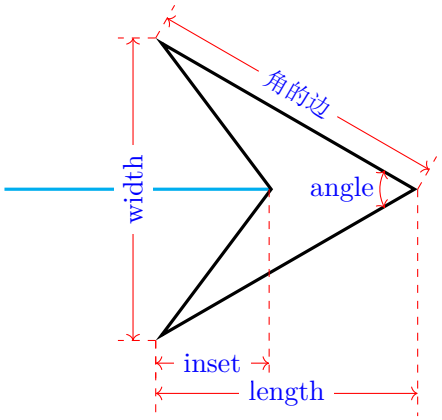
箭头的初始指向都是指向右侧的，箭头尖端的最末一个像素的位置是路径的端点。箭头不接受 rotate 选项。当箭头添加到路径上时，程序会自动将箭头绕着箭头尖端旋转，使得箭头的指示方向与路径 (在端点附近的) 方向一致。程序在确定箭头旋转状态时要参考箭头轮廓线与路径的公共点 (见 §16.3.8)，如果路径长度太短造成公共点缺失就导致程序计算失败，结果可能出人意料。所以被添加箭头的路径的长度与箭头尺寸要匹配，最好不要太短。

43.3 设置箭头的外观

对于标准的箭头，例如 Stealth，Latex，Bar，可以设置其尺寸、宽高之比、颜色等参数。设置箭头外观是通过设置箭头的选项来实现的，例如 Stealth[length=4pt,width=2pt].

43.3.1 箭头的“特征尺寸”

长、宽、高是一个立方体的“特征尺寸”，这些“特征尺寸”可以确定一个立方体。一个箭头也有各种“特征尺寸”，并且不同的箭头由于形态不同，具有不同的“特征尺寸”。下面以比较典型的 Stealth 箭头为例来说明各种“特征尺寸”。Stealth 箭头的定义出现在文件《pgflibraryarrows.meta.code》中。



上面图形中的青色横线代表主路径，箭头加载路径的右端；箭头尖端的最末一个像素位置是路径的端点；从箭头尖端到 inset 内凹的顶点这一段路径处于箭头的内部，这段路径会被“裁掉”，也就是说，路径的右端点会变成 inset 内凹的顶点。

/pgf/arrow keys/length=⟨*dimension*⟩⟨*line width factor*⟩⟨*outer factor*⟩ (no default)

这个选项设置箭头长度，注意箭头长度指的是从箭头最左侧的“像素”到最右侧的“像素”的水平距离。

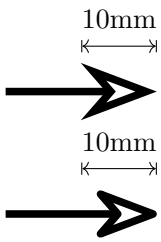
这个选项的值有 3 个参数，这 3 参数之间用空格分隔。⟨*dimension*⟩ 是带长度单位的尺寸，它规定箭头长度的基础尺寸，不能省略。⟨*line width factor*⟩ 和 ⟨*outer factor*⟩ 是纯数值，可以省略，由于它们之间用空格分隔，所以此二者的省略次序是从后向前的，如果省略，就默认其值为 0.

如果给出 ⟨*line width factor*⟩，那么箭头长度就是 ⟨*dimension*⟩ + ⟨*line width factor*⟩ * w，其中 w 是被添加箭头的路径的线宽，这就使得箭头长度与路径线宽具有相关性。例如 length=0pt 5，就保持箭头长度与路径线宽之比为 5:1.

⟨*outer factor*⟩ 仅在路径是双线时有效。当路径是双线时，路径有 3 种线宽：外侧线宽 wo，内部线宽 wi，总线宽 wt，它们之间有关系 wt=wi + 2wo. 当路径是双线且给出 ⟨*outer factor*⟩ 时，前面的 w 就由等式 w = ⟨*outer factor*⟩*wo + (1 - ⟨*outer factor*⟩)*wt 来确定。所以，如果 ⟨*outer factor*⟩ 是 0，则 w=wt.

例如，Latex 箭头的默认长度设置是 length=3pt 4.5 0.8，当路径线宽是 1.2pt 且为双线，双线间距为 2pt 时，该选项决定的箭头长度是：3pt + 4.5 * (0.8 * 1.2pt + (1 - 0.8) * 4.4pt) = 11.28pt.

Stealth 箭头是个封闭的路径 (四边形)，可以对其顶角设置 line join=⟨*round or miter*⟩ 选项，该选项的值 round 与 miter 会导致不同的箭头形态，但程序规定二者有相同的长度，观察下图，注意箭头尖端的最末一个像素的位置是路径的端点：




```
\tikz{
  \draw [line width=1mm, -{Stealth[length=10mm, open]}]
    (0,0) -- (2,0);
  \draw [|<->|] (2,.6) -- node[above=1mm] {10mm} ++(-10mm,0);
  }

\tikz{
  \draw [line width=1mm, -{Stealth[length=10mm, open, round]}]
    (0,0) -- (2,0);
  \draw [|<->|] (2,.6) -- node[above=1mm] {10mm} ++(-10mm,0);
  }
```

/pgf/arrow keys/width=⟨*dimension*⟩⟨*line width factor*⟩⟨*outer factor*⟩ (no default)

本选项指定箭头的宽度，键值中的参数作用与前一选项相同。

/pgf/arrow keys/width'=⟨*dimension*⟩⟨*length factor*⟩⟨*line width factor*⟩ (no default)

本选项指定箭头的宽度，参数 ⟨*length factor*⟩ 指定箭头宽度与长度之比，参数 ⟨*line width factor*⟩ 的作用如前一选项。

如果给出两个 `length` 选项，那么后一个 `length` 选项对 `width'` 有意义。例如，“`length=10pt, width'=5pt 2, length=7pt`”等价于 `length=7pt, width'=5pt 2`，箭头宽度就是 `5pt+2*7pt=19pt`.

/pgf/arrow keys/inset=⟨*dimension*⟩⟨*line width factor*⟩⟨*outer factor*⟩ (no default)

inset 尺寸指的是从箭头最左侧向箭头内部直到内凹顶点的长度。这个选项的参数意义与前面的类似。注意有的箭头，例如 Latex 箭头，因其本身的形状所限 (它没有内凹的顶点)，也就没有 inset 这个尺寸，因此这个选项对它无效。

/pgf/arrow keys/inset'=⟨*dimension*⟩⟨*length factor*⟩⟨*line width factor*⟩ (no default)

这个选项与 `width'` 类似。例如 Stealth 箭头的默认设置是 `inset'=0pt 0.325`，将内凹深度与长度之比保持为 0.325:1.

/pgf/arrow keys/angle=⟨*angle*⟩:⟨*dimension*⟩⟨*line width factor*⟩⟨*outer factor*⟩ (no default)

这个选项设置箭头尖角的角度与角的边长，其中参数 ⟨*angle*⟩ 设置角度，其余参数设置角的边长。在这个设置下，箭头的长度和宽度可以由“角度”和“角的边长”确定 (用三角函数计算)，但 inset 尺寸需要另外设置。

/pgf/arrow keys/angle'=⟨*angle*⟩ (no default)

这个选项设置箭头尖角的角度。

43.3.2 箭头的缩放

/pgf/arrows keys/scale=⟨*factor*⟩ (no default, initially 1)

当前文所述的尺寸选项处理完毕后，TikZ 再处理这个选项，将箭头按比例 ⟨*factor*⟩ 做缩放，即做位似变换，位似中心是路径端点。注意本选项对箭头的线宽、叠放箭头的间距 (`sep` 选项的设置) 无效。

/pgf/arrows keys/scale length=⟨*factor*⟩ (no default, initially 1)

这个选项类似 `scale`，只是只对箭头的 length，inset 起作用，对箭头的 width 无作用。

/pgf/arrows keys/scale width=⟨*factor*⟩ (no default, initially 1)

这个选项类似 `scale length`，只对箭头的 width 起作用。

43.3.3 圆弧箭头

有的箭头由圆弧组成，且圆弧长度 (角度) 可调。

/pgf/arrow keys/arc=⟨*degrees*⟩ (no default, initially 180)

设置圆弧的角度。

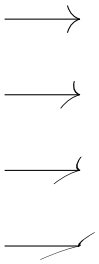
```
\tikz [ultra thick] {
  \draw [arrows = {-Hooks[scale=2]}] (0,2) -- (1,2);
  \draw [arrows = {-Hooks[arc=90,scale=2]}] (0,1) -- (1,1);
  \draw [arrows = {-Hooks[arc=270,scale=2]}] (0,0) -- (1,0);
  }
```

43.3.4 倾斜

可以让箭头像文字的 italic 形态那样倾斜。

/pgf/arrow keys/slant=⟨*factor*⟩ (no default, initially 0)

箭头的倾斜效果是使用“画布变换 (canvas transformation)”得到的。当 ⟨*factor*⟩ 比较大时，如下面例子中 slant=2，箭头会倾斜得比较严重，但箭头的尖端 (的最后一个像素) 始终保持在路径的端点处。



```
\tikz {
  \draw [arrows = {->[scale=2]}] (0,3) -- (1,3);
  \draw [arrows = {->[slant=.5,scale=2]}] (0,2) -- (1,2);
  \draw [arrows = {->[slant=1,scale=2]}] (0,1) -- (1,1);
  \draw [arrows = {->[slant=2,scale=2]}] (0,0) -- (1,0);
  }
```

43.3.5 Reversing, Halving, Swapping

/pgf/arrow keys/reverse (no value)

这个选项会使得箭头原地翻转，指向相反的方向。如果使用该选项两次，则会取消翻转。

```
\tikz [ultra thick]
  \draw [arrows = {-Stealth[reversed]}] (0,0) -- (1,0);\\
\tikz [ultra thick]
  \draw [arrows = {-Stealth[reversed, reversed]}] (0,0) -- (1,0);
```

/pgf/arrow keys/harpoon (no value)

这个选项只画出路径左侧的半个箭头，形似鱼叉。某些类型的箭头不支持本选项。


```
\tikz [ultra thick] \draw [arrows = {-Stealth[harpoon]}] (0,0) --
↪  (1,0);
```

/pgf/arrow keys/swap (no value)

这个选项与 harpoon 一起使用，只画出路径右半边的半个箭头。


```
\tikz [ultra thick] \draw [arrows = {-Stealth[harpoon,swap]}] (0,0) --
↪  (1,0);
```

/pgf/arrow keys/left (no value)

等效于 harpoon 选项。

/pgf/arrow keys/right (no value)

等效于 harpoon,swap 选项。

43.3.6 箭头颜色

箭头是个路径，例如 Stealth 这种箭头路径，它有自己的路径线条颜色、路径内部的填充色，这两种颜色通常与箭头所在路径的 draw 颜色一致。

```
\tikz\draw[very thick,draw=red,fill=cyan,-{Stealth[scale=1.5]}]
  (0,0)--++(-45:0.5)--++(45:1);
```

如果想让箭头有自己的颜色，可以使用下面的选项。

/pgf/arrow keys/color=⟨*color or empty*⟩ (no default, initially empty)

本选项可以为箭头单独设置颜色。color= 这一部分可以省略，只写出颜色名称。本选项指定的颜色 ⟨*color*⟩ 会用作箭头路径线条的颜色和箭头内部的填充色。如果本选项的值是空的，即写出 color=，那么箭头内部的填充色会与箭头路径线条的颜色一样。

```
\tikz\draw[very thick,draw=red,fill=cyan,-{Stealth[scale=2,blue]}]
  (0,0)--++(-45:0.5)--++(45:1);
```

```
\tikz [ultra thick]
  \draw [draw=red, fill=cyan,
    arrows = {-Stealth[length=10pt]}]
    (0,0) -- (1,1) -- (2,0);
```

上面例子中，由于箭头会把处于箭头内部的那一段路径裁掉，所以路径的填充色不能触及箭头的尖端。

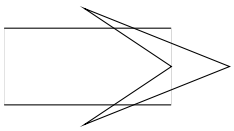
/pgf/arrow keys/fill=⟨*color or none*⟩ (no default)

这个选项指定箭头内部的填充色。fill=none 表示不填充颜色 (注意这不同于填充白色)。

```
\tikz {
  \draw [help lines] (0,-.5) grid [step=1mm] (1,.5);
  \draw [thick, red, arrows = {-Stealth[fill=white,length=15pt]}] (0,0)
  ↪  -- (1,0);
}
```

```
\tikz {
  \draw [help lines] (0,-.5) grid [step=1mm] (1,.5);
  \draw [thick, red, arrows = {-Stealth[fill=none,length=15pt]}] (0,0)
  ↪  -- (1,0);
}
```

箭头有 inset 尺寸的话，若箭头不填充颜色，可能会产生问题，下面是个极端的例子：



```
\tikz \draw [double distance=1cm,
  arrows = {-Stealth[length=2cm,width=1.6cm,inset=1.2cm,fill=none]}]
    (0,0) -- (3,0);
```

上面例子中，路径是双线，由于 inset 过深且双线间距过大，造成双线突出箭头外部。

/pgf/arrow keys/open (no value)

等效于 fill=none.

如果在箭头选项中同时使用 color 和 fill 选项，那么 fill 选项应该放在 color 选项之后，否则 fill 选项会被 color 选项重设。

如果当前主路径有填充颜色，那么 pgffillcolor 就是这个填充色的名称 (内部使用的别名)。下面例子中，箭头填充色设置为 pgffillcolor，使得箭头内部颜色与主路径填充色一致：

```
\tikz [ultra thick] \draw [draw=red, fill=red!50,
  arrows = {-Stealth[length=15pt, fill=pgffillcolor]}]
  (0,0) -- (1,1) -- (2,0);
```

43.3.7 线型

参数 line join, line cap, line width 对路径线条的外观有很大影响，箭头的路径线条也有这三种特征。

/pgf/arrow keys/line cap=⟨*round or butt*⟩ (no default)

设置箭头路径端点的“帽子”形态。如果箭头不是闭合的路径，它的端点就有默认的“帽子”，“帽子”可以用这个选项来修改。对于箭头来说只有 round 和 butt 两个选择。这个选项对箭头的各种尺寸没有影响，箭头最末端的点仍然是路径的端点。

```
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line cap=butt]}]
    (0,0) -- (1,0);\\
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line cap=round]}]
    (0,0) -- (1,0);
```

/pgf/arrow keys/line join=⟨*round or miter*⟩ (no default)

设置箭头路径上的角 (不包括路径端点) 的外缘形态，对于箭头来说只有 round(圆形)，miter(尖角形) 两个选择。这个选项对箭头的各种尺寸没有影响，箭头最末端的点仍然是路径的端点。

```
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line join=miter]}]
    (0,0) -- (1,0);\\
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line join=round]}]
    (0,0) -- (1,0);
```

/pgf/arrow keys/round (no value)

设置 line cap=round, line join=round 的简缩。

/pgf/arrow keys/sharp (no value)

设置 line cap=butt, line join=miter 的简缩。

/pgf/arrow keys/line width=⟨*dimension*⟩⟨*line width factor*⟩⟨*outer factor*⟩ (no default)

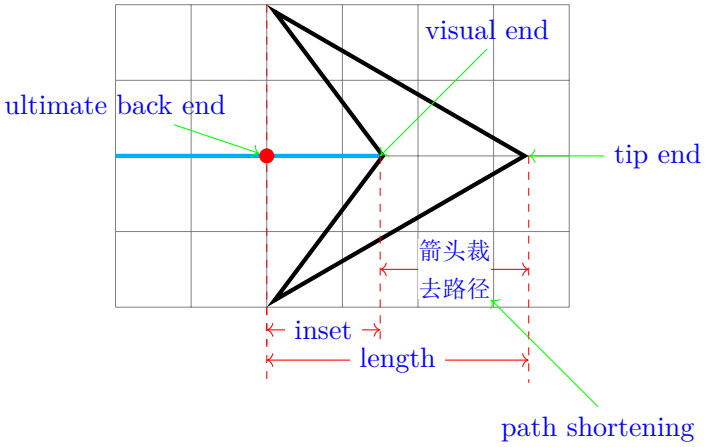
本选项设置箭头路径的线宽，各个参数的意义与选项 /pgf/arrow keys/length → P. 786 的相同。如果设置本选项为 0pt，那么对于封闭的箭头路径来说，就只能填充箭头内部。当箭头带有 bend 选项时，设置箭头线宽为 0pt 可能会比较好。

/pgf/arrow keys/line width'=⟨*dimension*⟩⟨*length factor*⟩ (no default)

本选项设置箭头路径的线宽。

43.3.8 Bending and Flexing

首先规定几个名词，参照下图



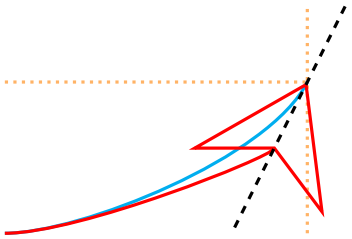
上面图中，首先画出网格，然后画出路径和箭头，箭头使用了 open 选项。箭头没有遮挡网格，却把处于箭头内部的那一段路径裁去了，被裁去的路径长度是 length - inset，这种裁去一段路径的行为称为 path shortening，即“裁线”。箭头最前端的那个像素是原来的路径端点，称之为 tip end. 路径与箭头轮廓线的公共点 (图中是箭头内凹的顶点) 称作 visual end.

一般情况下，箭头添加到路径上后，箭头的指向应当与箭头所处的路径端点处的切线方向一致。这对于直线段路径的情况比较容易实现，对于曲线路径就比较复杂。程序按照是否载入 bending 库的情况来分别决定箭头指向。

当不载入 bending 库时，默认使用选项 quick 来决定箭头指向：

/pgf/arrow keys/quick (no value)

下面以一段控制曲线为例来说明这个选项的作用。为了明显，箭头的尺寸设置得比较大。

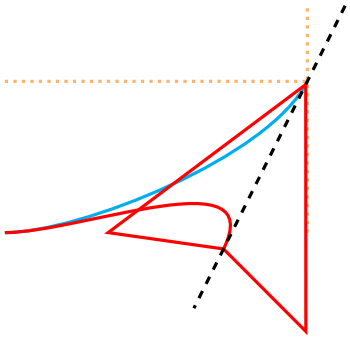


```
\begin{tikzpicture}[line width=1.2pt]
  \draw  [dotted,orange!60] (2,-2) -- (2,1) (-2,0) -- (2,0);
  \draw [cyan] (-2,-2) .. controls (-1,-2) and (1.5,-1) .. (2,0);
  \draw [red,-{Stealth[length=1.5cm,width=2cm,inset=0.5cm,open,quick]}]
    (-2,-2) .. controls (-1,-2) and (1.5,-1) .. (2,0);
  \draw [dashed] ($(2,0)!-1!(1.5,-1)$)--($(2,0)!2!(1.5,-1)$);
\end{tikzpicture}
```

上面例子中，红色控制曲线的端点处有个不填充颜色的 Stealth 箭头，箭头使用了 quick 选项。青色和红色的控制曲线的路径坐标是完全一样的，青色曲线是原本的形态。红色箭头的指向与青色曲线在点 tip end 处的切向是一致的 (注意 tip end 总是控制曲线的端点)，也就是图中黑色虚线的方向。但由于箭头的这种设置使得红色曲线偏离了原来的形态，这是 quick 选项的效果。

关于曲线偏离原来形态的问题，参考后文的 bending 库。

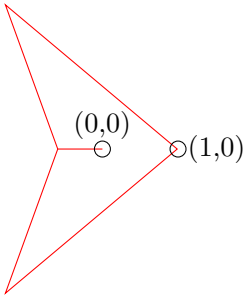
将上面例子中箭头的尺寸修改一下，得到下面比较极端的例子：



```
\begin{tikzpicture}[line width=1.2pt]
  \draw  [dotted,orange!60] (2,-2) -- (2,1) (-2,0) -- (2,0);
  \draw [cyan] (-2,-2) .. controls (-1,-2) and (1.5,-1) .. (2,0);
  \draw [red,-{Stealth[length=3cm,width=3cm,inset=0.5cm,open,quick]}]
    (-2,-2) .. controls (-1,-2) and (1.5,-1) .. (2,0);
  \draw [dashed] ($(2,0)!-1!(1.5,-1)$)--($(2,0)!3!(1.5,-1)$);
\end{tikzpicture}
```

可见，由于箭头尺寸过大造成了很意外的结果。

注意，如果不载入 bending 程序库，程序默认 quick 选项起作用 (即使箭头没有写出 quick 选项，程序也会按 quick 选项的效果画出箭头)。因此，箭头的尺寸应当与路径的长度相称。下面的例子中，线段长度是 1cm，明显小于箭头尺寸，意外地出现了一段线段：




```
\begin{tikzpicture}
  \draw [red,-{Stealth[angle=80:3cm,inset=0.7cm,open,quick]}]
    (0,0)--(1,0);
  \draw (0,0) circle(3pt) node [above] {(0,0)};
  \draw (1,0) circle(3pt) node [right] {(1,0)};
\end{tikzpicture}
```

另外，当串联多个箭头时，quick 选项还会导致箭头不在路径上的问题：

```
\begin{tikzpicture}[line width=1.2pt]
\draw [red!25,] (-1,0) -- (1,0) (1,-1)--(1,1);
\draw [cyan,-{[quick,sep=4pt]>>>>}]
  (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

TikZ Library bending

```
\usetikzlibrary{bending} % LaTeX and plain TeX
\usetikzlibrary[bending] % ConTeXt
```

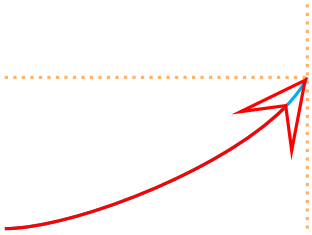
当载入 bending 库后，可以使用选项 flex, flex', bend 来调节箭头指向，其中的 flex 是默认选项 (除非指定其它选项)。

载入 arrows.meta 库后，最好也载入 bending 库，尤其是在计算两个路径的交点时，如果不载入 bending 库可能导致曲线偏离原本的形态，使得交点、曲线相离。前面 /pgf/arrow keys/quick →P. 791 选项下就是这样的例子。下面是 standalone 文类下的例子：

```
\documentclass[tikz]{standalone}
\usepackage{tikz}
\usetikzlibrary[intersections,
                arrows.meta,
                bending
               ]
\begin{document}
\begin{tikzpicture}
    \path [-Stealth,draw,name path=A] (-2,1)--+(6,0);
    \path [-Stealth,draw,name path=B] (0,0) ..controls +(-4,2)  and +(4,2)..(2,0);
    \fill [fill=red,name intersections={of=A and B,name=i}]
        (i-1) circle (1pt) (i-2) circle (1pt);
\end{tikzpicture}
\end{document}
```

如果其中不载入 bending 库，那么可以看出来，曲线会偏离交点（不是交点偏离曲线）。

/pgf/arrow keys/flex=⟨*factor*⟩ (default 1)



```
\begin{tikzpicture}[line width=1.2pt]
  \draw  [dotted,orange!60] (2,-2) -- (2,1) (-2,0) -- (2,0);
  \draw [cyan] (-2,-2) .. controls (-1,-2) and (1.5,-1) .. (2,0);
  \draw [red,-{Stealth[length=1cm,width=1cm,inset=0.5cm,open]}]
    (-2,-2) .. controls (-1,-2) and (1.5,-1) .. (2,0);
\end{tikzpicture}
```

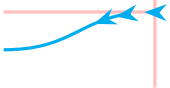
上面的例子是 flex 选项的默认效果：箭头不会像 quick 选项那样让路径曲线走样变形；在箭头尺寸适当的情况下，箭头绕点 tip end 旋转，使得 visual end 点恰好位于路径 (控制曲线) 上。这样会好看一些，不过好看的代价是—箭头方向也不再严格地指向曲线在端点处的切线方向。

如果需要让箭头方向尽量指向曲线在端点处的切线方向，可以调整本选项的值 ⟨*factor*⟩，这会使得箭头围绕其 tip end 点旋转。如果 flex=0，则箭头的指向与路径端点处的切线方向一致。如果 ⟨*factor*⟩ 是大于 0 的数值，则箭头顺时针旋转。如果 ⟨*factor*⟩ 是小于 0 的数值，则先把箭头旋转 180°，即把箭头变成它关于点 tip end 的中心对称图形，再将它逆时针旋转。

下面例子是 flex 取负值的情况：

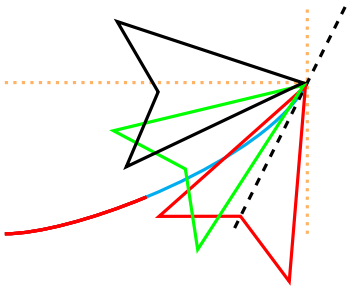
```
\begin{tikzpicture}[line width=1.2pt,>=Stealth]
  \draw [red!25,] (-1,0) -- (1,0) (1,-1)--(1,1);
  \draw [cyan,-{>>[flex=-2,sep=20pt]>}]
    (-1,-.5) .. controls (0,-0.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

如果在路径上串联多个箭头，将选项 flex 置于所有箭头符号之前，则该选项会对每个箭头起作用，不会像 quick 选项那样，出现箭头脱离路径曲线的情况：



```
\begin{tikzpicture}[line width=1.2pt,>=Stealth]
  \draw [red!25,] (-1,0) -- (1,0) (1,-1)--(1,1);
  \draw [cyan,-{[flex=-2,sep=4pt]>>>}]
    (-1,-.5) .. controls (0,-0.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

如果箭头尺寸相对于路径长度过大，也会出现问题：



```
\begin{tikzpicture}[line width=1.2pt]
  \draw  [dotted,orange!60] (2,-2) -- (2,1) (-2,0) -- (2,0);
  \draw [cyan] (-2,-2) .. controls (-1,-2) and (1.5,-1) .. (2,0);
  \draw [dashed] ($(2,0)!-1!(1.5,-1)$)--($(2,0)!2!(1.5,-1)$);

  \foreach \n / \c in {0/red,1/green,3/black}
    {\draw [red,-{Stealth[length=2.5cm,width=2cm,inset=0.5cm,open,flex=\n,color=\c]}]
      (-2,-2) .. controls (-1,-2) and (1.5,-1) .. (2,0);}
\end{tikzpicture}
```

在上面例子中，由于箭头尺寸过大，造成路径不能连接到 visual end 点 (箭头内凹的顶点)，使得路径与箭头之间有空白, 即使 `flex=1`，也不能让 visual end 点位于路径 (控制曲线) 上。

`/pgf/arrow keys/flex'`=⟨*factor*⟩ (default 1)

这个选项与 flex 类似，当 ⟨*factor*⟩ 等于 1 时，选项 `flex'` 使得箭头最后部的中间点“ultimate back end”位于路径上。不过 `flex=0` 与 `flex'=0` 等效。如果一个箭头没有 inset 尺寸，从而没有内凹的 visual end 点 (例如 Computer Modern Rightarrow 箭头)，那么就适合使用 `flex'` 选项来设置箭头的指向。

`/pgf/arrow keys/bend` (no value)

这个选项会使箭头轮廓线条随着路径的弯曲而弯曲，且箭头的指向会自动与路径的切向符合起来。在箭头的各个尺寸都适当的情况下，弯曲箭头比较美观，但如果箭头尺寸太大可能会走样：

```
\tikz \draw [red,line width=1mm,-{Stealth[bend,round,length=20pt]}]
  (0,-.5) .. controls (1,-.5) and (0.25,0) .. (1,0);\\
\tikz \draw [red,line width=1mm,-{Stealth[bend,length=40pt,
    inset=20pt,width=15pt]}]
  (0,-.5) .. controls (1,-.5) and (0.25,0) .. (1,0);
```

43.4 Arrow Tip Specifications

43.4.1 句法

作为针对箭头的选项，指定箭头样式的句法是：

⟨*start specification*⟩`-`⟨*end specification*⟩

即“⟨路径起点箭头样式⟩`-`⟨路径终点箭头样式⟩”，下面主要叙述路径终点箭头样式的设置。

路径端点的箭头样式设置的句法是：

`[`⟨*options for all tips*⟩`]`⟨*first arrow tip spec*⟩⟨*second arrow tip spec*⟩...

开头的方括号里的选项对后续的各个箭头都有效。方括号后的箭头又可以分为 3 种形式：

1. ⟨*arrow tip kind name*⟩`[`⟨*options*⟩`]`，例如 `Stealth[red]`，当某个箭头名称后面还有其它箭头符号、箭头名称等时，要注意保留该箭头名称后的方括号 (即使里面没有选项)。诸如“`Stealth >`”,“`Stealth Latex`”这种句式都是错误的，因为程序会把“`Stealth >`”看作箭头名称，但没有名称为“`Stealth`

>”的箭头。如果箭头名称后面有方括号，程序把方括号作为箭头名称结束的标志，如“Stealth[]>”，就不会出现这种无法判断名称的问题。

2. ⟨*shorthand*⟩[⟨*options*⟩]，这里的 ⟨*shorthand*⟩ 是用手柄 /.tip= 定义的 key，与上一种形式类似，也要在适当的位置用方括号来隔开箭头。
3. ⟨*single char shorthand*⟩[⟨*options*⟩]，这里的 ⟨*single char shorthand*⟩ 可以是“<，>，(，)，*，o，x 这几个符号之一，它们分别代表一种箭头。与前两种形式不同，这种符号箭头与后续的箭头之间并非必须用方括号来分隔。

程序对以上形式的箭头句式做如下处理，举例来说，假设把 abc[] 作为箭头句式写入文档，当程序处理到 a 这里时，首先检查从当前位置到下一个开方括号“[”之前的内容，即检查 abc，看看这个内容是否是 (属于以上列出的形式)⟨*arrow tip kind name*⟩ 或 ⟨*shorthand*⟩；如果是，执行之，否则，检查该内容的第一个符号，即检查 a，看看这个符号是否是 ⟨*shorthand*⟩ 或 ⟨*single char shorthand*⟩；如果是，执行之，然后按同样的规则检查剩余的内容，即检查 bc；否则，给出错误信息。

总结起来，可以用下面的例子来说明箭头句式是否正确：

- ->>>，正确，这是在路径末端加 3 个箭头。
- ->[]>>，正确，等效于上一个句式。
- -[]>>>，正确，等效于上一个句式。
- ->[]>[]>[]，正确，等效于上一个句式。
- ->Stealth，正确。
- -Stealth >，错误，因为 Stealth > 和字母 S 都不代表箭头.
- -Stealth[]>，正确。
- <Stealth-，正确，它是 -Stealth[]> 的翻转形式。
- -Stealth[length=5pt] Stealth[length=6pt]，正确，箭头名称后的方括号设置该箭头的长度。

在如下的箭头选项句式中

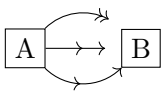
{[⟨*ops for all*⟩] {⟨*arrow1*⟩[⟨*ops1*⟩] ⟨*arrow2*⟩[⟨*ops2*⟩] ⟨*arrow3*⟩[⟨*ops3*⟩] ...}}

如果某个方括号里有 flex，flex'，bend 这 3 个选项之一，那么所有方括号里的 quick 选项都当作 flex 来处理。另外注意，如果某个箭头后面有方括号，还需要用花括号将所有箭头及其方括号选项括起来。

43.4.2 Specifying Paddings

/pgf/arrow keys/sep=⟨*dimension*⟩⟨*line width factor*⟩⟨*outer factor*⟩ (default 0.88pt .3 1)

这个选项可以用于所有箭头 (放在箭头序列之前的方括号里)，也可以只用作某一个箭头的选项，它在箭头的后面插入一个间隔，这样箭头之间或箭头与路径端点之间就不是紧紧相贴的，而是适当疏松的，会显得美观。注意，在相邻两个箭头之间，这个选项所造成的间隔不是空白 (箭头之间仍然有路径线条相连)，但在箭头与路径端点之间造成的间隔却是空白。



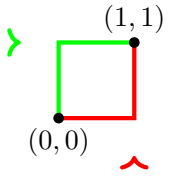

```
\tikz {
  \node [draw] (A) {A};
  \node [draw] (B) [right=of A] {B};
  \draw [-{>>[sep=6pt]}] (A) to [bend left=45] (B);
  \draw [-{>[sep=15pt]>}] (A) to [bend right=45] (B);
  \draw [-{[sep=6pt]>>}] (A) to (B);
}
```

下标线“_”是一个特殊的箭头，它不画出任何东西，它代表的长度为 0，但程序默认它带有一个 sep 选项，所以它的作用就是插入一个 sep 间距。它是一个 ⟨*single char shorthand*⟩，所以它与其前后的箭头之间不必用方括号来间隔。

```
\tikz \draw [->__________>] (0,0) -- (1,0);
```

显然，用“_”插入间距不如用 sep 选项来得快捷。

当给由符号 -| 或 |- 创建的路径添加箭头时需要注意箭头的位置，观察下面的图形：

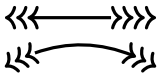



```
\begin{tikzpicture}
  \draw [red, ultra thick, -{>[sep=1.5cm]>[length=0pt]}]
    (0,0)-|(1,1);
  \draw [green, ultra thick, -{>[sep=1.5cm]>[length=0pt]}]
    (0,0)|-(1,1);
  \fill circle (2pt) node[below]{$(0,0)$};
  \fill (1,1) circle (2pt) node[above]{$(1,1)$};
\end{tikzpicture}
```

显然，箭头加到了虚拟的线段上了。

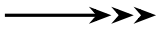
43.4.3 Specifying the Line End

有个 ⟨*single char shorthand*⟩ 符号“.”，即点号，它的作用是：把它与路径端点之间的那一段路径隐藏起来，造成间断效果。



```
\tikz [very thick] \draw [<<.<-.>>.>>] (0,0) -- (2,0);\\
\tikz [very thick] \draw [<<<.->.>>] (0,0) to [bend left] (2,0);
```

注意，如果将符号“.”放在某个非 ⟨*single char shorthand*⟩ 的箭头符号之后，还需要用花括号将所有箭头及其方括号选项，连同符号“.”都括起来，否则在解析箭头语句时可能会造成歧义。



```
\tikz [very thick] \draw [-{Stealth[] . Stealth[] Stealth[]}]
  (0,0) -- (2,0);
```

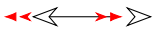
符号“.”之后可以带有方括号选项，但所带选项都无效。

43.4.4 定义箭头的简写形式

有时使用的箭头样式可能比较复杂，箭头带有很多选项，而这个箭头样式又要在多个地方使用，如果每次都写出那么多代码就显得过于繁琐。此时，可以把这个箭头样式保存在一个 key 中，用这个 key 代替复杂的代码。这用到下面的手柄。

Key handler ⟨*key*⟩**/.tip=**⟨*end specification*⟩

每当使用 ⟨*key*⟩ 时，⟨*key*⟩ 就会被 ⟨*end specification*⟩ 替换。这种设置是针对路径末端的箭头的，当在路径始端使用这个 ⟨*key*⟩ 时，箭头会自动翻转。



```
\tikz [xoo /.tip = {Stealth[sep] Latex[sep]},
    bar /.tip = {Stealth[length=10pt,open]}]
  \draw [{xoo[red] . bar}-{xoo[red] . bar}] (0,0) -- (2,0);
```

上面例子中，xoo[red] 的效果等效于 Stealth[sep,red] Latex[sep,red]，也就是说当 xoo[red] 展开为箭头设置时，颜色 red 对箭头 Stealth，Latex 适合“分配律”。

这样定义的 ⟨*key*⟩ 可以套嵌使用，即某个 ⟨*key*⟩ 可以用于其它 ⟨*key*⟩ 的定义中，例如：

```
\tikz [xoo /.tip = {Stealth[sep] Latex[sep]},
      bar /.tip = {xoo[length=10pt,open]}]
  \draw [bar-bar] (0,0) -- (3,0);
```

符号“<，>”设置箭头同时还指定箭头方向，“(，)”的作用类似。但是对于用箭头名称设置的箭头来说，在默认下，始端与末端的箭头指向相反：

```
\tikz \draw [>->] (0,0) -- (1,0);\\
\tikz \draw [Stealth-Stealth] (0,0) -- (1,0);
```

如果调用了 arrows.meta 库，那么默认“>”是 To(箭头名称，等效于 Computer Modern Rightarrow)的简缩。如果不载入 arrow.meta 库，To 相当于 to(旧库中的箭头名称)。

可以对“<, >”重新“赋值”，来一揽子地同时指定路径始端和末端的箭头样式，这用到类似下面的句法：

```
<-> /.tip ={⟨end arrow specification⟩}
>-> /.tip ={⟨end arrow specification⟩}
>-< /.tip ={⟨end arrow specification⟩}
<-< /.tip ={⟨end arrow specification⟩}
```

注意其中的 ⟨*end arrow specification*⟩ 是针对路径末端的箭头设置，在路径始端的箭头会被自动翻转。

```
\tikz [>-> /.tip = {Latex[length=10pt,red].Stealth[open,scale=2]}]
  \draw [>-<] (0,0) -- (2,0);
```

在上面例子中，用“>->”设置了始端和末端的箭头样式，在画路径时，路径末端的箭头符号用“<”，指示各个箭头原地反向 (相对于手柄定义中的“>”)；路径始端的箭头用“>”，结果箭头整体翻转 (相对于手柄定义中的“<”)。注意箭头样式设置中使用了间断符号“.”，由于 Latex 箭头后部没有内凹，所以间断符号“.”的效果只在路径始端显示出来。

这个手柄在文件《pgfcorearrows.code.tex》中定义：

```
% When you write "my name/.tip = arrow spec", this has the same
% effect as writing \pgfdeclarearrow{name = my name, means = arrow spec}.

\pgfkeys{/handlers/.tip/.code=%
  \expandafter\expandafter\expandafter\pgf@arrows@unravel\pgfkeyscurrentpath/
  ↪  \pgf@stop%
  \expandafter\pgf@arrows@key@call\expandafter{\pgf@arrows@unravelled}{#1}}
\def\pgf@arrows@unravel#1/{\pgfutil@ifnextchar\pgf@stop{\def\pgf@arrows@unravelled
↪  {#1}\pgfutil@gobble}{\pgf@arrows@unravel}}
\def\pgf@arrows@key@call#1#2{\pgfdeclarearrow{name={#1},means={#2}}}
```

这个手柄的处理是：

```
\pgfkeys{/a/b/c/.tip={⟨end arrow specification⟩}}
导致
\pgfdeclarearrow{name={c},means={⟨end arrow specification⟩}}
```

它定义一个 shorthand 箭头。

/tikz/>=⟨*end arrow specification*⟩ (no default)

这是 <->/.tip=⟨*end arrow specification*⟩ 的简写。如果把 ⟨*end arrow specification*⟩ 设为 {}，或者将它空置，则指定的是“空箭头”，然后再用 -> 画箭头时，画出的是“空箭头”，即没有箭头。


```
\tikz[>=] \draw [->,thick] (0,0)--(1,0);
```

这个选项的定义是：

```
\tikzoption{>}{\pgfdeclarearrow{name=<->,means={#1}}}%
```

43.4.5 Scoping of Arrow Keys

/tikz/arrows=[⟨*arrow keys*⟩] (no default)

这里的 ⟨*arrow keys*⟩ 是箭头选项。当某个环境带有这个选项后，环境内的任何一个路径命令的箭头都会带上选项 ⟨*arrow keys*⟩, 参考 \pgfsetarrows → P. 327, \pgf@arrows@options@scope → P. 328. 例如：

```
\tikz[arrows={[sep=10pt,bend]}]{
  \draw [red](0,0) to [bend left] (1,1);
  \draw [-{Stealth[]>}](0,0) to [bend left] (1,1);
}
```

这个选项的定义是：

```
\tikzoption{arrows}{\tikz@processarrows{#1}}%
\def\tikz@processarrows#1{%
  \def\tikz@current@arrows{#1}%
  \def\tikz@temp{#1}%
  \ifx\tikz@temp\pgfutil@empty%
  \else%
    \pgfsetarrows{#1}%
  \fi%
}%

\def\tikz@current@arrows{-}%
```

选项 `arrows=`⟨*arrow set*⟩ 的参数 ⟨*arrow set*⟩ 有以下几种形式：

- 空的，此时没有箭头
- 如果非空，按 `\pgfsetarrows`→P. 327 的处理，⟨*arrow set*⟩ 可以是：
  - `[`⟨*arrow keys*⟩`]`
  - `[-`⟨*something*⟩`]`
  - ⟨*start arrow spec*⟩`-`⟨*end arrow spec*⟩

下面总结一下各种箭头选项的作用次序：

1. 首先是选项的默认值，即 `/pgf/@arrows decl/defaults`→P. 323 的设置。
2. `arrows=[`⟨*keys*⟩`]` 中的选项设置。
3. 箭头序列前面方括号里的选项设置，如 `[`⟨*options*⟩`]>>>` 中的 ⟨*options*⟩
4. shorthand 箭头的选项。
5. 单个箭头后面方括号里的选项设置。

后作用的选项对前作用的选项有“优先权”。

把上面的例子与下面的例子做对比：

```
\tikz[arrows=Bar-{Stealth[sep=10pt]Bar}]{
  \draw (0,0) to [bend left] (1,1);
  \draw [>=,->](0,-.5) -- ++(1,0);
}
```

这个例子中第二个 `\draw` 命令使用选项 `>=` 将箭头设为空，然后用选项 `->` 画一个空箭头，空箭头对选项 `arrows=` 的设置有“优先权”，所以路径上没有箭头。

43.5 Reference: Arrow Tips

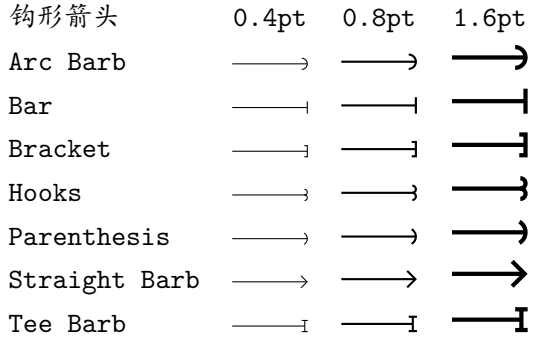
涉及箭头的库有 arrows，arrows.spaced，arrows.meta，前两个是旧的库。新的库 arrows.meta 定义了很多“标准箭头”，它们具有多种属性，可以比较细致地改变其外观。

唯一不默认自动加载这个库的原因是为了与旧版的 Ti*k*Z 兼容。当然，arrows.meta 可以与旧的 arrows，arrows.spaced 库同时加载使用。

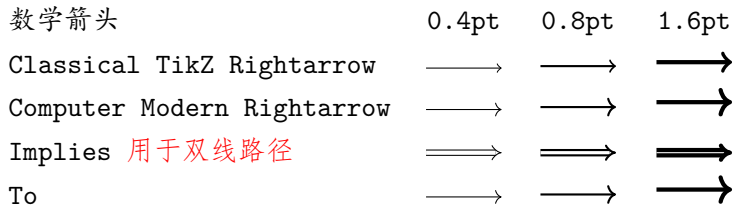
arrows.meta 定义的箭头可以分为以下几类：

- 钩形箭头 (barbed arrow tips)，其外形为开路径，没有填充特性，fill 选项对其无效。

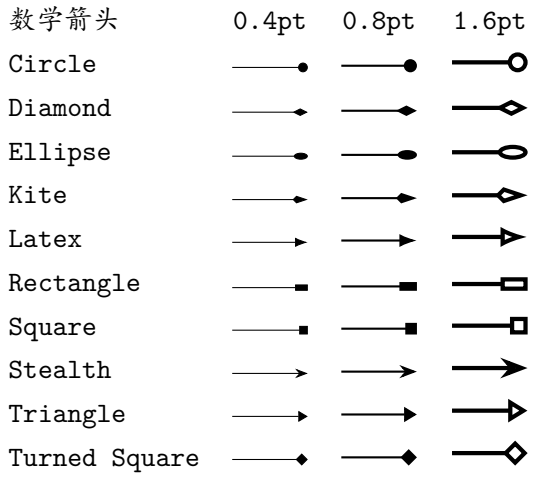
在默认下，若路径线宽为 0.4pt，则这种箭头的宽度是 (11pt 的 Computer Modern 字体中的) 字母“x”的高度。



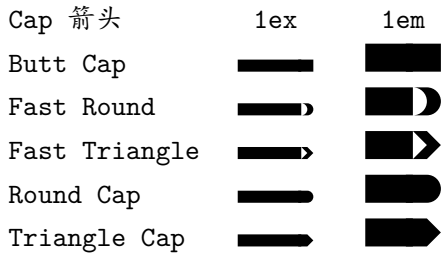
- 数学箭头 (Mathematical arrow tips)，这种箭头属于钩形箭头，但把它们单独列出，其外形与 TEX 的数学模式种的 `\to` 的外形一样。



- 几何箭头 (Geometric arrow tips)，是由几何图形做的箭头，并且其内部是默认填充的，可以用 `open` 选项取消填充。



- Cap 箭头 (Cap arrow tips)，路径的端点有 3 种基本的帽子：`round`，`rectangular`，`butt`，除了这 3 种，还可以使用以下箭头 cap:



选项 `open` 对 Cap 箭头无效。

- 射线箭头 (Rays)，其名称是 Rays，其形态是由一点发出的数条线段，象征放射状射线。默认它有 4 条射线。它可以带有选项 `n=`⟨*number*⟩ 来设置射线数目。

`/pgf/arrow keys/n`=⟨*number*⟩ (no default, initially `4`))

这个选项设置 Rays 箭头的射线数目，⟨*number*⟩ 应当是正整数。

射线箭头	0.4pt	0.8pt	1.6pt
Rays			
Rays[n=8]			
Rays[n=5]			

关于以上各种箭头的详细讲解参考 §16.5.1，§16.5.2，§16.5.3，§16.5.4，§16.5.5.

第四十四章　Nodes and Edges

44.1　Overview

通常, 一个 node 是“形状”与“内容”的结合体, 它的形状可以是矩形、圆形、或其它形状，而且形状上的某些点可以被引用。在形状内部可以放置盒子，盒子里可以是文字、LaTeX 的命令、环境等内容. node 是添加到路径上的, 但 node 不属于被添加的路径, 因此用于路径的某些选项, 例如 `scale`, `rotate`, `draw`, `fill` 等, 一般情况下对路径上的 node 无效.

44.2　Nodes and Their Shapes

44.2.1　Node 命令的句法

`\node`

这是 `\path node` 的简写.

`\path ... node`⟨*foreach statements*⟩`[`⟨*options*⟩`]`(⟨*name*⟩)`at`(⟨*coordinate*⟩)`:`
⟨*animation attribute*⟩`={`⟨*options*⟩`}{`⟨*node contents*⟩`}...;`

在这个句法中，从 `node` 到闭花括号 `}` 之间的东西一般情况下都在这个 `node` 的辖域内。

下面对这个命令做解释。

44.2.1.1　句法中各部分的次序

在 `node` 与开花括号 `{` 之间的几个部分, 即 ⟨*foreach statements*⟩, `[`⟨*options*⟩`]`, (⟨*name*⟩), `at`(⟨*coordinate*⟩), ⟨*animation attribute*⟩`={`⟨*options*⟩`}`, 都是可选的（根据需要可有可无的）。

如果使用 ⟨*foreach statements*⟩, 那么这一部分必须紧跟在 `node` 之后。其它几个部分的次序是随意的。

(⟨*name*⟩) 是 `node` 的名称, 如果给出两个“(⟨*name*⟩)”（例如 `(n1) (n2)`），则后一个名称有效。

如果给出两个“`at`(⟨*coordinate*⟩)”，则后一个坐标有效。⟨*coordinate*⟩ 会被命令 `\tikz@scan@one@point` 解析，所以 ⟨*coordinate*⟩ 可以是多种形式的坐标，例如相对坐标。

如果给出多个“`[`⟨*options*⟩`]`”，它们的作用会累计。

44.2.1.2　node 的内容

上面句法末尾的花括号和分号是必须有的，花括号内的部分 ⟨*node contents*⟩ 是 node 的内容。⟨*node contents*⟩ 内可以使用各种各样的内容，甚至可以使用脆弱命令（例如 `\verb`），在 ⟨*node contents*⟩ 内允许改变符号类别.

如果在“`[`⟨*options*⟩`]`”中使用下一个选项，则应去掉“`{`⟨*node contents*⟩`}`”:

/tikz/node contents=⟨*node contents*⟩ (no default)

这个 key 用在“[⟨*options*⟩]”中，设置 node 的内容。使用这个选项后，闭方括号“]”之后的代码都不被看作是属于当前 node 的代码，并且此时的 ⟨*node contents*⟩ 中可能不允许使用脆弱命令，因为在读取选项时，代码的类别就已经是固定的了。

本选项的参数 ⟨*node contents*⟩ 会被保存到宏 \tikz@node@content 中：

```
\tikzset{
  node contents/.code=\def\tikz@node@content{#1},
  pic type/.code=\def\tikz@node@content{#1}, % alias
}%
```

当 node contents 所处的方括号里的所有选项被处理完毕后，就会再处理 \tikz@node@content：

```
\expandafter\tikz@@scan@fig\expandafter{\tikz@node@content}%
```

其效果相当于

```
node [⟨options⟩]{⟨expandafter 展开的 \tikz@node@content⟩}
```

这就导致针对当前 node 的解析过程的结束，比较：


```
\tikz{
    \path (0,0)--(1,1)node[pos=0.5,shape=circle][draw=red]{A};
}
```

上面例子中，[draw=red] 是属于 node 的选项。

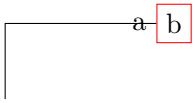

```
\tikz{
    \path (0,0)--(1,1)
        node[pos=0.5,shape=circle,node contents=A][draw=red];
}
```

上面例子中，只有第一个方括号是属于 node 的选项，因为其中含有 node contents，而 [draw=red] 是针对“--”的选项。当对 node 的解析过程结束时，会执行 \tikz@do@after@node，通常它等效于 \tikz@scan@next@command，即扫描下一个路径成分。当 \tikz@scan@next@command 遇到 [draw=red] 时，会把它看作是 \path 的选项。

44.2.1.3 指定 node 的位置

约定两个概念：“锚”和“锚定点”。打个比方，把一个 node 看作是一艘船，这艘船有多个抛锚口；把锚抛到水底固定后，锚所抓住的点是“锚定点”；尽管一个 node 有多个抛锚口，但有一个锚，所以只有一个锚定点；联系抛锚口与锚定点的是锚链，如果你不指定锚定点与抛锚口的距离，那么就默认这个距离是 0pt；如果你希望锚定点与抛锚口的距离不是 0pt，那么你不仅要指定这个距离是多少，还要指定船相对于锚定点在什么方位。这里 node 的 anchor 位置，通常指的是“抛锚口”。这个比方不是那么完美，但基本可以说明其中的意思。

如果你不指明 node 的锚定点，就把 node 之前出现的坐标位置作为 node 的锚定点。一个坐标点后可以跟随多个 node 语句，这些 node 的锚定点都是这个坐标点。




```
\tikz \draw (0,0) |- (2,1)
  node [left] {a}
  node [draw=red, anchor=west]{b};
```

at(⟨*coordinate*⟩) 这一部分指定 node 的锚定点。也可以在 [⟨*options*⟩] 中用下面的 key 指定 node 的锚定点：

/tikz/at=⟨*coordinate*⟩ (no default)

注意在“[⟨*options*⟩]”中用这个选项时，如果 ⟨*coordinate*⟩ 中有逗号，就用花括号要把 ⟨*coordinate*⟩ 括起来，例如 at={(1,1)}，否则 ⟨*coordinate*⟩ 中的逗号会引起错误。

⟨*coordinate*⟩ 会被命令 `\tikz@scan@one@point` 解析，所以 ⟨*coordinate*⟩ 可以是多种形式的坐标，例如相对坐标。

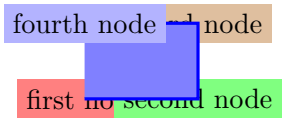
```
\tikzoption{at}{\tikz@scan@one@point\tikz@set@at#1}%
\def\tikz@set@at#1{\def\tikz@node@at{#1}}%
```

可见本选项将解析得到的坐标保存在 `\tikz@node@at`。

在路径上添加 node，如果路径与 node 有交叠，那么是路径遮挡 node, 还是 node 遮挡路径呢？如果先画出路径再画出 node，那就是 node 遮挡路径；如果先画出 node 再画出路径，就是路径遮挡 node. 遮挡别人的处于前端 (front)，被遮挡的处于后端 (behind)。通常情况下是 node 遮挡路径，二者的遮挡关系可以用下面的 key 调整：

`/tikz/behind path` (no value)

对于当前路径上的各个 node 来说，凡是带有这个选项的 node 属于同一组，在画出路径之前先画出这一组 nodes, 因此路径可能会遮挡这一组 nodes. 而对于这一组内的 nodes 来说，依照它们出现的次序依次画出它们，因此它们之间也可能会出现遮挡关系。实际上, 这一组 nodes 会在关于路径的 “pre-actions” 被执行前画出，参考 `/tikz/preaction` → P. 781.

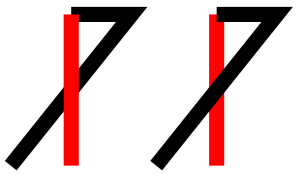



```
\tikz \fill [fill=blue!50, draw=blue, very thick]
  (0,0) node [behind path, fill=red!50] {first node}
  -- (1.5,0) node [behind path, fill=green!50] {second node}
  -- (1.5,1) node [behind path, fill=brown!50] {third node}
  -- (0,1) node [ fill=blue!30] {fourth node};
```

注意这个选项只能让 node 处于当前路径的后端，如果想让 node 处于其它路径的后端就应该使用图层 (layer)，参考 background 库, `\pgfdeclarelayer` → P. 358.
本选项的定义是：

```
\tikzset{
  behind path/.code=\def\tikz@whichbox{\tikz@figbox@bg},
  in front of path/.code=\def\tikz@whichbox{\tikz@figbox}
}%
\def\tikz@whichbox{\tikz@figbox}%
```

在 Ti*k*Z 解析 node 的过程中，node 会被放入盒子 `\tikz@whichbox` 中，选项 `behind path` 让盒子 `\tikz@whichbox` 等于盒子 `\tikz@figbox@bg`, 导致 node 被放入盒子 `\tikz@figbox@bg` 中。当 Ti*k*Z 遇到路径结束符号——分号 ; 时，就执行命令 `\tikz@finish`, 在此命令的展开过程中，先插入盒子 `\tikz@figbox@bg` 的内容，再用命令 `\pgfusepath` → P. 295 画出主路径，再插入盒子 `\tikz@figbox` 的内容。所以主路径遮挡盒子 `\tikz@figbox@bg`, 而盒子 `\tikz@figbox` 遮挡主路径。
另外，由 edge 操作 创建的路径也会被放入盒子 `\tikz@whichbox` 中，所以本选项可以对 `edge` 操作有作用。



```
\tikz\draw[line width=2mm](0,0)--(1.6,2)--(0.8,2)edge[red](0.8,0);
\tikz{
  \draw [line width=2mm,](0,0)--(1.6,2)--(0.8,2)
    [behind path] edge [line width=2mm,red](0.8,0);
}
```

注意上面例子中要把选项 `behind path` 放在 `edge` 之前执行，也就是说，先让盒子 `\tikz@whichbox` 等于盒子 `\tikz@figbox@bg`, 再执行 `edge` 操作，否则没有这种效果。

`/tikz/in front of path` (no value)

这个 key 的效果与 `behind path` 相反，这是 Ti*k*Z 的默认行为。

44.2.1.4 node 的名称

(⟨*name*⟩) 这一部分设置 node 的名称，有了名称后就可以用名称来引用 node. 也可以用下面的选项为 node 设置名称：

/tikz/name=⟨*node name*⟩ (no default)

用这个 key 设置 node 的名称，可供稍后引用。这个 key 设置的是“high-level”名称，驱动程序不会去“识别”它，所以 ⟨*node name*⟩ 的构成比较随意，其中可以含有空格、数字、字母、汉字、下标线等等，但不能包含逗号、点句号、冒号、分号等标点，因为逗号用于分隔坐标数据、分隔选项，点句号用于引用 node 坐标，冒号用于指定极坐标点。

```
\tikzset{
  name/.code={\edef\tikz@fig@name{\tikz@pp@name{#1}}\let\tikz@id@name
  ↪  \tikz@fig@name},%
  name prefix/.initial=,%
  name suffix/.initial=%
}%
\def\tikz@pp@name#1{\csname pgfk@/tikz/name prefix\endcsname#1\csname
↪  pgfk@/tikz/name suffix\endcsname}%
```

可见选项 name 会把 node 的全名，即“前缀”+“本名”+“后缀”的形式，保存到 \tikz@fig@name 中。

/tikz/alias=⟨*another node name*⟩ (no default)

给 node 另外起一个名字，多次使用这个 key 可以给 node 起多个名字，这些名字都可以用来引用 node. 使用 \path ... node also →P. 834 也可以给 node 另起一个名字。


```
\begin{tikzpicture}
  \node (A) [draw,alias=AA, alias=AAA] {\Huge A};
  \draw [->](A.north east) to[bend right] (AA.north west);
  \draw [->](AA.south west) to[bend right] (AAA.south east);
\end{tikzpicture}
```

此选项的定义是：

```
\tikzset{alias/.code={%
    \tikz@fig@mustbenamed
    \begingroup
    \toks0=\expandafter{\tikz@alias}%
    \edef\pgf@temp{\noexpand\pgfnodealias{\tikz@pp@name{#1}}{
    ↪  \noexpand\tikz@fig@name}}%
    \toks1=\expandafter{\pgf@temp}%
    \xdef\pgf@marshal{%
        \noexpand\def\noexpand\tikz@alias{\the\toks0 \the\toks1 }%
    }%
    \endgroup
    \pgf@marshal
}}%
```

可见这个选项调用 \pgfnodealias →P. 464 来定义 node 的别名，本选项全局地重定义 \tikz@alias，每当用户使用本选项时，它就把别名添加到这个宏中。另外：

\tikz@fig@mustbenamed

此命令的定义是：

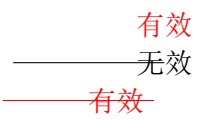
```
\def\tikz@fig@mustbenamed{%
  \ifx\tikz@fig@name\pgfutil@empty%
    % Assign a dummy name
```

```
    \global\advance\tikz@fig@count by1\relax
    \edef\tikz@fig@name{tikz@f@\the\tikz@fig@count}%
    \let\tikz@id@name\tikz@fig@name%
  \fi%
}%
```

可见命令 \tikz@fig@mustbenamed 的作用是确保 node 有名称，如果用户没有为 node 提供名称，那么本命令先将计数器 \tikz@fig@count 的值全局地加 1，再用其值来构造 node 的名称，其名称是 tikz@f@⟨*\tikz@fig@count 的当前值*⟩。

44.2.1.5 node 的选项

node 之后的选项设置“[⟨*options*⟩]”只在该 node 的辖域内有效，其它地方的选项可能对该 node 有效，也可能无效。在 node 之后可以设置多个方括号选项，但是注意选项 /tikz/node contents → P. 802.




```
\begin{tikzpicture}
  \path [color=red] (0,0.5)--(2,0.5) node {有效};
  \path [draw] (0,0)--(2,0) node {无效};
  \node [color=red,left] at(2,-0.5)
    {\tikz \draw (0,0)--(2,0) node {有效};};
\end{tikzpicture}
```

44.2.1.6 node 的形状

每个 node 都有自己的形状 (shape)，通常一个形状 (shape) 是用诸多命令定义的复杂路径，每个形状都有自己的名称，参考 \pgfnode → P. 463. 在默认下，node 的形状是名称为 rectangle 的 shape，即矩形。预定义的形状有 rectangle, circle, coordinate. 调用与 shapes 有关的库（例如 shapes.geometric, shapes.symbols, shapes.callouts, shapes.misc, shapes.arrows 等）后，可以使用程序库提供的更多形状。也可以自定义形状，参考 \pgfdeclareshape → P. 436.

如果想让 node 使用其它的形状，只需要在 [⟨*options*⟩] 中写出形状的名称，需要下面的选项：

/tikz/shape=⟨*shape name*⟩ (no default, initially rectangle)

这个 key 为 node 选定形状 (shape)，可以省略“shape=”，只写出 ⟨*shape name*⟩.

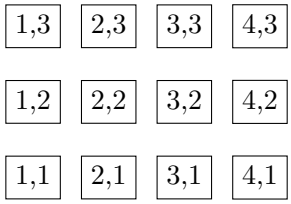
```
\tikzoption{shape}{\edef\tikz@shape{#1}}%
```

44.2.1.7 把 node 做成动画

当写出“:⟨*animation attribute*⟩=⟨*options*⟩”这一部分时，node 会出现动画效果。

44.2.1.8 node 中的 foreach 语句

注意 node 之后的 foreach 不带反斜线“\”. node 之后可以使用具有套嵌结构的 foreach 语句，能创建多个 node. foreach 语句的用法参考 \foreach → P. 166.



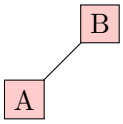

```
\tikz \node foreach \x in {1,...,4}
  foreach \y in {1,2,3}
  [draw] at (\x,\y) {\x,\y};
```

44.2.1.9 node 的样式

一个样式 (style) 实际上某些选项的组合体, 使用一个样式就相当于使用一组选项, 这会带来便利.

/tikz/every node (style, initially empty)

在这个 style 的有效范围内，它会把它所含的选项添加到所有 node 的开头。node 的选项会按照次序排成一个 “选项序列”，依次执行。这个 style 所含的选项总是处于选项序列的开头。




```
\begin{tikzpicture}[every node/.style={draw,fill=red!20}]
  \draw (0,0) node {A} -- (1,1) node {B};
\end{tikzpicture}
```

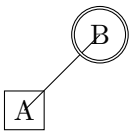
注意，执行样式 every node 的是命令 \node(实际是\tikz@normal@fig)，即样式 every node 在命令 \node 这里，通过命令：

```
\tikzset{every node/.try}%
```

发挥作用。

/tikz/every ⟨*shape*⟩ **node** (style, initially empty)

这个 style 类似 every node, 不过只是针对形状为 ⟨*shape*⟩ 的 node.




```
\begin{tikzpicture}
  [every rectangle node/.style={draw},
   every circle node/.style={draw,double}]
  \draw (0,0) node[rectangle] {A}
    -- (1,1) node[circle] {B};
\end{tikzpicture}
```

当 node 的所有选项都处理完毕，确定 node 的形状后，才会执行这个样式：

```
\tikzset{every \tikz@shape\space node/.try}%
```

/tikz/execute at begin node=⟨*code*⟩ (no default)

这个选项使得 ⟨*code*⟩ 在 node 的内容的开端处被执行，也就是把 ⟨*code*⟩ 插入到 node 内容的开头。如果多次使用本选项，则其作用累计，所给出的 ⟨*code*⟩ 会被依次执行。

/tikz/execute at end node=⟨*code*⟩ (no default)

这个选项使得 ⟨*code*⟩ 在 node 的内容的结尾处被执行，也就是把 ⟨*code*⟩ 插入到 node 内容的结尾。如果多次使用本选项，则其作用累计，所给出的 ⟨*code*⟩ 会被依次执行。

ABCD

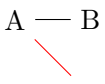
```
\begin{tikzpicture}
  [execute at begin node={A},
   execute at end node={D}]
  \node[execute at begin node={B}] {C};
\end{tikzpicture}
```

44.2.1.10 node 名称的前缀和后缀

用下面的选项规定 node 名称的前缀或后缀。

/tikz/name prefix=⟨*text*⟩ (no default, initially empty)

在这个 key 的有效范围内，它给每个 node 的名称规定前缀 ⟨*text*⟩. 这个 key 可以用作 scope 环境的选项，给当前 scope 环境内的所有 node 名称加前缀；当在这个 scope 环境内引用 (该环境的) node 时，只需要使用 node 的 “本名”(不必加前缀)；当超出这个 scope 环境范围并引用该环境内的 node 时，就应使用 node 的 “全名”，即在名称前要带上前缀 (前缀与名称之间不需要额外的分隔符号)。




```
\tikz {
  \begin{scope}[name prefix = top-]
    \node (A) at (0,1) {A};
    \node (B) at (1,1) {B};
    \draw (A) -- (B);
  \end{scope}
  \begin{scope}[name prefix = bottom-]
    \node (A) at (0,0) {A};
    \node (B) at (1,0) {B};
    \draw (A) -- (B);
  \end{scope}
  \draw [red] (top-A) -- (bottom-B);
}
```

注意，如果把样式 `every node/.style={name prefix=`⟨*text*⟩`}` 用作环境选项，那么在该环境内部引用（该环境内的）node 时，所引用的 node 名称要加前缀，这是因为，例如：

```
\tikz{
  \begin{scope}[every node/.style={name prefix=PREFIX}]
    \node (name) at(1,1) {};
  \end{scope}
}
```

等效于

```
\tikz{
  \begin{scope}
    \node (name)[name prefix=PREFIX] at(1,1) {};
  \end{scope}
}
```

这是因为样式 `/tikz/every node` → P. 806 在 `\node` 命令那里被执行。


```
\tikz {
  \begin{scope}[every node/.style={name prefix = pre fix }]
    \node (A) at (0,1) {A};
    \node (B) at (1,1) {B};
    \draw (pre fixA) -- (pre fixB);
  \end{scope}
}
```

上面例子还表明，选项 `name prefix=`⟨*text*⟩ 会把 ⟨*text*⟩ 中位于开头和结尾的空格忽略掉。

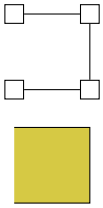
`/tikz/name suffix=`⟨*text*⟩ (no default, initially empty)

这个 key 类似 `name prefix`, 它给 node 名称规定后缀。

44.2.2 预定义的形状

如前述，预定义的 node 形状有 rectangle, circle, coordinate. 命令 `\coordinate` 创建的是 “坐标”—实际上就是一种简化版的 node. 在几何上，坐标（coordinate）对应没有内部尺寸的点，与这个观念相对应，当一个 node 的形状是 coordinate 时，就认为它的尺寸为 0, 它不能容纳任何内容，也没有自己的坐标系统（如 anchor 位置），它的作用是给一个坐标点命名，以便于引用该点。

通常，当用线条连接两个 node 时，线条的起点和终点都位于 node 的边界上，而不是位于 node 的中心点。当用线段把数个 node 相继连接起来后，所得到的并不是一个 “连续的” 路径，对于这样路径的填充效果并不好。但用线段把数个 coordinate 相继连接起来后却可以填充：



```
\begin{tikzpicture}[every node/.style={draw}]
  \path[yshift=1.5cm,shape=rectangle]
    (0,0) node(a1){} (1,0) node(a2){}
    (1,1) node(a3){} (0,1) node(a4){};
  \filldraw[fill=yellow!80!black]
    (a1) -- (a2) -- (a3) -- (a4) -- cycle;
  \path[shape=coordinate]
    (0,0) coordinate(b1) (1,0) coordinate(b2)
    (1,1) node (b3) {b3} (0,1) node (b4) {b4};
  \filldraw[fill=yellow!80!black]
    (b1) -- (b2) -- (b3) -- (b4);
\end{tikzpicture}
```

上面例子中，第一个 \filldraw 命令没有填充效果。

44.2.3 coordinate

\coordinate

这是 \path coordinate 的简写.

\path ... coordinate[⟨*options*⟩](⟨*name*⟩)at(⟨*coordinate*⟩)...;

等价于

```
\node[shape=coordinate,⟨options⟩](⟨name⟩)at(⟨coordinate⟩){};
```

在《tikz.code.tex》中，coordinate 句法的定义是：

```
\def\tikz@coordinate ordinate{%
  \pgfutil@ifnextchar[{\tikz@@coordinate@opt}{\tikz@@coordinate@opt[]}}%

\def\tikz@@coordinate@opt[#1]{%
  \pgfutil@ifnextchar({\tikz@@coordinate[#1]}
  {\tikz@fig ode[shape=coordinate,#1]{}}}%}%

\def\tikz@@coordinate[#1](#2){%
  \pgfutil@ifnextchar a{\tikz@@coordinate@at[#1](#2)}
  {\tikz@fig ode[shape=coordinate,#1](#2){}}}%

\def\tikz@@coordinate@at[#1](#2)a{%
  \pgfutil@ifnextchar t{\tikz@@coordinate@@at[#1](#2)a}%
  {\tikz@fig ode[shape=coordinate,#1](#2){}a}%
}%

\def\tikz@@coordinate@@at[#1](#2)at#3({%
  \def\tikz@coordinate@caller{\tikz@fig ode[shape=coordinate,#1](#2)at}%
  \tikz@scan@one@point\tikz@@coordinate@at@math(%
}%

\def\tikz@@coordinate@at@math#1{%
  \pgf@process{#1}%
  \edef\tikz@temp{(\the\pgf@x,\the\pgf@y)}%
  \expandafter\tikz@coordinate@caller\tikz@temp{}%
}%
```

按以上定义，coordinate 句法的形式可以是：

coodinate [⟨*options*⟩] (⟨*name*⟩) at ⟨*anything*⟩ (⟨*something*⟩)

其中的 [⟨*options*⟩] 与 at (⟨*something*⟩) 是可选的，而 ⟨*anything*⟩ 会被忽略。注意，这个句法中各个组成部分的次序是不可调换的 (这一点与 node 句法不一样)。

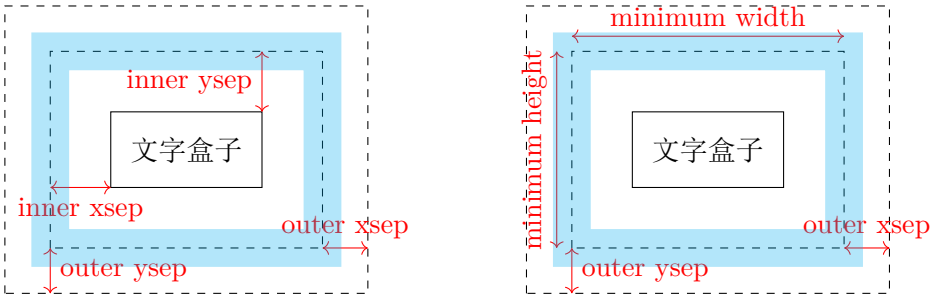
还要注意，当给出 at (⟨something⟩) 这一部分时，at 后面的圆括号是必须明确写出的，因为命令 \tikz@@coordinate@@at 以开圆括号 (为参数定界符号。此时对 (⟨something⟩) 的处理是：

- 先用 \tikz@scan@one@point →P. 710 解析 (⟨something⟩)，得到一个基本层的坐标，主要是寄存器 \pgf@x, \pgf@y 的值，
- 然后把 (\the\pgf@x,\the\pgf@y) 用作 at 的目标。

44.2.4 一般选项

下面的多个选项 inner xsep, inner ysep, inner sep, outer xsep, outer ysep, outer sep, minimum width, minimum height, minimum size 由 shapes 模块定义，它们的值通常会被 \pgfmathsetlength →P. 112 处理，处理结果是一个尺寸寄存器。

下面的多个选项针对的是“形状”(shape)，一个形状就是一个复杂的路径，它的某些特征是可以调整的，这些选项就是调整其特征的“手段”。例如，对于形状为 rectangle 的 node, 它的特征尺寸如下图所示：

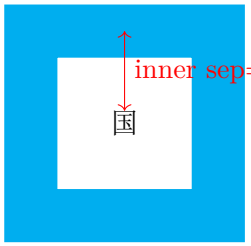


上面图形中的青色粗线代表 node 的背景路径 (即 shape 路径) 的线宽，在默认下，outer sep 的初始值是半个线宽，即恰好达到青色线的外缘。

/pgf/inner sep=⟨*dimension*⟩ (no default, initially .3333em)

/tikz/inner sep

这是 /pgf/inner sep 的别名，设置 node 的文字内容与其形状的背景路径 (background path) 的间距，包括 x 轴方向的间距和 y 轴方向的间距。这里说的“背景路径”并不包括线条的线宽 (line width 是图形状态参数，不属于路径)。




```
\begin{tikzpicture}
\node[line width=20pt,inner sep=30pt,draw=cyan]
  (g) {国};
\draw [<->,red](g.north)++(0,-10pt)
  --node[right]{inner sep=30pt}++(0,-30pt);
\end{tikzpicture}
```

/pgf/inner xsep=⟨*dimension*⟩ (no default, initially .3333em)

/tikz/inner xsep

这是 /pgf/inner xsep 的别名，设置 node 的文字内容与其形状的背景路径 (background path) 在 x 轴方向的间距。

/pgf/inner ysep=⟨*dimension*⟩ (no default, initially .3333em)

/tikz/inner ysep

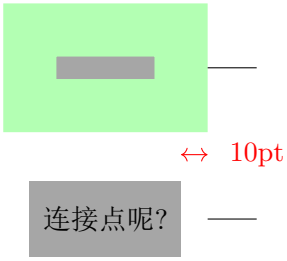
这是 /pgf/inner ysep 的别名，设置 node 的文字内容与其形状的背景路径 (background path) 在 y 轴方向的间距。

/pgf/outer sep=⟨*dimension or "auto"*⟩ (no default)

/tikz/outer sep

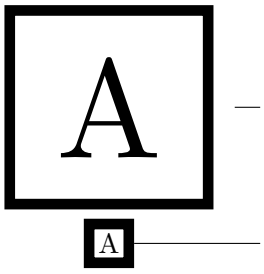
这是 /pgf/outer sep 的别名。若 ⟨*dimension*⟩ 为正值尺寸，则以 node 的背景路径为基础，向外扩展 ⟨*dimension*⟩, 但这个扩展是不可见的；若 ⟨*dimension*⟩ 为负值尺寸，则向内收缩，这个收缩也是不可见的。这个扩展 (收缩) 没有改变原来的背景路径，但会把 node 的各个锚位置 (anchor) 向外扩展 (向内收缩)。

如果你不自己指定本选项的值，那么本选项的值通常就是 "线宽的一半"。也就是说，node 的各个锚位置 (anchor) 并不是位于背景路径上的，而是位于 "背景路径线条" 的外侧缘上，这里说的线条包括线宽。但有时这会导致某些不太合适的地方，例如，如果一个 node 背景路径的线宽是 20pt，并且只是填充背景路径而不画出背景路径，那么 node 的各个 anchor 与填充色之间的间距就是线宽的一半，即 10pt；当用线条连接这个 node 时，线条端点就是某个 anchor，在视觉上线条并没有与 node 连起来。




```
\tikz[minimum width=2cm, minimum height=1cm]{
  \node (a) [fill=gray!70, line width=20pt] {连接点呢?};
  \draw (2,0)--(a);
  \node (b) [draw=green, draw opacity=0.3, fill=gray!70, line
  ↪  width=20pt] at(0,2){};
  \draw (2,2)--(b);
  \draw [<->,red]($(b.south east)!1!(a.north east)$)--+(-10pt,0)
  ↪  node[right]{10pt};
}
```

另外，当 node 带有 scale 选项时，outer sep 的值会被 scale 选项放大 (缩小)，但背景路径的线宽却不接受 scale 选项的作用，这也可能出现不合适的地方。例如，如果一个 node 背景路径的线宽是 4pt 并且画出背景路径，再给 node 添加选项 scale=5，那么 node 的各个锚 (anchor) 与路径线条 (包括线宽) 之间的间距就是 $4 \div 2 \times 5 - 4 \div 2 = 8$ pt.




```
\tikz{
  \node (a) [draw, line width=4pt] {A};
  \draw (2,0)--(a);
  \node (b) [draw, line width=4pt, scale=5] at(0,1.8){A};
  \draw (2,1.8)--(b);
}
```

解决这类问题的办法是使用 outer sep=auto，这个键值的作用是：若不画出背景路径，则设置 outer sep=0pt；若画出背景路径，则把 outer sep 的值设为 "半个线宽值与两个因子的乘积"，这里的两个因子，一个用于调整水平间隔 (horizontal separations)，一个用于调整垂直间隔 (vertical separations)，参考 \pgfhorizontaltransformationadjustment→P. 270, \pgfverticaltransformationadjustment→P. 270. 这个办法对放缩变换 (scale, xscale, yscale) 的效果还好，但对倾斜变换 (xslant, yslant) 的效果可能不够准确。

没有把 outer sep=auto 设为默认值是为了兼容之前的版本。

/pgf/outer xsep=⟨*dimension*⟩ (no default, initially .5\pgflinewidth)

/tikz/outer xsep

这是 /pgf/outer xsep 的别名。本选项类似 /pgf/outer sep，但只对水平方向的 outer sep 有效，对竖直方向的 outer sep 无效。如果同时写出 outer sep 和 outer xsep 选项，则 outer sep 选项具有优先地位。

/pgf/outer ysep=⟨*dimension*⟩ (no default, initially .5\pgflinewidth)

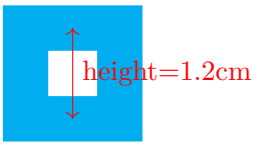
/tikz/outer ysep

这是 /pgf/outer ysep 的别名。本选项类似 /pgf/outer sep, 但只对竖直方向的 outer sep 有效，对水平方向的 outer sep 无效。如果同时写出 outer sep 和 outer ysep 选项，则 outer sep 选项具有优先地位。

/pgf/minimum height=⟨*dimension*⟩ (no default, initially 0pt)

/tikz/minimum height

这是 /pgf/minimum height 的别名。本选项决定形状 (shape) 的最小高度值 (形状的实际高度不小于这个高度值)，这个高度是背景路径的高度，不把背景路径线条的线宽算在内，也不把 outer sep 算在内。




```
\begin{tikzpicture}
  \draw (0,0) node[line width=0.6cm,minimum height=1.2cm, draw=cyan]
  ↪  (a){\rule{1cm}{0cm}};
  \draw [<->,red](a.south)++(0,0.3cm) -- node[right]{height=1.2cm}
  ↪  ++(0,1.2cm);
\end{tikzpicture}
```

/pgf/minimum width=⟨*dimension*⟩ (no default, initially 0pt)

/tikz/minimum width

这是 /pgf/minimum width 的别名，类似 minimum height, 本选项指定形状 (shape) 的最小宽度值。

/pgf/minimum size=⟨*dimension*⟩ (no default)

/tikz/minimum size

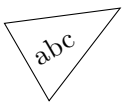
这是 /pgf/minimum size 的别名，本选项把形状 (shape) 的最小宽度值，高度值都指定为 ⟨*dimension*⟩.

/pgf/shape aspect=⟨*aspect ratio*⟩ (no default)

/tikz/shape aspect

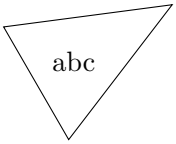
这是 /pgf/shape aspect 的别名，本选项指定形状 (shape) 的宽度与高度之比 $\frac{\text{宽度}}{\text{高度}}$ = ⟨*aspect ratio*⟩.

如果给 node 带上旋转选项 rotate，那么 node 的形状路径、文字内容都会被旋转：




```
\tikz \node [rotate=30,isosceles triangle,draw] {abc};
  % 形状 isosceles triangle 是等腰三角形，需要载入 shapes.geometric 库
```

如果想只旋转 node 的形状路径，不旋转文字内容，就要使用选项 shape border rotate=⟨*angle*⟩ (或者还需要同时使用选项 shape border uses incircle, 见下文)。



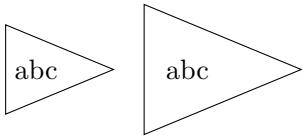

```
\tikz \node [shape border uses incircle,shape border rotate=30,isosceles
↪  triangle,draw] {abc};
  % 形状 isosceles triangle 是等腰三角形，需要载入 shapes.geometric 库
```

针对 node 形状路径的旋转有两种，一种是限制性旋转，另一种是非限制性旋转。当一个形状路径“装备”了内接圆时，它的文字内容放在内接圆内，选项 inner sep 的值指的是文字与内接圆的间距，此时对形状路径的旋转是比较随意的，是非限制性旋转。当一个形状路径没有“装备”内接圆时，形状路径与文字内容的间距显得比较紧密一些，此时对形状路径的旋转角度限制为 90° 的整数倍，是限制性旋转。布尔选项 shape border uses incircle 决定是否给 node 的形状路径“装备”内接圆。

/pgf/shape border uses incircle=⟨*boolean*⟩ (default true)

/tikz/shape border uses incircle=⟨*boolean*⟩ (default true)

如果给 node 使用 shape border uses incircle 或 shape border uses incircle=true, 那么 node 的形状路径会被自动调整，使得其有内接圆，并把 node 的文字内容放在内接圆内，文字内容与内接圆的间距就是选项 inner sep 的值。




```
\tikzstyle{every node}=[isosceles triangle, draw]
\begin{tikzpicture}
  \node {abc};
  \node [shape border uses incircle] at (2,0) {abc};
\end{tikzpicture}
```

/pgf/shape border rotate=⟨*angle*⟩ (no default, initially 0)

/tikz/shape border rotate=⟨*angle*⟩ (no default, initially 0)

这个选项将 node 的形状路径旋转 ⟨*angle*⟩ 角度，但不旋转内容。如果 node 没有带选项 shape border uses incircle, 则该选项只能将形状路径旋转 90° 的整数倍。你写出的 ⟨*angle*⟩ 可以不是 90° 的整数倍，但实际的旋转角度是与 ⟨*angle*⟩ 最接近的某个“90° 的整数倍”。

如果 node 带有选项 shape border uses incircle, 则转角就没有这个限制。

用这个选项旋转形状路径时，node 坐标系中的罗盘位置（与方向名称 north, east 等有关的位置，以及用角度指出的位置）、与内容的基线相关的位置（与 base 有关的位置）都不被旋转，其它的锚位置（用“上、下、左、右”指出的位置）会随同形状路径一起旋转.

并非所有的 node 形状都支持选项 shape border rotate 的旋转，例如 rectangle 形状就不支持。支持这个选项的形状不区分 outer xsep 和 outer ysep 这两个选项值，而是将这两个选项值中的较大者作为 outer sep 的值。

44.3 Multi-Part Nodes

一个形状为，例如，circle 的 node, 可以将其中间画一横线，分为上下两部分，在上下部分别添加文字，得到一个有两个部分的 node.

参考 shapes.multipart 库.

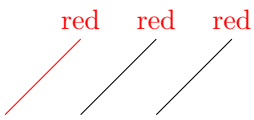
44.4 node 中的文字

44.4.1 文字参数：颜色、不透明度

node 中的文字的颜色由之前的 color= 选项来规定，也可以由下一选项来设置：

/tikz/text=⟨*color*⟩ (no default)

本选项设置文字的颜色。




```
\begin{tikzpicture}
  \draw[red] (0,0) -- +(1,1) node[above] {red};
  \draw[text=red] (1,0) -- +(1,1) node[above] {red};
  \draw (2,0) -- +(1,1) node[above,red] {red};
\end{tikzpicture}
```

文字的不透明度用选项 /tikz/text opacity→P. 905 设置。

44.4.2 文字参数：字体

字体包括字族、字形、尺寸等文字属性。

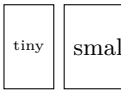
/tikz/node font=⟨*font command*⟩ (no default)

这个选项设置 node 内容中的文字字体。有时会希望文字的尺寸与 node 的形状尺寸相匹配，例如 `minimum size=4em` 规定 node 的最小尺寸，其中用了相对长度单位 em，em 会随着正文默认字体尺寸的变化而变化。选项 `node font` 设置的文字尺寸会对以 em，ex 定义的 node 形状尺寸产生影响，即在当前分组或路径内，`minimum size=4em` 中的 `em` 的实际长度决定于选项 `node font` 所规定的字体尺寸。


```
\tikzstyle{every node}=[rectangle, draw]
\tikz \node [node font={\it \tiny}, minimum height=3em, draw] {tiny};
\tikz \node [node font={\sf \small}, minimum height=3em, draw] {small};
```

/tikz/font=⟨*font commands*⟩ (no default)

这个选项设置 node 内容中的文字字体, 但不会重设当前分组或路径内的 em, ex 的长度。




```
\tikz \node [font=\tiny, minimum height=3em, draw] {tiny};
\tikz \node [font=\small, minimum height=3em, draw] {small};
```

44.4.3 文字参数：文字换行、对齐方式、文字行宽

一般情况下，node 的文字内容会被放入一个水平盒子中，不会自动换行，在文字内容中使用 \\ 手动换行也无效。创建多行内容的办法有以下几种：

1. 使用 LaTeX 环境或其它宏包提供的环境, 例如表格环境 `{tabular}`, 矩阵环境.
2. 使用对齐选项 `align`, 然后在内容中使用两个反斜线 \\ 来换行（必须先设置对齐方式）。可以使用 \\[⟨*dimension*⟩] 在换行时追加垂直间距，这个间距可以是负的。
3. 使用选项 `text width` 设置文本行的宽度，这样可以自动换行，也可以用 \\ 手动换行，注意这个选项会使得 node 的宽度不小于所设置的文本行的宽度。

/tikz/text width=⟨*dimension*⟩ (no default)

这个选项会将 node 的内容放入一个宽度为 <dimension> 的盒子中，盒子作用类似 `{minipage}` 环境，其中可以自动换行，也可手动换行，也会使得 node 的宽度不小于 ⟨*dimension*⟩. 给出这个选项后，内容会自动使用左对齐方式（`align=left`）。文本行的行末可能出现断词（连字符）。

如果 ⟨*dimension*⟩ 留空，则本选项无效，会取消自动换行。

/tikz/align=⟨*alignment option*⟩ (no default)

这个选项设置多行 node 内容的对齐方式。如果使用了 `text width`=⟨*dimension*⟩ 且 ⟨*dimension*⟩ 非空，则 `align` 选项会参照 ⟨*alignment option*⟩ 设置 \leftskip 以及 \rightskip 来实现对齐和换行，

这是“有断行对齐”（alignment with line breaking）。如果没有使用 `text width=`⟨*dimension*⟩ 或 ⟨*dimension*⟩ 是空的，则 ⟨*align*⟩ 选项会使用选项 `node halign header` 确定的机制来实现对齐，这是“无断行对齐”（alignment without line breaking），此时可以使用 `\\` 来手动换行。

对齐方式 ⟨*alignment option*⟩ 有以下几种：

align=`left` 这个对齐方式使用 plain TEX 所定义的左对齐（ragged right）方式，TEX 会尽量平衡文本行以降低文本右侧的不平整程度，因此可能会在行末自动断词并添加连字符号。

align=`flush left` 这个对齐方式使用 LATEX 样式，不会自动平衡文本行，不会在行末出现断词（连字符），故文本右侧可能很是参差不齐。

align=`right`

align=`flush right`

align=`center` 当 `text width=`⟨*dimension*⟩ 设置的宽度很长，但 node 的文字内容很短时，TEX 会给出由 `align=center` 创建的盒子所引起的水平方向的劣质警告信息（horizontal badness warnings），这是无法避免的，而默认 TikZ 关闭这些水平方向的劣质警告以及其它某些（可能有用的）警告，可以使用下一选项恢复这些警告：

`/tikz/badness warnings for centered text=`⟨*true or false*⟩ (no default, initially `false`)

如果这个选项的值设为 `true`, 则会发出针对由 `align=center` 创建的盒子的各种警告，如果这些盒子是你需要的设计效果，当然可以置之不理。

align=`flush center` 对于文字内容很短的情况，这个键值不会像 `center` 那样导致水平方向的劣质警告信息。

align=`justify` 在“无断行对齐”时，即没有使用 `text width=`⟨*dimension*⟩ 或者 ⟨*dimension*⟩ 为空时，这个选项等效于 `align=left`；在“有断行对齐”时, 即使用了 `text width=`⟨*dimension*⟩ 且 ⟨*dimension*⟩ 非空时，这个选项会使文本行分散对齐（文字在一行中均匀分布）。

align=`none` 取消文字对齐，换行命令 `\\` 也无效。

`/tikz/node halign header=`⟨*macro storing a header*⟩ (no default, initially empty)

在“无断行对齐”时，即没有使用 `text width=`⟨*dimension*⟩ 或者 ⟨*dimension*⟩ 为空，并且使用了选项 `align` 时，本选项有效。这个选项会使用命令 `\halign` 的机制。命令 `\halign` 是 TEX 制造表格的基本命令，其句法如下：

```
\halign{列格式行\cr 表格行 1\cr ... 表格行 n\cr}
```

例如

China Apple $1.0
France Banana $0.574

```
\halign{%
  \it#\tabskip=1em & \hfil#\hfil & \hfil\$#\cr
  China            & Apple       & 1.0\cr
  France           & Banana      & 0.574\cr}
```

这个制表例子中，`\cr` 是换行标志。第一行定义列格式，`#` 代表单元格数据，`&` 是分列符号，所以第一行定义了一个 3 列表格，第一列（按默认编辑方式）左对齐，第二列用两个 `\hfil` 命令包围单元格数据实现居中对齐，第三列右对齐。命令 `\tabskip=1em` 在当前列与右侧各列之间插入 1em 的间距。选项 `node halign header` 借用了命令 `\halign` 的机制，但这个选项只能定义“只有一列的表格”。这个选项将该命令的换行符号 `\cr` 改为双反斜线 `\\`, 在文本中用 `\\` 手动换行。文本中的每个换行都对应表格的“新行”，每行文本都放入一个水平盒子中。最后一行（包括只有一行的文本）不必使用 `\\` 来结束。

对齐方式由 ⟨*macro storing a header*⟩ 规定，例如：

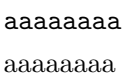

```
\def\myheader{\hfil\hfil##\hfil\cr}
\tikz [node halign header=\myheader] \node[draw] {ABCD\\EF};
```

即首先定义宏 `\myheader`, 这个宏的内容就是命令 `\halign` 的列格式定义，但只能定义一列（不出现 `&` 符号）。然后将 `\myheader` 作为选项 `node halign header` 的值。注意如果直接写出

```
node halign header=\hfil\hfil##\hfil\cr
```

会导致错误。

注意，由于本选项把每行文本都放入一个水平盒子中，所以一行之内的命令一般只在该行之内有效，例如第一行内的字体声明命令 `\ttfamily` 只对第一行有效，对第二行无效。




```
\tikz{
\node [align=left]{\ttfamily aaaaaaaa\\
                                aaaaaaaa};
}
```

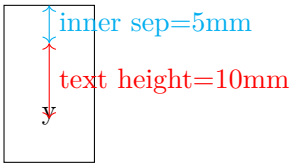
这种表格机制缺少弹性，如有必要，可以使用 `tabular` 环境或 `matrix` 环境。

44.4.4 文字参数：文字的高度和深度

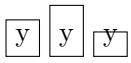
选项 `text height` 和 `text depth` 可以调整文字盒子的高度和深度，从而控制 node 的尺寸。

`/tikz/text height`=⟨*dimension*⟩ (no default)

本选项指定文字盒子的高度，如果 ⟨*dimension*⟩ 空置，则使用文字盒子的“自然高度”。注意不要混淆“文字盒子高度”和 `inner xsep`.




```
\begin{tikzpicture}
\node (y)[draw,text height=10mm,inner sep=5mm]{y};
\draw [<->,red](y.base)--
  node[right]{text height=10mm}++(0,10mm);
\draw [<->,cyan](y.base)++(0,10mm)--
  node[right]{inner sep=5mm}++(0,5mm);
\end{tikzpicture}
```




```
\tikz \node[draw] {y};
\tikz \node[draw,text height=10pt] {y};
\tikz \node[draw,text height=-5pt] {y};
```

`/tikz/text depth`=⟨*dimension*⟩ (no default)

本选项指定文字盒子的深度。

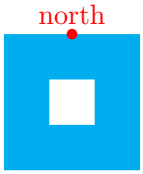
设置文字的高度和深度可以实现某种对齐效果。

44.5 Positioning Nodes

一般情况下，把 ndoe 放在“锚定点”上时，node 的中心会处于锚定点上。

44.5.1 利用 anchor 来确定 node 的位置

PGF 可以使用锚机制（anchoring mechanism）来调整 ndoe 的位置。每个 node 形状都有自己的坐标系统，坐标原点位于 node 形状的中心。node 的锚位置就是这个坐标系统中的点，这些点（除了 `center`, `base`）都位于形状路径线条的外侧缘上（不是线条的中间）。



在文件《pgfmoduleshapes.code》中有 coordinate，rectangle，circle 这三种 node 形状的定义。库文件《pgflibraryshapes.geometric.code》中也定义了很多 node 形状。对于预定义的 node 来说，node 的形状一般都会有以下锚位置：

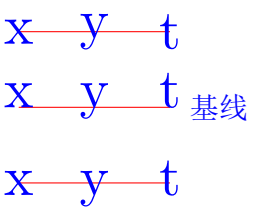
- 带有方向名词的位置：north，east，north east 等。
- 与内容文字的基线 base 有关的位置：base，base west，base east.
- 与 node 形状的竖直方向的中点 mid 有关的位置：mid，mid west，mid east.
- 形状 coordinate 也有多种锚位置，如 center，east，north east，mid，mid west 等，但都指向一个位置 center.

各种形状的锚位置参考相应的 Shape Library.

/tikz/anchor=⟨*anchor name*⟩ (no default)

这个选项平移 node，使得 node 的名称为 ⟨*anchor name*⟩ 的位置放在当前点（锚定点）上。注意“锚与船的位置是相对的”，如果将 north 位置放在某个坐标点上，那么 node 的中心将位于该点的“南方”；如果将 north east 位置放在某个坐标点上，那么 node 的中心将位于该点的“西南方”。

使用关于 base，mid 的锚位置可以实现某种对齐效果：




```
\begin{tikzpicture}[scale=2,red,text=blue,transform shape]
  \draw[anchor=center] (0,1) node{x} -- (0.5,1) node{y} -- (1,1) node{t};
  \draw[anchor=base] (0,.5) node{x} -- (0.5,.5) node{y} -- (1,.5) node{t}
                                                    node[anchor=west]{\tiny 基线};
  \draw[anchor=mid] (0,0) node{x} -- (0.5,0) node{y} -- (1,0) node{t};
\end{tikzpicture}
```

本选项的定义是：

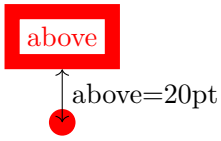
```
\tikzoption{anchor}{\def\tikz@anchor{#1}\let\tikz@do@auto@anchor=\relax}%
```

宏 \tikz@anchor 将作为命令 \pgfmultipartnode 的参数，来创建 node，参考 \tikz@fig@continue → P. 842。在命令 \pgfmultipartnode 的处理过程中，锚位置 \tikz@anchor 导致一个平移变换。

44.5.2 基本的平移选项

/tikz/above=⟨*offset*⟩ (default 0pt)

这里的 ⟨*offset*⟩ 是带长度单位的尺寸。如果不指定⟨*offset*⟩，那么这个选项的作用等于 anchor=south；如果指定 ⟨*offset*⟩，那么 node 会相对于锚定点向上平移，使得 node 的 south 位置位于锚定点之上的 ⟨*offset*⟩ 距离处。




```
\tikz {
  \fill [red](0,0) circle (5pt)
    node(上)[above=20pt,draw,line width=2mm,inner sep=2mm] {above};
  \draw [<->](上.south)-- node[right]{above=20pt} ++(0,-20pt);
  }
```

/tikz/below=⟨*offset*⟩ (default 0pt)

类似 above.

/tikz/left=⟨*offset*⟩ (default 0pt)

类似 above.

/tikz/right=⟨*offset*⟩ (default 0pt)

类似 above.

/tikz/above left=⟨*offset*⟩ (no value)

等效于 anchor=south east, 注意 [above,left] 这两个选项只有后者有效，故不等于 [above left].

/tikz/above right=⟨*offset*⟩ (no value)

类似 above left.

/tikz/below left=⟨*offset*⟩ (no value)

类似 above left.

/tikz/below right=⟨*offset*⟩ (no value)

类似 above left.

/tikz/centered=⟨*offset*⟩ (no value)

等于 anchor=center.

44.5.3 高级平移选项

TikZ Library positioning

```
\usetikzlibrary{positioning} % LaTeX and plain TeX
\usetikzlibrary[positioning] % ConTeXt
```

这个库允许用稍微复杂但更有控制能力的句式来移动 node.

当载入 positioning 库后，选项 above, above left 等会被重定义，见文件《tikzlibrarypositioning.code.tex》，例如：

```
\tikzset{above/.code=\tikz@lib@place@handle@{#1}{south}{0}{1}{north}{1}}%
\tikzset{above left/.code=\tikz@lib@place@handle@{#1}{south east}{-1}{1}{north west
↪  }{0.707106781}}%
```

此时的选项参数，例如“above=⟨*specification*⟩”中的 ⟨*specification*⟩, 其格式可以有以下几种：

- 只是一个 ⟨*expression*⟩, 例如 2cm 或 3cm/2+4cm, 或者 3+{sin(60)}, 表达式 ⟨*expression*⟩ 会被命令 \pgfmathparse→P. 110 解析。实际上这个格式等价于 ⟨*expression*⟩ and ⟨*expression*⟩, 如下。
- 可以是由 and 连接的两个表达式，⟨*expression1*⟩ and ⟨*expression2*⟩, 这两个表达式 ⟨*expression1*⟩ 和 ⟨*expression2*⟩ 都会被命令 \pgfmathparse→P. 110 解析。表达式 ⟨*expression1*⟩ 和 ⟨*expression2*⟩ 的用处是确定一个平移向量，⟨*expression1*⟩ 用来构成 y 轴向的平移因子，⟨*expression2*⟩ 用来构成 x 轴向的平移因子。
- 可以含有单词 of, 如
  - of ⟨*point spec*⟩, 其中的参数 ⟨*point spec*⟩ 是能被命令 \tikz@scan@one@point→P. 710 解析的 TikZ 点，这个点将作为当前 node 的锚定点。如果 ⟨*point spec*⟩ 是某个 node 名称，那么还会考虑真值 \iftikz@lib@ignore@size, 也就是选项 on grid 的值，来决定当前 node 的锚定点。
  - ⟨*expression*⟩ of ⟨*point spec*⟩, 类似。

– ⟨*expression1*⟩ and ⟨*expression2*⟩ of ⟨*point spec*⟩, 类似。

总之，⟨*specification*⟩ 的用处是：确定一个平移向量，(也可能) 指定 node 的锚定点 (如果有单词 of 的话)。

命令 \tikz@lib@place@handle@ 是文件《tikzlibrarypositioning.code.tex》定义的主要命令，它需要 6 个参数：

```
\def\tikz@lib@place@handle@#1#2#3#4#5#6{%
%......
}
```

命令 \tikz@lib@place@handle@ 的参数格式是：

\tikz@lib@place@handle@{⟨*specification*⟩}{⟨*anchor1*⟩}{⟨*number1*⟩}{⟨*number2*⟩}{⟨*anchor2*⟩}{⟨*number3*⟩}

它的第一个参数就是选项参数 ⟨*specification*⟩. 命令 \tikz@lib@place@handle@ 会检查参数 ⟨*specification*⟩ 中是否含有单词 of 或 and, 根据检查结果做相应的处理，最后的处理结果有几种可能：

1. 如果没有写出 ⟨*specification*⟩, 例如只是写出 above, 那么规定当前 node 的 anchor 位置为

   ```
   \def\tikz@anchor{south}%
   ```

2. 如果写出 ⟨*specification*⟩, 但其中没有单词 of, 例如写出 above=1, 那么会
   - 规定当前 node 的 anchor 位置

     ```
     \def\tikz@anchor{south}%
     ```

   - 定义保存平移变换命令的宏

     ```
     \edef\tikz@lib@pos@call{\noexpand\pgftransformshift{\noexpand\pgfqpoint{0pt}
     ↪  {1cm}}}%
     ```

3. 如果写出 ⟨*specification*⟩, 其中有单词 of, 但没有 ⟨*expression*⟩, 那么默认 ⟨*expression*⟩ 是 “1cm and 1cm”, 例如写出 above= of (1,2), 实际等价于 above=1cm and 1cm of (1,2), 这会
   - 规定当前 node 的 anchor 位置为

     ```
     \def\tikz@anchor{south}%
     ```

   - 定义保存平移变换命令的宏

     ```
     \edef\tikz@lib@pos@call{\noexpand\pgftransformshift{\noexpand\pgfqpoint{0pt}
     ↪  {1cm}}}%
     ```

   - 规定当前 node 的锚定点为 (1,2), 因为有定义

     ```
     \tikz@scan@one@point\tikz@lib@place@remember(#2)%
     \def\tikz@lib@place@remember#1{\def\tikz@node@at{#1}}%
     ```

     所以锚定点定义为

     ```
     \def\tikz@node@at{\pgfqpoint{1cm}{2cm}}
     ```

文件《tikzlibrarypositioning.code.tex》的开头执行

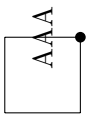
```
\pgfutil@g@addto@macro\tikz@node@reset@hook{\tikz@addtransform{\tikz@lib@pos@call}
↪  \let\tikz@lib@pos@call=\relax}%
```

这是全局地重定义宏 \tikz@node@reset@hook.

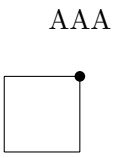
执行 \tikz@node@reset@hook 的作用是：把 \tikz@lib@pos@call 添加到变换命令序列中，即添加到宏 \tikz@transform 的定义中，参考 \tikz@addtransform → P. 881。需要注意的是，当 TikZ 解析 “node[⟨*options*⟩]” 语句时，即执行命令 \tikz@fig ode 时，会依次执行：

- \tikz@node@reset@hook
- \tikzset{every node/.try}
- \tikzset{⟨*options*⟩}

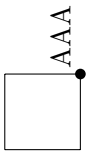
这会导致一个难以察觉的细节，例如，比较：




```
\tikz{
    \draw (0,0)grid(1,1);
    \fill (1,1) circle [radius=2pt];
    \node [rotate=90,shift={(0,0.5)}] at(1,1) {AAA};
}
```




```
\tikz{
    \draw (0,0)grid(1,1);
    \fill (1,1) circle [radius=2pt];
    \node [above=0.5] at(1,1) {AAA};
}
```




```
\tikz{
    \draw (0,0)grid(1,1);
    \fill (1,1) circle [radius=2pt];
    \node [rotate=90,above=0.5] at(1,1) {AAA};
}
```

上面例子中的选项 above=0.5 会把 \tikz@lib@pos@call, 也就是把平移变换命令

```
\pgftransformshift{\pgfqpoint{0pt}{0.5cm}}
```

添加到宏 \tikz@transform 的定义中。当 Ti*k*Z 解析 node 语句时，会执行 \tikz@transform 引入变换命令，(参考 \tikz@node@transformations→P. 843)，这个平移变换命令会在旋转命令 rotate=90 之前起作用 (尽管选项 above=0.5 写在选项 rotate=90 之后)，这是因为先执行 \tikz@node@reset@hook, 把平移变换命令添加到宏 \tikz@transform 的中，然后再执行 \tikzset{rotate=90}, 把旋转变换命令添加到宏 \tikz@transform 的中。

/tikz/above=⟨*specification*⟩ (default 0pt)

规定 node 移动方式的 ⟨*specification*⟩ 有两种情况：

- 只有 ⟨*shifting part*⟩, 即只有指定平移距离的句式，这种句式又有以下 3 种形式：
  1. 关于长度（带有长度单位）的算式，例如 2cm 或 3cm/2+4cm, 如同前面所述的（相当于不载入 positioning 程序库的）above 选项的作用，这会使得 node 的 south 位置位于锚定点之上。
  2. 关于纯数字（不带单位）的算式，例如 2 或 3+{sin(60)}, 如果算式的计算结果是 ⟨*number*⟩, 则 node 的 south 位置位于锚定点之上，平移向量为 (0,⟨*number*⟩).
  3. 用 and 给出两个数据,⟨*number or dimension 1*⟩ and ⟨*number or dimension 2*⟩, 例如 above=.2 and 3mm, 这里 .2 的默认长度单位是 cm. 此时程序会构造一个平移向量：(⟨*number or dimension 2*⟩,⟨*number or dimension 1*⟩), 例如 (3mm,0.2cm), 注意其中将 and 后的数据作为横标，and 前的数据作为纵标。横标指示横向平移距离，纵标指示纵向平移距离。而 above 是纵向平移，所以只有纵标，即 and 前的数据对 above 选项有意义。
- 带有 ⟨*of-part*⟩, 即用 of 指定 node 的锚定点。⟨*of-part*⟩ 可以用以下形式：
  1. ⟨*of-part*⟩ 是 of ⟨*coordinate*⟩ 这种坐标形式，注意 ⟨*coordinate*⟩ 不能带圆括号。例如，

```
of 1,2
of 1,2 -| 2,1
of {$(0,0)!0.5!(2,2)$} 其中必须用花括号把坐标算式括起来
of {{sqrt(2)},{exp(0.5)}} 其中必须使用二重套嵌花括号
of a.north 其中 a 是某个 node 的名称
```

有了锚定点，然后平移 node, 使其 south 位置位于锚定点之上，二点的距离由 of 之前的部分指定。所以

```
\node [above=5mm of somenode.north east]{};
```

等效于

```
\node [above=5mm] at(somenode.north east){};
```

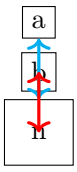
如果不指定平移距离，即等号“=”后没有算式，例如 above= of somenode.north, 那么平移距离就由选项 node distance 规定。

2. ⟨*of-part*⟩ 是 of ⟨*node name*⟩ 这种形式，例如

```
\node (a)[above=5mm of somenode]{};
```

这会使得 (a) 的锚位置 south 位于 (somenode) 的 north 位置之上 5mm 处。如果不指定平移距离，即等号“=”后没有算式，那么平移距离就由选项 node distance 规定。

如果在选项中同时给出锚位置和平移距离，例如 [above=6mm,anchor=center], 那么后者优先。如果将某个 node 的名称用做 at 的参数，如 at(⟨*node name*⟩), 则 at 决定的指向点位于 ⟨*node name*⟩ 的中心，即其 center 位置。



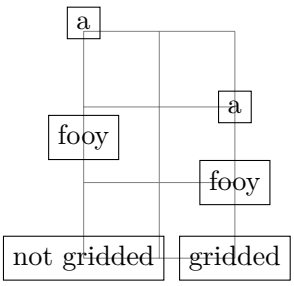

```
\tikz {\node [draw,inner sep=10pt] (n){n};
  \node (a)[above=8mm of n,draw]{a};
  \node (b)[above=8mm,anchor=center,draw] at(n){b};
  \draw [<->,cyan,very thick] (n.north)--++(0,8mm);
  \draw [<->,red,very thick] (n.center)--++(0,8mm);
}
```

/tikz/on grid=⟨*boolean*⟩ (no default, initially false)

前面说的 above 选项在平移 node 时，平移距离都是以 node 的形状线条的外侧缘为测量界限的。当设置选项 on grid=true 后，举例来说，

```
\node (a)[above=5mm of somenode]{};
```

使得 (a) 的中心 center 位置位于 (somenode) 的中心 center 之上 5mm 处。




```
\begin{tikzpicture}[every node/.style=draw]
  \draw[help lines] (0,0) grid (2,3);
% Not gridded
  \node (a1) at (0,0) {not gridded};
  \node (b1) [above=1cm of a1] {fooy};
  \node (c1) [above=1cm of b1] {a};
% gridded
  \node (a2) at (2,0) {gridded};
  \node (b2) [on grid,above=1cm of a2] {fooy};
  \node (c2) [on grid,above=1cm of b2] {a};
\end{tikzpicture}
```

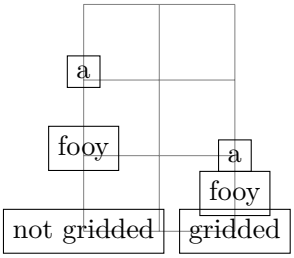
本选项的定义是：

```
\tikzset{on grid/.is if=tikz@lib@ignore@size}%
```

本选项会设置 \iftikz@lib@ignore@size 的真值。

/tikz/node distance=⟨*shifting part*⟩ (no default, initially 1cm and 1cm)

如果不指定平移距离，即等号“=”后没有算式，那么平移距离就由该选项规定。




```
\begin{tikzpicture}[every node/.style=draw,node distance=5mm]
\draw[help lines] (0,0) grid (2,3);
% Not gridded
\node (a1) at (0,0) {not gridded};
\node (b1) [above=of a1] {fooy};
\node (c1) [above=of b1] {a};
% gridded
\begin{scope}[on grid]
\node (a2) at (2,0) {gridded};
\node (b2) [above=of a2] {fooy};
\node (c2) [above=of b2] {a};
\end{scope}
\end{tikzpicture}
```

这里的 ⟨*shifting part*⟩ 可以是：

- 带长度单位的尺寸
- 数值
- （有单位或无单位的）算式
- ⟨*number or dimension 1*⟩ `and` ⟨*number or dimension 2*⟩

⟨*shifting part*⟩ 会被插入到，例如 `above=` 之后，从而获得 `above=`⟨*shifting part*⟩ 这种形式的效果。

`/tikz/below=`⟨*specification*⟩ (no default)

类似 `above`.

`/tikz/left=`⟨*specification*⟩ (no default)

类似 `above`.

`/tikz/right=`⟨*specification*⟩ (no default)

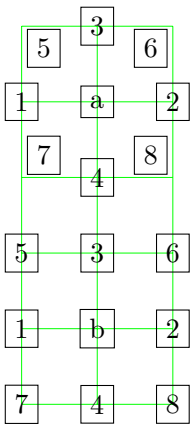
类似 `above`.

`/tikz/above left=`⟨*specification*⟩ (no default)

类似 `above`，但 ⟨*shifting part*⟩ 的形式决定的平移方式有所不同：

1. 如果 ⟨*shifting part*⟩ 的形式是 ⟨*number or dimension 1*⟩ `and` ⟨*number or dimension 2*⟩, 程序会构造一个平移向量：(⟨*number or dimension 2*⟩,⟨*number or dimension 1*⟩), 注意其中将 `and` 后的数据作为横标, `and` 前的数据作为纵标。横标指示横向平移距离，纵标指示纵向平移距离。
2. 如果 ⟨*shifting part*⟩ 的形式是 ⟨*number or dimension*⟩, 那么平移向量是极坐标 `(135:`⟨*number or dimension*⟩`)`。

下面的例子显示不同形式的 ⟨*shifting part*⟩ 造成的差别：




```
\begin{tikzpicture}[every node/.style={draw,rectangle},on grid]
  \draw[help lines,green] (1,-1) grid (3,4);
  \begin{scope}[node distance=1]
    \node (a) at (2,3) {a};
    \node [left=of a] {1}; \node [right=of a] {2};
    \node [above=of a] {3}; \node [below=of a] {4};
    \node [above left=of a] {5}; \node [above right=of a] {6};
    \node [below left=of a] {7}; \node [below right=of a] {8};
  \end{scope}
  \begin{scope}[node distance=1 and 1]
    \node (b) at (2,0) {b};
    \node [left=of b] {1}; \node [right=of b] {2};
    \node [above=of b] {3}; \node [below=of b] {4};
    \node [above left=of b] {5}; \node [above right=of b] {6};
    \node [below left=of b] {7}; \node [below right=of b] {8};
  \end{scope}
\end{tikzpicture}
```

上面例子中，第二个 scope 环境的选项 `node distance=1 and 1`，会把 `1 and 1` 作为 `above`, `above left` 等选项的值，获得 `above=1 and 1 of b`, `above left=1 and 1 of b` 这样的效果。

/tikz/below left=⟨*specification*⟩ (no default)

类似 above left.

/tikz/above right=⟨*specification*⟩ (no default)

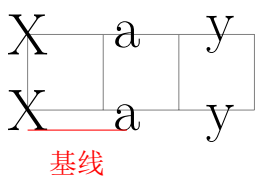
类似 above left.

/tikz/below right=⟨*specification*⟩ (no default)

类似 above left.

/tikz/base left=⟨*specification*⟩ (no default)

左移 node, 使得其 base east 位置位于锚定点的左侧。如果 ⟨*specification*⟩ 中有 of ⟨*node name*⟩ 这种成分，则新 node 的文字基线与旧 ⟨*node name*⟩ 的文字基线在同一水平上，新 node 的 base east 位置位于旧 ⟨*node name*⟩ 的 base west 位置的左侧。




```
\begin{tikzpicture}[node distance=1ex]
  \draw[help lines] (0,0) grid (3,1); \huge
  \node (X) at (0,1) {X};          \node (a) [right=of X] {a};
  \node (y) [right=of a] {y};      \node (X) at (0,0) {X};
  \node (a) [base right=of X] {a};   \node (y) [base right=of a] {y};
  \draw [red] (X.base) -- node [below] {\normalsize 基线} (a.base);
\end{tikzpicture}
```

/tikz/base right=⟨*specification*⟩ (no default)

类似 base left.

/tikz/mid left=⟨*specification*⟩ (no default)

类似 base left.

/tikz/mid right=⟨*specification*⟩ (no default)

类似 base left.

44.5.4 排布 node 的高级方法

参考 graphdrawing 库和 matrix 库。

44.6 Fitting Nodes to a Set of Coordinates

参考 fit 库。

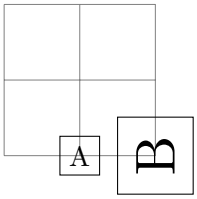
44.7 变换

路径上的 node 不属于路径，所以路径选项中的变换，例如 scale, rotate 等对路径上的 node 无效：

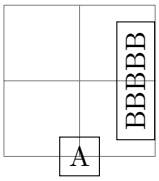

```
\begin{tikzpicture}[every node/.style=draw,]
  \draw [rotate=45](0,0)--(1,0) node [right]{x};
\end{tikzpicture}
```

可以给 node 加变换选项来实现 node 的放缩、旋转、平移等变换。注意有以下次序：

1. 首先把 node 自己的坐标系 (也就是绘制 node 的形状 (背景路径) 的坐标系) 的原点平移到 node 的锚定点处
2. 然后再考虑 node 自己的变换选项，其参照系、变换对象都是 node 自己的坐标系
3. 然后再考虑选项 /tikz/anchor → P. 888=⟨*anchor name*⟩ 导致的平移




```
\begin{tikzpicture}[every node/.style={draw}]
  \draw[help lines](0,0) grid (2,2);
  \draw (1,0) node{A}
        (2,0) node[rotate=90,scale=2] {B};
\end{tikzpicture}
```



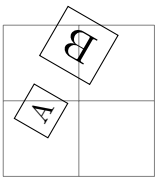

```
\begin{tikzpicture}[every node/.style={draw}]
  \draw[help lines](0,0) grid (2,2);
  \draw (1,0) node{A}
        (2,0) node[shift={(0,1)},rotate=90,anchor=south] {BBBBB};
\end{tikzpicture}
```

上面例子中，第二个 node 的坐标系的原点位于 (2,1), 与其 anchor=south 位置重合。

如果希望路径选项中的变换对路径上的 node 有效，可以给 node 带上下面的选项：

/tikz/transform shape=⟨*true or false*⟩ (default **true**, initially **false**)

如果 node 带有这个选项，那么路径选项中的变换选项将对该 node 起作用。




```
\begin{tikzpicture}[every node/.style={draw}]
  \draw[help lines](0,0) grid (2,2);
  \draw[rotate=60] (1,0) node[transform shape] {A}
                   (2,0) node[transform shape,rotate=90,scale=1.5] {B};
\end{tikzpicture}
```

上面例子中的第二个 node 本身带有变换选项，由于它有 **transform shape** 选项，它还受到路径选项中的变换选项的作用，所以它总共旋转了 150°（相对于它自己的中心点）。如果 **transform shape** 选项用作环境选项，则会对环境内的所有 node 有效。

在《tikz.code.tex》中有：

```
\tikzoption{transform shape}[true]{%
  \csname tikz@fullytransformed#1\endcsname%
  \iftikz@fullytransformed%
    \pgfresetnontranslationattimefalse%
  \else%
    \pgfresetnontranslationattimetrue%
  \fi%
}%

\newif\iftikz@fullytransformed
\pgfresetnontranslationattimetrue%
```

可见本选项先设置 \iftikz@fullytransformed 的真值,然后再决定 \ifpgfresetnontranslationattime 的真值。

TEX-if 命令 \ifpgfresetnontranslationattime 定义在《pgfcoretransformations.code.tex》中：

```
\newif\ifpgfresetnontranslationattime
```

通常,如果有真值 \pgfresetnontranslationattimetrue, 那么就会执行 \pgftransformresetnontranslations.

参考 \tikz@node@transformations→P. 843, \pgftransformresetnontranslations→P. 269.

/tikz/transform shape nonlinear=⟨*true or false*⟩ (no default, initially **false**)

如果本选项的值是 **true**，那么 tikz 会把当前的非线性变换用于 node. 目前，默认 TikZ 关闭“非线性变换”功能，参考 \pgftransformnonlinear→P. 392.

44.8 在直线段或曲线上显式地摆放 node

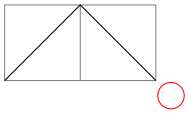
本节下文列举的选项仅对“--”，“arc”，控制曲线符号“..”，纵横线符号“-|”这 4 种操作生成的路径（子路径）上的 node 有效。这里所说的“显示地（explicitly）”的意思是，node 语句位于这些操作创建的路径（子路径）的终点之后，例如 `(0,0)--(1,1)node{a}`. 如果 node 语句位于这些操作符号之后，例如 `(0,0)--node{a}(1,1)`, 则是“隐式地（implicitly）”。

/tikz/pos=⟨*fraction*⟩ (no default)

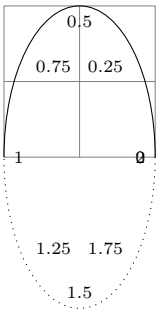
当 node 带有这个选项后，tikz 会把该 node 与其之前的路径操作联系起来，比方说，设 node 之前的路径操作创建的路径是 $p(t)$，则 node 的锚定点就是 $p(\langle fraction\rangle)$. 这里所说的路径操作目前仅限于“--”，“arc”，控制曲线符号“..”，纵横线符号“-|”这 4 种.

如果路径操作画出的路径有起点和终点，那么当 ⟨*fraction*⟩ 是 1 时，该 node 的锚定点在终点；当 ⟨*fraction*⟩ 是 0 时，该 node 的锚定点在起点；当 ⟨*fraction*⟩ 大于 0 小于 1 时，该 node 的锚定点在起点与终点之间；当 ⟨*fraction*⟩ 是其它值时，该 node 的锚定点在路径之外。

一个路径操作后面可以有多个 node.

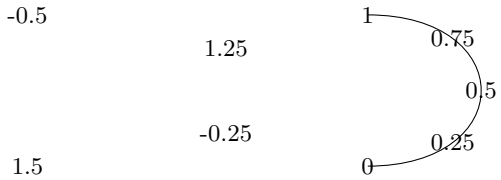


```
\begin{tikzpicture}[every node/.style={draw},transform shape]
  \draw[help lines](0,0) grid (2,1);
  \draw (0,0)--(1,1)--(2,0)node [circle,draw=red,pos=1.2]{} ;
\end{tikzpicture}
```




```
\tikz {
  \draw [help lines] (0,0) grid (2,2);
  \draw (2,0) arc (0:180:1 and 2)
        node foreach \t in {0,0.25,...,2} [pos=\t,auto,font=
        ↪  \scriptsize] {\t};
  \draw [y={(0,-1)},dotted] (2,0) arc (0:180:1 and 2);
}
```

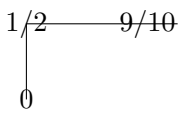
上面例子显示，对于 arc 操作而言，当 ⟨*fraction*⟩ 大于 1 时，node 的锚定点会沿着 arc 操作决定的椭圆路径摆放。




```
\tikz \draw (0,0) .. controls +(right:2) and +(right:2) .. (0,2)
  node foreach \p in {-0.5,-0.25,...,1.5} [pos=\p]{\small \p};
```

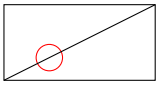
对于控制曲线而言，⟨*fraction*⟩ 决定 node 的锚定点的数学机制稍微复杂，pos=0.5 未必把 node 的锚定点确定在曲线长度的一半位置，具体可以参考“计算机图形学”或“Bézier”曲线方面的资料。

对于纵横线操作“|-”而言，pos=0.5 决定的锚定点在拐角点处：




```
\tikz \draw (0,0) |- (2,1)
            node[pos=0]{0}
            node[pos=0.5]{1/2}
            node[pos=0.9]{9/10};
```

对于其它的路径操作，目前 pos=⟨*fraction*⟩ 选项还不能获得期望的位置，例如该选项尚未与“circle”，“sin”操作创建的路径联系起来。对于“rectangle”操作，该选项决定的位置在对角线上：



```
\begin{tikzpicture}[every node/.style={draw},transform shape]
  \draw (0,0) rectangle (2,1) node[pos=0.3,draw=red, circle]{};
  \draw (0,0)--(2,1); % 对角线
\end{tikzpicture}
```

本选项的定义是：

```
\tikzoption{pos}{\edef\tikz@time{#1}\ifx\tikz@time\pgfutil@empty
↪  \else\pgfmathsetmacro\tikz@time{\tikz@time}\fi}%
```

可见本选项通过定义 \tikz@time 来发挥作用。在 tikzpicture 环境的开头 (参考 \tikz@picture)，TikZ 会自动定义

```
\def\tikz@time{.5}%
```

之后，根据用户给出的具体命令、选项，宏 \tikz@time 的值还可能会被修改。

/tikz/auto=⟨*direction*⟩ (default is scope's setting)

node 带有这个选项后，它将位于路径的左侧或右侧，而不是在路径上。这里的 ⟨*direction*⟩ 可以是：

- left，会使得 node 位于路径的左侧。
- right，会使得 node 位于路径的右侧。
- false，取消自动“站边”的功能。如果你使用 anchor 选项指定了 node 的方位，或者使用平移选项 above 等指定了 node 的位置，那么自然就有 auto=false.
- 如果不写出 =⟨*direction*⟩，就使用上一次所设置的 auto=left 或 auto=right 值。

本选项只对那些添加到（由 -- 创建的）线段或控制曲线上的 node 才有效。

此选项的定义是：

```
\tikzoption{auto}[]{\csname tikz@install@auto@anchor@#1\endcsname}%
```

与此选项相关的命令是

```
\def\tikz@install@auto@anchor@{\let\tikz@do@auto@anchor=\tikz@auto@anchor@on}%
\def\tikz@install@auto@anchor@false{\let\tikz@do@auto@anchor=\relax}%
\def\tikz@install@auto@anchor@left{\let\tikz@do@auto@anchor=\tikz@auto@anchor@on
↪  \def\tikz@auto@anchor@direction{left}}%
\def\tikz@install@auto@anchor@right{\let\tikz@do@auto@anchor=\tikz@auto@anchor@on
↪  \def\tikz@auto@anchor@direction{right}}%
\let\tikz@do@auto@anchor=\relax%
\def\tikz@auto@anchor@on{\csname tikz@auto@anchor@\tikz@auto@anchor@direction
↪  \endcsname}
\def\tikz@auto@anchor@left{\tikz@auto@pre\tikz@auto@anchor\tikz@auto@post}%
\def\tikz@auto@anchor@right{\tikz@auto@pre\tikz@auto@anchor@prime\tikz@auto@post}%
\def\tikz@auto@anchor@direction{left}%
```

/tikz/swap (no value)

如果 auto 选项决定了 left 一侧，那么 swap 选项就把 node 转换到 right 一侧，即它翻转路径的左右侧。

此选项的定义是：

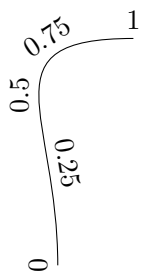
```
\tikzoption{swap}[]{%
  \def\tikz@temp{left}%
  \ifx\tikz@auto@anchor@direction\tikz@temp%
    \def\tikz@auto@anchor@direction{right}%
  \else%
    \def\tikz@auto@anchor@direction{left}%
  \fi%
}%
```

/tikz/' (no value)

这是 swap 选项的简写。

/tikz/sloped (no value)

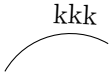
这个选项旋转 node，使其内容沿着路径的切线方向展开。假设 node 的锚定点是曲线（或曲线延伸线）上的点 p，则 node 的 south 位置锚定 p，并且 node 处于曲线“凸出”的一侧，也就是说，node 与曲线的曲率圆不在同一侧。



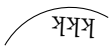

```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
  node foreach \p in {0,0.25,...,1} [sloped,above,pos=\p]{\p};
```

/tikz/allow upside down=⟨*boolean*⟩ (default true, initially false)

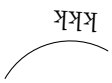
当 node 带有 allow upside down 或 allow upside down=true 时，该 node 的内容会“上下颠倒”。




```
\begin{tikzpicture}[auto]
  \draw (0,0) arc (60:150:1)node[pos=0.3,sloped]{kkk};
\end{tikzpicture}
```




```
\begin{tikzpicture}[auto]
  \draw (0,0) arc (60:150:1)node[pos=0.3,sloped,
    allow upside down]{kkk};
\end{tikzpicture}
```




```
\begin{tikzpicture}[auto]
  \draw (0,0) arc (60:150:1)node[pos=0.3,sloped,swap,
    allow upside down]{kkk};
\end{tikzpicture}
```

/tikz/midway (style, no value)

等于 pos=0.5.

/tikz/near start (style, no value)

等于 pos=0.25.

/tikz/near end (style, no value)

等于 pos=0.75.

/tikz/very near start (style, no value)

等于 pos=0.125.

/tikz/very near end (style, no value)

等于 pos=0.875.

/tikz/at start (style, no value)

等于 pos=0.

/tikz/at end (style, no value)

等于 pos=1.

44.8.1 收集 node 的命令

当“--”引起 line-to 操作时，执行的是 \tikz@lineto, 这个命令会识别“--node”这种符号组合，并执行 \tikz@collect@label@onpath, 这会把“--”后面的 node 语句保存起来，在适当的时候，再解析这些 node 语句，创建 node.

44.9 在直线段或曲线上隐式地摆放 node

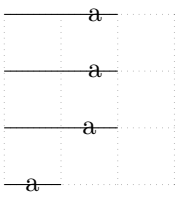
“隐式地（implicitly）”的意思是把 node 语句放在路径操作符号之后。在多数情况下，“显示”与“隐式”的效果是一样的，但程序对二者的处理方式不同，以致有特殊的差别。举例而言，如果 node 的内容中有抄录命令，那么显式的 (0,0) -- (1,1) node{\verb&\small&} 可以接受，但隐式的 (0,0) -- node{\verb&\small&} (1,1) 不能接受。

当读取

(0,0) -- ⟨*node specification*⟩ (1,1)

时，会把 ⟨*node specification*⟩ 另外保存，读完坐标 (1,1) 后，另存的 ⟨*node specification*⟩ 才被调出并处理。当 ⟨*node specification*⟩ 被保存时，其中各个符号的类别就已确定，因此 node 的内容中不能有抄录命令。

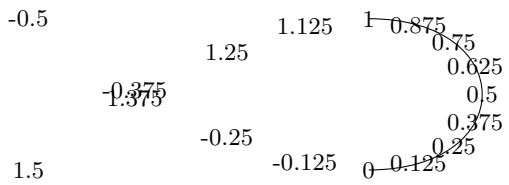
如果写出 (0,0)--node{a}(1,1), 那么 node 的内容 a 将位于点 (0,0) 与点 (1,1) 的中点上。如果写出 (x)--(y)--node{a}(z), 那么 node 的内容 a 将位于点 (y) 与点 (z) 的中点位置。




```
\tikz[scale=0.75]{
  \draw [help lines, dotted] (0,0) grid (3,3);
  \draw (0,0) -- node{a} (1,0);
  \draw (0,1) -- (1,1) -- node{a} (2,1);
  \draw (0,2) -- (1,2) -- node[pos=0.6]{a} (2,2);
  \draw (0,3) -- (1,3) -- (2,3) node[pos=0.6]{a};
}
```

如果把 node 语句放在 rectangle 操作之后，那么 node 将位于矩形的中心。


```
\begin{tikzpicture}
  \draw [rotate=30] [cyan] (0,0) rectangle node[sloped]{旋转} (1,0.5) ;
\end{tikzpicture}
```




```
\tikz\draw (0,0) ..  node foreach \p in
↪  {-0.5,-0.375,...,1.5} [pos=\p]{\small \p}controls
↪  +(right:2) and +(right:2) .. (0,2);
```

44.10 label 和 pin 选项

44.10.1 Overview

选项 label 和 pin 是用于 node 的选项。node 本身可以给它的锚定点加标签，而选项 label 和 pin 又可以要给 node 加标签。

44.10.2 label 选项

/tikz/label=[⟨*options*⟩]⟨*angle*⟩:⟨*text*⟩ (no default)

若一个 node 带有这个选项，该 node 完成后，tikz 会另创建一个 node 作为它的标签。这两个 node 暂时分别称为 main node 和 label node, 选项 `label` 的各个值解释如下：

1. ⟨*angle*⟩ 是 main node 的坐标系统中的“角度”，指定 main node 的坐标系统中的某个点，这个点就是需要加标签的点，即 label node 的锚定点。⟨*angle*⟩ 的值可以是数值（代表角度），锚位置名称（`north`, `east` 等），平移位置名称（`above`, `left` 等），tikz 序会把 `above` 转换成角度 90, 把 `left` 转换成角度 180, 等等。

   注意，对于 label node 来说，⟨*angle*⟩`=north east` 与 ⟨*angle*⟩`=above right` 的效果未必相同。⟨*angle*⟩`=north east` 确定的点是 main node 的锚位置 `north east`；而 ⟨*angle*⟩`=above right` 的作用则是平移 label node, 即假定从 main node 的中心点出发且角度是 45° 的射线与 main node 的边界线交于点 P, 点 P 就是 label node 的锚定点，平移 label node 使其锚位置 `south west` 与点 P 重合。

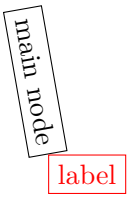
   如果不给出 ⟨*angle*⟩, 那么就默认使用下一个选项的值：

   `/tikz/label position`=⟨*angle*⟩ (no default, initially `above`)

   设置 label node 的默认位置。

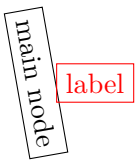
   `/tikz/absolute`=⟨*true or false*⟩ (default true)

   如果 main node 受到变换，例如旋转变换，tikz 会默认 main node 的坐标系统是固着在 main node 上随之一起旋转的，因此 label 选项中的 ⟨*angle*⟩ (所确定的 main node 边界上的点) 也会被旋转。如果设置 `absolute` 或 `absolute=true`, 那么 tikz 会认为 main node 的坐标系统是不随着 main node 一起旋转的，因而是绝对的（坐标轴的方向不变），不过 ⟨*angle*⟩ 所指出的点仍然在 main node 的边界上。这个选项所说的“绝对（absolute）”就是这个意思。


   ```
   \tikz [rotate=-80,every label/.style={draw,red}]
     \node [transform shape,rectangle,draw,
       label=right:label] {main node};
   ```

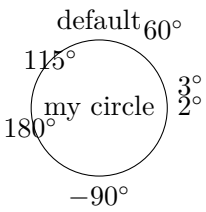
   上面例子中, main node 被旋转了 −80°, 它的 `right` 位置也被旋转了 −80°, label 标签锚定这个 `right` 位置，不过 label 标签的“`north west`”位置位于 main node 的 `right` 位置上。


   ```
   \tikz [rotate=-80,every label/.style={draw,red},absolute]
     \node [transform shape,rectangle,draw,
       label=right:label] {main node};
   ```

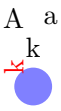
   上面例子中使用了 `absolute` 选项，main node 的 `right` 方向不跟随旋转，`right` 位置仍然在水平向右一侧的边界上，此时 label 标签的“`left`”位置处于 main node 的 `right` 位置上。

2. ⟨*angle*⟩ 确定了 label node 的锚定点，tikz 会把相应的 label node 的锚位置放在这个点上，规则是：0, 90, 180, 270, 这四个角度是“主角度（major angle）”；如果 ⟨*angle*⟩ 是 0 ± 2，即它确定的点在 main node 的 `east` 位置附近，则 label node 的 `west` 位置锚定该点；如果 ⟨*angle*⟩ 是 90 ± 2，即它确定的点在 main node 的 `north` 位置附近，则 label node 的 `south` 位置锚定该点；如果 ⟨*angle*⟩ 是 180 ± 2，即它确定的点在 main node 的 `west` 位置附近，则 label node 的 `east` 位置锚定该点；如果 ⟨*angle*⟩ 是 270 ± 2，即它确定的点在 main node 的 `south` 位置附近，则 label node 的 `north` 位置锚定该点；如果 ⟨*angle*⟩ 是其它角度，则按“就近”的原则确定 label node 的锚位置，例如，若 ⟨*angle*⟩ 是 30，即它确定的点在 main node 的 `north east` 位置附近，则 label node 的 `south west` 位置锚定该点。



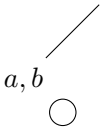

```
\tikz
  \node [circle, draw,
    label=default,
    label=60:$60^\circ$,
    label=below:$-90^\circ$,
    label=3:$3^\circ$,
    label=2:$2^\circ$,
    label={[below]180:$180^\circ$},
    label={[centered]135:$115^\circ$}] {my circle};
```

3. 如果 ⟨*angle*⟩ 是 `center`, 则 label node 会处于 main node 的中心。
4. 如果使用多个 `label` 选项，这些标签会依次画出，而且 `label` 选项可以套嵌使用，例如：



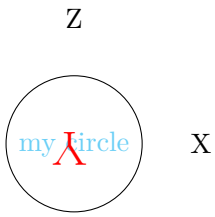

```
\tikz \node [circle,fill=blue!50,minimum size=0.5cm,
  label=90:k,
  label={[text=red,rotate=90,
    label={[absolute,label distance=1ex,
      label=right:a]90:A}]
    90:k}] {};
```

`label` 选项实际生成一个 node, 所以用于 node 的那些选项也能够成为 `label` 的选项，这些选项要放在 `angle` 之前的方括号里，此时还要把整个 `label` 选项的值用花括号括起来。




```
\begin{tikzpicture}
  \node [circle,draw,label={[name=label node]above left:$a,b$}] {};
  \draw (label node) -- +(1,1);
\end{tikzpicture}
```

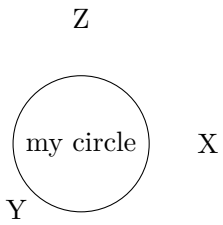
如果 `label` 带有 `rotate` 选项，则 label node 会围绕它的锚定点整体（包括它的内容）旋转。




```
\tikz[label distance=5mm]
  \node [circle,draw,label=right:X,
    label={[rotate=180,red]above right:{\LARGE Y}},
    label=above:Z] {\color{cyan!50}my circle};
```

`/tikz/label distance`=⟨*distance*⟩ (no default, initially `0pt`)

这个选项设置 label node 的锚定点与 label node 的锚位置之间的距离，可以是负值尺寸。




```
\tikz[label distance=5mm]
  \node [circle,draw,label=right:X,
    label={[label distance=-2.5cm]above right:Y},
    label=above:Z] {my circle};
```

`/tikz/every label` (style, initially `draw=none,fill=none`)

设置每个 label node 的样式，默认值 `draw=none, fill=none`.

44.10.3 The Pin Option

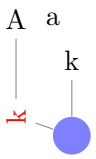
`/tikz/pin`=`[`⟨*options*⟩`]`⟨*angle*⟩`:`⟨*text*⟩ (no default)

这个选项与 `label` 选项类似，用作 node 的选项，给该 node 添加一个作为标签的 node, 所加的标签像大头针。⟨*angle*⟩ 确定 main node 坐标系统中的点，是大头针的针尖所锚定的位置；针后端是与 ⟨*angle*⟩ 相应的 label node 的锚位置。

可以给一个 node 加多个 pin 选项，产生多个标签。

pin 选项可以与 pin 选项套嵌使用，也可以与 label 选项套嵌使用。

如果 pin 带有 rotate 选项，标签会围绕它的锚定点整体（包括它的内容）旋转，但是“针”并未必能获得正确的方位。




```
\tikz \node [circle,fill=blue!50,minimum size=0.5cm,
  pin=90:k,
  pin={[text=red,rotate=90,
    pin={[absolute,pin distance=5ex,
      label=right:a]90:A}]
    90:k}] {};
```

/tikz/pin distance=⟨*distance*⟩ (no default, initially 3ex)

设置 pin 标签的锚位置与其锚定点的距离。

/tikz/every pin (style, initially draw=none,fill=none)

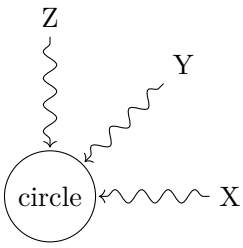
设置每个 pin 标签的样式。

/tikz/pin position=⟨*angle*⟩ (no default, initially above)

类似 label position，设置默认的锚定点。

/tikz/every pin edge (style, initially help lines)

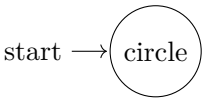
设置每个 pin 标签的“针”的外观，即“edge”的样式。




```
\tikz [pin distance=15mm,
    every pin edge/.style={<-,shorten <=1pt,decorate,
    decoration={snake,pre length=4pt}}]
  \node [circle,draw,
    pin=right:X,
    pin=above right:Y,
    pin=above:Z] {circle};
```

/tikz/pin edge=⟨*options*⟩ (no default, initially empty)

设置 pin 标签的“针”，即“edge”的外观。注意 pin 选项中的颜色、线宽、线型等选项只是针对大头针的“头”的设置，“针”（即边）的默认样式为 help lines, 如果需要修改其外观就得用这个选项单独设置。




```
\tikz [every pin edge/.style={},
    initial/.style={pin={[pin distance=5mm,
      pin edge={<-,shorten <=1pt}]left:start}}]
  \node [circle,draw,initial] {circle};
```

44.10.4 引用句法

TikZ Library quotes

```
\usetikzlibrary{quotes} % LaTeX and plain TeX
\usetikzlibrary[quotes] % ConTeXt
```

载入这个库后，可以把双引号引起来的一串符号作为 node 的选项，这串符号会成为 node 的标签。可以用这个办法给 node, label, pin, pic, edge 加标签，这个加标签办法简捷一些。

引用句法不使用 ⟨*key*⟩=⟨*value*⟩ 这种赋值句式，其句式为：

"⟨*text*⟩"⟨*options*⟩

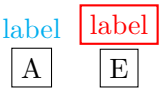
TikZ 会检查一串符号是否以双引号开头（TikZ 的 key 不以双引号开头），若是则确认为引用句法。

- 在 ⟨*options*⟩ 中的选项就是那些能用于 node 的选项，不过这些选项 不用方括号括起来。
- 如果 ⟨*options*⟩ 中有逗号 (相邻两个选项之间就是用逗号分隔的)，就必须用花括号把整个 ⟨*options*⟩ 括起来，否则可以不加花括号（选项前可以加空格）。

在默认下，上面的句式会被转换为

label={[⟨*options*⟩]⟨ 平移位置⟩:⟨*text*⟩}

所以可用于 label 的选项也可以用于引用句式。




```
\tikz \node ["label" color=cyan,draw] {A};
\tikz \node [draw, "label" {red,draw,thick,below,label distance=-11mm}] {E};
```

/tikz/quotes mean label (no value)

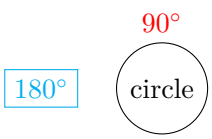
这个选项的字面意思是：“引用内容意味着 label”，就是将引用句法转换为 label 选项的句法。当调用 quotes 库后，这个选项是默认的。

/tikz/every label quotes (style, no value)

为那些由引用句法产生的、等效于 label 选项句法的标签（node）设置外观样式。

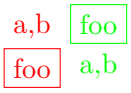
关于"⟨*text*⟩"⟨*options*⟩ 要注意以下几点：

- 在默认下，可以在 ⟨*text*⟩ 中可以使用 “⟨*direction*⟩:⟨*actual text*⟩ ” 的形式来设置标签的方位和内容（这原本是属于 label 选项的用法），例如：



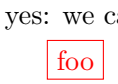

```
\tikz \node ["90:$90^\circ$" red,
             "west:$180^\circ$" {cyan,draw,label distance=5mm},
             circle, draw] {circle};
```

- 因为 Ti*k*Z 把逗号当作分隔两个 key 的标志，所以如果 ⟨*text*⟩ 的文字内容含有逗号，就必须用花括号把逗号括起来，或者用花括号把文字内容括起来。




```
\tikz [red]\node ["{a,b}", draw] {foo};
\tikz [green]\node ["-90:a{,}b", draw] {foo};
```

- 冒号可以用于极坐标，如 (30:1)，或标签，如 ⟨*angle*⟩:⟨*text*⟩（冒号的作用是将方向、方位与其它内容分隔开来），所以如果 ⟨*text*⟩ 的文字内容含有冒号，就必须用花括号把冒号括起来，或者用花括号把文字内容括起来。




```
\tikz \node [red, "yes{:} we can", draw] {foo};
```

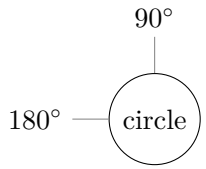
- 在 ⟨*options*⟩ 部分可以使用撇号 “'”，撇号（apostrophe）一般是选项 swap 的简写。撇号的使用规则如下：

```
"foo"'         等效于 "foo" {'}
"foo"' red     等效于 "foo" {',red}
"foo"'{red}    等效于 "foo" {',red}
"foo"{',red}   等效于 "foo" {',red}
"foo"{red,'}   等效于 "foo" {red,'}
"foo" {'red}   是非法句式，因为 tikz 会把“'red”当作一个选项
"foo" {red'}   是非法句式，因为 tikz 会把“red'”当作一个选项
```

这个规则的要点是：在撇号与某个选项之间没有逗号分隔时，不要用花括号把撇号与该选项“绑在一起”。

/tikz/quotes mean pin (no value)

这个选项的字面意思是：引用内容意味着 pin，就是将引用句法转换为 pin 选项的句法，故可用于 pin 的那些选项也可以用于引用句法。




```
\tikz [quotes mean pin]
  \node ["$90^\circ$" above, "$180^\circ$" left, circle, draw] {circle
  ↪  };
```

/tikz/every label quotes (style, no value)

为那些由引用句法产生的、等效于 pin 选项句法的标签（node）设置外观样式。

/tikz/node quotes mean=⟨*replacement*⟩ (no default)

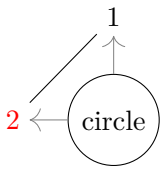
这个 key 可以规定引用句法的实际作用。当把"⟨*text*⟩"'⟨*options*⟩ 用作选项后，tikz 就会用 ⟨*replacement*⟩ 来替换它。

⟨*replacement*⟩ 是一组 key 设置，其中包含 #1 和 #2 两个变量。⟨*text*⟩ 对应 #1，⟨*options*⟩ 对应 #2，tikz 会使用 \pgfkeys 来解析 ⟨*replacement*⟩.

在文件《tikzlibraryquotes.code》中有如下代码：

```
\tikzset{
% 省略若干
  quotes mean pin/.style={node quotes mean={
      pin={[direction shorthands,every pin quotes/.try,##2]##1}}},
  quotes mean label/.style={node quotes mean={
      label={[direction shorthands,every label quotes/.try,##2]##1}}},
  quotes mean label,
% 省略若干
}
```

以上代码定义了 quotes mean pin 和 quotes mean label，还启用了 quotes mean label。




```
\tikzset{
  node quotes mean={
    pin={[#2,pin distance=0.5cm,
    pin edge={-{>[scale=2]}},
    name={#1}]#1}
  }
}
\tikz {
  \node ["1", "2" {red,pin position=left}, circle, draw] {circle};
  \draw (1) -- (2);
  }
```

44.11 Connecting Nodes: Using Nodes as Coordinates

44.12 Connecting Nodes: 用 edge 操作

44.13 Referencing Nodes Outside the Current Picture

44.13.1 Referencing a Node in a Different Picture

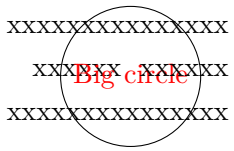
通常，一个 {tikzpicture} 环境创建一个图片，一个图片内的 node 坐标系统中的点只能在本图片内引用，因为完成一个图片后，TikZ 就会忘记该图片中各个点在页面上的位置。通过适当操作可以“跨图

片”引用 node 坐标系统中的点。

如果要在图形 G_2 中引用图形 G_1 中的点（例如，将 G_2 中的某个点与 G_1 中的某个点用线连起来），那么首先要让程序记住这两个图形中的点在页面中的位置，然后才能跨图引用这些点。当给 G_1 和 G_2 的环境选项都加上选项 `remember picture` 后，Ti*k*Z 会记住图形中的点在页面中的位置，这需要后台驱动的支持（如果驱动不支持就不能实现这一点），并且还需要运行 TEX 两次才能引用这些点。

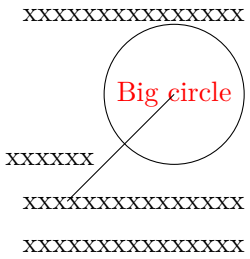
当在图形 G_2 中引用图形 G_1 中的点时，Ti*k*Z 总是会试图扩大图形 G_2 的边界盒子（bounding box）以包含图形 G_1 中被引用的点，这样图形 G_2 就可能变得很大以致页面比较难看。这就需要给图形 G_2 的环境选项加 `overlay` 选项，或者给图形 G_2 中的那个引用图形 G_1 中点的子环境或路径加 `overlay` 选项。

如果图形的某个环境选项中有 `overlay` 选项，那么该环境内的所有内容不影响图形边界盒子的位置、尺寸。




```
xxxxxxxxxxxxxxx \\
xxxxxx
\begin{tikzpicture}[overlay]
  \node (c) [circle,draw,text=red] {Big circle};
\end{tikzpicture}
xxxxxx
xxxxxxxxxxxxxxx
```

如果图形中的某个路径有 `overlay` 选项，那么该路径不影响图形边界盒子的位置、尺寸。




```
xxxxxxxxxxxxxxx \\
xxxxxx
\begin{tikzpicture}
  \node (c) [circle,draw,text=red] {Big circle};
  \draw [overlay](0,0)--(-135:2);
\end{tikzpicture}
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx
```

`/tikz/remember picture`=⟨*boolean*⟩ (no default, initially `false`)

这个选项决定是否让 Ti*k*Z 记住当前图形在页面中的位置。下面的设置：

```
\tikzset{every picture/.append style={remember picture}}
```

执行这个设置命令后，tikz 会记住此后所有图形在页面中的位置。这会在.aux 文件中的每个 picture 中加上一行代码。

`/tikz/overlay`=⟨*boolean*⟩ (default `true`)

看一个例子。先画一个 node (带填充色的圆)：

```
\tikz[remember picture]
  \node[circle,fill=red!50] (n1) {};
```

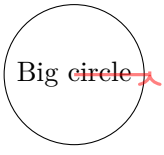
再画一个 node （方块）：

```
\tikz[remember picture]
  \path(0,0) (1,0)node[fill=blue!50] (n2) {};
```

将上面两个图形中的名称为 `(n1)`, `(n2)` 的 node 用带箭头的（黑色）线段连起来：

```
\tikz[remember picture,overlay]
  \draw[->,very thick] (n1) -- (n2);
```

在 `(n1)` 与 `(n2)` 之间再画一条（红色）线：




```
\begin{tikzpicture}[remember picture]
  \node (c) [circle,draw] {Big circle};
  \draw [overlay,->,very thick,red,opacity=.5]
    (c) to[bend left] (n1) -| (n2);
\end{tikzpicture}
```

注意，在跨图引用 node 时，所涉及的各个图形应当在同一个页面内，否则结果会很意外。

44.13.2 引用 Current Page Node——绝对位置

有个名称是 current page 的预定义 node, 它对应的是“当前页面”，它的形状是矩形，它的锚位置 south west 是当前页面的左下角，它的锚位[illegible]或命令加上 remember picture 和 overlay 选项后，就可以在[illegible]统中的位置。

This is an absolutely positioned text in the lower left corner. No shipout-hackery is used.

下面的代码在当前页面的左下角添加文字：

```
\begin{tikzpicture}[remember picture,overlay]
  \node at (current page.south west)
    [text width=7cm,fill=red!20,rounded corners,above right]
  {
This is an absolutely positioned text in the
lower left corner. No shipout-hackery is used.
  };
\end{tikzpicture}
```

下面的代码在当前页面的中心添加文字：

```
\begin{tikzpicture}[remember picture,overlay]
  \node [rotate=60,scale=10,text opacity=0.2]
    at (current page.center) {Example};
\end{tikzpicture}
```

可以用这个办法给正文内容做一个边注：

大佛頂如來密因修證了義諸菩薩萬行首楞嚴經：楞嚴經：

```
大佛頂如來密因修證了義諸菩薩萬行首%
\tikz[baseline=(lyj.base),inner sep=0pt, remember picture]{
  \node(lyj) {楞嚴經};
  \node [text=red,font=\kaishu,text width=1.5cm, below right=0pt and 2mm,
         overlay]at(lyj.north-|current page.west)
    {楞嚴經：注释注释注释注释注释注释};}%
文句
```

44.14 Late Code and Late Options

假设已经创建了名称为 ⟨*name*⟩ 的 node, 但在之后的编辑过程中发现需要给 ⟨*name*⟩ 添加某些选项来对它做修改，此时可以用下面的句法或选项来实现这种“事后修改”。

`\path ... node also[`⟨*late options*⟩`](`⟨*name*⟩`)...;`

其中的 ⟨*name*⟩ 是之前创建的 node 名称，是待修改的 node. 方括号里的选项 ⟨*late options*⟩ 是需要添加给 ⟨*name*⟩ 的选项，这些选项会被放在一个局部域中执行。注意，⟨*name*⟩ 原来的外观是不能被直接改变的，也就是说 ⟨*late options*⟩ 中的选项可能不会直接起作用，例如，若 ⟨*name*⟩ 原来的填充色是红色，那么 ⟨*late options*⟩ 中的 fill=green 就不能把 ⟨*name*⟩ 的填充色改为绿色。在 ⟨*late options*⟩ 中使用选项 append after command, prefix after command 可能会有帮助。

world
Hello

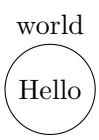
```
\begin{tikzpicture}
  \node [draw, circle, fill=red] (a) {Hello};
  \node also [label=above:world, fill=green] (a);
\end{tikzpicture}
```

\tikzlastnode

这个宏代表之前的、最近创建的 node 的名称。由于这个宏是局部定义的，所以其有效性受到组的限制，通常只在当前路径内使用。

/tikz/late options=⟨*options*⟩ (no default)

这个选项可以用作路径的选项，但不能用作 node 的选项。这个选项也能实现 node also 操作的功能，但必须在 ⟨*options*⟩ 中使用 name=⟨*name*⟩ 指定所针对的 node：




```
\begin{tikzpicture}
  \node [draw,circle] (a) {Hello};
  \path [late options={name=a, label=above:world}];
\end{tikzpicture}
```

44.15 TikZ 解析 node 的命令

在一个 TikZ 路径中，当遇到字母“n”时，一般会引起命令 \tikz@fig node, 参考 \tikz@handle → P. 757. TikZ 处理 node 的基本过程是：

1. \tikz@fig node
2. \tikz@normal@fig
3. \tikz@@scan@fig
4. \tikz@@fig@main
5. \tikz@do@fig
6. \tikz@fig@collectresetcolor
7. \tikz@fig@boxdone
8. \tikz@fig@continue
9. \tikz@node@finish
10. \tikz@do@after@node

\tikz@fig ode

```
\def\tikz@fig ode{%
  \pgfutil@ifnextchar a\tikz@test@also{
    \pgfutil@ifnextchar f{\tikz@nodes@start}\tikz@normal@fig}}%
```

这个命令对应“node”这个单词。

\tikz@test@also a

```
\def\tikz@test@also a{\pgfutil@ifnextchar l\tikz@node@also{\tikz@normal@fig a}}%
```

这个命令检查是否 \path ... node also...;, 如果是，就执行 \tikz@node@also.

\tikz@nodes@start

处理 node foreach 句式。

\tikz@normal@fig

这个命令处理通常的 node, 其处理是：

1. 保存线宽 \edef\tikz@save@line@width{\the\pgflinewidth}
2. 用 \begingroup 开启一个组
3. 清空 \let\tikz@fig@name=\pgfutil@empty

4. 用 \begingroup 开启一个组
5. 清空或重置

```
\tikz@is@matrixfalse%
\let\nodepart=\tikz@nodepart%
\let\tikz@atbegin@scope=\pgfutil@empty%
\let\tikz@atend@scope=\pgfutil@empty%
\let\tikz@do@after@node=\tikz@scan@next@command%
\let\tikz@options=\pgfutil@empty%
\tikz@clear@rdf@options%
\let\tikz@id@name=\pgfutil@empty%
\let\tikz@after@path=\pgfutil@empty%
\let\tikz@transform=\pgfutil@empty%
\let\tikz@mode=\pgfutil@empty%
\tikz@decoratepathfalse%
\let\tikz@preactions=\pgfutil@empty%
\let\tikz@postactions=\pgfutil@empty%
\let\tikz@alias=\pgfutil@empty%
```

   其中 \tikz@transform 被 let 为 \pgfutil@empty, 这将使得坐标变换命令被保存到 \tikz@transform 中，参考 \tikz@addtransform→P. 881.
6. 定义 \def\tikz@node@at{\pgfqpoint{\the\tikz@lastx}{\the\tikz@lasty}}, 宏 \tikz@node@at 保存的是 node 的锚定点，初始之下，锚定点是“当前点”。
7. 规定 \let\tikz@time@for@matrix\tikz@time
8. 规定 \let\tikz@node@content\relax
9. 保存当前的软路径 \pgfgetpath\tikzpathuptonow
10. 检查 node 是否路径标签，即，检查是否要把 node (用“显式”或“隐式”的方式) 沿着路径放置

```
\iftikz@node@is@a@label%
\else%
    \let\tikz@time\pgfutil@empty%
\fi%
```

11. 执行 \tikz@node@reset@hook, 这个钩子的初始值是 \pgfutil@empty
12. 执行 \tikzset{every node/.try}
13. 执行 \tikz@@scan@fig

\tikz@@scan@fig

这个命令执行一系列的 if 判断，参考 \pgfutil@ifnextchar→P. 29.

```
\def\tikz@@scan@fig{%
  \pgfutil@ifnextchar a{\tikz@fig@scan@at}
  {\pgfutil@ifnextchar({\tikz@fig@scan@name}
    {\pgfutil@ifnextchar[{\tikz@fig@scan@options}%
      {\pgfutil@ifnextchar:{\tikz@fig@scan@animation}%
        {\pgfutil@ifnextchar\bgroup{\tikz@fig@main}%
          {\tikzerror{A node must have a (possibly empty) label text}%
            \tikz@fig@main{}}}}}}}%}}%
```

这个命令检查 node 的锚定点（at），名称（开圆括号 (），选项（开方括号 [），动画设置（冒号 :），内容（开花括号 {, 即 \bgroup）。当遇到 \bgroup 时，执行 \tikz@fig@main.

\tikz@fig@scan@at at

本命令解析“at(⟨TikZ coordinate⟩)”这一部分。参考 \tikz@scan@one@point→P. 710.

```
\def\tikz@fig@scan@at at{%
  \tikz@scan@one@point\tikz@@fig@scan@at}%
\def\tikz@@fig@scan@at#1{%
  \def\tikz@node@at{#1}\tikz@@scan@fig}%
```

解析 ⟨*TikZ coordinate*⟩ 的结果（PGF 的点）被保存到 `\tikz@node@at`.

注意前面的命令 `\tikz@normal@fig` →P. 835 先定义：

```
\def\tikz@node@at{\pgfqpoint{\the\tikz@lastx}{\the\tikz@lasty}}
```

本命令最后回过头来执行 `\tikz@@scan@fig`.

`\tikz@fig@scan@name`(⟨*node name*⟩)

本命令解析圆括号里的 node 名称。

```
\def\tikz@fig@scan@name(#1){%
    \pgfkeysvalueof{/tikz/name/.@cmd}#1\pgfeov
    ↪  % CF : this is now ALWAYS consistent with 'name=' option; allows overrides.
    \tikz@@scan@fig}%
```

`/tikz/name=`⟨*node name*⟩

这个选项的定义是：

```
% General node options
\tikzset{
  name/.code={\edef\tikz@fig@name{\tikz@pp@name{#1}}\let\tikz@id@name
  ↪  \tikz@fig@name},%
  name prefix/.initial=,%
  name suffix/.initial=%
}%
\def\tikz@pp@name#1{\csname pgfk@/tikz/name prefix\endcsname#1\csname
↪  pgfk@/tikz/name suffix\endcsname}%
```

可见 node 的名称保存在 `\tikz@fig@name` 中，而且是带有前缀、后缀的名称。

`\tikz@fig@scan@options`[⟨*options*⟩]

本命令解析 node 后面方括号里的选项。

```
\long\def\tikz@fig@scan@options[#1]{%
    \iftikz@node@is@pic%
        \tikz@enable@pic@quotes%
    \else%
        \tikz@enable@node@quotes%
    \fi%
    \tikzset{#1}%
    \ifx\tikz@node@content\relax%
        \expandafter\tikz@@scan@fig%
    \else%
        \tikz@expand@node@contents%
    \fi
}%
```

本命令会检查 `\iftikz@node@is@pic` 的真值，因为命令 `\pic` 也会使用相同的命令来解析句子成分。在执行 ⟨*options*⟩ 后，还会回头执行 `\tikz@@scan@fig` 或者 `\tikz@@scan@fig{`⟨*node 的内容*⟩`}`（如果使用了选项 `node contents={`⟨*node 的内容*⟩`}` 的话）。可见，如果使用了选项 `node contents={`⟨*node 的内容*⟩`}`, 就不要再用花括号设置 `{`⟨*node 的内容*⟩`}`.

`\tikz@node@content`

这个宏与选项 `/tikz/node contents` →P. 802, `/tikz/pic type` →P. 848 对应：

```
\tikzset{
  node contents/.code=\def\tikz@node@content{#1},
  pic type/.code=\def\tikz@node@content{#1}, % alias
}%
```

这个宏保存键 node contents 的值，即 node 的内容。

\tikz@expand@node@contents

```
\def\tikz@expand@node@contents{%
  \expandafter\tikz@@scan@fig\expandafter{\tikz@node@content}%
}%
```

本命令调用 \tikz@@scan@fig 来解析保存在 \tikz@node@content 中的内容，导致

\tikz@fig@main{⟨*exp* 展开的 *\tikz@node@content*⟩}

```
\tikzset{
  behind path/.code=\def\tikz@whichbox{\tikz@figbox@bg},
  in front of path/.code=\def\tikz@whichbox{\tikz@figbox}
}%
\def\tikz@whichbox{\tikz@figbox}%
```

这两个选项决定把 node 放入哪个盒子里，盒子 \tikz@figbox@bg 会在画出路径之前插入文档，因此可能被路径遮挡；盒子 \tikz@figbox 会在画出路径之后插入文档，因此可能遮挡路径。参考 \tikz@finish →P. 759.

```
\let\tikz@node@reset@hook=\pgfutil@empty%
\let\tikz@node@begin@hook=\pgfutil@empty%
```

\tikz@fig@main

```
\def\tikz@fig@main{%
  \iftikz@node@is@pic%
    \tikz@node@is@picfalse%
    \expandafter\tikz@subpicture@handle%
  \else%
    \afterassignment\tikz@@fig@main\expandafter\let\expandafter\next\expandafter=%
  \fi}%
```

因为命令 \pic 也会使用相同的命令来解析句子成分，所以这里要检查是否 pic 的句子，如果是，则执行 \tikz@subpicture@handle, 正式进入 pic 句子的构建。如果不是 pic 的句子，则

\afterassignment\tikz@@fig@main\let\next={⟨*node* 的内容⟩}

这会把开花括号 { 吃掉，并保存到 \next 中，再执行 \tikz@@fig@main, 也就是

\tikz@@fig@main⟨*node* 的内容⟩}

在形式上，最右侧多了一个 }, 这需要在 \tikz@@fig@main 中给出 \bgroup 来平衡它（实际是在 \tikz@do@fig 中给出）。

\tikz@@fig@main

其定义是：

```
\def\tikz@@fig@main{%
    \pgfutil@ifundefined{pgf@sh@s@\tikz@shape}%
    {\tikzerror{Unknown shape ``\tikz@shape.'' Using ``rectangle'' instead}%
      \def\tikz@shape{rectangle}}%
    {}%
    \expandafter\xdef\csname tikz@dcl@coord@\tikz@fig@name\endcsname{%
```

```
      \csname tikz@scan@point@coordinate\endcsname}%
    \tikzset{every \tikz@shape\space node/.try}%
    \tikz@node@textfont%
    \tikz@node@begin@hook%
    \iftikz@is@matrix%
      \let\tikz@next=\tikz@do@matrix%
    \else%
      \let\tikz@next=\tikz@do@fig%
    \fi%
    \tikz@next%
}%
```

本命令：

1. 检查形状 \tikz@shape 是否已定义，如果未定义就报错。
2. 将之前被解析的 TikZ 点 \tikz@scan@point@coordinate 全局地保存到 \csname tikz@dcl@coord@\tikz@f
3. 执行样式选项 every \tikz@shape node
4. 声明字体
5. 执行钩子 \tikz@node@begin@hook, 它的初始值是 \pgfutil@empty
6. 检查 \iftikz@is@matrix 的真值，如果不是 matrix, 就执行 \tikz@do@fig

```
\let\tikz@nodepart@list\pgfutil@empty
```

\tikz@do@fig

本命令的定义是：

```
\def\tikz@do@fig{%
    % Ok, reset all node part boxes
    \pgfutil@for\tikz@temp:=\tikz@nodepart@list\do{%
      \expandafter\setbox\csname pgfnodepart\tikz@temp box\endcsname=
      ↪  \box\pgfutil@voidb@x%
    }%
    \setbox\pgfnodeparttextbox=\hbox%
      \bgroup%
        \pgfinterruptpicture%
        \pgfsys@begin@text%
          \pgfsys@text@to@black@hook%
          \tikzset{every text node part/.try}%
          \ifx\tikz@textopacity\pgfutil@empty%
          \else%
            \pgfsetfillopacity{\tikz@textopacity}%
            \pgfsetstrokeopacity{\tikz@textopacity}%
          \fi%
          \ifx\tikz@text@width\pgfutil@empty%
            \tikz@textfont%
          \else%
            \begingroup%
                \pgfmathsetlength{\pgf@x}{\tikz@text@width}%
              \pgfutil@minipage[t]{\pgf@x}\leavevmode\hbox{}%
                \tikz@textfont%
                \tikz@text@action%
          \fi%
          \tikz@atbegin@node%
          \bgroup%
            \aftergroup\unskip%
            % Some color stuff has been moved from here to outside; this is
            % necessary for support of dvisvgm and of animation
```

```
            % snapshots.
            \ifx\tikz@textcolor\pgfutil@empty%
            \else%
              \pgfutil@colorlet{.}{\tikz@textcolor}%
              \pgfutil@color{\tikz@textcolor}%
            \fi%
            \setbox\tikz@figbox=\box\pgfutil@voidb@x%
            \setbox\tikz@figbox@bg=\box\pgfutil@voidb@x%
            \tikz@uninstallcommands%
            \iftikz@handle@active@code%
              \tikz@orig@shorthands%
              \let\tikz@orig@shorthands\pgfutil@empty%
            \fi%
            \ifnum\the\catcode`\;=
            ↪  \active\relax\expandafter\let\tikz@activesemicolon=\tikz@origsemi
            ↪  \fi%
            \ifnum\the\catcode`\:=\active\relax\expandafter\let\tikz@activecolon=
            ↪  \tikz@origcolon\fi%
            \ifnum\the\catcode`\|=\active\relax\expandafter\let\tikz@activebar=
            ↪  \tikz@origbar\fi%
            \aftergroup\tikz@fig@collectresetcolor%
            \tikz@signal@halign@check%
            \tikz@text@reset%
            \tikz@halign@check%
            \ignorespaces%
}%
```

关于这个命令：

1. \tikz@nodepart@list 中保存的应该是一个用逗号分隔的列表，列表项是 shape 的各个文字部分的名称。先用 \pgfutil@for→P. 31 定义各个文字盒子的初始状态。
2. 开始盒子 \pgfnodeparttextbox 的定义：

   ```
   \setbox\pgfnodeparttextbox=\hbox%
       \bgroup
       ...
   ```

   这个定义会在命令 \tikz@fig@boxdone 那里结束。
3. 盒子 \pgfnodeparttextbox 的开头使用了 \pgfinterruptpicture.
4. 选项 /tikz/text width→P. 813 会定义宏 \tikz@text@width.
   这个宏的初始值是 (被 let 为) \pgfutil@empty.
   如果这个宏不等于 \pgfutil@empty, 则设置一个 minipage 来盛放 node 的文字内容，这个 minipage 的宽度就是 \tikz@text@width. 这个 minipage 将在 \tikz@fig@boxdone 那里结束。这个 minipage 被放在一个组中。
5. \tikz@atbegin@node 与选项 /tikz/execute at begin node→P. 806 相关。
6. 设置一个 \bgroup, 在与这个 \bgroup 对应的 \egroup 后面将会执行

   ```
   \unskip
   \tikz@fig@collectresetcolor
   ```

7. 在文件《pgfutil-latex.def》中有:

   ```
   \def\pgfutil@definecolor{\definecolor}
   \def\pgfutil@color{\color}
   \def\pgfutil@colorlet#1#2{%
     % If the color is a defined named color, we have to use the [named]
   ```

```
  % option for colorlet to force xcolor to perform color model
  % conversion.
  \expandafter\ifx\csname\expandafter\string\csname color@#2\endcsname
  ↪  \endcsname\relax
    \colorlet{#1}{#2}%
  \else
    \colorlet[named]{#1}{#2}%
  \fi
}
%......
\let\pgfutil@minipage=\minipage
\let\pgfutil@endminipage=\endminipage
```

8. 处理活动符。
9. 执行 \tikz@signal@halign@check 来处理选项 /tikz/node halign header →P. 814 的意图。
10. 然后

```
\ignorespaces⟨node 的内容⟩}
```

其中最右侧的闭花括号 } 与前面的 \bgroup 对应，在这个 } 之后会执行

```
\unskip
\tikz@fig@collectresetcolor
```

\tikz@fig@collectresetcolor

其定义是：

```
\def\tikz@fig@collectresetcolor{%
  % Hacks for special packages that mess with \aftergroup
  \pgfutil@ifnextchar\reset@color% hack for color package
  {\reset@color\afterassignment\tikz@fig@collectresetcolor\let\tikz@temp=}
  ↪  \tikz@fig@boxdone%
}%
```

这里的 \reset@color 是一个占位符。本命令最终导致执行 \tikz@fig@boxdone.

\tikz@fig@boxdone

其定义是：

```
\def\tikz@fig@boxdone{%
            \tikz@atend@node%
          \ifx\tikz@text@width\pgfutil@empty%
          \else%
              \pgfutil@endminipage%
            \endgroup%
          \fi%
        \pgfsys@end@text%
        \endpgfinterruptpicture%
      \egroup%
    \pgfutil@ifnextchar c{\tikz@fig@mustbenamed\tikz@fig@continue}%
    {\pgfutil@ifnextchar[{\tikz@fig@mustbenamed\tikz@fig@continue}%
      {\pgfutil@ifnextchar t{\tikz@fig@mustbenamed\tikz@fig@continue}
        {\pgfutil@ifnextchar e{\tikz@fig@mustbenamed\tikz@fig@continue}
          {\ifx\tikz@after@path\pgfutil@empty\expandafter\tikz@fig@continue
          ↪  \else\expandafter\tikz@fig@mustbenamed\expandafter\tikz@fig@continue
          ↪  \fi}}}}}%}%
```

关于这个命令：

1. \tikz@atend@node 与选项 /tikz/execute at end node →P. 806 相关。

2. 如果 \tikz@text@width 不是空的，则结束之前开启的 minipage.
3. 执行 \endpgfinterruptpicture.
4. 执行 \egroup 结束之前对盒子 \pgfnodeparttextbox 的定义。
5. 继续解析之后的记号，
   - 如果之后的一个记号是 c，[，t，e 之一，则

     ```
     \tikz@fig@mustbenamed\tikz@fig@continue
     ```

   - 否则，检查 \tikz@after@path 是否非空，
     - 如果是空的，则

       ```
       \tikz@fig@continue
       ```

     - 如果不是空的，则

       ```
       \tikz@fig@mustbenamed\tikz@fig@continue
       ```

     宏 \tikz@after@path 与选项 /tikz/append after command→P. 719, /tikz/prefix after command→P. 719 有关。

\tikz@fig@mustbenamed

其定义是：

```
\def\tikz@fig@mustbenamed{%
  \ifx\tikz@fig@name\pgfutil@empty%
    % Assign a dummy name
    \global\advance\tikz@fig@count by1\relax
    \edef\tikz@fig@name{tikz@f@\the\tikz@fig@count}%
    \let\tikz@id@name\tikz@fig@name%
  \fi%
}%
```

如果当前的 node 没有名称，就为它创建一个名称：

```
\edef\tikz@fig@name{tikz@f@\the\tikz@fig@count}
```

\tikz@fig@continue

其定义是：

```
\def\tikz@fig@continue{%
    \ifx\tikz@text@width\pgfutil@empty%
    \else%
      \pgfmathsetlength{\pgf@x}{\tikz@text@width}%
      \wd\pgfnodeparttextbox=\pgf@x%
    \fi%
    \ifx\tikz@text@height\pgfutil@empty%
    \else%
      \pgfmathsetlength{\pgf@x}{\tikz@text@height}%
      \ht\pgfnodeparttextbox=\pgf@x%
    \fi%
    \ifx\tikz@text@depth\pgfutil@empty%
    \else%
      \pgfmathsetlength{\pgf@x}{\tikz@text@depth}%
      \dp\pgfnodeparttextbox=\pgf@x%
    \fi%
    %
    % Node transformation
    %
    \tikz@node@transformations%
```

```
    \tikz@nlt%
    %
    \setbox\tikz@whichbox=\hbox{%
      \unhbox\tikz@whichbox%
      \hbox{{%
          \pgfinterruptpath%
            \pgfscope%
              \tikz@options%
              \tikz@do@rdf@pre@options%
              \tikz@is@nodetrue%
              \tikz@call@id@hook%
              \pgfidscope%
                \tikz@do@rdf@post@options%
                \let\tikz@id@name\pgfutil@empty%
                \setbox\tikz@figbox=\box\pgfutil@voidb@x%
                \setbox\tikz@figbox@bg=\box\pgfutil@voidb@x%
                % Add color modifications to text box
                \setbox\pgfnodeparttextbox=\hbox{{%
                    \pgfsys@begin@text% Colors moved here...
                    \ifx\tikz@textcolor\pgfutil@empty%
                    \else%
                      \pgfutil@colorlet{.}{\tikz@textcolor}%
                    \fi%
                    \pgfsetcolor{.}%
                    \pgfusetype{.text}%
                    \pgfidscope%
                      \box\pgfnodeparttextbox%
                    \endpgfidscope%
                    \pgfsys@end@text%
                  }}%
                \pgfmultipartnode{\tikz@shape}{\tikz@anchor}{\tikz@fig@name}{%
                  \pgfutil@tempdima=\pgflinewidth%
                  {\begingroup\tikz@finish}%
                  \global\pgflinewidth=\pgfutil@tempdima%
                }%
              \endpgfidscope%
            \endpgfscope%
          \endpgfinterruptpath%
      }}%
    }%
    %
    \tikz@alias%
    \tikz@node@finish%
}%
```

本命令：

1. 利用 \tikz@text@width, \tikz@text@height, \tikz@text@depth 来决定 node 的文字盒子的相关尺寸。
2. 执行 \tikz@node@transformations 计算针对 node 的变换矩阵，即 node 所在的标架，
3. 宏 \tikz@nlt 与 /tikz/transform shape nonlinear →P. 823 有关。
4. 重定义盒子 \tikz@whichbox, 在这个盒子中执行 \pgfmultipartnode 创建 node, 即将这个 node 添加到盒子 \tikz@whichbox 中。每个 node 都被放入一个 \hbox 中。
5. 对盒子 \tikz@whichbox 的重定义在 \tikz@node@finish 那里结束。

\tikz@node@transformations

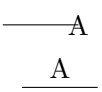
其定义是：

```
\def\tikz@node@transformations{%
  %
  % Possibly, we are ``online''
  %
  \ifx\tikz@time\pgfutil@empty%
    \pgftransformshift{\tikz@node@at}%
    \iftikz@fullytransformed%
    \else%
      \pgftransformresetnontranslations%
    \fi%
  \else%
    \tikz@do@auto@anchor%
    \tikz@timer%
  \fi%
  % Invoke local transformations
  \tikz@transform%
}%
```

这个命令计算针对 node 的变换矩阵（即 node 所在的标架）。先把标架平移到 node 的锚定点，使得 node 本身的坐标系 (其形状 shape 的坐标系) 的原点与其锚定点重合，然后执行 \tikz@transform 引入其他的变换命令，修改变换矩阵。

分为两种情况：

情况一：宏 \tikz@time 保存的内容不是空的 此时使用一个“时刻”(如选项 pos=0.2) 决定的路径上的某个点，这个点作为 node 的锚定点。当 \iftikz@node@is@a@label 的真值为 true 时，一般会导致这一情况，例如，to 操作的标签 (to 后面的 node 语句) 就是这种情况，参考 \tikztonodes →P. 743，对比：




```
\tikz\draw (0,0)--(1,0)node{A};\par
\makeatletter
\tikz\draw (0,0)--(1,0)
    \pgfextra{\tikz@node@is@a@labeltrue}
    node[auto]{A};
\makeatother
```

情况二：宏 \tikz@time 保存的内容是空的 当 \iftikz@node@is@a@label 的真值为 false 时，一般会导致这一情况，此时通常使用 at 直接指定的 node 的锚定点，参考 \tikz@normal@fig →P. 835.

上面代码中的命令 \tikz@do@auto@anchor 是与选项 /tikz/auto →P. 825, /tikz/swap →P. 825 相关的。

上面代码中的命令 \tikz@timer 由构建路径的操作定义，例如在命令 \tikz@@lineto 的展开过程中会规定：

```
\let\tikz@timer=\tikz@timer@line%
```

而 \tikz@timer@line 的定义是：

```
\def\tikz@timer@line{%
  \pgftransformlineattime{\tikz@time}{\tikz@timer@start}{\tikz@timer@end}%
}%
```

其中 \tikz@time 保存的值 (初始值是 0.5) 可以用选项 /tikz/pos →P. 824 修改。参考 \pgftransformlineattime →P.

再如在命令 \tikz@curveC 的展开过程中会规定：

```
\let\tikz@timer=\tikz@timer@curve%
```

而 \tikz@timer@curve 的定义是：

```
\def\tikz@timer@curve{%
  \pgftransformcurveattime{\tikz@time}{\tikz@timer@start}{\tikz@timer@cont@one}{
  ↪  \tikz@timer@cont@two}{\tikz@timer@end}%
}%
```

参考 \pgftransformcurveattime→P. 268.

命令 \tikz@node@transformations 之后的步骤，将把 node 放入盒子中保存起来。

\tikz@node@finish

其定义是：

```
\def\tikz@node@finish{%
    \global\let\tikz@last@fig@name=\tikz@fig@name%
    \global\let\tikz@after@path@smuggle=\tikz@after@path%
    % shift box outside group
    \global\setbox\tikz@tempbox=\box\tikz@figbox%
    \global\setbox\tikz@tempbox@bg=\box\tikz@figbox@bg%
  \endgroup\endgroup%
  \setbox\tikz@figbox=\box\tikz@tempbox%
  \setbox\tikz@figbox@bg=\box\tikz@tempbox@bg%
  \global\pgflinewidth=\tikz@save@line@width%
  \tikz@do@after@path@smuggle%
  \tikz@node@is@picfalse
  \tikz@do@after@node%
}%
```

本命令把当前 node 的名称全局地转存到 \tikz@last@fig@name 中。

本命令把 node 放入盒子中保存起来，等到 \tikz@finish→P. 759 那里再释放盒子。

```
\let\tikz@fig@continue@orig=\tikz@fig@continue

\def\tikz@do@after@node{\tikz@scan@next@command}%
```

\tikz@do@after@path@smuggle

```
\def\tikz@do@after@path@smuggle{%
  \let\tikz@to@last@fig@name=\tikz@last@fig@name%
  \let\tikz@to@use@whom=\tikz@to@use@last@fig@name%
  \let\tikzlastnode=\tikz@last@fig@name%
  \ifx\tikz@after@path@smuggle\pgfutil@empty%
  \else%
    \ifpgflatenodepositioning%
      \expandafter\expandafter\expandafter\tikz@call@late%
      \expandafter\expandafter\expandafter{\expandafter\tikz@last@fig@name
      ↪  \expandafter}\expandafter{\tikz@after@path@smuggle}%
    \else%
      \tikz@scan@next@command{\tikz@after@path@smuggle}\pgf@stop%
    \fi%
  \fi%
}%
```

命令 \tikzlastnode→P. 835 局部地等于 \tikz@last@fig@name.

\tikz@call@late{⟨*node name*⟩}{⟨*code*⟩}

```
\def\tikz@call@late#1#2{\pgfnodepostsetupcode{#1}{\path
↪  [late options={name={#1},append after command={#2}}];}}%
```

参考 \pgfnodepostsetupcode→P. 469.

44.16 node 与变换矩阵

TikZ 在执行创建 node 的命令时，`\path ... node (⟨node name⟩)...;`, 会把创建 node 时的当前变换矩阵全局地保存到

`\csname pgf@sh@nt@⟨node name⟩\endcsname`

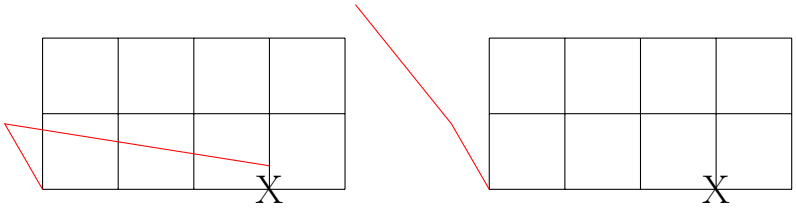
中，此处简记为矩阵 A.

之后如果引用与 (⟨node name⟩) 相关的点,例如 `\path ... (⟨node name⟩.⟨anchor⟩)...;` 或者 `\path ... (⟨node name⟩.60)...;`, 会如下计算：

把当前的变换矩阵记为 B, 此时用命令 \pgfpointanchor→P. 469 计算这个点的坐标。实际上，命令 `\pgfpointanchor` 调用 \pgf@sh@reanchor→P. 456 在 node 自己的坐标系中计算出这个被引用点的坐标。这里所谓“在 node 自己的坐标系中计算”，指的是，当初用命令 \pgfdeclareshape→P. 436 定义 node 的 shape 时，所给出的计算被引用点的方法，所计算出的坐标是没有经过矩阵 A 变换的原始数据。

然后，命令 \pgfpointanchor→P. 469 利用逆矩阵 B^{-1} 来变换坐标数据。再之后，`\path` 的 move-to 或 line-to 之类的操作再利用矩阵 B 来变换坐标数据。这就使得当前变换矩阵 B 对被引用点没有效果。

\pgfpathmoveto→P. 276, \pgfpathlineto→P. 277 之类的命令会用矩阵 B 来变换坐标数据。




```
\begin{tikzpicture}
  \draw (0,0) grid (4,2);
  {
    \tikzset{shift=(0:3)}
    \node (x) at(0,0) {\Large X};
  }
  \draw [red,rotate=120](0,0) -- (1,0) -- (x.north);
\end{tikzpicture}
%
\begin{tikzpicture}
  \draw (0,0) grid (4,2);
  \node [shift=(0:3)](x) at(0,0) {\Large X};
  \path (x.north);
  \pgfgetlastxy{\macrox}{\macroy}
  \draw [red,rotate=120](0,0) -- (1,0) -- (\macrox,\macroy);
\end{tikzpicture}
```

44.17 node foreach

句子 `\path ... node foreach ... ;` 实际会创建

```
\path node ...
      node ...
      ... ;
```

44.18 作为路径标签的 node

参考 \tikz@collected@onpath→P. 766.

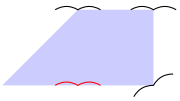
第四十五章　Pics: Small Pictures on Paths

45.1　Overview

pic 是 picture 的省写，代表“小图”。凡是可以用 node 语句的地方都可以用 pic 语句，适用于 node 的绝大多数选项（key）也都适用于 pic. 单词 shape 描述 node 的形状，而 pic 的形状则由单词 type 来描述。pic 是添加到路径上的图形，它（与 node 一样）不属于路径，不计入路径的边界盒子，路径选项中的某些 key 对该路径上的 pic 图形无效。每当使用某种 type 的 pic 时，都会开启一个局部 scope，定义这种 type 的代码会在这个局部 scope 中被执行。定义 type 的代码可以是各种 Ti*k*Z 的绘图命令。

与 node 不同的是，pic 不能被引用，但可以引用 pic 图形中的 node；在 node 之间可以画线，但不能直接在 pic 之间画线；如果 pic 图形很复杂，那么处理起来可能会慢一些。

下面是个例子：



```
\tikzset{ % 定义一个海鸟形状的 pic, 其 type 的名称为 seagull
  seagull/.pic={
    \draw (-3mm,0) to [bend left] (0,0) to [bend left] (3mm,0);
  }
}
\tikz \fill [fill=blue!20]
    (1,1) % 下面用 pic 语句插入 seagull
  --(2,2) pic {seagull}
  --(3,2) pic {seagull}
  --(3,1) pic [rotate=30] {seagull}
  --(2,1) pic [red] {seagull};
```

45.2　The Pic Syntax

\pic

在 {tikzpicture} 环境中，这个命令是 \path pic 的简写。

pic 语句根 node 语句非常相似，实际上，解析这两种语句的是同样的解析代码（parser code）。

\path ... pic⟨*foreach statements*⟩[⟨*options*⟩](⟨*prefix*⟩)at(⟨*coordinate*⟩)
:⟨*animation attribute*⟩={⟨*options*⟩}{⟨*pic type*⟩}...;

pic 语句与 node 语句非常类似，Ti*k*Z 会把“pic”与开花括号“{”之间的各个部分看作是属于该 pic 操作的，这几个部分都是可有可无的（视情况而定），各部分的次序如同 node 语句，⟨*foreach statements*⟩ 必须放在 pic 之后（如果有的话）。

可以用于 [⟨*options*⟩] 中的选项 (前缀路径为 /tikz) 如下文。

⟨*pic type*⟩ 实际上应当是一个键值对列表，键的前缀应当是 /tikz/pics, 其中至少应给出一个 pic type 名称，或应给出 (或者隐含) code={⟨*pic type code*⟩} 这个键。

45.2.1 指定所用的 pic type

/tikz/pic type=⟨*pic type*⟩ (no default)

本选项指定当前 pic 所使用的 type，如果解析器在 pic 的选项块中遇到这个选项，那么对这个 pic 的解析过程就终止于方括号，此时花括号里的 {⟨*pic type*⟩} 就是多余的。这一点类似选项 /tikz/node contents→P. 802，实际上这两个选项是可以相互替换使用的。

```
\tikzset{
  node contents/.code=\def\tikz@node@content{#1},
  pic type/.code=\def\tikz@node@content{#1}, % alias
}%
```



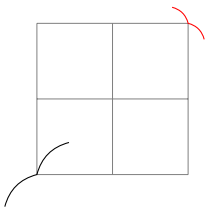
```
\tikz {
  \path (0,0) pic [pic type = seagull]
        (1,0) pic {seagull};
}
```



```
\tikz {
  \path (0,0) pic [node contents = seagull]
        (1,0) pic {seagull};
}
```

45.2.2 指定 pic 图形的位置

pic 图形的锚定点，或者是 pic 之前的点，或者由 at(⟨*coordinate*⟩) 这一部分指定，或者用选项 at=(⟨*coordinate*⟩) 指定，pic 图形的原点与其锚定点重合。注意 pic 命令自己的变换选项，如 shift 选项，会影响 pic 图形的原点与（这个 pic 图形中的）其它点的相对距离，从而影响 pic 图形的尺寸。



```
\tikz {
  \draw [help lines](0,0) grid (2,2);
  \pic [red,shift={(1,1)},rotate=-45]at(1,1){seagull};
  \pic [at={(0,0)},rotate=45,scale=2]{seagull};
}
```

与 node 命令一样，如果 pic 命令带有 transform shape 选项，则 pic 图形接受路径命令的变换选项的作用；可以在一个点之后使用多个 pic 语句，它们会依次画出；可以给 pic 使用 sloped, pos, near end, near start 等选项来调整 pic 图形在路径上的位置。

45.2.3 选项的有效与无效

当在构建主路径的过程中遇到 pic 时，主路径的构建过程会暂时中断，然后开启一个内部域（internal scope），在这个内部域中 ⟨*options*⟩ 和 ⟨*pic type*⟩ 被执行。把“使用路径”的选项（例如 draw, fill, clip 等）用在 pic 的 [⟨*options*⟩] 中时，都不会直接起作用。

45.2.4 定义 pic code

```
\tikzset{
  pics/setup code/.initial=,
  pics/code/.initial=,
  pics/background code/.initial=,
  pics/foreground code/.initial=
}%
```

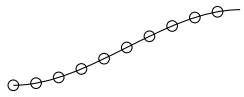
通常，⟨*pic type*⟩ 保存了 pic 图形的代码。如果没有视线定义 ⟨*pic type*⟩，那么可以使用选项 `/tikz/pics/code` 临时定义 pic 图形。

`/tikz/pics/code`=⟨*code*⟩ (no default)

这个选项保存的 ⟨*code*⟩ 就是当前 pic 图形的代码。如果把 ⟨*pic type*⟩ 留空（但要保留花括号），那就可以使用这个选项；如果指定了 {⟨*pic type*⟩}，那么本选项无效。

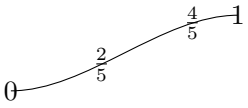

```
\tikz \pic [pics/code={\draw (-3mm,0) to[bend left] (0,0)
                                    to[bend left] (3mm,0);}] {};
```

⟨*pic type*⟩ 实际是个 key 列表，各个 key 会被冠以前缀 `/tikz/pics/` 来执行。在原则上，可以用这些 key 提供任何能被 `\pgfkeys` 解析的内容。前面定义了 `seagull`，当处理 `pic{seagull}` 时会执行 `\pgfkeys{/tikz/pics/seagull}`，这等价于执行选项 `code={\draw(-3mm,0)...;}`.



```
\tikz \draw (0,0) .. controls(1,0) and (2,1) .. (3,1)
  foreach \t in {0, 0.1, ..., 1} {
    pic [pos=\t] {code={\draw circle [radius=2pt];}}
};
```

上面例子中，处理花括号内的 `code={\draw...` 时，实际上执行 `\pgfkeys{/tikz/pics/code={\draw...}`。




```
\tikz \draw (0,0) .. controls(1,0) and (2,1) .. (3,1)
  foreach \t in {0,0.4,0.8,1} {
    pic [pos=\t] {code={
      \pgftext{\pgfmathprintnumber[/pgf/number format/frac]{\t}}}}
};
```

45.2.5 pic 的选项的传递


```
\tikz \draw (0,0)
  pic [red,fill=green,draw,] {code={
    \draw (-1,0)--(1,0);
    \node[circle]{A};}};
```

上面例子表明，当写出 `pic[`⟨*options*⟩`]{`⟨*pic type*⟩`}` 后，⟨*options*⟩ 中的 `color=` 选项能对 ⟨*pic type*⟩ 起作用，但“使用路径”的选项 `draw`，`fill` 选项（另外还有 `shading`，`clip` 等选项）对 ⟨*pic type*⟩ 没有作用。

为了能够让 `pic` 选项中的 `fill`，`draw`，`shading`，`clip` 等选项对 ⟨*pic type*⟩ 有效，需要在定义 ⟨*pic type*⟩ 的代码中的路径语句里添加下面的选项：

`/tikz/pic actions` (no value)

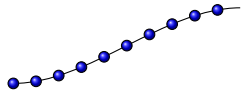
本选项的定义是：

```
\tikzset{
  pic actions/.code=\tikz@addmode{\tikz@picmode}
}%
```

参考 `\tikz@subpicture@handle@` →P. 854 的处理过程。

对比下面两个图形：

```
\tikz \draw (0,0) .. controls(1,0) and (2,1) .. (3,1)
  foreach \t in {0, 0.1, ..., 1} {
    pic [draw,fill,pos=\t] {
      code={\path circle [radius=2pt];}}
};
```



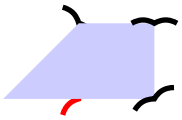
```
\tikz \draw (0,0) .. controls(1,0) and (2,1) .. (3,1)
  foreach \t in {0, 0.1, ..., 1} {
    pic [draw,shading=ball,pos=\t] {
      code={\path [pic actions]circle [radius=2pt];}}
};
```

上面例子中，当执行选项 pic actions 时，导致 \tikz@addmode{\tikz@picmode}，此时的 \tikz@picmode 等于 \tikz@mode.

参考 \tikz@addmode→P. 765, \tikz@mode→P. 766.

45.2.6 指定 pic 图形的遮挡次序

与 node 类似，pic 操作可以带有 /tikz/behind path→P. 803 或 /tikz/in front of path→P. 803 选项。



```
\tikz[line width=2pt] \fill [fill=blue!20]
    (1,1)
  --(2,2) pic [rotate=-45,behind path] {seagull}
  --(3,2) pic {seagull}
  --(3,1) pic [rotate=30] {seagull}
  --(2,1) pic [red,rotate=45,behind path] {seagull};
```

此外，还可以给 pic 使用下面的 key：

/tikz/pics/foreground code=⟨*code*⟩ (no default)

/tikz/pics/background code=⟨*code*⟩ (no default)

45.2.7 设置每个 pic 图形的样式

/tikz/every pic (style, initially empty)

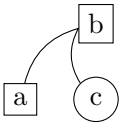
这个选项将用于每个 pic 图形的构建过程的开头。

45.2.8 设置 pic 图形中 node 名称的前缀并引用它

不能像 node 那样用选项 name=⟨*name*⟩ 为 pic 图形定义名称，然后引用它，不过 pic 图形中的 node 是可以被引用的。如果 pic 图形中的 node 有名称，可以用这个名称直接引用它。另外还可以为 pic 图形中 node 的名称加前缀，有 3 种加前缀的方式：

1. 在 pic 语句中使用“名称”(⟨*name*⟩)，类比 node 命令的句法，表面上看这个 ⟨*name*⟩ 是 pic 图形的名称，但实际上它是 pic 图形中各个 node 的名称前缀；
2. 给 pic 添加 name=⟨*name*⟩ 选项，这个办法的实际效果与上一个办法一样。
   以上两种方式实际上都会启用选项 /tikz/name prefix→P. 806 来为 pic 图形中的各个 node 定义名称前缀。
3. 在 pic 图形内直接给 node 命令加选项 name prefix 来定义其名称前缀，这个方式定义的前缀比前两种方式定义的前缀具有优先地位。

当 pic 图形中的 node 有了名称前缀后，若要在这个 pic 图形之外引用其中的 node，就需要带有这个前缀，前缀之后紧随 node 名称，二者之间不需要分隔符号。




```
\tikz{
  \pic (x) [pics/code={
    \node [draw] (a) at(0,0) {a};
    \node [draw,name prefix=y] (b) at(1,1) {b};}] {};
  \pic [name=z,pics/code={\node [circle,draw](c) {c};}] at(1,0) {};
  \draw (xa) to [bend left] (yb) to [bend right] (zc);
}
```

上面例子中三个 node 的名称分别是 (xa)，(yb)，(zc)，没有 (xb).

下面修改前文定义的 seagull，给其中的坐标加名称：

```
\tikzset{
  seagull/.pic={
    \coordinate (-left wing) at (-3mm,0);
    \coordinate (-head) at (0,0);
    \coordinate (-right wing) at (3mm,0);
    \draw (-left wing) to [bend left] (0,0) (-head) to [bend left] (-right wing);
  }
}
```

然后可以引用 seagull 中的坐标：

```
\tikz {
  \pic (Emma) {seagull};
  \pic (Alexandra) at (0,1) {seagull};
  \draw (Emma-left wing) -- (Alexandra-right wing);
}
```

也可以用下面的办法引用 pic 图形中的 node：

```
\tikzset{
  pics/seagull/.style={
    code={
      \coordinate (#1-left wing) at (-3mm,0);
      \coordinate (#1-head) at (0,0);
      \coordinate (#1-right wing) at (3mm,0);
      \draw (#1-left wing) to [bend left] (0,0)
            (#1-head) to [bend left] (#1-right wing);
    }
  }
}
\tikz {
  \pic {seagull={Emma}};
  \pic at (0,1) {seagull={Alexandra}};
  \draw (Emma-left wing) -- (Alexandra-right wing);
}
```

45.2.9 用 pic 制作动画

这与用 node 制作动画类似。

45.2.10 引用句法

载入 quotes 库后，可在 `pic` 的选项中使用引用句法。当在 `pic` 的选项中写出 "⟨*text*⟩"'⟨*options*⟩ 后，它会被转换为：

every pic quotes/.try,pic text=⟨*text*⟩, pic text options={⟨*options*⟩}

/tikz/pic text=⟨*text*⟩ (no default)

这个选项把 ⟨*text*⟩ 保存到宏 `\tikzpictext`，这个宏被 `\let` 默认为 `\relax`. quotes 库会把引用句法选项中的“文字”部分（⟨*text*⟩部分）映射到这个 key 中。

/tikz/pic text options=⟨*options*⟩ (no default)

这个选项把 ⟨*options*⟩ 保存到宏 `\tikzpictextoptions`，这个宏被 `\let` 默认为“empty string”. quotes 库会把引用句法选项中的“选项”部分（⟨*options*⟩部分）映射到这个 key 中。

/tikz/every pic quotes (style, initially empty)

angles 库定义了一个名称为 `angle` 的 pic type，专门用于描绘角度。

45.3 定义 pic type

前面已经有例子展示如何定义一个 pic type, 即定义 pic 图形。定义一个 pic type 有两个要点：

1. 定义一个路径为 /tikz/pic 的 key，记为 ⟨*key*⟩.
2. 所定义的 ⟨*key*⟩ 能够利用选项 /tikz/pic/code 来规定一组画图命令。

除了直接用 pic 的 code 选项规定一个 pic 图形外，还可以使用“手柄”定义 pic 图形，主要用到与 /.pic, /.style 有关的手柄。

Key handler ⟨*pic type*⟩/.pic=⟨*pic type code*⟩

这个手柄定义的 ⟨*key*⟩ 只能用在 TikZ 命令或者 \tikzset 命令中，因为 TikZ 命令和 \tikzset 命令会自动给选项 ⟨*pic type*⟩ 加路径前缀 /tikz/.

```
\tikzset{
  seagull/.pic = { % 定义名称为 seagull 的 pic type
    \draw (...) ... ;
    ...
  }
}
```

这个手柄的定义是：

```
\pgfkeysdef{/handlers/.pic}{%
  \edef\pgf@temp{\pgfkeyscurrentpath}%
  \edef\pgf@temp{\expandafter\tikz@smuggle@pics@in\pgf@temp\pgf@stop}%
  \expandafter\pgfkeys\expandafter{\pgf@temp/.style={code={#1}}}%
}%
\def\tikz@smuggle@pics@in/tikz/#1\pgf@stop{/tikz/pics/#1}%
```

由定义，当执行 \tikzset{⟨*pic type*⟩/.pic=⟨*pic type code*⟩} 时，导致

```
\pgfkeys{/tikz/pics/⟨pic type⟩/.style={code={⟨pic type code⟩}}}
```

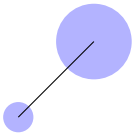
这就把 ⟨*pic type*⟩ 定义成了一个键。

使用 /.style 手柄，例如：

```
\tikzset{
  pics/seagull/.style ={ % 定义名称为 seagull 的 pic type, 注意它的名称前有路径 pics/
    code = { % 注意这里必须用选项 code
      \draw ... ;
    }
  }
}
```

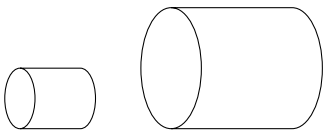
手柄 /.style 最多可以使用 1 个变量，如果需要使用 2 个或更多变量，可以用手柄 /.style 2 args, /.style args, /.style n args.

如果用以上手柄定义 pic type 的代码中含有变量，那么使用该 pic 图形时要给 type 名称赋值，如下面的例子所示：



```
\tikzset{
  pics/my circle/.style = {
    background code = { \fill circle [radius=#1]; }
  }
}
\tikz [fill=blue!30]
  \draw (0,0) pic {my circle=2mm} -- (1,1) pic {my circle=5mm};
```

在下面的例子中定义 pic 图形的手柄用了 /.style 2 args, 其中用了 code={} 来约束绘图代码，而且在命令 \draw 的选项中使用了 pic actions 选项，这是为了使得选项 scale= 有效：



```
\begin{tikzpicture}
\tikzset{
  pics/cylinder/.style 2 args={
    code={\draw [pic actions](0,0) arc (-90:90:1 and #1) coordinate(coord1)--(#2,#1+#1) arc (90:-90:1
    ↪  and #1)--cycle;
    \draw [pic actions](0,0) arc (270:90:1 and #1);}
  }
}
\draw (0,0) pic [scale=0.2]{cylinder={2}{4}};
\draw  (2,0) pic [scale=0.4]{cylinder={2}{4}};
\end{tikzpicture}
```

45.4 TikZ 对 pic 的处理

\iftikz@node@is@pic

TikZ 对 pic 的处理类似对 node 的处理，在处理过程中，根据这个 TEX-if 的真值来区分 pic 与 node，分别对待。

在路径命令 \path 中遇到 pic 时会导致命令 \tikz@subpicture ic,

\tikz@subpicture ic

这个命令的定义是：

```
\def\tikz@subpicture ic{\tikz@node@is@pictrue\tikz@scan@next@command node}%
```

本命令：

1. 设置真值 \tikz@node@is@pictrue
2. 执行解析 node 的命令 \tikz@scan@next@command node, 这又导致 \tikz@fig ode

可见 TikZ 对 pic 的处理类似对 node 的处理，pic 的句法应该类似 node 的句法。在处理过程中，只有 \tikz@node@is@pictrue 的部分会有区别，这些区别是：

- \tikz@fig@scan@options →P. 837[⟨*options*⟩], 此命令有下面的区分：

```
\iftikz@node@is@pic%
  \tikz@enable@pic@quotes%
\else%
  \tikz@enable@node@quotes%
\fi%
```

  \tikz@enable@pic@quotes, \tikz@enable@node@quotes 的初始值都是 \relax, 库 quotes 会重定义它们。

- \tikz@fig@main →P. 838, 此命令有下面的区分：

```
\iftikz@node@is@pic%
  \tikz@node@is@picfalse%
  \expandafter\tikz@subpicture@handle%
\else%
  \afterassignment\tikz@@fig@main\expandafter\let\expandafter\next\expandafter=%
\fi
```

- \tikz@nodes@start →P. 835, 此命令有下面的区分：

```
\iftikz@node@is@pic%
  \def\tikz@nodes@collect{pic }%
\else%
  \def\tikz@nodes@collect{node }%
\fi%
```

\tikz@subpicture@handle{⟨被 *expandafter* 展开的 *\tikz@node@content*, 即 ⟨*pic type*⟩⟩}

本命令导致

```
\tikz@subpicture@handle@{⟨pic type⟩}
```

⟨*pic type*⟩ 应当是一个键值对列表,键的前缀应当是 /tikz/pics, 其中至少应给出 (或者隐含) code={⟨*pic type code*⟩} 这个键。也可以给出 (或者隐含) setup code, background code, foreground code 这些预定义的键。

\tikz@subpicture@handle@{⟨*pic type*⟩}

本命令的处理是:

1. 执行键值对列表 ⟨*pic type*⟩,

   ```
   \pgfkeys{/tikz/pics/.cd,⟨pic type⟩}
   ```

   其中至少应当执行

   ```
   \pgfkeys{/tikz/pics/code={⟨pic type code⟩}}
   ```

   这会把 ⟨*pic type code*⟩ 保存到控制序列 \csname pgfk@/tikz/pics/code\endcsname 中
2. 执行 \tikz@node@transformations →P. 843
3. \let\tikz@transform=\relax
4. \let\tikz@picmode\tikz@mode, 这个步骤可以配合选项 /tikz/pic actions →P. 849. 因为后面步骤会先清空宏 \tikz@mode 保存的“模式”，然后再执行绘制 pic type 图形的代码，所以此时转存这些“模式”，可以在后面用命令 \tikz@addmode\tikz@picmode 调用这些“模式”。
5. 处理 pic 的名称前缀

   ```
   \tikzset{name prefix ../.style/.expanded={/tikz/name prefix=\pgfkeysvalueof
   ↪  {/tikz/name prefix}}}%
   \ifx\tikz@fig@name\pgfutil@empty\else%
     \tikzset{name prefix/.expanded=\tikz@fig@name}%
   \fi%
   ```

6. 执行键 /tikz/pics/setup code 保存的代码

   ```
   \pgfkeysvalueof{/tikz/pics/setup code}
   ```

7. 处理键 /tikz/pics/code 保存的代码
   (a) 转存键 /tikz/pics/code 保存的代码到 \tikz@pic@code

   ```
   \pgfkeysgetvalue{/tikz/pics/code}{\tikz@pic@code}
   ```

   (b) 用 \ifx 检查 \tikz@pic@code 是否空的,
   - 如果是，就什么也不做。
   - 如果不是，就向盒子 \tikz@whichbox 中添加内容:

   ```
   \setbox\tikz@whichbox=\hbox\bgroup%
   \unhbox\tikz@whichbox%
     \hbox\bgroup
       \bgroup%
         \pgfinterruptpath%
           \pgfscope%
   ```

```
          \tikz@options%
          \setbox\tikz@figbox=\box\pgfutil@voidb@x%
          \setbox\tikz@figbox@bg=\box\pgfutil@voidb@x%
          \tikz@atbegin@scope%
          \scope[every pic/.try]%
            \tikz@pic@code%
          \endscope%
          \tikz@atend@scope%
        \endpgfscope%
      \endpgfinterruptpath%
    \egroup
  \egroup%
\egroup%
```

注意：

- 以上代码向盒子 \tikz@whichbox 中添加的内容被 \bgroup 和 \egroup 包裹。
- 盒子 \tikz@whichbox 一般用于保存路径的附加物，如 node, edge. 参考 /tikz/behind path → P. 803.
- 之前的命令 \tikz@normal@fig → P. 835 已经 \let\tikz@options=\pgfutil@empty, 所以上面的 \tikz@options 有可能是空的。参考 \tikz@addoption → P. 676.
- \tikz@atbegin@scope 对应选项 /tikz/execute at begin scope → P. 672.
- \tikz@atend@scope 对应选项 /tikz/execute at end scope → P. 672.
- 样式 every pic 作为 scope 环境的选项被执行。
- 绘制 pic type 的代码被放在 scope 环境内执行。
- 环境命令 \scope, 路径命令 \tikz@@command@path 都会 \let\tikz@mode=\pgfutil@empty, 所以下面代码

```
\tikz \draw (0,0) .. controls(1,0) and (2,1) .. (3,1)
  foreach \t in {0, 0.1, ..., 1} {
    pic [draw,fill,pos=\t] {
      code={\path circle [radius=2pt];}}
  };
```

中的 draw, fill 对 pic 无效。

8. 处理键 /tikz/pics/foreground code 保存的代码
   (a) 转存键 /tikz/pics/foreground code 保存的代码到 \tikz@pic@code

```
\pgfkeysgetvalue{/tikz/pics/foreground code}{\tikz@pic@code}
```

   (b) 用 \ifx 检查 \tikz@pic@code 是否空的，
   - 如果是，就什么也不做。
   - 如果不是，就向盒子 \tikz@figbox 中添加内容：

```
\setbox\tikz@figbox=\hbox\bgroup%
\unhbox\tikz@figbox%
  \hbox\bgroup
    \bgroup%
      \pgfinterruptpath%
        \pgfscope%
          \tikz@options%
          \setbox\tikz@figbox=\box\pgfutil@voidb@x%
          \setbox\tikz@figbox@bg=\box\pgfutil@voidb@x%
          \tikz@atbegin@scope%
          \scope[every pic/.try]%
```

```
            \tikz@pic@code%
          \endscope%
          \tikz@atend@scope%
        \endpgfscope%
      \endpgfinterruptpath%
    \egroup
  \egroup%
\egroup%
```

9. 处理键 /tikz/pics/background code 保存的代码
   (a) 转存键 /tikz/pics/background code 保存的代码到 \tikz@pic@code

```
\pgfkeysgetvalue{/tikz/pics/background code}{\tikz@pic@code}
```

   (b) 用 \ifx 检查 \tikz@pic@code 是否空的，
   - 如果是，就什么也不做。
   - 如果不是，就向盒子 \tikz@figbox@bg 中添加内容：

```
\setbox\tikz@figbox@bg=\hbox\bgroup%
\unhbox\tikz@figbox@bg%
  \hbox\bgroup
    \bgroup%
      \pgfinterruptpath%
        \pgfscope%
          \tikz@options%
          \setbox\tikz@figbox=\box\pgfutil@voidb@x%
          \setbox\tikz@figbox@bg=\box\pgfutil@voidb@x%
          \tikz@atbegin@scope%
          \scope[every pic/.try]%
            \tikz@pic@code%
          \endscope%
          \tikz@atend@scope%
        \endpgfscope%
      \endpgfinterruptpath%
    \egroup
  \egroup%
\egroup%
```

10. 执行 \tikz@node@finish →P. 845.

第四十六章　变换

46.1　各种坐标系统

Ti*k*Z 的线性变换以 PGF 的 2 种线性坐标系：xyz 坐标系和 canvas 坐标系为基础 (详见相关章节)。Ti*k*Z 的坐标会被命令 `\tikz@scan@one@point` →P. 710 解析，然后转为 PGF 的坐标命令，然后用变换矩阵对坐标作变换，然后再转为软路径命令。Ti*k*Z 的各种坐标系统实际上是解析坐标数据的各种方式，集中体现在命令 `\tikz@scan@one@point` 的处理过程中。

例如对于坐标 `(x,y)`, 命令 `\tikz@scan@one@point` 会对坐标数据 `x, y` 分别进行处理 (详见该命令的解释)。对坐标的 x 分量数据的处理方式称为“x 轴”；对坐标的 y 分量数据的处理方式称为“y 轴”；xyz 坐标系和 canvas 坐标系都有各自的“x 轴”与“y 轴”。

46.1.1　各种标架

以下只分析二维图形的情况。当给出坐标数据 `(1,1)` 时，必须同时给出相应的标架才能确定 `(1,1)` 究竟代表页面上的哪个点。在不同标架中，`(1,1)` 可能代表页面上的不同点。

首先假设 2 个标架：固定标架、平动标架：

- 固定标架：当绘图环境开始时，假想先创建一个固定标架 $U = \{u_1, u_2; O\}$, 其中 O 是标架 U 的原点；基向量 u_1 在页面上的方向是水平向右的，长度为 1cm；基向量 u_2 在页面上的方向是竖直向上的，长度为 1cm. 标架 U 是固定不变的单位正交标架，其他标架都用 U 来描述。
- 平动标架：设点 P 是页面上任意一点，在固定标架 U 下，以 $\overrightarrow{OP}$ 为平移向量，以标架 U 为参照系来平移标架 U 得到的标架 $V_P = \{u_1, u_2; P\}$ 是一个平动标架。

xyz 坐标系和 canvas 坐标系的“x 轴”与“y 轴”可以组合成如下的标架：

(1) XY 标架，即画布 (canvas) 坐标系，它的两个轴记为 X, Y;
(2) xy 标架，即 xyz 坐标系，它的两个轴记为 x, y;
(3) Xy 标架，它由当前画布标架的 X 轴和当前 xy 标架的轴 y 组成；
(4) xY 标架，它由当前 xy 标架的轴 x 和当前画布标架的 Y 轴组成。

46.1.1.1　画布标架与变换矩阵

这里说的“画布标架”就是手册中说的“变换矩阵”，参考 `\pgftransformcm` →P. 263, 例如，

$$\begin{pmatrix} 1 & 0 & 0\text{pt} \\ 0 & 1 & 0\text{pt} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1\text{cm} \\ 1\text{cm} \\ 1 \end{pmatrix} = \begin{pmatrix} 1\text{cm} \\ 1\text{cm} \\ 1 \end{pmatrix}$$

按上面的式子，坐标 $\binom{1\mathrm{cm}}{1\mathrm{cm}}$ 可以看作是直角标架 $\{\binom{1}{0},\binom{0}{1};\binom{0}{0}\}$ 中的点，这个直角标架的基向量看作是固定标架 U 中的向量。而按照式子

$$\begin{pmatrix} \frac{\sqrt{3}}{2} & \frac{-1}{2} & 1\mathrm{cm} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 1\mathrm{cm} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1\mathrm{cm} \\ 1\mathrm{cm} \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{3}+1}{2}\mathrm{cm} \\ \frac{\sqrt{3}+3}{2}\mathrm{cm} \\ 1 \end{pmatrix}$$

则可以把坐标 $\binom{1\mathrm{cm}}{1\mathrm{cm}}$ 可以看作是仿射标架 $\left\{\begin{pmatrix}\frac{\sqrt{3}}{2}\\ \frac{1}{2}\end{pmatrix},\begin{pmatrix}\frac{-1}{2}\\ \frac{\sqrt{3}}{2}\end{pmatrix};\binom{1\mathrm{cm}}{1\mathrm{cm}}\right\}$ 中的点，这个仿射标架的基向量看作是固定标架 U 中的向量。所以，改变变换矩阵的选项、命令都是在改变画布标架。

当一个坐标数据带有长度单位时，就用画布标架的 X 轴或者 Y 轴来处理这个坐标数据。

- 例如坐标 `(1cm,2cm)` 中的数据 `1cm` 对应画布标架的 X 轴，数据 `2cm` 对应画布标架的 Y 轴。
- 再如坐标 `(1cm,2)` 中的数据 `1cm` 对应画布标架的 X 轴，而数据 `2` 对应 xy 标架的 y 轴。

详情参考 `\tikz@scan@one@point` →P. 710 的定义。

46.1.1.2 xy 标架

这里说的 xy 标架——这种计算方式由命令 `\pgfsetxvec` →P. 253, `\pgfsetyvec` →P. 253, `\pgfsetzvec` →P. 253 决定：

- 命令 `\pgfsetxvec` 设置 xy 标架的 x 轴基向量：`\pgf@xx`, `\pgf@xy` (两个尺寸寄存器)
- 命令 `\pgfsetyvec` 设置 xy 标架的 y 轴基向量：`\pgf@yx`, `\pgf@yy` (两个尺寸寄存器)
- 命令 `\pgfsetzvec` 设置 xy 标架的 z 轴基向量：`\pgf@zx`, `\pgf@zy` (两个尺寸寄存器)

当一个坐标数据不带长度单位时，就用 xy 标架的 x 轴或者 y 轴来处理这个坐标数据。例如坐标 `(2,3)`，先把这个坐标看作是标架 $\left\{\begin{pmatrix}\texttt{\textbackslash pgf@xx}\\ \texttt{\textbackslash pgf@xy}\end{pmatrix},\begin{pmatrix}\texttt{\textbackslash pgf@yx}\\ \texttt{\textbackslash pgf@yy}\end{pmatrix};\binom{0}{0}\right\}$ 中的坐标，计算 $P=\begin{pmatrix}\texttt{2*\textbackslash pgf@xx+3*\textbackslash pgf@xy}\\ \texttt{2*\textbackslash pgf@yx+3*\textbackslash pgf@yy}\end{pmatrix}$，然后再把 P 看作是当前画布标架中的坐标，从而确定页面上的一个点 $\mathbf{P}$. 详情参考 `\tikz@scan@one@point` →P. 710 的定义。

也就是说，xy 标架是在画布标架的基础上来工作的，它专门处理不带长度单位的坐标数据，它比画布标架多了一层计算手续。

46.1.2 如何看待变换选项

一个变换选项可以看作是一个变换 T, 它是对轴做变换的四元映射：

$$T\colon (X_P, Y_P\ x_P, y_P) \to (X'_{P'}, Y'_{P'}, x'_{P'}, y'_{P'}),$$

上面记号中的 P 是变换前各个轴的原点，P' 是变换后各个轴的原点；P 与 P' 的相对位置取决于变换 T. 轴 X_P, Y_P 组成画布标架 W_P, 轴 x_P, y_P 组成 xy 标架 I_P, 所以变换 T 也可以看作是对标架做变换的二元映射：

$$T\colon (W_P, I_P) \to (W'_{P'}, I'_{P'}),$$

其中 W_P 和 $W'_{P'}$ 是画布标架；I_P 和 $I'_{P'}$ 是 xy 标架；T 把 W_P 变成 $W'_{P'}$，把 I_P 变成 $I'_{P'}$；W_P 与 I_P 的原点重合，$W'_{P'}$ 与 $I'_{P'}$ 的原点重合。在初始之下，画布标架和 xy 标架都与固定标架 U 重合。设 T_1 和 T_2 是前后相继出现的两个变换，则有

$$(W_{P_0}, I_{P_0}) \xrightarrow{T_1} (W_{P_1}, I_{P_1}) \xrightarrow{T_2} (W_{P_2}, I_{P_2}).$$

注意在变换过程中，变换得到的画布标架或 xy 标架可以是“退化的”。

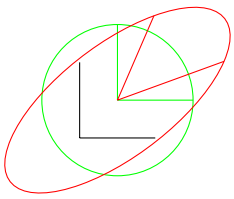
对于 $T\colon (W_P, I_P) \to (W'_{P'}, I'_{P'})$, 约定：

- W_P 是变换 T 的当前画布标架、输入画布标架

- W'_P, 是变换 T 的输出画布标架
- I_P 是变换 T 的当前 xy 标架、输入 xy 标架
- I'_P, 是变换 T 的输出 xy 标架
- 对 W_P, I_P 做变换时所参照的标架是当前参照标架，当前参照标架可能是画布标架，也可能是 xy 标架，也可能是其他标架

46.1.3 例子

在标架变换下，画圆的操作 `circle (1cm)` 与 `circle (1)` 有不同表现：



```
\begin{tikzpicture}
    \draw (0,0)--(1,0)(0,0)--(0,1);
  {[shift={(5mm,5mm)},x={(20:1.5)},y={(70:1)}]
    \draw [green](0,0) circle (1cm);
    \draw [green](0,0)--(1cm,0)(0,0)--(0,1cm);
    \draw [red](0,0) circle (1);
    \draw [red](0,0)--(1,0)(0,0)--(0,1);
  }
\end{tikzpicture}
```

上面图形中，操作 `circle (1cm)` 带有长度单位 `cm`, 所以执行的是

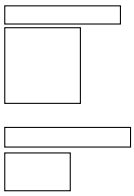
```
\pgfpathellipse{\pgfpointorigin}%
  {\pgfqpoint{\tikz@ellipse@x pt}{0pt}}%
  {\pgfpoint{0pt}{\tikz@ellipse@x pt}}%
```

这个圆上点的坐标是当前画布标架（绿色）下的坐标，所以画出的是圆。操作 `circle (1)` 不带长度单位，所以执行的是

```
\pgfpathellipse{\pgfpointorigin}%
  {\pgfpointxy{\tikz@ellipse@x}{0}}%
  {\pgfpointxy{0}{\tikz@ellipse@x}}%
```

这个圆上点的坐标是当前 xy 标架（红色）下的坐标，所以画出的是椭圆。

对于画矩形的操作有如下效果：



```
\tikzset{x={(-30:1)},y={(40:1)}}
\tikz \draw (0,0) rectangle (1,1); \par
\tikz \draw (0,0) rectangle (1cm,1cm); \par
\tikz \draw (0,0) rectangle (1cm,1); \par
\tikz \draw (0,0) rectangle (1,1cm);
```

```
%\tikzset{xscale=-2,yscale=-0.5,rotate=30}
\tikz[xscale=-2,yscale=-0.5,rotate=30] \draw (0,0) rectangle (1,1);
```

46.2 变换选项

46.2.1 改变 xyz 标架的选项

下面的选项用于指定或改变 xy 标架、xyz 标架，它们不改变画布标架。

`/tikz/x=`⟨*value*⟩ (no default, initially `1cm`)

这里 ⟨*value*⟩ 可以是带单位的尺寸，也可以是不带单位的数值，也可以是坐标点。

在《tikz.code.tex》中定义：

```
% Coordinate options
\tikzoption{x}{\tikz@handle@vec{\pgfsetxvec}{\tikz@handle@x}#1\relax}%
\tikzoption{y}{\tikz@handle@vec{\pgfsetyvec}{\tikz@handle@y}#1\relax}%
\tikzoption{z}{\tikz@handle@vec{\pgfsetzvec}{\tikz@handle@z}#1\relax}%

\def\tikz@handle@vec#1#2{\pgfutil@ifnextchar({\tikz@handle@coordinate#1}{
↪  \tikz@handle@single#2}}%
\def\tikz@handle@coordinate#1{\tikz@scan@one@point#1}%
\def\tikz@handle@single#1#2\relax{#1{#2}}%
\def\tikz@handle@x#1{\pgfsetxvec{\pgfpoint{#1}{0pt}}}%
\def\tikz@handle@y#1{\pgfsetyvec{\pgfpoint{0pt}{#1}}}%
\def\tikz@handle@z#1{\pgfsetzvec{\pgfpoint{#1}{#1}}}%
```

参考 \tikzoption→P. 676, \pgfsetxvec→P. 253, \pgfutil@ifnextchar→P. 29. 可见对于 x=⟨*value*⟩ 来说，

- 如果 ⟨*value*⟩ 以开圆括号“(”开头，则导致

  ```
  \tikz@handle@coordinate\pgfsetxvec⟨value⟩\relax
  导致
  \tikz@scan@one@point\pgfsetxvec⟨value⟩\relax
  ```

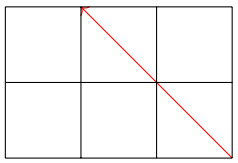
  此时 Ti*k*Z 会把 ⟨*value*⟩ 当作一个“坐标点”并解析它，解析结果通常是一个基本层的点，看作是画布坐标系中的一个向量，用作 xyz 坐标系的 x 轴的单位向量。

- 如果 ⟨*value*⟩ 不以开圆括号“(”开头，则导致

  ```
  \tikz@handle@single\tikz@handle@x{⟨value⟩}
  导致
  \tikz@handle@x{⟨value⟩}
  导致
  \pgfsetxvec{\pgfpoint{⟨value⟩}{0pt}}
  ```

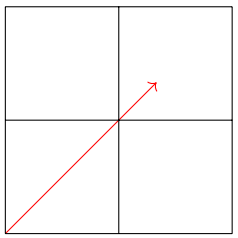
  此时 Ti*k*Z 会把 ⟨*value*⟩ 当作一个数学表达式对待，用 \pgfmathparse→P. 110 解析它。如果在 ⟨*value*⟩ 中有长度单位，则长度单位都会被转为 pt 来计算。如果 ⟨*value*⟩ 不含单位，例如 x=10，则它等价于 x={(10pt,0pt)}. 命令 \pgfsetxvec 直接在画布坐标系中的向量指定为 xyz 坐标系的 x 轴的单位向量。

- 注意,如果 ⟨*value*⟩ 是包含逗号的坐标点,那么应该把坐标 ⟨*value*⟩ 用花括号括起来,例如 x={(3,1)}, x={(3,20pt)}, x={(3cm,20pt)}, 因为逗号会被命令 \pgfkeys 作为前后两个选项的分界标志。



```
\tikz {
  \draw[->,red] (0,0) -- (-2,2);
  \draw[x=-1.5cm] (0,0) grid (2,2);}
```

上面例子中，第 2 个命令设置 x 轴单位向量的实际长度为 −1.5cm，故点 (-2,2) 的横标 -2 代表的实际长度是 −3cm，在默认下，操作 grid 的步长选项 step=1cm，所以画出的网格中有 3 条竖线。也就是说，由于选项 step=1cm 所规定的尺寸是带单位的绝对长度，这个长度在当前画布标架内确定，而变换选项 x=-1.5cm 并不改变画布标架，故网格步长不受 x=-1.5cm 的影响。对比下面变换选项 scale 的作用：



```
\tikz {
  \draw[->,red] (0,0) -- (2,2);
  \draw[scale=1.5] (0,0) grid (2,2);}
```

由于变换选项 scale 改变画布标架，所以网格的外观与前面的例子不同。

/tikz/y=⟨value⟩ (no default, initially 1cm)

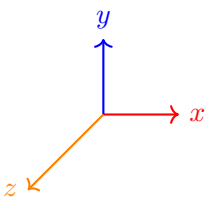
类似 x=⟨value⟩. 如果 ⟨value⟩ 是带单位的尺寸，则这个选项设置 y 轴的单位向量为 (0pt,⟨value⟩).

/tikz/z=⟨value⟩ (no default, initially -3.85mm)

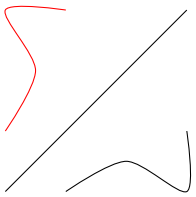
类似 x=⟨value⟩. 如果 ⟨value⟩ 是带单位的尺寸，这个选项设置 z 轴的单位向量为 (⟨value⟩,⟨value⟩). 在平面上画 3 维轴只是一种视觉上的模拟，给定平面上 4 个合适的点可以模拟 3 维标架，所以这里用 2 维点来指定 z 轴的单位向量。这个选项的初始值由文件《pgfcorepoints.code.tex》中的命令

```
\pgfsetzvec{\pgfpoint{-0.385cm}{-0.385cm}}
```

来指定。



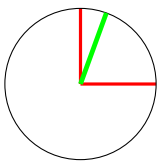

```
\begin{tikzpicture}[z=-1cm,->,thick]
  \draw[color=red] (0,0,0) -- (1,0,0) node[right]{$x$};
  \draw[color=blue] (0,0,0) -- (0,1,0) node[above]{$y$};
  \draw[color=orange] (0,0,0) -- (0,0,1) node[left]{$z$};
\end{tikzpicture}
```



```
\begin{tikzpicture}[smooth,scale=0.8]
  \draw plot coordinates{(1,0) (2,0.5) (3,0) (3,1)};
  \draw[x={(0cm,1cm)},y={(1cm,0cm)},color=red]
    plot coordinates{(1,0) (2,0.5) (3,0) (3,1)};
  \draw (0,0) -- (3,3);
\end{tikzpicture}
```

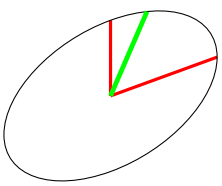
在以上选项中,若 ⟨value⟩ 是画布坐标点,即坐标分量都是带单位的形式,如 x=(30:20mm),x={(10pt,20mm)}, 则 (30:20mm), (10pt,20mm) 是当前画布标架之下的坐标。若 ⟨value⟩ 是不带单位的 xy 坐标形式，如 x=(30:2)，x={(1.5,2)}，则 (30:2)，(1.5,2) 是当前 xy 标架中的坐标。

再看一下选项 x={(30:2)} 的输出 xy 标架如何确定。选项 x={(30:2)} 的参照标架是当前 xy 标架，坐标 (30:2) 是参照标架中的坐标。若参照标架不是单位正交标架，如何解释极坐标 (30:2) 所代表的点呢？观察下面的例子：



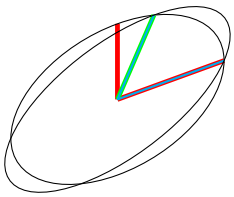
```
\begin{tikzpicture}
  \draw [red,line width=1.2pt](0,0)--(1,0)(0,0)--(0,1);
  \draw (0,0)circle(1);
  \draw [green,line width=1.8pt](0,0)--(70:1);
\end{tikzpicture}
```

在这个图形中，绘图所用的标架是固定标架 U，此标架中的极坐标 (70:1) 所代表的向量如绿色线段所示。现在用一个仿射变换作用于上图，使得 xy 标架变为 {(20:1.5cm),(0cm,1cm);(0,0)}，这不是个单位正交标架，需要在这个标架中解释极坐标 (70:1) 究竟代表哪个向量。变换后的图形如下：



```
\begin{tikzpicture}
  {[x={(20:1.5)},]
    \draw [red,line width=1.2pt](0,0)--(1,0)(0,0)--(0,1);
    \draw (0,0)circle(1);
    \draw [green,line width=1.8pt](0,0)--(70:1);
  }
\end{tikzpicture}
```

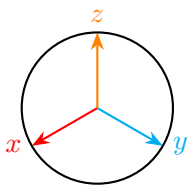
可以猜测，在 xy 标架 {(20:1.5cm),(0cm,1cm);(0,0)} 下极坐标 (70:1) 代表的向量（绿色线段）由之前图形中的绿色线段变换而来。



```
\begin{tikzpicture}
  {[x={(20:1.5)},]
    \draw [red,line width=1.8pt](0,0)--(1,0)(0,0)--(0,1);
    \draw (0,0)circle(1);
    \draw [green,line width=1.8pt](0,0)--(70:1);
  }
  {[x={(20:1.5)},y={(70:1)},]
    \draw [cyan,line width=0.8pt](0,0)--(1,0)(0,0)--(0,1);
    \draw (0,0)circle(1);
  }
\end{tikzpicture}
```

分析一个例子

下面设置一个 xyz 坐标系，而且画出 x, y, z 坐标轴的单位向量，要求让它们看起来长度相等且 3 等分单位圆：

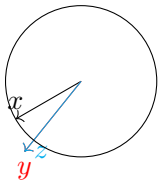



```
\tikzmath {\a=0.5*sqrt(3);}
\begin{tikzpicture}[x={(-\a cm,-0.5cm)}, y={(\a cm,-0.5cm)},
    z={(0cm,1cm)}, -Stealth,thick]
  \draw[color=red] (0,0,0) -- (1,0,0) node[left]{$x$};
  \draw[color=cyan] (0,0,0) -- (0,1,0) node[right]{$y$};
  \draw[color=orange] (0,0,0) -- (0,0,1) node[above]{$z$};
  \draw (0,0) circle(1cm);
\end{tikzpicture}
```

上面例子中用画布坐标形式 `x={(-\a cm,-0.5cm)}, y={(\a cm,-0.5cm)}, z={(0cm,1cm)}` 把当前画布标架 W_O（即固定标架 U）中的 3 个向量分别作为 x, y, z 轴的单位向量，从而得到输出 xy 标架 C:

$$C=\left\{\begin{array}{ll} \text{原点} & :O\to O,\\ x\text{ 轴单位向量} & :(1\text{cm},0\text{pt})\to(\frac{-\sqrt{3}}{2}\text{cm},\frac{-1}{2}\text{cm}),\\ y\text{ 轴单位向量} & :(0\text{pt},1\text{cm})\to(\frac{\sqrt{3}}{2}\text{cm},\frac{-1}{2}\text{cm}),\\ \text{z 轴单位向量} & :(-3.58\text{mm},-3.85\text{mm})\to(0\text{cm},1\text{cm}).\end{array}\right\}\text{固定标架 } U \text{ 下的坐标}$$

在输出的 xyz 标架 C 内画出的路径可以反映“画图要求”。
对比下面的例子：




```
\tikzmath {\a=0.5*sqrt(3);}
\tikz[x={(-\a,-0.5)},y={(\a,-0.5)},z={(0,1)}]
{
  \draw [->] (0,0,0) -- (1,0,0) node [above] {$x$};
  \draw [->,red] (0,0,0) -- (0,1,0) node [below] {$y$};
  \draw [->,cyan] (0,0,0) -- (0,0,1) node [right] {$z$};
  \draw (0,0) circle(1cm);
}
```

在上面例子中，选项 `x={(-\a,-0.5)}` 把当前参照标架——即固定标架 U 中的向量 $(\frac{-\sqrt{3}}{2}\text{cm},\frac{-1}{2}\text{cm})$ 作为 x 轴的单位向量，而 y 轴与 z 轴的单位向量还是默认的，本选项的输出 xyz 标架 C_1 是：

$$C_1=\left\{\begin{array}{ll} \text{原点} & :O,\\ x\text{ 轴单位向量} & :(\frac{-\sqrt{3}}{2}\text{cm},\frac{-1}{2}\text{cm}),\\ y\text{ 轴单位向量} & :(0\text{pt},1\text{cm}),\\ \text{z 轴单位向量} & :(-3.58\text{mm},-3.85\text{mm}).\end{array}\right\}\text{固定标架 } U \text{ 下的坐标}$$

然后选项 `y={(\a,-0.5)}` 以当前 xy 标架 C_1 为参照标架，对 C_1 做变换，即在 C_1 中找出坐标为 $(\frac{\sqrt{3}}{2},\frac{-1}{2})$ 的点 P_y，并以 OP_y 为 y 轴的单位向量，而 z 轴的单位向量还是不变。点 P_y 在 C_1

中的坐标为 $(\frac{\sqrt{3}}{2}, \frac{-1}{2})$，由此计算出点 P_y 在固定标架 U 中的坐标是 $(\frac{3}{4}, \frac{2+\sqrt{3}}{4})$，所以本选项的输出 xyz 标架 C_2 是：

$$C_2 = \left\{\begin{array}{ll} \text{原点} & : O, \\ x \text{ 轴单位向量} & : (\frac{-\sqrt{3}}{2}\text{cm}, \frac{-1}{2}\text{cm}), \\ y \text{ 轴单位向量} & : (\frac{-3}{4}\text{cm}, \frac{-2-\sqrt{3}}{4}\text{cm}), \\ \text{z 轴单位向量} & : (-3.58\text{mm}, -3.85\text{mm}). \end{array}\right\} \text{固定标架 } U \text{ 下的坐标}$$

然后选项 `z={(0,1)}` 以当前 xy 标架 C_2 为参照标架对 C_2 做变换，即在 C_2 中找出坐标为 $(0,1)$ 的点 P_z，并以 OP_z 为 z 轴的单位向量。在 C_2 中坐标为 $(0,1)$ 的点就是点 P_y，实际上点 P_z 与点 P_y 重合。因此本选项的输出 xy 标架 C_3 是：

$$C_3 = \left\{\begin{array}{ll} \text{原点} & : O, \\ x \text{ 轴单位向量} & : (\frac{-\sqrt{3}}{2}\text{cm}, \frac{-1}{2}\text{cm}), \\ y \text{ 轴单位向量} & : (\frac{-3}{4}\text{cm}, \frac{-2-\sqrt{3}}{4}\text{cm}), \\ \text{z 轴单位向量} & : (\frac{-3}{4}\text{cm}, \frac{-2-\sqrt{3}}{4}\text{cm}). \end{array}\right\} \text{固定标架 } U \text{ 下的坐标}$$

然后在最终输出 xyz 标架 C_3 中画出路径 `(0,0,0) -- (1,0,0)` 等，当然不能满足“画图要求”。

`/tikz/rotate around x=`⟨*angle*⟩ (no default)

本选项使得 xyz 标架的 z 轴与 y 轴围绕 x 轴做旋转，以右手握 x 轴，拇指指向 x 轴的正向，手指螺旋方向为旋转的正向。

本选项的定义是：

```
% Code for rotating the xyz coordinate system
% around the x, y, or z vector.
%
\def\tikz@xyz@rotate@let{%
  \let\pgf@z=\pgf@yc%
  \let\pgf@za=\pgf@xc%
}%

\def\tikz@xyz@rotate@xyz@xaxis#1#2#3#4{%
  \tikz@xyz@rotate@let%
  \pgf@x=#1\relax%
  \pgf@ya=#2\relax%
  \pgf@za=#3\relax%
  \pgfmathsin@{#4}\let\tikz@xyz@sin=\pgfmathresult%
  \pgfmathcos@{#4}\let\tikz@xyz@cos=\pgfmathresult%
  \pgf@y=\tikz@xyz@cos\pgf@ya%
  \advance\pgf@y by-\tikz@xyz@sin\pgf@za%
  \pgf@z=\tikz@xyz@sin\pgf@ya%
  \advance\pgf@z by\tikz@xyz@cos\pgf@za%
}%

\tikzset{rotate around x/.code={%
    \tikz@xyz@rotate@let%
    \pgfmathparse{#1}\let\tikz@xyz@angle=\pgfmathresult%
    \tikz@xyz@rotate@xyz@xaxis{0pt}{1pt}{0pt}{\tikz@xyz@angle}%
    \pgfextract@process\tikz@xyz@rotate@yvec{\pgfpointxyz{\pgf@sys@tonumber{\pgf@x
    ↪  }}{\pgf@sys@tonumber{\pgf@y}}{\pgf@sys@tonumber{\pgf@z}}}%
```

```
    \tikz@xyz@rotate@xyz@xaxis{0pt}{0pt}{1pt}{\tikz@xyz@angle}%
    \pgfsetzvec{\pgfpointxyz{\pgf@sys@tonumber{\pgf@x}}{\pgf@sys@tonumber{\pgf@y
    ↪  }}{\pgf@sys@tonumber{\pgf@z}}}%
    \pgfsetyvec{\tikz@xyz@rotate@yvec}%
  },
}
```

命令 \pgfextract@process 见文件《pgfcorepoints.code.tex》，其定义是：

```
% Save a point.
%
% #1 = macro for storing point.
% #2 = code for point (should define x and y)
%
% Example:
%
% \pgfextract@process\mypoint{\pgf@x=10pt \pgf@y10pt}
% \pgfextract@process\myarcpoint{\pgfpointpolar{30}{5cm and 2cm}}

\def\pgfextract@process#1#2{%
    \pgf@process{#2}%
    \edef#1{\noexpand\global\pgf@x=\the\pgf@x
    ↪  \noexpand\relax\noexpand\global\pgf@y=\the\pgf@y\noexpand\relax}%
}
```

选项 rotate around x=⟨*value*⟩ 的值 ⟨*value*⟩ 会被 \pgfmathparse 解析，解析结果——看作一个角度值，记为 θ ——保存在 \tikz@xyz@angle; 假设 d_1, d_2, d_3 是 3 个单位为 pt 的尺寸，执行

```
\tikz@xyz@rotate@xyz@xaxis{d1}{d2}{d3}{θ}%
```

的结果是

$$\texttt{\textbackslash pgf@x} = d_1,\ \begin{bmatrix}\cos(\theta) & -\sin(\theta)\\ \sin(\theta) & \cos(\theta)\end{bmatrix}\begin{pmatrix}d_2\\ d_3\end{pmatrix} = \begin{pmatrix}\texttt{\textbackslash pgf@y}\\ \texttt{\textbackslash pgf@z}\end{pmatrix}$$

本选项实际上依次执行

```
\pgfsetzvec{\pgfpointxyz{0}{-sin(θ)}{cos(θ)}}
\pgfsetyvec{\pgfpointxyz{0}{cos(θ)}{sin(θ)}}
```

也就是先把当前 xyz 标架中的向量 $(0, -\sin(\theta), \cos(\theta))$ 作为新的 z 轴——得到新的 xyz 标架，然后把新 xyz 标架中的向量 $(0, \cos(\theta), \sin(\theta))$ 作为新的 y 轴——得到更新的 xyz 标架——这就是本选项的结果。

/tikz/rotate around y=⟨*angle*⟩ (no default)

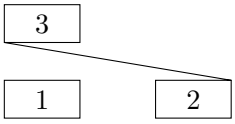
本选项使得 xyz 标架的 z 轴与 x 轴围绕 y 轴做旋转，以右手螺旋为正。

/tikz/rotate around z=⟨*angle*⟩ (no default)

本选项使得 xyz 标架的 y 轴与 x 轴围绕 z 轴做旋转，以右手螺旋为正。

46.2.2 改变 canvas 标架的选项

PGF 和 Ti*k*Z 可以对坐标应用坐标变换矩阵 (在 canvas 标架中解释坐标数据)。坐标变换矩阵只对坐标有效，对线宽、虚线样式、颜色渐变的方向等项目没有影响。坐标变换的有效范围受到 TEX 分组的限制。环境是个分组，一个绘图命令也是个分组。当在一个路径之内用方括号列出变换选项时，该变换只对其后的子路径有效，对其前的子路径无效。




```
\tikz \draw
  (0,0) rectangle node{1} (1,0.5)
  {[xshift=2cm] (0,0) rectangle node{2} (1,0.5)}
  --(0,1) rectangle node{3} (1,1.5) ;
```

注意坐标变换由 TEX 完成，限于 TEX 的计算能力，当计算过程涉及变换矩阵的逆矩阵时，不良条件数的矩阵或者奇异矩阵会导致意外的结果。

以下选项的变换对象、参照标架都是当前的画布标架 (即变换矩阵)。

/tikz/shift=⟨*coordinate*⟩ (no default)

平移变换，平移向量为 ⟨*coordinate*⟩. 参数 ⟨*coordinate*⟩ 应当是 TikZ 的坐标。

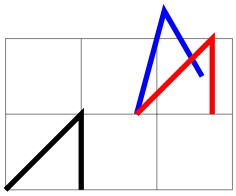
```
\tikzoption{shift}{\tikz@addtransform{\tikz@scan@one@point\pgftransformshift#1
↪  \relax}}%
```

参数 ⟨*coordinate*⟩ 会被 `\tikz@scan@one@point` 解析，解析结果用作 `\pgftransformshift` 的参数。

/tikz/shift only (no default)

本选项把当前画布标架变成当前平动标架。

```
\tikzoption{shift only}[]{\tikz@addtransform{\pgftransformresetnontranslations}}%
```



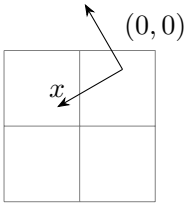
```
\begin{tikzpicture}[line width=2pt]
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (1,1) -- (1,0);
  \draw[rotate=30,xshift=2cm,blue]
    (0,0) -- (1,1) -- (1,0);
  \draw[rotate=30,xshift=2cm,shift only,red]
    (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/xshift=⟨*value*⟩ (no default)

参数 ⟨*value*⟩ 会被 `\pgfmathparse` 解析，如果 ⟨*value*⟩ 不带长度单位，则默认其单位为 pt. ⟨*value*⟩ 可以是复杂的表达式。

```
\def\pgftransformxshift#1{\pgftransformcm{1}{0}{0}{1}{\pgfpoint{#1}{+0pt}}}
\tikzoption{xshift}{\tikz@addtransform{\pgftransformxshift{#1}}}%
```

可见本选项决定的平移向量是 `\pgfpoint{`⟨*value*⟩`}{+0pt}`.




```
\begin{tikzpicture}
  \draw[help lines] (-2,-2) grid (0,0);
  \node [above]{$(0,0)$};
  {[>=Stealth,rotate=30,xscale=-1,xshift=0.5cm]
    \draw [->](0,0)--(1,0)node[above]{$x$};
    \draw [->](0,0)--(0,1);
  }
\end{tikzpicture}
```

/tikz/yshift=⟨*dimension*⟩ (no default)

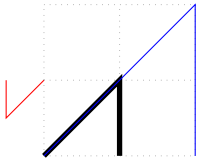
类似 xshift.

```
\def\pgftransformyshift#1{\pgftransformcm{1}{0}{0}{1}{\pgfpoint{+0pt}{#1}}}
\tikzoption{yshift}{\tikz@addtransform{\pgftransformyshift{#1}}}%
```

/tikz/scale=⟨*factor*⟩ (no default)

以当前画布标架的原点为中心做位似变换，即放缩，⟨*factor*⟩ 代表放缩比例（放缩后的尺寸比上放缩前的尺寸）。参数 ⟨*factor*⟩ 会被 `\pgfmathparse` 解析，如果解析结果是负值，就先以当前画布标架的原点为中心对当前画布标架做中心对称，再以原点为中心对当前画布标架做放缩。

```
\def\pgftransformscale#1{\pgftransformcm{#1}{0}{0}{#1}{\pgfpointorigin}}
\tikzoption{scale}{\tikz@addtransform{\pgftransformscale{#1}}}%
```



```
\begin{tikzpicture}
\draw[help lines,dotted] (0,0) grid (2,2);
\draw [line width=2pt](0,0) -- (1,1) -- (1,0);
\draw[scale=2,blue] (0,0) -- (1,1) -- (1,0);
\draw[yshift=1cm,scale=-0.5,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/scale around={⟨*factor*⟩:⟨*coordinate*⟩} (no default)

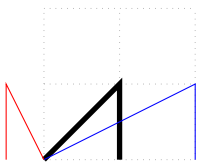
与 scale 类似，只是变换的中心改为 ⟨*coordinate*⟩. 参数 ⟨*factor*⟩ 会被 \pgfmathparse 解析，⟨*coordinate*⟩ 会被 \tikz@scan@one@point 解析。

```
\tikzoption{scale around}{\tikz@addtransform{\def\tikz@aroundaction{
↪  \pgftransformscale}\tikz@doaround{#1}}}%
\def\tikz@doaround#1{%
  \edef\tikz@temp{#1}% get rid of active stuff
  \expandafter\tikz@doparseA\tikz@temp%
}%
\def\tikz@doparseA#1:{%
  \def\tikz@temp@rot{#1}%
  \tikz@scan@one@point\tikz@doparseB%
}%
\def\tikz@doparseB#1{%
  \pgf@process{#1}%
  \pgf@xc=\pgf@x%
  \pgf@yc=\pgf@y%
  \pgftransformshift{\pgfqpoint{\pgf@xc}{\pgf@yc}}%
  \tikz@aroundaction{\tikz@temp@rot}%
  \pgftransformshift{\pgfqpoint{-\pgf@xc}{-\pgf@yc}}%
}%
```

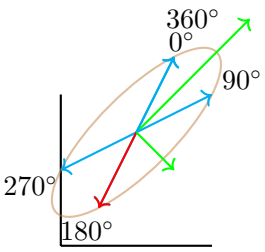
从定义看，scale around=⟨*value*⟩ 的参数 ⟨*value*⟩ 先被 \edef 展开，所以 ⟨*value*⟩ 可以是保存 ⟨*factor*⟩:⟨*coordinate*⟩ 的宏。

/tikz/xscale=⟨*factor*⟩ (no default)

对当前画布标架的 x 轴做放缩，⟨*factor*⟩ 为放缩比例。如果 ⟨*factor*⟩ 是负值，就先以当前画布标架的 y 轴为对称轴对当前画布标架做镜像对称，再以原点为中心做放缩。所以，如果 ⟨*factor*⟩$= -1$, 则当前画布标架与输出画布标架关于当前画布标架的 Y 轴对称。参数 ⟨*factor*⟩ 会被 \pgfmathparse 解析。\tikz@scan@one@point 解析 ⟨*coordinate*⟩ 的结果用作 \tikz@doparseB 的参数，作为平移向量。



```
\begin{tikzpicture}
  \draw[help lines,dotted] (0,0) grid (2,2);
  \draw [line width=2pt](0,0) -- (1,1) -- (1,0);
  \draw[xscale=2,blue] (0,0) -- (1,1) -- (1,0);
  \draw[xscale=-0.5,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```




```
\begin{tikzpicture}[thick]
  \draw (0,0)--(2,0) (0,0)--(0,2);
  \begin{scope}[cm={0.5,1,1,0.5,(1,1.5)}]
    \draw [brown,,opacity=0.5](0,0)circle(1);
    \draw [->,green](0,0)--(1,1); % 绿色线将变成下面的绿色线
    \foreach \ang / \dis in {0/2mm,90/4mm,180/3mm,270/4mm,360/5mm}
      {\draw [->,cyan](0,0)--(\ang:1);
        \node at ($(\ang:1)+(\ang:\dis)$){$\ang^\circ$};}
    \draw [->,xscale=-1,red] (0,0)--(1,0);
    \draw [->,xscale=-1,green] (0,0)--(1,1); % 由上面的绿色线变换来
  \end{scope}
\end{tikzpicture}
```

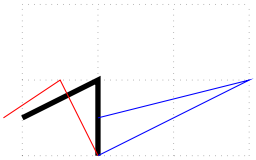
/tikz/yscale=⟨*factor*⟩ (no default)

类似 xscale.

/tikz/xslant=⟨*factor*⟩ (no default)

这个变换是 $(x, y) \to (x + y \cdot \langle factor \rangle, y)$.

```
\tikzoption{xslant}{\tikz@addtransform{\pgftransformxslant{#1}}}%
```

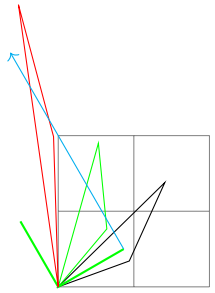


```
\begin{tikzpicture}
\draw [help lines,dotted] (0,0) grid (3,2);
\draw [line width=2pt](0,0.5) -- (1,1) -- (1,0);
\draw [xslant=2,blue] (0,0.5) -- (1,1) -- (1,0);
\draw [xslant=-0.5,red] (0,0.5) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/yslant=⟨*factor*⟩ (no default)

这个变换是 $(x, y) \to (x, y + x \cdot \langle factor \rangle)$.

```
\tikzoption{yslant}{\tikz@addtransform{\pgftransformyslant{#1}}}%
```



```
\tikz{
  \draw [help lines](0,0)grid(2,2);
  {[x={(20:1)},y={(60:1)}]
    \draw (0,0)--(1,0)--(1,1)--cycle;
    {[rotate=30]
      \draw [green,thick](0cm,0cm)--(1cm,0cm) (0cm,0cm)--(0cm,1cm);
      \draw [green] (0,0)--(1,0)--(1,1)--cycle;
      \draw [red,yslant=1.5] (0,0)--(1,0)--(1,1)--cycle;
      \draw [->,cyan](1cm,0cm)--(1cm,3cm);
    }}}
```

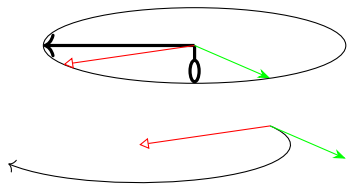
上面图形中，青色箭头指示选项 yslant=1.5 的倾斜方向。

/tikz/rotate=⟨*degree*⟩ (no default)

以当前画布标架的原点为中心做旋转。在右手系内，转角以逆时针方向为正。在左手系内，转角以逆时针方向为负。⟨*degree*⟩ 会被 \pgfmathparse 解析。

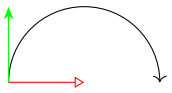
```
\tikzoption{rotate}{\tikz@addtransform{\pgftransformrotate{#1}}}%
```

注意这个选项可以把正交标架变成非正交标架，参考 \pgftransformrotate→P. 265, 例如：



```
\begin{tikzpicture}
  \scoped[xscale=-2,yscale=-0.5,]{
    \draw (0,0) circle (1);
    \draw[->,very thick](0,0)--(1,0);
    \draw[-{Ellipse[open]},very thick](0,0)--(0,1);
    \draw [-{Triangle[open]},red](0,0)--(30:1cm);
    \draw [-Stealth,green](0,0)--(90+30:1);
  }
\end{tikzpicture}
\par\vspace{5mm}
\begin{tikzpicture}
  \scoped[xscale=-2,yscale=-0.5,rotate=30,]{
    \draw[->,](0,0)arc(180:0:1);
    \draw[-{Triangle[open]},red](0,0)--(1cm,0cm);
    \draw[-Stealth,green](0,0)--(0cm,1cm);
  }
\end{tikzpicture}
```

看一下选项 `shift only` 的作用：



```
\begin{tikzpicture}
  \scoped[xscale=-2,yscale=-0.5,rotate=30,shift only]{
    \draw[->,](0,0)arc(180:0:1);
    \draw[-{Triangle[open]},red](0,0)--(1,0);
    \draw[-Stealth,green](0,0)--(0,1);
  }
\end{tikzpicture}
```

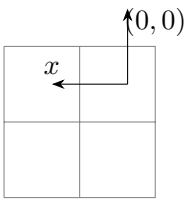
可见选项 `shift only` 把前面各个选项的作用都取消了。

`/tikz/rotate around`={⟨*degree*⟩:⟨*coordinate*⟩} (no default)

本选项的参照标架是当前画布标架。以 ⟨*coordinate*⟩ 为中心做旋转。参数 ⟨*degree*⟩ 会被 `\pgfmathparse` 解析，⟨*coordinate*⟩ 会被 `\tikz@scan@one@point` 解析。

```
\tikzoption{rotate around}{\tikz@addtransform{\def\tikz@aroundaction{
↪  \pgftransformrotate}\tikz@doaround{#1}}}%
```

类似 `scale around`, 选项 `rotate around`=⟨*value*⟩ 的参数 ⟨*value*⟩ 先被 `\edef` 展开，所以 ⟨*value*⟩ 可以是保存 ⟨*degree*⟩:⟨*coordinate*⟩ 的宏。




```
\begin{tikzpicture}
  \draw[help lines] (-2,-2) grid (0,0);
  \node [above]{$(0,0)$};
  {[>=Stealth,rotate=30,xscale=-1,xshift=0.5cm,
      rotate around={30:(0.5,0)}]
    \draw [->](0,0)--(1,0)node[above]{$x$};
    \draw [->](0,0)--(0,1);
  }
\end{tikzpicture}
```

`/tikz/cm`={⟨*a*⟩,⟨*b*⟩,⟨*c*⟩,⟨*d*⟩,⟨*coordinate*⟩} (no default)

这个变换是一般的变换形式，它直接设置变换矩阵。

```
\tikzoption{cm}{\tikz@addtransform{\tikz@parse@cm#1\relax}}%
\def\tikz@parse@cm#1,#2,#3,#4,{%
  \def\tikz@p@cm{{#1}{#2}{#3}{#4}}%
  \tikz@scan@one@point\tikz@parse@cmA}%
\def\tikz@parse@cmA#1{%
  \expandafter\pgftransformcm\tikz@p@cm{#1}%
}%
```

Ti*k*Z 的坐标 ⟨*coordinate*⟩ 会被 `\tikz@scan@one@point` 解析，假设被解析为 ⟨*coordinate'*⟩, 那么本选项执行

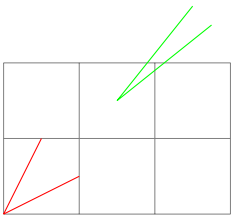
\pgftransformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{⟨coordinate'⟩}

其中的参数 ⟨a⟩, ⟨b⟩, ⟨c⟩, ⟨d⟩ 会被 \pgfmathparse 解析，参考 \pgftransformcm→P. 263.
假设 ⟨coordinate'⟩ 代表向量 $\binom{t_x}{t_y}$，针对点 $\binom{x}{y}$ 做变换，则变换结果是

$$\begin{pmatrix} a & c \\ b & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}.$$

本选项是矩阵乘积变换和平移变换的复合，两个复合成分的参照标架都是当前画布标架，所以：

```
cm={a,b,c,d,(p, q)}
等价于
shift={(p, q)}, cm={a,b,c,d,(0, 0)}
但不等价于
cm={a,b,c,d,(0, 0)}, shift={(p, q)}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \begin{scope}[cm={0.5,1,1,0.5,(0,0)}]
    \draw [red] (0,0)--(1,0)(0,0)--(0,1);
    \draw [green,cm={0.5,1,1,0.5,(1,1)}] (0,0)--(1,0)(0,0)--(0,1);
  \end{scope}
\end{tikzpicture}
```

/tikz/reset cm (no value)

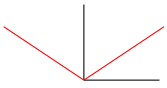
将变换矩阵设为单位矩阵。

```
\tikzoption{reset cm}[]{\tikz@addtransform{\pgftransformreset}}%
```

46.2.3 注意的问题

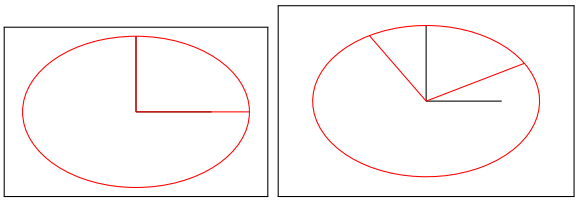
关于坐标变换要注意以下问题。

- 对于画布标架的变换：
  - 选项 rotate 可以改变画布标架基向量的方向，但不能改变画布标架的“手性”。
  - 选项 xscale=⟨负数⟩ 或 yscale=⟨负数⟩ 可以改变画布标架的“手性”、基向量的方向、单位长度，但不能把画布标架变成非正交的。
  - 选项 xscale=⟨负数⟩ 或 yscale=⟨负数⟩ 配合选项 rotate 可以改变画布标架的“手性”、方向、单位长度、正交性。



```
\begin{tikzpicture}
    \draw (0,0)--(1,0);
    \draw (0,0)--(0,1);
    \begin{scope}[draw=red,xscale=1.5,rotate=45]
        \draw (0,0)--(1,0);
        \draw (0,0)--(0,1);
    \end{scope}
\end{tikzpicture}
```

- 对比下面两个图形：



```
\fbox{
\begin{tikzpicture}
    \draw (0,0)--(1,0);
    \draw (0,0)--(0,1);
    \begin{scope}[draw=red,xscale=1.5,]
        \draw (0,0)--(1,0);
        \draw (0,0)--(0,1);
        \draw (0,0) circle [radius=1];
    \end{scope}
\end{tikzpicture}
}
\fbox{
\begin{tikzpicture}
    \draw (0,0)--(1,0);
    \draw (0,0)--(0,1);
    \begin{scope}[draw=red,xscale=1.5,rotate=30]
        \draw (0,0)--(1,0);
        \draw (0,0)--(0,1);
        \draw (0,0) circle [radius=1];
    \end{scope}
\end{tikzpicture}
}
```

上面例子中，选项 xscale=1.5 把圆变成椭圆，表面上看，选项 rotate=30 没有起到任何作用，但实际上，第一个椭圆所面临的画布标架是 $\begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 1 & 0 \\ 0\text{pt} & 0\text{pt} & 1 \end{bmatrix}$，等效于命令

```
\pgfpathellipse{\pgfpointorigin}{\pgf@x=1.5cm\relax\pgf@y=0cm\relax}{\pgf@x=0cm
↪  \relax\pgf@y=1cm\relax}
```

第二个椭圆所面临的画布标架是 $\begin{bmatrix} 1.5\cos(30) & \sin(30) & 0 \\ -1.5\sin(30) & \cos(30) & 0 \\ 0\text{pt} & 0\text{pt} & 1 \end{bmatrix}$，等效于命令

```
\pgfpathellipse{\pgfpointorigin}
{
    \pgfmathparse{1.5*cos(30)}%
    \expandafter\pgf@x\expandafter=\pgfmathresult cm\relax%
    \pgfmathparse{sin(30)}%
    \expandafter\pgf@y\expandafter=\pgfmathresult cm\relax%
}
{
    \pgfmathparse{-1.5*sin(30)}%
    \expandafter\pgf@x\expandafter=\pgfmathresult cm\relax%
    \pgfmathparse{cos(30)}%
    \expandafter\pgf@y\expandafter=\pgfmathresult cm\relax%
}
```

- 如果只是使用标架变换选项 x={(-60:1)}，那么绘图时的最终输出标架就是仿射标架

$$\left\{\begin{pmatrix} \frac{1}{2} \\ -\frac{\sqrt{3}}{2} \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\} = \left\{\begin{pmatrix} \cos(-60^\circ) \\ \sin(-60^\circ) \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\},$$

在这个标架下画出点线正方形路径如下：

```
\begin{tikzpicture}
  \draw [x={(-60:1)}]
    (0,0) -- (1,0) -- (1,1) -- (0,1) -- cycle;
\end{tikzpicture}
```

现在要把标架转为如下左手单位直角标架

$$\left\{\begin{pmatrix} \frac{1}{2} \\ -\frac{\sqrt{3}}{2} \end{pmatrix}, \begin{pmatrix} -\frac{\sqrt{3}}{2} \\ -\frac{1}{2} \end{pmatrix}\right\} = \left\{\begin{pmatrix} \cos(-60^\circ) \\ \sin(-60^\circ) \end{pmatrix}, \begin{pmatrix} \sin(-60^\circ) \\ -\cos(-60^\circ) \end{pmatrix}\right\},$$

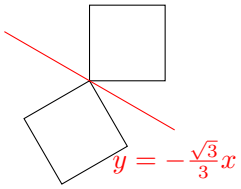
可以先在标架

$$\left\{\begin{pmatrix} \frac{1}{2} \\ -\frac{\sqrt{3}}{2} \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\}$$

中计算向量 $(-\frac{\sqrt{3}}{2}, -\frac{1}{2})$ 的坐标，它的坐标是 $(-\sqrt{3}, -2)$，将这个坐标设为 y= 的值，如下

```
[x={(-60:1)},y={({-sqrt(3)},-2)}]
```

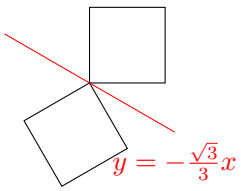
重新画上面的图形：




```
\begin{tikzpicture}
  \draw (0,0) -- (1,0) -- (1,1) -- (0,1) -- cycle;
  \draw [x={(-60:1)},y={({-sqrt(3)},-2)}]
    (0,0) -- (1,0) --   (1,1) -- (0,1) -- cycle;
  \draw [red]({-0.65*sqrt(3)},0.65)--({0.65*sqrt(3)},-0.65)
    node [below] {$y=-\frac {\sqrt{3}} 3 x$};
\end{tikzpicture}
```

其中第 2 个 \draw 命令就是在标架 $\left\{(\frac{1}{2}, -\frac{\sqrt{3}}{2}), (-\frac{\sqrt{3}}{2}, -\frac{1}{2})\right\}$ 下画图形的。由于这个标架与初始标架 $\{(1,0), (0,1)\}$ 关于直线 $y = -\frac{\sqrt{3}}{3}x$ 对称，故上面画出的两个正方形关于该直线对称，此直线的方向是 $(\cos(-30°), \sin(-30°))$.

上面的对称图需要一定的手工计算，可以直接指定左手标架来避免手工计算：




```
\begin{tikzpicture}
  \draw (0,0) -- (1,0) -- (1,1) -- (0,1) -- cycle;
  \draw [x={(-60:1cm)},y={(-150:1cm)}]
    (0,0) -- (1,0) -- (1,1) -- (0,1) -- cycle;
  \draw [red]({-0.65*sqrt(3)},0.65)--({0.65*sqrt(3)},-0.65)
    node [below] {$y=-\frac {\sqrt{3}} 3 x$};
\end{tikzpicture}
```

下面的命令

```
\draw [y={({-sqrt(3)},-2)},x={(-60:1)}]
  (0,0) -- (1,0) -- (1,1) -- (0,1) -- cycle;
```

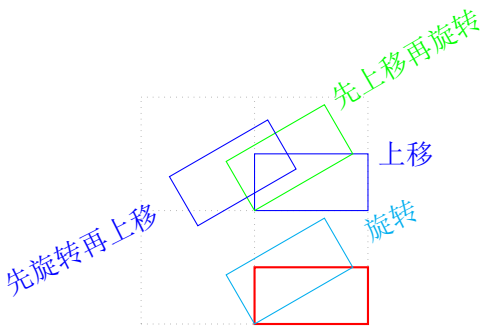
所做的标架变换是：

$$(0,0) \to (0,0), \quad (1,0) \to (\frac{5}{4}, \frac{\sqrt{3}}{4}), \quad (1,1) \to (\frac{5-2\sqrt{3}}{4}, \frac{\sqrt{3}-2}{4}), \quad (0,1) \to (-\frac{\sqrt{3}}{2}, -\frac{1}{2}),$$

画出的图形是：

```
\begin{tikzpicture}
  \draw [y={(-150:1)},x={(-60:1)}]
    (0,0) -- (1,0) --   (1,1) -- (0,1) -- cycle;
\end{tikzpicture}
```

- 不同的变换次序对结果可能有不同影响。



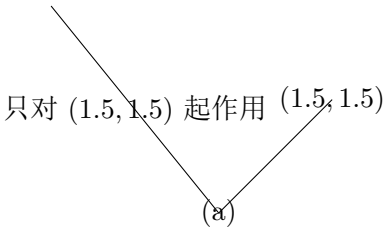
```
\begin{tikzpicture}[scale=1.5]
\draw [help lines,dotted] (-1,0) grid (1,2);
\draw [red, thick] (0,0) rectangle (1,0.5);
\draw [yshift=1cm] [blue] (0,0) rectangle (1,0.5) node[right]{上移};
\draw [rotate=30] [cyan] (0,0) rectangle node[sloped,right=1cm]{旋转} (1,0.5) ;
\draw [yshift=1cm,rotate=30] [green] (0,0) rectangle (1,0.5)
node[rotate=30,right]{先上移再旋转};
\draw [rotate=30,yshift=1cm] [blue] (0,0) rectangle node[sloped,left=1cm]{先旋转再上移}
(1,0.5);
\end{tikzpicture}
```

- 类似 (c)，(a)，(a.east)，(a.60) 这样的 Ti*k*Z 坐标名，不接受变换矩阵的作用，因为 Ti*k*Z 的 \tikz@scan@one@point →P. 710 命令，或者说 \tikz@parse@node 命令，调用 \pgfpointanchor →P. 469 来计算这种点的坐标，而命令 \pgfpointanchor 会在 node 的逆矩阵下计算坐标。

  例如，假设写出

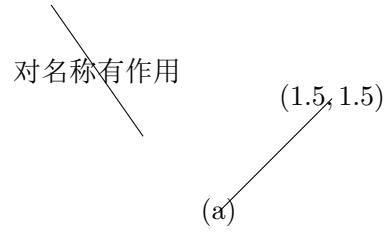
```
\coordinate (a) at (1,1);
% 或者
\tikzmath{
  \bx=(3-sqrt(3))/2;  \by=0;  \cx=3/2;  \cy=-1/2;
coordinate \b, \c;
  \b=(\bx, \by);
  \c=(\cx, \cy);
}
```

  尽管其中的 (a)，(\b)，(\c) 都代表坐标点，但是坐标变换对“名称”(a) 通常无效，也就是说，如果路径中用到了名称 (a)，该名称代表的点 (1,1) 在坐标变换下保持不动；其它的凡是以 (x，y) 或 (d:l) 形式，或者以坐标运算形式 ($ ... $)、($...!...!...$) 提供的点，都接受变换选项的作用，其中 x，y，d，l 可以是命令 \tikzmath{} 提供的实数型数值名称。

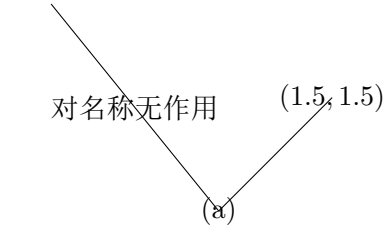



```
\begin{tikzpicture}
  \coordinate (a) at (0,0);
  \coordinate (b) at (1.5,1.5);
  \draw [shift={(-1,1)},rotate=80]
    (a) -- node{只对$(1.5,1.5)$起作用} (1.5,1.5);
  \draw (0,0) node{(a)}--(1.5,1.5) node{$(1.5,1.5)$};
\end{tikzpicture}
```

  如果把变换选项用作环境选项，那么对于坐标点的命名要在环境内完成，否则变换选项对坐标点的命名无效，比较下面两个图形：




```
\begin{tikzpicture}
  \begin{scope}[shift={(-1,1)},rotate=80]
    \coordinate (a) at (0,0);
    \coordinate (b) at (1.5,1.5);
    \draw (a) -- node{对名称有作用} (1.5,1.5);
  \end{scope}
  \draw (0,0) node{(a)}--(1.5,1.5) node{$(1.5,1.5)$};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \coordinate (a) at (0,0);
  \coordinate (b) at (1.5,1.5);
  \begin{scope}[shift={(-1,1)},rotate=80]
    \draw (a) -- node{对名称无作用} (1.5,1.5);
  \end{scope}
  \draw (0,0) node{(a)}--(1.5,1.5) node{$(1.5,1.5)$};
\end{tikzpicture}
```

46.2.4 平面上的轴对称

下面分析平面上的轴对称作图。给定一个线段 AB，一个图形 G，要作出 G 关于直线 AB 对称的图形 G'。对不同情况可以有不同方法，下面列举几种方法。

方法一： 如果图形 G 是由几个点决定的直线形或简单曲线，可以先找出这几个点的对称点，然后把这些对称点连接起来即可得到对称图形。可以使用 `($ (A)!(G)!(B) $)` 等表达式确定对称点。

方法二： 使用 `cm` 变换选项。先对一个点做一般分析。设点 $\mathbf{A}=(a_1,a_2)$，$\mathbf{B}=(b_1,b_2)$ 确定直线 AB，点 $\mathbf{P}=(p_1,p_2)$ 关于直线 AB 的对称点是 $\mathbf{P}'=(p_1',p_2')$，用 $\mathbf{A}$，$\mathbf{B}$，$\mathbf{P}$ 表达 $\mathbf{P}'$：

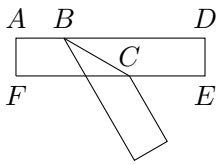
$$\begin{aligned}\mathbf{P}' &= \mathbf{P}-\mathbf{A}-2\,(\mathbf{e},\mathbf{P}-\mathbf{A})\,\mathbf{e}+\mathbf{A}\\ &= \left(\mathbf{I}-2\,(\mathbf{e},\mathbf{e}^T)\right)(\mathbf{P}-\mathbf{A})+\mathbf{A}\\ &= \frac{1}{(a_1-b_1)^2+(a_2-b_2)^2}\begin{pmatrix}(a_1-b_1)^2-(a_2-b_2)^2 & 2(a_1-b_1)(a_2-b_2)\\ 2(a_1-b_1)(a_2-b_2) & (a_2-b_2)^2-(a_1-b_1)^2\end{pmatrix}\begin{pmatrix}p_1-a_1\\ p_2-a_2\end{pmatrix}+\begin{pmatrix}a_1\\ a_2\end{pmatrix}\\ &= \begin{pmatrix}a & c\\ b & d\end{pmatrix}\begin{pmatrix}p_1-a_1\\ p_2-a_2\end{pmatrix}+\begin{pmatrix}a_1\\ a_2\end{pmatrix}=\begin{pmatrix}p_1'\\ p_2'\end{pmatrix}\end{aligned}$$

其中 $\mathbf{e}$ 是直线 AB 的单位法向量，$\left(\begin{smallmatrix}p_1-a_1\\ p_2-a_2\end{smallmatrix}\right)$ 是以 $-\mathbf{A}$ 为平移向量对 $\mathbf{P}$ 做的平移，矩阵 $\left(\begin{smallmatrix}a & c\\ b & d\end{smallmatrix}\right)$ 是反射矩阵。只要计算出上式中的各个元素就可以用 `cm` 选项作轴对称变换。可以调用数学程序库进行计算。

假设图形 G 的绘图代码是 `\draw` ⟨*G-code*⟩，那么对称图形 G' 可以用下面的代码画出：

```
\draw [cm={a,b,c,d,(a1,a2)}] [shift={(-a1,-a2)}] ⟨G-code⟩;
```

注意代码中两个变换选项的次序不能调换。注意，⟨*G-code*⟩ 中不能出现坐标名称，因为变换对坐标名称无效。下面是一个例子，将一个矩形纸条沿着 BC 折叠：




```
\tikzmath{
  \bx=(3-sqrt(3))/2;  \by=0;  \cx=3/2;  \cy=-1/2; % 设置点 B, C 的坐标
  \ff=((\bx-\cx)^2+(\by-\cy)^2)^(-1); % 分母
  \fa=(\bx-\cx)^2-(\by-\cy)^2;  \fd=-\fa;  \fbc=2*(\bx-\cx)*(\by-\cy); % 分子
  \a=\ff*\fa;  \b=\ff*\fbc;  \c=\b;  \d=-\a; % 矩阵元素
}

\begin{tikzpicture}
  \coordinate [label=above:$A$] (a) at (0,0);
  \coordinate [label=above:$B$] (b) at (\bx,\by);
  \coordinate [label=above:$C$] (c) at (\cx,\cy);
  \coordinate [label=above:$D$] (d) at (2.5,0);
  \coordinate [label=below:$E$] (e) at (2.5,-0.5);
  \coordinate [label=below:$F$] (f) at (0,-0.5);
  \draw (b)--(a)--(f);
  \draw (f)--(c);
  \draw (b)--(c);
  \draw (b)--(d)--(e)--(c);
  \draw [cm={\a,\b,\c,\d,(\bx,\by)}] [shift={(-\bx,-\by)}]
    (0:\bx) -- (2.5,0) -- (2.5,-0.5) -- (\cx,\cy);
\end{tikzpicture}
```

方法三： 利用标架变换选项 `x={}`，`y={}` 和选项 `shift={}`.

在坐标系 xOy （不变标架）中，设 $\mathbf{P}$ 是原图形 G 上的任一点，要找出点 $\mathbf{P}$ 关于直线 $\mathbf{AB}$ 的对称点 $\mathbf{P}'$，按以下步骤分析：

1. 先做平移：$\mathbf{P}\to\mathbf{P}-\mathbf{A}$.
2. 找出 xOy 坐标系关于直线 $t\cdot(\mathbf{A}-\mathbf{B})$ 对称的坐标系 $x'Oy'$.
3. 找出点 $\mathbf{Q}$，该点在 $x'Oy'$ 中的坐标为 $\mathbf{Q}_{x'Oy'}=\mathbf{P}-\mathbf{A}$，在 xOy 中的坐标记为 $\mathbf{Q}_{xOy}=(q_1,q_2)$.
4. 在 xOy 坐标系中做平移：$\mathbf{Q}_{xOy}\to\mathbf{Q}_{xOy}+\mathbf{A}=\mathbf{P}'$ 得到要找的点 $\mathbf{P}'$.

算式

$$\left(\mathbf{I}-2\left(\mathbf{e},\mathbf{e}^T\right)\right)(\mathbf{P}-\mathbf{A})=\mathbf{Q},$$

表示点 $\mathbf{P}-\mathbf{A}$ 关于参数直线 $t\cdot(\mathbf{A}-\mathbf{B})$ 的对称点。

坐标系 $x'Oy'$ 是左手系，可设其单位向量在 xOy 坐标系内的表达式是

$$x'\text{轴单位向量}:\begin{pmatrix}\cos\varphi\\-\sin\varphi\end{pmatrix}=(-\varphi:1),\quad y'\text{轴单位向量}:\begin{pmatrix}-\sin\varphi\\-\cos\varphi\end{pmatrix}=(-\varphi-90^\circ:1)$$

设 $\mathbf{P}-\mathbf{A}$ 在 xOy 坐标系内的坐标是 (α,β)，则

$$xOy\text{系}:\mathbf{P}-\mathbf{A}=\alpha\begin{pmatrix}1\\0\end{pmatrix}+\beta\begin{pmatrix}0\\1\end{pmatrix}$$

$$x'Oy'\text{系}:\mathbf{Q}_{x'Oy'}=\alpha\begin{pmatrix}\cos\varphi\\-\sin\varphi\end{pmatrix}+\beta\begin{pmatrix}-\sin\varphi\\-\cos\varphi\end{pmatrix}$$

再找出 $\mathbf{A}$，$\mathbf{B}$ 与 $(-\varphi:1)$ 和 $(-\varphi-90^\circ:1)$ 的关系。

参数直线 $t\cdot(\mathbf{A}-\mathbf{B})$ 的方向是 $\mathbf{A}-\mathbf{B}$，如果能求得这个方向与 xOy 系的单位向量之间的夹角，用对称的办法就能容易地知道系 $x'Oy'$ 的单位向量的坐标。设从方向向量 $\mathbf{A}-\mathbf{B}$ 到单位向量 $(1,0)$ 的夹角是 θ_1，从方向向量 $\mathbf{A}-\mathbf{B}$ 到单位向量 $(0,1)$ 的夹角是 θ_2，$|\theta_i|\leqslant 180^\circ, i=1,2$，计算外积

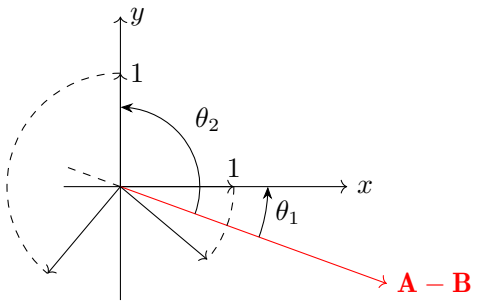
$$(\mathbf{A}-\mathbf{B})\times(1,0)=|\mathbf{A}-\mathbf{B}|\sin\theta_1=b_2-a_2$$

$$(\mathbf{A}-\mathbf{B})\times(0,1)=|\mathbf{A}-\mathbf{B}|\sin\theta_2=a_1-b_1$$

注意反三角函数的值域 $0^\circ\leqslant\arccos t\leqslant 180^\circ$，$-90^\circ\leqslant\arcsin t\leqslant 90^\circ$，可以分以下几种情况分析

(i) 如果 $b_2-a_2>0$ 且 $a_1-b_1>0$，则 $\mathbf{A}-\mathbf{B}$ 在第四象限，此时

$$\theta_1=\arcsin\frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|},\quad \theta_2=\theta_1+90^\circ$$



坐标系 xOy 变为坐标系 $x'Oy'$，则 x 轴的单位向量 $(1,0)$ 变为 x' 轴的单位向量

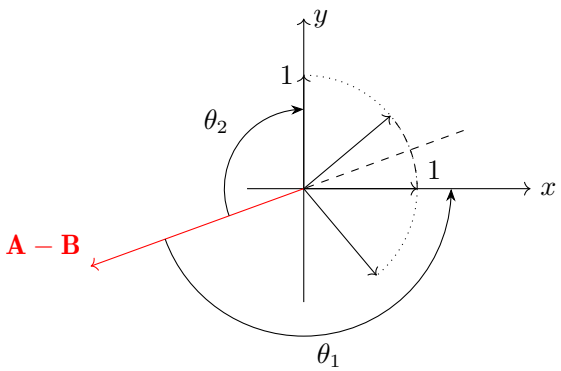
$$(\cos(-2\theta_1),\sin(-2\theta_1))=(\cos 2\theta_1,-\sin 2\theta_1)$$

因为坐标系 $x'Oy'$ 是左手系，故 y 轴的单位向量 $(0,1)$ 变为 y' 轴的单位向量

$$(-\sin 2\theta_1,-\cos 2\theta_1)$$

(ii) 如果 $b_2-a_2>0$ 且 $a_1-b_1<0$，则 $\mathbf{A}-\mathbf{B}$ 在第三象限，此时

$$\theta_1=180^\circ-\arcsin\frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|},\quad \theta_2=\theta_1-270^\circ$$



x 轴的单位向量 $(1,0)$ 变为 x' 轴的单位向量

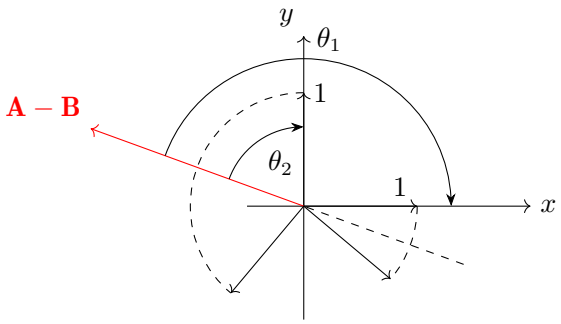
$$(\cos 2(180^\circ-\theta_1),\sin 2(180^\circ-\theta_1))=(\cos 2\theta_1,-\sin 2\theta_1)$$

y 轴的单位向量 $(0,1)$ 变为 y' 轴的单位向量

$$(-\sin 2\theta_1,-\cos 2\theta_1)$$

(iii) 如果 $b_2-a_2<0$ 且 $a_1-b_1<0$，则 $\mathbf{A}-\mathbf{B}$ 在第二象限，此时

$$\theta_2=\arcsin\frac{a_1-b_1}{|\mathbf{A}-\mathbf{B}|},\quad \theta_1=\theta_2-90^\circ=-180^\circ-\arcsin\frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$$



x 轴的单位向量 $(1,0)$ 变为 x' 轴的单位向量

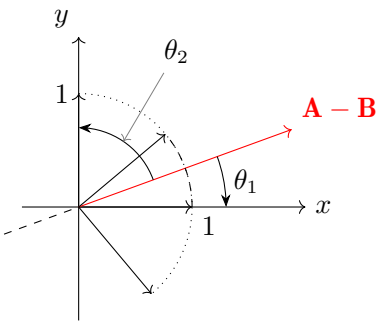
$$(\cos(-2(180^\circ+\theta_1)),\sin(-2(180^\circ+\theta_1)))=(\cos 2\theta_1,-\sin 2\theta_1)$$

y 轴的单位向量 $(0,1)$ 变为 y' 轴的单位向量

$$(-\sin 2\theta_1,-\cos 2\theta_1)$$

(iv) 如果 $b_2-a_2<0$ 且 $a_1-b_1>0$，则 $\mathbf{A}-\mathbf{B}$ 在第一象限，此时

$$\theta_2=\arcsin\frac{a_1-b_1}{|\mathbf{A}-\mathbf{B}|},\quad \theta_1=\theta_2-90^\circ=\arcsin\frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$$



x 轴的单位向量 $(1,0)$ 变为 x' 轴的单位向量

$$(\cos(-2\theta_1),\sin(-2\theta_1))=(\cos 2\theta_1,-\sin 2\theta_1)$$

y 轴的单位向量 $(0,1)$ 变为 y' 轴的单位向量

$$(-\sin 2\theta_1,-\cos 2\theta_1)$$

(v) 对于 $b_2-a_2=0$ 或 $a_1-b_1=0$ 的情况，显然，x 轴的单位向量 $(1,0)$ 变为 x' 轴的单位向量

$$(\cos(-2\theta_1),\sin(-2\theta_1))=(\cos 2\theta_1,-\sin 2\theta_1)$$

y 轴的单位向量 $(0,1)$ 变为 y' 轴的单位向量

$$(-\sin 2\theta_1,-\cos 2\theta_1)$$

总结起来，有

$\mathbf{A}-\mathbf{B}$	$-180^\circ\leqslant\theta_1\leqslant 180^\circ$	反正弦值	与 $-2\theta_1$ 终边相同的角度
第一象限	$\theta_1=\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}$	$-90^\circ\leqslant\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}\leqslant 0^\circ$	$-2\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}$
第二象限	$\theta_1=-180^\circ-\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}$	$-90^\circ\leqslant\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}\leqslant 0^\circ$	$2\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}$
第三象限	$\theta_1=180^\circ-\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}$	$0^\circ\leqslant\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}\leqslant 90^\circ$	$2\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}$
第四象限	$\theta_1=\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}$	$0^\circ\leqslant\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}\leqslant 90^\circ$	$-2\arcsin\frac{b_2-a_2}{\lvert\mathbf{A}-\mathbf{B}\rvert}$

并且总有

xOy系的标架		$x'Oy'$系的标架
$(1,0)$	$\longrightarrow$	$(\cos 2\theta_1,-\sin 2\theta_1)=(-2\theta_1:1)$
$(0,1)$	$\longrightarrow$	$(-\sin 2\theta_1,-\cos 2\theta_1)=(-2\theta_1-90^\circ:1)$

令

$$\xi=\frac{a_1-b_1}{|a_1-b_1|}\arcsin\frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|},$$

则 -2ξ 与 $-2\theta_1$ 的终边相同，所以

xOy系的标架		$x'Oy'$系的标架
$(1,0)$	$\longrightarrow$	$(\cos 2\xi,-\sin 2\xi)=(-2\xi:1)$
$(0,1)$	$\longrightarrow$	$(-\sin 2\xi,-\cos 2\xi)=(-2\xi-90^\circ:1)$

还有

$\mathbf{A}-\mathbf{B}$	ξ	θ_1
第一象限	$-90^\circ \leqslant \xi \leqslant 0^\circ$	$\theta_1 = \xi$
第二象限	$0^\circ \leqslant \xi \leqslant 90^\circ$	$\theta_1 = \xi - 180^\circ$
第三象限	$-90^\circ \leqslant \xi \leqslant 0^\circ$	$\theta_1 = \xi + 180^\circ$
第四象限	$0^\circ \leqslant \xi \leqslant 90^\circ$	$\theta_1 = \xi$

由此容易得到 ξ 的正弦值是

$$\sin\xi = \frac{a_1-b_1}{|a_1-b_1|}\frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|},$$

再用 $\cos^2 t + \sin^2 t = 1$ 可以得到 ξ 的余弦值是

$$\cos\xi = \frac{a_1-b_1}{|\mathbf{A}-\mathbf{B}|},$$

所以

$$\cos(-2\theta_1) = \cos(-2\xi) = \frac{(a_1-b_1)^2-(a_2-b_2)^2}{(a_1-b_1)^2+(a_2-b_2)^2}$$
$$\sin(-2\theta_1) = \sin(-2\xi) = -2\frac{a_1-b_1}{|a_1-b_1|}\frac{(a_1-b_1)(a_2-b_2)}{(a_1-b_1)^2+(a_2-b_2)^2}$$

可以调用数学程序库计算 ξ，例如，

```
\tikzmath{
  coordinate \dirvec;
    \dirvec=(a1 - b1, a2 - b2); % 方向向量 A-B
    \xival=sign(a1 - b1) * asin(-\dirvecy / veclen(\dirvec)); % 计算 xi 的值
    }
```

由于变换选项会改变变换所参照的当前标架，所以为了能转换到 $x'Oy'$ 系的标架，需要计算 $(-\sin 2\theta_1, -\cos 2\theta_1)$ 在标架 $\{(\cos 2\theta_1, -\sin 2\theta_1), (0,1)\}$ 下的坐标，先计算逆矩阵

$$T = \begin{bmatrix} \cos 2\theta_1 & 0 \\ -\sin 2\theta_1 & 1 \end{bmatrix}, \quad T^{-1} = \begin{bmatrix} \frac{1}{\cos 2\theta_1} & 0 \\ \frac{\sin 2\theta_1}{\cos 2\theta_1} & 1 \end{bmatrix}$$

需要计算的坐标就是

$$T^{-1}\begin{pmatrix} -\sin 2\theta_1 \\ -\cos 2\theta_1 \end{pmatrix} = \begin{pmatrix} -\frac{\sin 2\theta_1}{\cos 2\theta_1} \\ -\frac{\sin^2 2\theta_1}{\cos 2\theta_1} - \cos 2\theta_1 \end{pmatrix} = \frac{-1}{\cos 2\theta_1}\begin{pmatrix} \sin 2\theta_1 \\ 1 \end{pmatrix} = \frac{-1}{\cos 2\xi}\begin{pmatrix} \sin 2\xi \\ 1 \end{pmatrix},$$

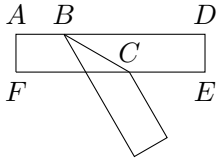
即

$$\left(-2\frac{a_1-b_1}{|a_1-b_1|}\frac{(a_1-b_1)(a_2-b_2)}{(a_1-b_1)^2-(a_2-b_2)^2}, -\frac{(a_1-b_1)^2-(a_2-b_2)^2}{(a_1-b_1)^2+(a_2-b_2)^2}\right)^T.$$

还是假设图形 G 的绘图代码是 `\draw` ⟨*G-code*⟩，那么对称图形 G' 可以用下面的代码画出：

```
\draw [shift={(a1, a2)},x={(-2*\xival : 1)},
       y={($-(1/cos(2*\xival))$*({sin(2*\xival)}, 1)$)},shift={(-a1, -a2)}]
      ⟨G-code⟩ ;
% 注意 ⟨G-code⟩ 中不能用 coordinate 名称
```

用这个方法重画前面折叠矩形纸条的例子：



```
\tikzmath{
  \bx=(3-sqrt(3))/2;  \by=0;  \cx=3/2;  \cy=-1/2; % 设置点 B, C 的坐标
  coordinate \dirvec;
    \dirvec=(\bx - \cx, \by - \cy); % 方向向量 B-C
    \xival=sign(\bx-\cx)*asin(-\dirvecy / veclen(\dirvec)); % 计算 xi 的值
    }

\begin{tikzpicture}
  \coordinate [label=above:$A$] (a) at (0,0);
  \coordinate [label=above:$B$] (b) at (\bx,\by);
  \coordinate [label=above:$C$] (c) at (\cx,\cy);
  \coordinate [label=above:$D$] (d) at (2.5,0);
  \coordinate [label=below:$E$] (e) at (2.5,-0.5);
  \coordinate [label=below:$F$] (f) at (0,-0.5);
  \draw (b)--(a)--(f);
  \draw (f)--(c);
  \draw (b)--(c);
  \draw (b)--(d)--(e)--(c);
  \draw [shift={(\bx,\by)},x={(-2*\xival:1)},
    y={({-sin(2*\xival)/cos(2*\xival)}, -1/cos(2*\xival))},shift={(-\bx,-\by)}]
    (0:\bx) -- (2.5,0) -- (2.5,-0.5) -- (\cx,\cy);
\end{tikzpicture}
```

方法四： 利用标架变换选项 `y={}`，`shift={}` 和选项 `rotate={}`.

继续沿用前面设定的符号。用下面的思路找出点 $\mathbf{P}$ 的对称点 $\mathbf{P}'$：

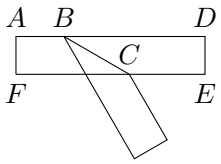
1. 先做平移：$\mathbf{P} \to \mathbf{P} - \mathbf{A}$.
2. 转换标架
$$\left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}\right\} \longrightarrow \left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\-1\end{pmatrix}\right\}.$$
3. 假设向量 $\mathbf{A}-\mathbf{B}$ 到单位向量 $(1,0)$ 的夹角是 θ_1，将标架 $\{\binom{0}{1}, \binom{0}{-1}\}$ 旋转 $-2\theta_1$ 角度，相当于旋转 -2ξ 角度：
$$\left\{\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}0\\-1\end{pmatrix}\right\} \longrightarrow \left\{\begin{pmatrix}\cos 2\xi\\-\sin 2\xi\end{pmatrix}, \begin{pmatrix}-\sin 2\xi\\-\cos 2\xi\end{pmatrix}\right\}.$$
由此得到的坐标系 $x'Oy'$ 与坐标系 xOy 关于直线 $t\cdot(\mathbf{A}-\mathbf{B})$ 对称.
4. 在 $x'Oy'$ 中找出坐标为 $\mathbf{Q}_{x'Oy'} = \mathbf{P} - \mathbf{A}$ 的点 $\mathbf{Q}$，它在 xOy 中的坐标记为 $\mathbf{Q}_{xOy} = (q_1, q_2)$。
5. 在 xOy 坐标系中做平移：$\mathbf{Q}_{xOy} \to \mathbf{Q}_{xOy} + \mathbf{A} = \mathbf{P}'$.

还是利用数学程序库计算 ξ，绘图代码可以是

```
\tikzmath{
  coordinate \dirvec;
    \dirvec=(a1 - b1, a2 - b2); % 方向向量 A-B
    \xival=sign(a1 - b1) * asin(-\dirvecy / veclen(\dirvec)); % 计算 xi 的值
    }

\draw [shift={(a1, a2)}]
      [y={(0, -1)}, rotate=-2*\xival]
      [shift={(-a1, -a2)}]
      ⟨G-code⟩ ;
% 注意 ⟨G-code⟩ 中不能用 coordinate 名称
```

用这个思路重画前面折叠矩形纸条的图形：




```
\tikzmath{
  \bx=(3-sqrt(3))/2;  \by=0;  \cx=3/2;  \cy=-1/2; % 设置点 B, C 的坐标
  coordinate \dirvec;
    \dirvec=(\bx - \cx, \by - \cy); % 方向向量 B-C
    \xival=sign(\bx-\cx)*asin(-\dirvecy / veclen(\dirvec)); % 计算 xi 的值
    }

\begin{tikzpicture}
  \coordinate [label=above:$A$] (a) at (0,0);
  \coordinate [label=above:$B$] (b) at (\bx,\by);
  \coordinate [label=above:$C$] (c) at (\cx,\cy);
  \coordinate [label=above:$D$] (d) at (2.5,0);
  \coordinate [label=below:$E$] (e) at (2.5,-0.5);
  \coordinate [label=below:$F$] (f) at (0,-0.5);
  \draw (b)--(a)--(f);
  \draw (f)--(c);
  \draw (b)--(c);
  \draw (b)--(d)--(e)--(c);
  \draw [shift={(\bx,\by)}]
        [y={(0, -1)}, rotate=-2*\xival]
        [shift={(-\bx,-\by)}]
    (0:\bx) -- (2.5,0) -- (2.5,-0.5) -- (\cx,\cy);
\end{tikzpicture}
```

与前两种方法相比，由于用了旋转变换选项，这个方法的代码稍微简洁一些。

方法五： 利用标架变换选项 `x={}`，`y={}` 指定标架。

符号约定如前，得到点 $\mathbf{P}'$ 的步骤与方法三相同，只不过在确定坐标系 $x'Oy'$ 时利用选项 `x={}`，`y={}` 来直接指定标架。

在方法四中已经得到：

$$\begin{array}{ccc} xOy\text{系的标架} & & x'Oy'\text{系的标架} \\ (1,0) & \longrightarrow & (\cos 2\xi, -\sin 2\xi) = (-2\xi : 1) \\ (0,1) & \longrightarrow & (-\sin 2\xi, -\cos 2\xi) = (-2\xi - 90^\circ : 1) \end{array}$$

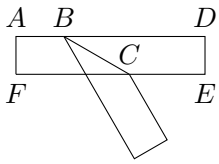
计算 ξ 的值并保存：

```
\tikzmath{
  coordinate \dirvec;
    \dirvec=(a1 - b1, a2 - b2); % 方向向量 A-B
    \xival=sign(a1 - b1) * asin(-\dirvecy / veclen(\dirvec)); % 计算 xi 的值
    }
```

然后用下面的代码画出对称图形 G'：

```
\draw [shift={(a1, a2)}]
      [x={(-2*\xival:1cm)},y={(-2*\xival-90:1cm)}]
      [shift={(-a1, -a2)}]
      ⟨G-code⟩ ;
% 注意 ⟨G-code⟩ 中不能用 coordinate 名称
```

用这个方法重画前面折叠矩形纸条的例子：




```
\tikzmath{
  \bx=(3-sqrt(3))/2;  \by=0;  \cx=3/2;  \cy=-1/2; % 设置点 B, C 的坐标
  coordinate \dirvec;
    \dirvec=(\bx - \cx, \by - \cy); % 方向向量 B-C
    \xival=sign(\bx-\cx)*asin(-\dirvecy / veclen(\dirvec)); % 计算 xi 的值
    }

\begin{tikzpicture}
  \coordinate [label=above:$A$] (a) at (0,0);
  \coordinate [label=above:$B$] (b) at (\bx,\by);
  \coordinate [label=above:$C$] (c) at (\cx,\cy);
  \coordinate [label=above:$D$] (d) at (2.5,0);
  \coordinate [label=below:$E$] (e) at (2.5,-0.5);
  \coordinate [label=below:$F$] (f) at (0,-0.5);
  \draw (b)--(a)--(f);
  \draw (f)--(c);
  \draw (b)--(c);
  \draw (b)--(d)--(e)--(c);
  \draw [shift={(\bx,\by)}]
        [x={(-2*\xival:1cm)},y={(-2*\xival-90:1cm)}]
        [shift={(-\bx,-\by)}]
    (0:\bx) -- (2.5,0) -- (2.5,-0.5) -- (\cx,\cy);
\end{tikzpicture}
```

方法六： 如果图形 G 由数个绘图命令构成，那么可以把 ⟨*G-code*⟩ 放入 {scope} 环境中，并给 {scope} 环境使用变换选项。

46.3 画布变换

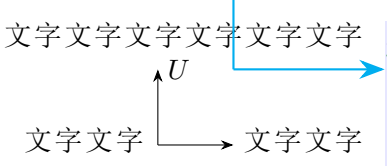
打个比方说，当气球充气时，气球上的图画变化就可以比喻成“画布变换”。线条、文字以及其它的项目都发生变化。当做画布变换时，PGF 不再跟踪 node 的位置，也不再计算图形的尺寸，因为它目前还不能将画布变换的结果纳入计算之内。在将来可能会改进这一点。

画布变换会作用于整个路径，不能只作用于某一段子路径，这一点与坐标变换不同。

应尽量避免使用画布变换。

/tikz/transform canvas=⟨*options*⟩ (no default)

这个选项引入画布变换，其中 ⟨*options*⟩ 所包含的选项，就是前面所讲的关于坐标变换的选项。画布变换与坐标变换会叠加。




```
文字文字文字文字文字文字文字文字文字文字文字
\begin{tikzpicture}[>=Stealth]
  \draw [->](0,0)--(1,0);
  \draw [->](0,0)--(0,1)node[right]{$U$};
  {[cyan,transform canvas={shift={(1cm,1cm)},scale=2}]
    \draw [->](0,0)--(1,0);
    \draw [->](0,0)--(0,1)node[right]{$U$};
  }
\end{tikzpicture}
文字文字文字文字文字文字文字文字文字文字文字
```

上面的图形突入周围的文字中，是因为在画布变换下 PGF 无法跟踪图形的边界盒子。

注意，改变 xyz 标架的选项不能用作画布变换，也就是说，下面代码：

```
\begin{tikzpicture}[transform canvas={x={(1cm,1cm)}}]
  \draw (0,0)--(1,0) (0,0)--(0,1);
\end{tikzpicture}
```

其中的选项 x={(1cm,1cm)} 没有作用。

46.4 TikZ 内部的变换命令

TikZ 的那些改变 canvas 标架 (变换矩阵) 的变换选项基本上都会执行 \tikz@addtransform, 例如选项 /tikz/scale 的定义是：

```
\tikzoption{scale}{\tikz@addtransform{\pgftransformscale{#1}}}%
```

选项 /tikz/shift 的定义是：

```
\tikzoption{shift}{\tikz@addtransform{\tikz@scan@one@point\pgftransformshift#1\relax}}
↪  %
```

\tikz@addtransform{⟨*transform command*⟩}

此命令的定义是：

```
\def\tikz@addtransform#1{%
  \ifx\tikz@transform\relax
    #1%
  \else
    \expandafter\def\expandafter\tikz@transform\expandafter{\tikz@transform#1}%
  \fi
}%
```

如果 \tikz@transform 等于 \relax, 那么就返回其参数 ⟨*transform command*⟩, 通常就是直接执行 ⟨*transform command*⟩; 如果 \tikz@transform 不等于 \relax, 那么重定义 \tikz@transform, 将 ⟨*transform command*⟩ 添加到 \tikz@transform 中。

当需要单独保存、执行某些变换命令时，可以

1. 先

   ```
   \let\tikz@transform=\pgfutil@empty
   ```

   使得 \tikz@transform 不等于 \relax.
2. 然后执行变换选项或 \tikz@addtransform{⟨*transform command*⟩}, 将各变换命令保存到 \tikz@transform 中。
3. 再将保存在 \tikz@transform 中的变换命令转存到其他宏中,也可以直接执行 \tikz@transform 来执行其中保存的变换命令。
4. 然后再

   ```
   \let\tikz@transform=\relax
   ```

\tikz@transform

参考 \tikz@addtransform.

这个宏的值经常变化：

- 有的命令，如 \begin{tikzpicture}(实际是 \tikz@picture) 会设置

  ```
  \let\tikz@transform=\relax
  ```

- 在文件《tikz.code.tex》中，有的命令，如
  - \tikz@do@edge(edge 操作)

– \tikz@@@plot(plot 操作)
– \tikz@circle@opt(circle 操作)
– \tikz@normal@fig(解析 node 语句)
– \tikz@late@options
– \tikz@children@collected
– \tikz@childnode
– \tikz@scan@handle@options(解析坐标点自己带的选项，如 ([rotate=90]1,2))

都会设置

```
\let\tikz@transform=\pgfutil@empty
```

然后再处理选项 (包括变换选项)，将变换命令被保存到 \tikz@transform 中。然后再执行

```
\let\tikz@transform=\relax
```

对点坐标做变换的是底层的 PGF 的命令，例如 PGF 的命令 \pgfpathmoveto{⟨*basic point*⟩} 会调用 \pgfpointtransformed→ P. 255{⟨*basic point*⟩} 来对 {⟨*basic point*⟩} 做变换。也就是说，TikZ 只是提供坐标和变换信息，并不执行变换——不过有例外，例如，当命令 \tikz@scan@one@point 解析 ([shift={(1,2)}]3,4) 时，会直接用平移 shift={(1,2)} 对 (3,4) 做变换 (见 \tikz@scan@handle@options)。

命令 \pgfpathmoveto{⟨*basic point*⟩} 会转变为底层的软路径形式

\pgfsyssoftpath@moveto{⟨\the\pgf@x⟩}{⟨\the\pgf@y⟩}

其中的坐标 ⟨\the\pgf@x⟩, ⟨\the\pgf@y⟩ 是用变换矩阵对 ⟨*basic point*⟩ 做变换后的坐标；当在 TikZ 的路径命令中引用坐标 ⟨\the\pgf@x⟩, ⟨\the\pgf@y⟩ 时，应当把针对这个坐标的变换矩阵设置成单位矩阵，即使用命令 \pgftransformreset, 或者选项 /tikz/reset cm, 否则不能得到期望的位置点。

“变换”属于图形状态参数，其有效范围受到 pgfscope→ P. 238 环境的限制。scope 环境会在自己内部创建一个 pgfscope 环境。实际上，变换命令都利用 \pgftransformcm→ P. 263 来修改变换矩阵，命令 \pgftransformcm 对变换矩阵的修改“不是”全局的，所以变换命令的作用受到 TEX 组的限制。

第四十七章　矩阵及其对齐方式

47.1　Overview

TikZ 的矩阵功能基于 PGF 的 matrix 模块，参考 \pgfmatrix → P. 479.

在 matrix 库中定义了几个有用的选项。

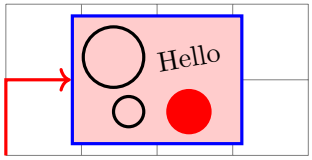
一个 TikZ 的矩阵（matrix）类似于 LaTeX 的 {tabular} 或 {array} 环境，只不过矩阵的元素（cell）是 TikZ 的绘图代码（或者留空）。矩阵的每一行（包括最后一行）都用 \\ 结束，一行内的相邻元素之间用 & 分隔。注意 & 和 \\ 的后面可以带有方括号选项。

47.2　Matrices are Nodes

必须用 node 路径来构造矩阵。当 node 带有 matrix 选项后，这个 node 就用于构造矩阵，也就是说，一个矩阵其实是个 node，该 node 的名称就是矩阵名称，该 node 的形状就是矩阵的形状。

/tikz/matrix=⟨*true or false*⟩　(default true)

这个选项用于 node，使得该 node 用来构造矩阵。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,2);
  \node [matrix,fill=red!20,draw=blue,very thick] (my matrix) at (2,1)
  {
    \draw (0,0) circle (4mm);   & \node[rotate=10] {Hello}; \\
    \draw (0.2,0) circle (2mm); & \fill[red] (0,0) circle (3mm); \\
  };
  \draw [very thick,red,->] (0,0) |- (my matrix.west);
\end{tikzpicture}
```

/tikz/every matrix　(style, initially empty)

这个选项设置的样式（在有效范围内）用于每个矩阵。

\matrix

在 {tikzpicture} 环境里，这是 \path node[matrix] 的简写。

矩阵（作为 node）也可以添加到别的路径上，也可以引用矩阵的 node 坐标系统或者引用矩阵内部的 node. 针对 node 的大多数（不是全部的）操作、选项都可以用于矩阵。针对整个矩阵的旋转和放缩变换无效，针对矩阵元素的变换有效。对于命令 \matrix（或其等效语句）而言，以 text 开头的选项（如 text width）无效。

47.3　元素图形

矩阵有确定的行数和列数，如果某一行的元素个数不足 (即分列符 & 比其他行少) 就自动用“空元素”（empty cells）填补。

8 1
3 5 7
4 9

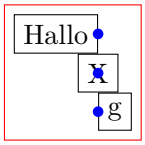
```
\begin{tikzpicture}
  \matrix [matrix of nodes]
  {
    8 & 1  \\
    3 & 5 & 7 \\
    4 & 9  \\
};
\end{tikzpicture}
```

矩阵的元素是“图形”，绘制图形当然需要一个坐标系。当创建某个元素图形时，程序会开启一个专属于这个元素图形的“私有”坐标系，定义该元素图形的各条绘图命令就在这个坐标系内画图。元素图形是在简化的 {pgfpicture} 环境中画出的，这导致元素图形（cell pictures）没有“图层”（layer）的概念。

47.3.1 元素图形的对齐方式

在默认之下：对于一行的元素图形而言，它们的坐标系的原点处于同一水平线上，以此线为界，元素图形在此线之上的部分属于该元素图形的高度，在此线之下的部分属于该元素图形的深度；一行的高度等于该行中诸元素图形的最大高度，是该行的上界；一行的深度等于该行中诸元素图形的最大深度，是该行的下界；对于相邻的两行而言，上行的下界紧邻下行的上界，上下两行的间距为 0，除非用 row sep 选项或用其它方式设置两行间距。

在默认之下：对于一列的元素图形而言，它们的坐标系的原点处于同一竖直线上；对于相邻的两列而言，左列的右边界紧邻右列的左边界，左右两列间距为 0，除非用 column sep 选项或其它方式设置两列间距。




```
\begin{tikzpicture}[every node/.style={draw}]
  \matrix [draw=red]
  {
    \node[left] {Hallo}; \fill[blue] (0,0) circle (2pt); \\
    \node {X}; \fill[blue] (0,0) circle (2pt); \\
    \node[right] {g}; \fill[blue] (0,0) circle (2pt); \\
  };
\end{tikzpicture}
```

上面图形中，第一个 node 的锚位置 left 位于（第一个元素图形坐标系的）原点；第三个 node 的锚位置 right 位于（第三个元素图形坐标系的）原点。

a X g
a X g

a X g
a X g

```
\tikz[font=\Large]\matrix [draw=red]
  {
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
  };
\quad
\tikz[font=\Large,anchor=base]\matrix [draw=red]
  {
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
  };
```

上面例子中，第二个图形用了 anchor=base，这个选项对该图形内的所有 node 有效，将各图的 base 位置置于各图的原点，使得对齐效果不同于第一个图形。

47.3.2 调整行距和列距

调整行距和列距有 2 种方式：

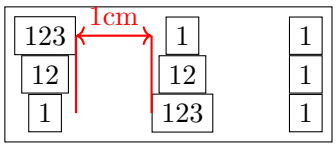
- 用 column sep 和 row sep.
- 给 & 或 \\ 带上长度选项。

/tikz/column sep=⟨*spacing list*⟩ (default 0pt,between borders)

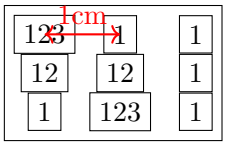
这个选项用于指定列间距。⟨*spacing list*⟩ 有两种情况：

- 是一个长度尺寸，例如 10pt，也可以是负值尺寸，如 -2mm.
- 是一个长度与一个词组的组合，例如 {5mm,between origins}, 或者 {between borders,-2mm}。

词组 between origins 决定行距、列距的计算方式是“从原点到原点”的；词组 between borders 决定行距、列距的计算方式是“从边界到边界”的；between borders 是默认的计算方式。




```
\begin{tikzpicture}
  \matrix [draw,column sep=1cm,nodes=draw]
  {
    \node(a) {123}; & \node (b) {1}; & \node {1}; \\
    \node {12}; & \node {12}; & \node {1}; \\
    \node(c) {1}; & \node (d) {123}; & \node {1}; \\
  };
  \draw [red,thick] (a.east) -- (a.east |- c)
                    (d.west) -- (d.west |- b);
  \draw [<->,red,thick] (a.east) -- (d.west |- b)
    node [above,midway] {1cm};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [draw,column sep={1cm,between origins},nodes=draw]
  {
    \node(a) {123}; & \node (b) {1}; & \node {1}; \\
    \node {12}; & \node {12}; & \node {1}; \\
    \node {1}; & \node {123}; & \node {1}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center)
    node [above,midway] {1cm};
\end{tikzpicture}
```

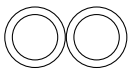
如果左右两个元素图形有重叠，则右侧的元素图形遮挡左侧的。


```
\tikz[font=\Huge]
  \matrix [draw=red,column sep=-4mm]
    {
    \node [fill=red]{a}; & \node [fill=cyan]{X}; \\
    \node {a}; & \node {X}; \\
    };
```

/tikz/row sep=⟨*spacing list*⟩ (default 0pt,between borders)

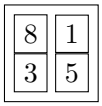
这个选项指定行距，⟨*spacing list*⟩ 的情况参照上一选项。如果上下两个元素图形有重叠，则下部的元素图形遮挡上部的。

在分列符 & 或换行符 \\ 的后面可以使用方括号，把 ⟨*spacing list*⟩ 放入方括号内，⟨*spacing list*⟩ 中的间距会叠加在选项 column sep 或 row sep 指定的间距上。下面例子中，在两个地方指定行距，总行距是 1mm-1mm=0mm：

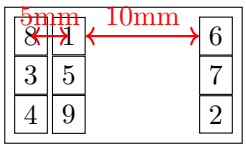


```
\begin{tikzpicture}
  \matrix [row sep=1mm]
  {
    \draw (0,0) circle (4mm); & \draw (0,0) circle (4mm); \\
    ↪  [-1mm,between origins]
    \draw (0,0) circle (3mm); & \draw (0,0) circle (3mm); \\
  };
\end{tikzpicture}
```

当用 &[⟨*spacing list*⟩] 指定某两列的间距时，只能用在第一行，否则无效。




```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep=1mm]
  {
    \node {8}; &                          \node{1}; \\
    \node {3}; &[20mm,between origins] \node{5}; \\
  };
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep={0.5cm,between origins}]
  {
    \node (a){8}; & \node(b){1}; &[1cm,between borders] \node(c){6}; \\
    \node {3};    & \node {5};   & \node {7}; \\
    \node {4};    & \node {9};   & \node {2}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {5mm};
  \draw [<->,red,thick] (b.east) -- (c.west) node [above,midway] {10mm};
\end{tikzpicture}
```

47.3.3 设置元素图形样式的选项

注意 \matrix 自己的 draw，fill 选项不能传递给矩阵的元素图形，但颜色选项能传递给元素的文字。

/tikz/every cell (style, no default, initially empty)

这个 key 设置的样式会加在每个元素图形的开头。注意这个样式中的 draw，fill 等选项不会传递给元素图形，但颜色选项则能传递。

有两个预定义的宏 \pgfmatrixcurrentrow 和 \pgfmatrixcurrentcolumn，分别代表当前元素的行号和列号（是计数器数值）。

/tikz/cells=⟨*options*⟩ (no default)

等效于 every cell/.append style=⟨*options*⟩.

/tikz/nodes=⟨*options*⟩ (no default)

等效于 every node/.append style=⟨*options*⟩. 如果把这个选项作为 matrix 的选项，则这个选项的设置对每个元素图形有效，但对矩阵本身无效。这个选项会把 draw，fill 等选项传递给所有元素图形。

以下样式（style）的可以在一个矩阵中重复使用，它们设置的样式会被叠加，它们都会附加在 every cell 之后。注意它们不会把 draw，fill 等选项传递给元素图形。

/tikz/column⟨*number*⟩ (style, no value)

这个样式针对第 ⟨*number*⟩ 列的所有元素。

/tikz/every odd column⟨*number*⟩ (style, no value)

这个样式针对第奇数列的所有元素。

/tikz/every even column⟨*number*⟩ (style, no value)

这个样式针对第偶数列的所有元素。

/tikz/row⟨*number*⟩ (style, no value)

这个样式针对第 ⟨*number*⟩ 行的所有元素。

/tikz/every odd row⟨*number*⟩ (style, no value)

这个样式针对第奇数行的所有元素。

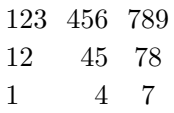
/tikz/every even row⟨*number*⟩ (style, no value)

这个样式针对第偶数行的所有元素。

/tikz/row ⟨*row number*⟩ **column** ⟨*col number*⟩ (style, no value)

这个样式针对第 ⟨*row number*⟩ 行、⟨*col number*⟩ 列元素。

```
\begin{tikzpicture}
  [row 1/.style={red},
   column 2/.style={green!50!black},
   row 3 column 3/.style={font=\Large}]
  \matrix
  {
    \node {8}; & \node{1}; & \node {6}; \\
    \node {3}; & \node{5}; & \node {7}; \\
    \node {4}; & \node{9}; & \node {2}; \\
  };
\end{tikzpicture}
```



```
\begin{tikzpicture}
  [column 1/.style={anchor=base west},
  column 2/.style={anchor=base east},
  column 3/.style={anchor=base}]
\matrix
  {
  \node {123}; & \node{456}; & \node {789}; \\
  \node {12}; & \node{45}; & \node {78}; \\
  \node {1}; & \node{4}; & \node {7}; \\
  };
\end{tikzpicture}
```

有的矩阵的各个元素具有极其类似的代码，各元素的开头和结尾就都是一样的，如果为所有元素设置相同的开头和结尾就比较便利，这要用以下两个 key，它们针对非空元素：

/tikz/execute at begin cell=⟨*code*⟩ (no default)

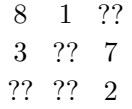
⟨*code*⟩ 会在所有非空元素的开头处被执行。

/tikz/execute at end cell=⟨*code*⟩ (no default)

⟨*code*⟩ 会在所有非空元素的结尾处被执行。

/tikz/execute at empty cell=⟨*code*⟩ (no default)

⟨*code*⟩ 会在所有空元素处被执行，即用 ⟨*code*⟩ 填补空元素。



```
\begin{tikzpicture}
  [matrix of nodes/.style={
   execute at begin cell=\node\bgroup,
   execute at end cell=\egroup;,%
   execute at empty cell=\node{??};%
  }]
  \matrix [matrix of nodes]
  {
    8 & 1 &   \\
    3 &   & 7 \\
      &   & 2 \\
  };
\end{tikzpicture}
```

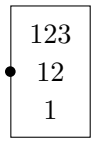
47.4 矩阵的位置

一个 node 有自己的坐标系统，其中有各种位置，这里只涉及罗盘位置（north，east 等）和角度位置，不涉及平移位置（above，left 等）。

关于矩阵的 anchor 有两种：第一，矩阵是 node，它有自己的各种 anchor；第二，如果矩阵的某个元素图形中含有 node，则这个 node 有自己的各种 anchor；这两种 anchor 都可以用来调整矩阵的位置。

/tikz/matrix anchor=⟨*anchor or node.anchor*⟩ (no default)

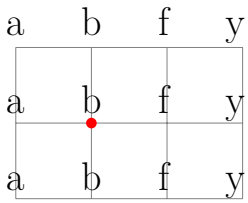
这里的 ⟨*anchor or node.anchor*⟩ 可以是矩阵自己的 anchor 位置，也可以是某个元素图形中的 node 的 anchor 位置。这个选项将 ⟨*anchor or node.anchor*⟩ 位置放在矩阵的锚定点上。矩阵的锚定点就是选项 at 指定的位置，或者其它类似的定位形式确定的位置。




```
\tikz{
  \matrix [draw,matrix anchor=west] at (0,0)
  {
    \node {123}; \\
    \node {12}; \\
    \node {1}; \\
  };
  \fill (0,0) circle (2pt);
}
```

/tikz/anchor=⟨*anchor*⟩ (no default)

当这个 key 用作 matrix 的选项时，这个选项只是针对矩阵内的、各个元素图形中的 node，此时将各个 node 的 ⟨*anchor*⟩ 位置放在各自的锚定点上，其锚定点默认为各元素图形坐标系的原点。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \matrix[matrix anchor=inner node.south,anchor=base,
    row sep=3mm,column sep=5mm,font=\Large] at (1,1)
  {
    \node {a}; & \node {b};              & \node {f}; & \node {y}; \\
    \node {a}; & \node(inner node) {b}; & \node {f}; & \node {y}; \\
    \node {a}; & \node {b};              & \node {f}; & \node {y}; \\
  };
  \fill [red](inner node.south) circle (2pt);
\end{tikzpicture}
```

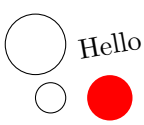
上面例子中矩阵的锚定点是 (1,1)，选项 matrix anchor=inner node.south 将第二行、第二列元素的 inner node.south 位置与 (1,1) 重合，这在整体上决定了矩阵的位置。选项 anchor=base 将各个元素图形的 base 位置与该元素图形坐标系的原点重合，这决定了元素的对齐方式。

47.5 自定义分列符

在构造矩阵时，Ti*k*Z 用 & 作为分列符，而 PGF 使用命令 \pgfmatrixnextcell 来分隔左右相邻的两个元素。由于在 LaTeX 的 {tabular} 环境中也是用 & 作为分列符的，所以，如果在 {tabular} 环境中使用 Ti*k*Z 矩阵，将 & 作为分列符会导致歧义。此时，可以用下面的选项自定义 Ti*k*Z 矩阵的分列符：

/tikz/ampersand replacement=⟨*macro name or empty*⟩ (no default)

如果这个选项值是个宏，那么这个宏就等价于命令 \pgfmatrixnextcell，此时就不再把 & 作为矩阵的分列符。




```
\tikz \matrix [ampersand replacement=\spc]
  {
    \draw (0,0) circle(4mm);   \spc \node[rotate=10]{Hello}; \\
    \draw (0.2,0) circle(2mm); \spc \fill[red] (0,0) circle(3mm); \\
  };
```

47.6 Examples

关于矩阵的其它内容参考 matrix 库。

第四十八章　函数绘图

如果你想比较简便地绘制科技方面的图形，请参考 PGFPLOTS 和本手册的第六部分 Data Visualization.

48.1　Overview

对于绘制十分复杂、精细的函数图形来说，Ti*k*Z 不如专业的数学软件（例如 gnuplot，mathematica），但在 TEX 中使用 Ti*k*Z 绘图的优势是它与 TEX 的兼容性。

用 Ti*k*Z 绘图的方式主要有 3 种：

1. 使用 plot 路径操作。
2. 使用 datavisualization 路径命令。
3. 使用 PGFPLOTS 宏包。

48.2　plot 路径操作

`plot` 操作绘制的路径可以作为主路径的一个子路径。`plot` 操作的句法有多个版本。

`\path...--plot`⟨*further arguments*⟩`...;`

符号 `--` 把当前路径与 `plot` 操作创建的路径用 line-to 方式连接起来。

`\path...plot`⟨*further arguments*⟩`...;`

用 move-to 方式把当前路径与 `plot` 操作创建的路径联系起来。

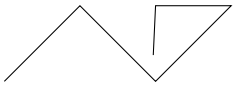
以“line-to”的联系方式为例，`plot` 操作的句法可以是：

1. `--plot[`⟨*local options*⟩`]coordinates{`⟨*coordinate 1*⟩⟨*coordinate 2*⟩`...`⟨*coordinate n*⟩`}`
2. `--plot[`⟨*local options*⟩`]file{`⟨*filename*⟩`}`
3. `--plot[`⟨*local options*⟩`]`⟨*coordinate expression*⟩
4. `--plot[`⟨*local options*⟩`]function{`⟨*gnuplot formula*⟩`}`

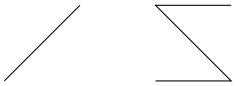
注意以上句式中的 ⟨*local options*⟩ 可以是专门针对 `plot` 操作设计的选项，例如 `smooth`，`variable`，`domain`，`mark` 等，也可以是变换选项，这些选项的作用范围也仅限于其所从属的当前 `plot` 操作，不影响其它 `plot` 操作。而其它的选项，例如 `draw`，`fill`，`color=red` 等针对整个路径的选项，用在这里是无效的，因为 `plot` 操作创建的是当前路径的一段“子路径”。

48.3 连点成线

可以用 plot 操作将多个点用直线段或有一定弯曲状态的曲线连接起来。



```
\tikz \draw plot coordinates {(0,0) (1,1) (2,0) (3,1) (2,1) (10:2cm)};
```



```
\tikz \draw (0,0) -- (1,1) plot coordinates {(2,0) (3,0)}
  --plot coordinates {(2,1)(3,1)};
```

48.4 从外部文件中读取数据绘图

用语句--plot[⟨*local options*⟩]file{⟨*filename*⟩} 从外部文件中读取数据绘图。

目前限于 Ti*k*Z 的读取能力，被读取的外部文件需要参照以下规则编辑：一行可以是空行；如果一行以 # 或 % 开头，则认为该行是空行，所以文件的注释内容可以用这两个符号开头；非空行必须以两个数字开头，两个数字之间用空格分隔；非空行的第二个数字之后可以跟随文字，但除了字母“o”和“u”，其它文字都会被忽略；每一行的两个数字都看作一个坐标点；如果某一行以字母 o 开头，或者在第一个（或第二个）数字之后是字母 o，则该行被看作是 outlier 点，它的作用类似函数的“间断点”，路径在此被截断，开始一段新的子路径；如果某一行以字母 u 开头，或者在第一个（或第二个）数字之后是字母 u，则该行被看作是 undefined 点，路径也会在此被截断，开始一段新的子路径。

```
\tikz \draw plot[mark=x,smooth] file {pgfmanual-sine.table};
```

文件《pgfmanual-sine.table》的样子如下：

```
#Curve 0, 20 points
#x y type
0.00000 0.00000  i
0.52632 0.50235  i
1.05263 0.86873  i
1.57895 0.99997  i
2.10526 0.86054  i
2.63158 0.48819  i
3.15789 -0.01630  i
3.68421 -0.51638  i
4.21053 -0.87669  i
4.73684 -0.99970  i
5.26316 -0.85212  i
5.78947 -0.47390  i
6.31579 0.03260  i
6.84211 0.53027  i
7.36842 0.88441  i
7.89474 0.99917  i
8.42105 0.84348  i
8.94737 0.45948  i
9.47368 -0.04889  i
10.00000 -0.54402  i
```

这个数据文件是用 gnuplot 生成的：

```
set table "../plots/pgfmanual-sine.table"
set format "%.5f"
set samples 20
```

```
plot [x=0:10] sin(x)
```

48.5 用函数表达式绘图

用句式--plot[⟨*local options*⟩](⟨*coordinate expression*⟩)，其中的 ⟨*coordinate expression*⟩ 是用圆括号括起来的坐标形式；如果圆括号里有 2 个表达式（之间由逗号分隔），第一个表达式的计算结果是横标，第二个表达式的计算结果是纵标；如果圆括号里有 3 个表达式，其意义也是类似的。如果某个表达式中有圆括号，则该表达式要用花括号括起来。表达式中的变量要使用宏的形式，默认变量是 \x，可以用选项 variable 设置其它的变量名。

用 plot 绘制函数图的大概过程是：先确定定义域 (保存在内部宏 \tikz@plot@domain 中)，自变量样本点列表 (保存在内部宏 \tikz@plot@samplesat 中)，自变量 (保存在内部宏 \tikz@plot@var 中，默认是 \x)；然后执行 \tikz@plot@expression, 这个命令的定义是：

```
\def\tikz@plot@expression(#1){%
  \edef\tikz@plot@data{\noexpand\pgfplotfunction{\expandafter\noexpand\tikz@plot@var}{
  ↪  \tikz@plot@samplesat}}%
  \expandafter\def\expandafter\tikz@plot@data\expandafter{\tikz@plot@data{
  ↪  \tikz@scan@one@point\pgfutil@firstofone(#1)}}%
  \tikz@@@plot%
}%
```

执行 \tikz@@@plot 的过程中将 \tikz@plot@data 展开，由命令 \pgfplotfunction→P. 491 得到一个图流：

```
\pgfplotfunction{⟨ 展开的 \tikz@plot@var⟩}{⟨ 展开的 \tikz@plot@samplesat⟩}{
↪  \tikz@scan@one@point\pgfutil@firstofone(⟨ 坐标表达式⟩)}
```

按命令 \pgfplotfunction→P. 491 的定义，这导致

```
\pgfplotstreamstart%
\foreach \⟨var⟩ in{⟨ 展开的 \tikz@plot@samplesat⟩}%
{%
\pgf@process{\tikz@scan@one@point\pgfutil@firstofone(⟨ 坐标表达式⟩)}%
\edef\pgf@marshal{\noexpand\pgfplotstreampoint{\noexpand\pgfqpoint{\the\pgf@x}{\the
↪  \pgf@y}}}%
\pgf@marshal%
}
\pgfplotstreamend%
```

默认下,图柄 \tikz@plot@handler 等于 \pgfplothandlerlineto; 如果使用选项 /tikz/smooth→P. 900, 那么这个图柄就等于 \pgfplothandlercurveto→P. 657.

/tikz/variable=⟨*macro*⟩ (no default, initially \x)

这个选项设置 ⟨*coordinate expression*⟩ 中使用的变量名称，变量要使用宏的形式。

```
\tikzoption{variable}{\def\tikz@plot@var{#1}}%
```

/tikz/samples=⟨*number*⟩ (no default, initially 25)

设置变量 ⟨*macro*⟩ 的采样点数目。样本点数目保存在内部宏 \tikz@plot@samples 中。

```
\tikzoption{samples}{\pgfmathsetmacro\tikz@plot@samples{max(2,#1)}
↪  \expandafter\tikz@plot@samples@recalc\tikz@plot@domain\relax}%
```

执行本选项导致执行命令 \tikz@plot@samples@recalc.

\tikz@plot@samples@recalc⟨*a*⟩:⟨*b*⟩\relax

此命令会计算一个用逗号分隔的列表。

```
\def\tikz@plot@samples@recalc#1:#2\relax{%
  \begingroup
  \pgfmathparse{#1}%
  \let\tikz@temp@start=\pgfmathresult%
  \pgfmathparse{#2}%
  \let\tikz@temp@end=\pgfmathresult%
  \pgfmathsetmacro\tikz@temp@diff{(\tikz@temp@end-\tikz@temp@start)/(
  ↪  \tikz@plot@samples-1)}%
  %
  % this particular item is for backwards compatibility.
  % Pgfplots <= 1.8 called 'samples' in a context where the 'fpu' was
  % active... and I fear there is no simple solution to replace the
  % new \ifdim below. Sorry.
  \pgfkeys{/pgf/fpu/output format/fixed/.try}%
  %
  \pgfmathsetmacro\tikz@temp@diff@abs{abs(\tikz@temp@diff)}%
  \ifdim\tikz@temp@diff@abs pt<0.0001pt\relax%
    \edef\tikz@plot@samplesat{\tikz@temp@start,\tikz@temp@end}%
  \else%
    \pgfmathparse{\tikz@temp@start+\tikz@temp@diff}%
    \edef\tikz@plot@samplesat{\tikz@temp@start,\pgfmathresult,...,
    ↪  \tikz@temp@end}%
  \fi%
  \pgfmath@smuggleone\tikz@plot@samplesat
  \endgroup
}%
```

假设选项 `domain=`a`:`b 规定了区间 $[a,b]$, 选项 `samples=`$n+1$ 规定的样本点数目是 $n+1$, 那么命令 `\tikz@plot@samples@recalc` 计算 $\frac{b-a}{n}$.

如果 $\left|\frac{b-a}{n}\right| < 0.0001$, 那么定义逗号分隔的列表 `\tikz@plot@samplesat` 为

```
\edef\tikz@plot@samplesat{a,b}
```

如果 $\left|\frac{b-a}{n}\right| \geqslant 0.0001$, 那么计算 $s = a + \frac{b-a}{n}$, 定义列表为

```
\edef\tikz@plot@samplesat{a,s,...,b}
```

列表 `\tikz@plot@samplesat` 将用作 `\pgfplotfunction` →P. 491 的参数，也就是用作 `\foreach` 命令中的列表，作为函数自变量的取值点。由于 `\foreach` 命令会对含有省略号的列表做特别地处理，可能会导致定义域的端点 b 被忽略（参考 `\foreach` 部分），例如

```
\draw plot [samples=4.5, domain=1:10] (\x,\x);
等价于
\draw plot [samples at={1,3,...,10}] (\x,\x);
等价于
\draw plot [samples at={1,3,...,9}] (\x,\x);
```

`/tikz/domain=`⟨*start*⟩`:`⟨*end*⟩ (no default, initially `-5:5`)

设置变量 ⟨*macro*⟩ 的变化范围，即样本点 `samples` 的采样范围。

本选项的定义是：

```
\tikzoption{domain}{\edef\tikz@plot@domain{#1}
↪  \expandafter\tikz@plot@samples@recalc\tikz@plot@domain\relax}%
```

执行本选项导致执行命令 `\tikz@plot@samples@recalc`.

`/tikz/samples at=`⟨*sample list*⟩ (no default)

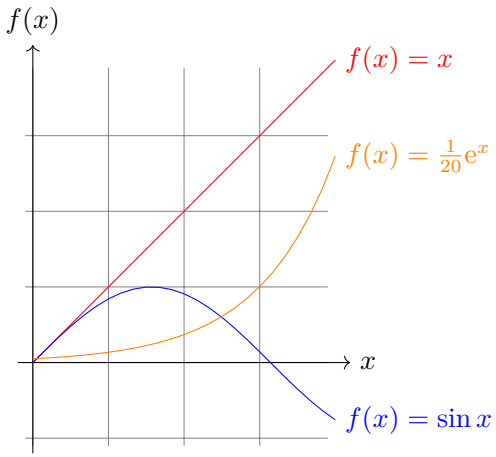
直接指定变量 ⟨*macro*⟩ 的采样点。⟨*sample list*⟩ 是个列表，此列表采用 `\foreach` 句法的列表形式。

本选项直接定义列表 \tikz@plot@samplesat

```
\tikzoption{samples at}{\def\tikz@plot@samplesat{#1}}%
```

有以下默认值：

```
\def\tikz@plot@samples{25}%
\def\tikz@plot@domain{-5:5}%
\def\tikz@plot@var{\x}%
\def\tikz@plot@samplesat{-5,-4.5833333,...,5}%
```



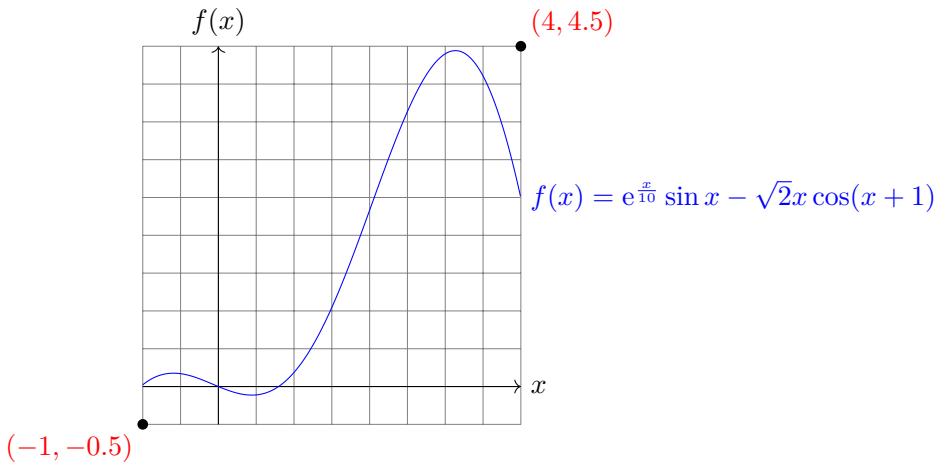

```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);
  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};
  \draw[color=red]    plot (\x,\x)             node[right] {$f(x) =x$};
  \draw[color=blue]   plot (\x,{sin(\x r)})    node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot (\x,{0.05*exp(\x)}) node[right] {$f(x) = \frac{1}{20} \mathrm e^x$};
\end{tikzpicture}
```

上面例子中，符号 \x r 是将角度 \x 转换为弧度。

```
\tikz \draw[domain=0:360,smooth,variable=\t]
  plot ({sin(\t)},\t/360,{cos(\t)});
```

上面的例子绘制 3 维曲线。

下面例子中的函数稍微复杂一些：

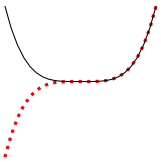



```
\begin{tikzpicture}[samples=500,domain=-1:4]
  \draw [help lines,step=0.5cm]
    (-1,-0.5)node[below left,red]{$(-1,-0.5)$} grid (4,4.5)node[above right,red]{$(4,4.5)$};
```

```
  \draw[->] (-1,0) -- (4,0) node[right] {$x$};
  \draw[->] (0,-0.5) -- (0,4.5) node[above] {$f(x)$};
  \draw[color=blue] plot (\x,{exp(\x / 10) * (sin(\x r)) - (2 ^ (1/2)) * \x * (cos(\x r + 1))})
    node[right] {$f(x) = \mathrm{e}^{\frac x {10}} \sin x - \sqrt{2} x \cos (x+1)$};
  \fill (-1,-0.5) circle [radius=2pt] (4,4.5) circle [radius=2pt];
\end{tikzpicture}
```

48.5.1 注意

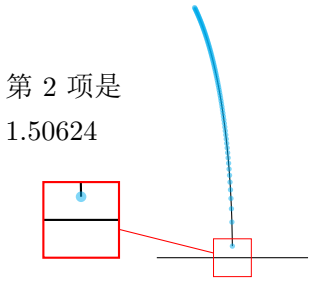
当 ⟨*coordinate expression*⟩ 中有变量的幂运算时，例如 x^4, 注意区别 `(\x)^4` 与 `\x^4`，比较下图中的两个曲线：



```
\tikz{
  \draw [red,very thick,dotted] plot [domain=-1:1] (\x,\x^4);
  \draw plot [domain=-1:1] (\x,{(\x)^4});
}
```

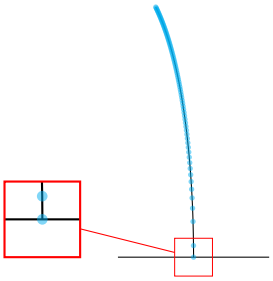
上图中的实线由 `plot (\x,{(\x)^4})` 画出，是 $x^4(x \in [-1,1])$ 的图象；而点线由 `plot (\x,\x^4)` 画出，是 $f(x) = \begin{cases} x^4, x \in [0,1], \\ -x^4, x \in [-1,0] \end{cases}$ 的图象。

下面的例子中，`plot`, 即 `\foreach` 忽略了绘图区间的右端点：




```
\begin{tikzpicture}
  [spy using outlines={lens={scale=2}, size=1cm, connect spies}]
  \draw(1,0)--(3,0);
  \draw [domain=1.5:2,samples=81,smooth] plot (\x,{sqrt(16-(\x)^4)});
  \pgfmathparse{1.5+(2-1.5)/80}
  \let\mytestresult\pgfmathresult
  \node [text width=2cm] at(0,2) {第 2 项是\\ $\mytestresult$};
  \foreach \x in {1.5,\mytestresult,...,2}
  {\fill[fill=cyan,fill opacity=0.5]
   (\x,{sqrt(16-(\x)^4)}) circle [radius=1pt];}
  \spy [red] on (2,0) in node at (0,0.5);
\end{tikzpicture}
```

上面例子中，定义域的右端点 $x = 2$ 被忽略了。函数 $f(x) = \sqrt{16 - x^4}$ 的导数在 $x \to 2\_$ 时趋于无穷，较小的 δ 会导致 $f(2+\delta) - f(2)$ 有较大的值。在这种情况下，由于 TEX 本身的计算精度有限，即使增加样本点的数目 `samples=`⟨大数值⟩ 也未必有改观 (有一定的概率)。此时可以考虑使用 `plot coordinates` 或者选项 `samples at`.



```
\begin{tikzpicture}
  [spy using outlines={lens={scale=2}, size=1cm, connect spies}]
  \draw(1,0)--(3,0);
  \pgfmathparse{1.5+(2-1.5)/80}
  \let\mytestresult\pgfmathresult
  \draw [samples at={1.5,\mytestresult,...,2,2},smooth]
    plot (\x,{sqrt(16-(\x)^4)});
  \foreach \x in {1.5,\mytestresult,...,2,2}
  {\fill[fill=cyan,fill opacity=0.5]
   (\x,{sqrt(16-(\x)^4)}) circle [radius=1pt];}
  \spy [red] on (2,0) in node at (0,0.5);
\end{tikzpicture}
```

48.6 调用 gnuplot 绘制函数图形

首先确保安装了 gnuplot. 参考 `\pgfplotgnuplot` → P. 492.

调用 gnuplot 绘制函数图形的句法是：

`plot[`⟨*local options*⟩`]function`⟨*gnuplot formula*⟩

其中的 ⟨*gnuplot formula*⟩ 是用 gnuplot 句法构造的函数表达式。“`plot function`”会调用命令 `\pgfplotgnuplot`→P. 492 来工作，这个命令进一步调用 gnuplot, 这种调用需要使用 shell escape 选项来编译。

假设当下编辑的 `.tex` 文件中的图形代码里有语句 `plot[id=`⟨*id*⟩`] function{x*sin(x)}`，如果不使用 shell escape 选项来编译，一般情况下也能通过编译，可以得到名称为 ⟨*prefix*⟩`.`⟨*id*⟩`.gnuplot` 的文件，也能得到图形，但所生成的图形中没有函数 $x\sin x$ 的图像，这说明 gnuplot 并没有被调用。在命令行编译 `.tex` 文件并调用 gnuplot，需要：(a) 在命令行进入 `.tex` 文件所在的文件目录位置；(b) 执行类似 `pfdlatex --shell-escape` ⟨*tex* 文件名⟩`.tex` 或 `xelatex --shell-escape` ⟨*tex* 文件名⟩`.tex` 这样的命令。

当 Ti*k*Z 首次遇到 `plot[id=`⟨*id*⟩`] function{x*sin(x)}` 这样的语句时，会创建一个名称为 ⟨*prefix*⟩`.`⟨*id*⟩`.gnuplot` 的文件，其中默认 ⟨*prefix*⟩ 是宏 `\jobname` 所保存的值，`\jobname` 的默认值是当下正在编辑的 `.tex` 文件的名称。如果不给出 ⟨*id*⟩ 就将其留空，这也是可接受的，但最好给出 ⟨*id*⟩.

然后，Ti*k*Z 会向文件 ⟨*prefix*⟩`.`⟨*id*⟩`.gnuplot` 中写入一些内部代码，这些代码以 `plot x*sin(x)` 结尾。文件 ⟨*prefix*⟩`.`⟨*id*⟩`.gnuplot` 会被 gnuplot 处理，得到另一个文件 ⟨*prefix*⟩`.`⟨*id*⟩`.table`, 此文件中有绘图所需的坐标数据，然后像使用语句 `plot file{`⟨*prefix*⟩`.`⟨*id*⟩`.table}` 那样绘图。

命令 `\pgfplotgnuplot` 会执行

```
\immediate\write18{gunplot \pgf@plotgnuplotfile}
```

其中的 `\pgf@plotgnuplotfile` 就是文件名 ⟨*prefix*⟩`.`⟨*id*⟩`.gnuplot`，这样就得到文件 ⟨*prefix*⟩`.`⟨*id*⟩`.table`。

（参考《pgfmoduleplot.code.tex》中 `\pgfplotgnuplot` 的定义，以及《pgfutil-common.tex》中 `\pgfutil@shellescap` 的定义）

然后执行

```
\pgfplotxyfile{\pgf@plottablefile}%
```

其中的 `\pgf@plottablefile` 就是文件 ⟨*prefix*⟩`.`⟨*id*⟩`.table`, 命令 `\pgfplotxyfile`→P. 490 绘制图形。

例如，若能顺利编译下面的代码：

```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);
  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};
  \draw[color=red]    plot[id=x]   function{x} node[right] {$f(x) =x$};
  \draw[color=blue]   plot[id=sin] function{sin(x)} node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot[id=exp] function{0.05*exp(x)} node[right] {$f(x) = \frac{1}
  ↪  {20} \mathrm e^x$};
\end{tikzpicture}
```

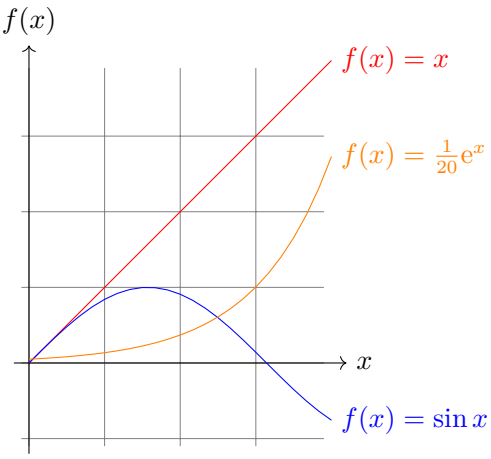
会得到文件

⟨*prefix*⟩`.x.gnuplot`, ⟨*prefix*⟩`.x.table`

⟨*prefix*⟩`.sin.gnuplot`, ⟨*prefix*⟩`.sin.table`

⟨*prefix*⟩`.exp.gnuplot`, ⟨*prefix*⟩`.exp.table`

还生成下面的图形：



为了能顺利调用 gnuplot 绘制函数图形，必须具备以下两个条件：

1. 对 TEX 来说，gnuplot 是外部程序。必须允许 TEX 调用外部程序，为此需要使用 `--shell-escape` 或 `enable-write18` 选项来编译。这样编译后就会得到图形以及文件：⟨*prefix*⟩.⟨*id*⟩`.gnuplot` 和 ⟨*prefix*⟩.⟨*id*⟩`.table`.
2. 必须提前安装好 gnuplot，并且 TEX 在编译文件时能找到并调用 gnuplot.

当 Ti*k*Z 第二次遇到语句 `plot[id=`⟨*id*⟩`] function{x*sin(x)}` 时，如果文件 ⟨*prefix*⟩.⟨*id*⟩`.gnuplot` 和 ⟨*prefix*⟩.⟨*id*⟩`.table` 都已经存在，那么文件 ⟨*prefix*⟩.⟨*id*⟩`.table` 就会被立即用于绘图，而不会再次调用 gnuplot，此时不必再使用 `--shell-escape` 选项来编译。

这样的机制有以下好处：

1. 如果你与朋友都需要调用 gnuplot 绘制同一个图形，但你朋友没有安装 gnuplot，那么你只需要把文件 ⟨*prefix*⟩.⟨*id*⟩`.gnuplot` 和 ⟨*prefix*⟩.⟨*id*⟩`.table` 发送给你朋友，你朋友可以不调用 gnuplot 就能得到想要的图形。
2. 若 `\write18` 特性被关闭,则 TEX 不能调用 gnuplot,此时编译 `.tex` 文件得到文件 ⟨*prefix*⟩.⟨*id*⟩`.gnuplot`, 然后再用 gnuplot 处理此文件得到文件 ⟨*prefix*⟩.⟨*id*⟩`.table`, 这样这两个文件就都有了，然后 Ti*k*Z 可以处理这两个文件得到最后的图形。
3. 如果你修改了语句中的函数，那么 Ti*k*Z 会自动重新产生新的文件 ⟨*prefix*⟩.⟨*id*⟩`.table`.
4. 如果不给出 ⟨*id*⟩，例如，若顺利编译下面的代码

```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);
  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};
  \draw[color=red]    plot function{x} node[right] {$f(x) =x$};
  \draw[color=blue]   plot function{sin(x)} node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot function{0.05*exp(x)} node[right] {$f(x) = \frac{1}{20}
  ↪    \mathrm e^x$};
\end{tikzpicture}
```

会得到文件 ⟨*prefix*⟩`.pgf-plot.gnuplot` 和 ⟨*prefix*⟩`.pgf-plot.table`.
文件 ⟨*prefix*⟩`.pgf-plot.gnuplot` 的内容如下：

```
set table "wenti.pgf-plot.table"; set format "%.5f"
set samples 25; plot [x=0:4] 0.05*exp(x)
```

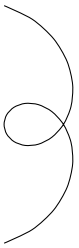
其中只涉及函数 `0.05*exp(x)`，文件 ⟨*prefix*⟩`.pgf-plot.table` 的内容也只是函数 `0.05*exp(x)` 的数据点。不过最后得到的图形还是前面的图形。这表明，程序逐个处理语句 `plot function{`⟨*gnuplot formula*⟩`}`，每处理一个绘图语句就向坐标系中添加一个函数图形，因此即使不给出 ⟨*id*⟩ 也能得到预

期的图形，但是函数的 .gnuplot 文件和 .table 文件却可能保存不下来。

操作 plot 调用 gnuplot 绘制函数图形时，可以使用选项 samples，domain 来设置样本点数目和样本点的取样范围，另外还可以使用以下选项：

/tikz/parametric=⟨*boolean*⟩ (default true)

本选项决定所绘制的图形是否是参数图。如果是，则函数的自变量必须使用 t，横坐标函数和坐标轴函数都以 t 为自变量，观察下面的例子：



```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth]
  plot[parametric,id=parametric-example] function{t*sin(t),t*cos(t)};
```

/tikz/range=⟨*start*⟩:⟨*end*⟩ (no default)

这里 ⟨*start*⟩ 和 ⟨*end*⟩ 都是纵轴上的坐标，图形上的任何一个点的纵坐标都位于 ⟨*start*⟩ 到 ⟨*end*⟩ 之间。

```
\tikz \draw
↪  [scale=0.5,domain=-3.141:3.141, samples=100, smooth, range=-0.8:0.8]
  plot[id=sin-example] function{sin(x)};
```

/tikz/yrange=⟨*start*⟩:⟨*end*⟩ (no default)

等效于 range.

/tikz/xrange=⟨*start*⟩:⟨*end*⟩ (no default)

这里 ⟨*start*⟩ 和 ⟨*end*⟩ 都是横轴上的坐标，图形上的任何一个点的横坐标都位于 ⟨*start*⟩ 到 ⟨*end*⟩ 之间。

/tikz/id=⟨*id*⟩ (no default)

这个选项用于标示所绘制的 gnuplot 函数，见前文。由于 ⟨*id*⟩ 会用于文件名称中，所以 ⟨*id*⟩ 中不能含有“*”或“$”等特殊符号，最好也不要有空格。

/tikz/prefix=⟨*prefix*⟩ (no default)

如前文所述，⟨*prefix*⟩ 用于文件名称中，它的默认值是 \jobname 的值。如果你要绘制多个 gnuplot 函数图像，最好把 ⟨*prefix*⟩ 设为其它，例如 plots/，并把所有的图形放到一个目录位置中。

/tikz/raw gnuplot (no value)

这个选项直接把 ⟨*gnuplot formula*⟩ 传递给 gnuplot，同时无需使用操作 plot 的各种选项（如 samples，domain 等）来设置图形。gnuplot 函数的样本点、定义域等内容可以在 ⟨*gnuplot formula*⟩ 中，使用 gnuplot 的句法做设置。例如

```
\tikz \draw plot[raw gnuplot,id=raw-example] function{set samples 25;
↪  plot [0:pi][0:0.8] sin(x)};
```

如果需要用 gnuplot 的语法作某些复杂的事情，那么这个选项比较有用。

/tikz/every plot (style, initially empty)

这是个样式，针对每个 plot 操作，例如

```
\tikzset{every plot/.style={prefix=plots/}}
```

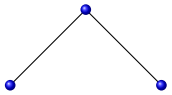
48.7 给 plot 路径上的样本点加标记

点标记（mark）放在样本点上来标识样本点。与 node 类似，当路径被使用（`draw/fill/shade`）后，点标记才会添加到路径的样本点上。点标记的类型、外观可以用选项来设置。

/tikz/mark=⟨*mark mnemonic*⟩ (no default)

这个选项选定某种类型的点标记。默认 ⟨*mark mnemonic*⟩ 有 3 种选择：`*`，`+`，`x`，分别代表圆点，加号，叉号。载入 plotmarks 库后可以使用更多点标记类型。使用命令 `\pgfdeclareplotmark` → P. 663 可以自定义一种类型的点标记。

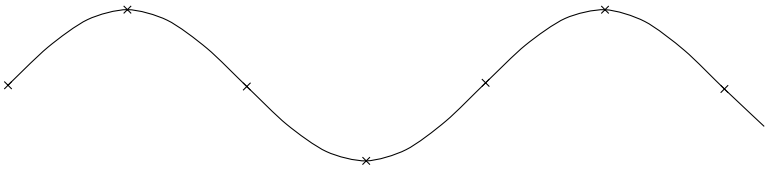
有个名称为 `ball` 的特殊点标记，它只能用在 TikZ 中，选项 `ball color` 可以设置 `ball` 的颜色。如果样本点的数量很多且用 PostScript 来渲染，那最好不要使用 `ball`，否则可能会比较费时。



```
\begin{tikzpicture}
  \draw plot [mark=ball,ball color=blue]
    coordinates {(0,0)(1,1)(2,0)};
\end{tikzpicture}
```

/tikz/mark repeat=⟨*r*⟩ (no default)

这个选项会使得第 1，⟨*r*⟩+1，2*⟨*r*⟩+1，3*⟨*r*⟩+1... 个样本点被标记。



```
\tikz \draw plot[mark=x,mark repeat=3,smooth] coordinates
  {
    (0.00000, 0.00000)  (0.52632, 0.50235)  (1.05263, 0.86873)  (1.57895, 0.99997)
    (2.10526, 0.86054)  (2.63158, 0.48819)  (3.15789, -0.01630) (3.68421, -0.51638)
    (4.21053, -0.87669) (4.73684, -0.99970) (5.26316, -0.85212) (5.78947, -0.47390)
    (6.31579, 0.03260)  (6.84211, 0.53027)  (7.36842, 0.88441)  (7.89474, 0.99917)
    (8.42105, 0.84348)  (8.94737, 0.45948)  (9.47368, -0.04889) (10.00000, -0.54402)
  };
```

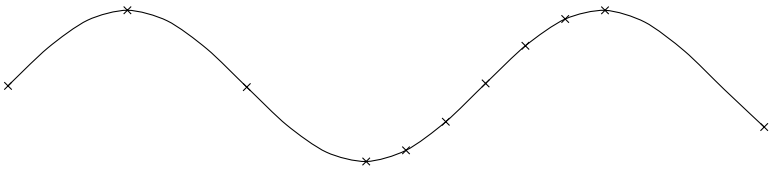
上面的图形中有 20 个样本点，被标记的是第 1, 4, 7, 10, 13, 16, 19 个点，共七个点。

/tikz/mark phase=⟨*p*⟩ (no default)

如果要使用这个选项，则应当与 `mark repeat=`⟨*r*⟩ 配合使用。这个选项使得 TikZ 先标记第 ⟨*p*⟩ 个点，然后标记第 ⟨*p*⟩+⟨*r*⟩ 个点，然后标记第 ⟨*p*⟩+2*⟨*r*⟩ 个点……

/tikz/mark indices=⟨*list*⟩ (no default)

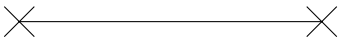
这里 ⟨*list*⟩ 是个正整数列表，是样本点的序号，序号对应的样本点会被标记。⟨*list*⟩ 中可以使用省略号，按照 `\foreach` 语句的规则来解释省略号。



```
\tikz \draw plot[mark=x,mark indices={1,4,...,10,11,12,...,16,20},smooth] coordinates
  {
    (0.00000, 0.00000)  (0.52632, 0.50235)  (1.05263, 0.86873)  (1.57895, 0.99997)
    (2.10526, 0.86054)  (2.63158, 0.48819)  (3.15789, -0.01630) (3.68421, -0.51638)
    (4.21053, -0.87669) (4.73684, -0.99970) (5.26316, -0.85212) (5.78947, -0.47390)
    (6.31579, 0.03260)  (6.84211, 0.53027)  (7.36842, 0.88441)  (7.89474, 0.99917)
    (8.42105, 0.84348)  (8.94737, 0.45948)  (9.47368, -0.04889) (10.00000, -0.54402)
  };
```

/tikz/mark size=⟨*dimension*⟩ (no default)

这个选项设置点标记的“半径”，注意 ⟨*dimension*⟩ 带有长度单位。也可以用 scale=⟨*factor*⟩ 来调节标记尺寸 (⟨*factor*⟩ 是目标尺寸与默认尺寸的比值)，相对地说 mark size 要快一些。如果 ⟨*dimension*⟩ 是负值尺寸则把它当作正值尺寸看待。



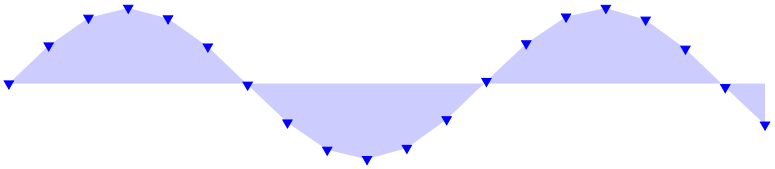
```
\tikz \draw plot[mark=x,scale=4,smooth] coordinates {(0,0) (1,0)};
```

/tikz/every mark (style, no value)

这个 style 会加在每个点标记的开头。

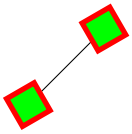
/tikz/mark options=⟨*options*⟩ (no default)

用 ⟨*options*⟩ 重新定义样式 every mark。当用 TikZ 的一般选项来设置点标记的外观样式时，这些选项要放入 ⟨*options*⟩ 中。



```
\tikz \fill[fill=blue!20]
  plot[mark=triangle*,mark options={color=blue,rotate=180}] coordinates
  {
    (0.00000, 0.00000)  (0.52632, 0.50235)  (1.05263, 0.86873)  (1.57895, 0.99997)
    (2.10526, 0.86054)  (2.63158, 0.48819)  (3.15789, -0.01630) (3.68421, -0.51638)
    (4.21053, -0.87669) (4.73684, -0.99970) (5.26316, -0.85212) (5.78947, -0.47390)
    (6.31579, 0.03260)  (6.84211, 0.53027)  (7.36842, 0.88441)  (7.89474, 0.99917)
    (8.42105, 0.84348)  (8.94737, 0.45948)  (9.47368, -0.04889) (10.00000, -0.54402)
  } |- (0,0);
```

上面例子中，纵横线操作“|-”把点 (10.00000, -0.54402) 与点 (0,0) 连接起来了。



```
\begin{tikzpicture}
  \draw plot[mark=square*,mark size=6pt,
      mark options={line width=2pt,draw=red,fill=green,rotate=30,}]
    coordinates{(0,0)(1,1)};
\end{tikzpicture}
```

/tikz/no marks (style, no value)

等于 mark=none, 取消点标记。

/tikz/no markers (style, no value)

等于 mark=none, 取消点标记。

48.8 直线、曲线、柱状图、条形图等

如果没有特别的设置，plot 操作会用直线段来连接样本点。使用下面的选项可以让 plot 对样本点执行其它操作。

/tikz/sharp plot (no value)

这个选项指示 plot 操作用直线段来连接样本点，这是默认的做法。

/tikz/smooth (no value)

这个选项指示 plot 操作用曲线段来连接样本点，并让曲线在样本点处（转角处）光滑。注意这个选项作用并不够智能，其效果可能不如意。连接点处的转弯角度越小（最好小于 30°），并且各点的间距越是均匀，则曲线效果越好。

在《tikz.code》中有定义：

```
\tikzoption{smooth}[]{\let\tikz@plot@handler=\pgfplothandlercurveto}%
......
\let\tikz@plot@handler=\pgfplothandlerlineto
```

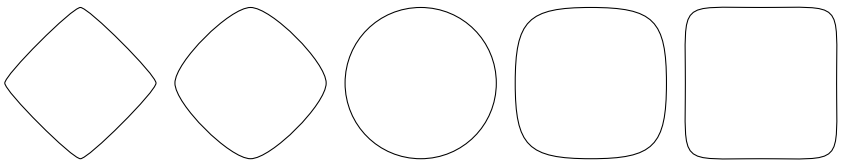
选项 smooth 使得 plot 利用图柄 \pgfplothandlercurveto 来绘图，参考 plothandlers 库。

/tikz/tension=⟨*value*⟩ (default 0.5)

本选项配合选项 smooth 使用 (参考 plothandlers 库)。在《tikz.code》中有定义：

```
\tikzoption{tension}{\pgfsetplottension{#1}}%
```

这个选项调节曲线的“张力”，即弯曲状态，数值越大，弯曲愈著。若有 4 个样本点均匀分布于一个圆上且张力值 ⟨*value*⟩ 是 1，则绘制的曲线是个圆。⟨*value*⟩ 的默认值是 0.5.



```
\begin{tikzpicture}[smooth cycle]
  \draw plot[tension=0.2] coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[xshift=2.25cm] plot[tension=0.5] coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[xshift=4.5cm] plot[tension=1] coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[xshift=6.75cm] plot[tension=1.5] coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[xshift=9cm] plot[tension=2] coordinates{(0,0) (1,1) (2,0) (1,-1)};
\end{tikzpicture}
```

/tikz/smooth cycle (no value)

这个选项指示 plot 操作用曲线段来连接样本点，让曲线在样本点处（转角处）光滑，且曲线是封闭的。

```
\tikz[scale=0.5]
  \draw plot[smooth cycle] coordinates{(0,0) (1,0) (2,1) (1,2)}
    plot coordinates{(0,0) (1,0) (2,1) (1,2)} -- cycle;
```

/tikz/only marks (no value)

这个选项指示 plot 操作只标记样本点，不画线。

/tikz/const plot (no value)

```
\tikz\draw plot[const plot] file{pgfmanual-sine.table};
```

/tikz/const plot mark left (no value)

```
\tikz\draw plot[const plot mark left,mark=*] file{pgfmanual-sine.table};
```

/tikz/const plot mark right (no value)

```
\tikz\draw plot[const plot mark right,mark=*] file{pgfmanual-sine.table};
```

/tikz/const plot mark mid (no value)

```
\tikz\draw plot[const plot mark mid,mark=*] file{pgfmanual-sine.table};
```

/tikz/jump mark left (no value)

```
\tikz\draw plot[jump mark left, mark=*] file{pgfmanual-sine.table};
```

/tikz/jump mark right (no value)

```
\tikz\draw plot[jump mark right, mark=*] file{pgfmanual-sine.table};
```

/tikz/jump mark mid (no value)

```
\tikz\draw plot[jump mark mid, mark=*] file{pgfmanual-sine.table};
```

/tikz/ycomb (no value)

comb 的意思是“梳子，篦子”，梳子的每个“齿”—一条“细棒”—的起点都位于直线 $y=0$ 上。

```
\tikz\draw[ultra thick] plot[ycomb,thin,mark=*] file{pgfmanual-sine.table};
```

```
\begin{tikzpicture}[ycomb]
  \draw[color=red,line width=6pt]
    plot coordinates{(0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9)};
  \draw[color=red!50,line width=4pt,xshift=3pt]
    plot coordinates{(0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5)};
\end{tikzpicture}
```

上面例子表明，影响 comb 外观的是“线宽”和线条颜色，没有填充色。

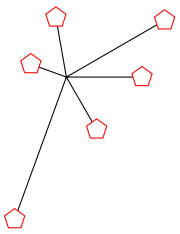
/tikz/xcomb (no value)

各个“细棒”的起点都位于直线 $x=0$ 上。

```
\tikz \draw plot[xcomb,mark=x,mark options={color=red}]
  coordinates{(1,0) (0.8,0.2) (0.6,0.4) (0.2,1)};
```

/tikz/polar comb (no value)

绘制一个放射状的图形，放射中心在 $(0,0)$.



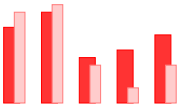
```
\tikz \draw plot[polar comb,
    mark=pentagon*,
    mark options={fill=white,draw=red},
    mark size=4pt]
  coordinates {(0:1cm) (30:1.5cm) (160:.5cm)
    (250:2cm) (-60:.8cm) (100:.8cm)};
```

/tikz/ybar (no value)

各个 bar 的起点都位于直线 $y=0$ 上。

```
\tikz{\draw[draw=orange,fill=blue!60!white] plot[ybar] file{pgfmanual-sine.table};
  \draw plot[mark=*,only marks,mark options={color=red}] file{pgfmanual-sine.table};}
```

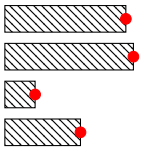
上面例子表明，影响 ybar 外观的有线条颜色和填充色。使用选项 /pgf/bar width →P. 660 和 /pgf/bar shift →P. 661 可以调整“bar”的宽度、位置。



```
\begin{tikzpicture}[ybar]
\draw[color=red,fill=red!80,bar width=6pt]
  plot coordinates{(0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9)};
\draw[color=red!50,fill=red!20,bar width=4pt,bar shift=3pt]
  plot coordinates{(0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5)};
\end{tikzpicture}
```

/tikz/xbar (no value)

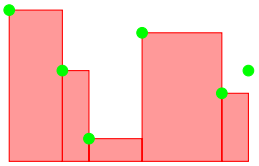
各个 bar 的起点都位于直线 $x = 0$ 上。



```
\tikz {
  \draw[pattern=north west lines] plot[xbar]
    coordinates{(1,0) (0.4,0.5) (1.7,1) (1.6,1.5)};
  \draw[pattern=north west lines] plot[mark=*,only marks,
      mark options={color=red}]
    coordinates{(1,0) (0.4,0.5) (1.7,1) (1.6,1.5)};}
```

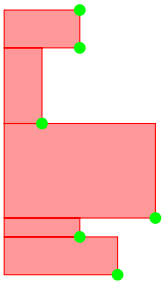
/tikz/ybar interval (no value)

这个选项用样本点构造柱状图。前后相继的两个数据点确定一个竖直柱，设在前的数据点是 (x_i, y_i)，在后的数据点是 (x_{i+1}, y_{i+1})，这两个数据点确定的竖直柱的宽度是 $|x_i - x_{i+1}|$，高度是 y_i.



```
\begin{tikzpicture}[ybar interval,x=10pt]
  \draw[color=red,fill=red!40!white] plot
    coordinates{(0,2) (2,1.2) (3,.3) (5,1.7) (8,.9) (9,.9)};
  \draw plot[mark=*,only marks,mark options={color=green}]
    coordinates{(0,2) (2,1.2) (3,.3) (5,1.7) (8,.9) (9,1.2)};
\end{tikzpicture}
```

/tikz/xbar interval (no value)



```
\begin{tikzpicture}[xbar interval,x=0.5cm,y=0.5cm]
  \draw[color=red,fill=red!40!white] plot
    coordinates {(3,0) (2,1) (4,1.5) (1,4) (2,6) (2,7)};
  \draw plot[mark=*,only marks,mark options={color=green}]
    coordinates {(3,0) (2,1) (4,1.5) (1,4) (2,6) (2,7)};
\end{tikzpicture}
```

对于 ybar, xbar 类型的柱状图，可以用选项 /pgf/bar width →P. 660 和 /pgf/bar shift →P. 661 调整其外观。对于 ybar interval, xbar interval 类型的柱状图，可以用选项 /pgf/bar interval width →P. 661 和 /pgf/bar interval shift →P. 661 调整其外观。

在《tikz.code》中有定义：

```
% Plot options
\tikzoption{smooth}[]{\let\tikz@plot@handler=\pgfplothandlercurveto}%
\tikzoption{smooth cycle}[]{\let\tikz@plot@handler=\pgfplothandlerclosedcurve}%
\tikzoption{sharp plot}[]{\let\tikz@plot@handler\pgfplothandlerlineto}%
\tikzoption{sharp cycle}[]{\let\tikz@plot@handler\pgfplothandlerpolygon}%

\tikzoption{tension}{\pgfsetplottension{#1}}%

\tikzoption{xcomb}[]{\let\tikz@plot@handler=\pgfplothandlerxcomb}%
\tikzoption{ycomb}[]{\let\tikz@plot@handler=\pgfplothandlerycomb}%
\tikzoption{polar comb}[]{\let\tikz@plot@handler=\pgfplothandlerpolarcomb}%
\tikzoption{ybar}[]{\let\tikz@plot@handler=\pgfplothandlerybar}%
\tikzoption{ybar interval}[]{\let\tikz@plot@handler=\pgfplothandlerybarinterval}%
```

```
\tikzoption{xbar interval}[]{\let\tikz@plot@handler=\pgfplothandlerxbarinterval}%
\tikzoption{xbar}[]{\let\tikz@plot@handler=\pgfplothandlerxbar}%
\tikzoption{const plot}[]{\let\tikz@plot@handler=\pgfplothandlerconstantlineto}%
\tikzoption{const plot mark left}[]{\let\tikz@plot@handler=
↪  \pgfplothandlerconstantlineto}%
\tikzoption{const plot mark right}[]{\let\tikz@plot@handler=
↪  \pgfplothandlerconstantlinetomarkright}%
\tikzoption{const plot mark mid}[]{\let\tikz@plot@handler=
↪  \pgfplothandlerconstantlinetomarkmid}%
\tikzoption{jump mark right}[]{\let\tikz@plot@handler=\pgfplothandlerjumpmarkright}%
\tikzoption{jump mark mid}[]{\let\tikz@plot@handler=\pgfplothandlerjumpmarkmid}%
\tikzoption{jump mark left}[]{\let\tikz@plot@handler=\pgfplothandlerjumpmarkleft}%

\tikzoption{raw gnuplot}[true]{\csname tikz@plot@raw@gnuplot#1\endcsname}%
\tikzoption{prefix}{\def\tikz@plot@prefix{#1}}%
\tikzoption{id}{\def\tikz@plot@id{#1}}%

\tikzoption{samples}{\pgfmathsetmacro\tikz@plot@samples{max(2,#1)}\expandafter
↪  \tikz@plot@samples@recalc\tikz@plot@domain\relax}%
\tikzoption{samples at}{\def\tikz@plot@samplesat{#1}}%
\tikzoption{parametric}[true]{\csname tikz@plot@parametric#1\endcsname}%

\tikzoption{variable}{\def\tikz@plot@var{#1}}%

\tikzoption{only marks}[]{\let\tikz@plot@handler\pgfplothandlerdiscard}%
```

第四十九章　透明度

49.1　Overview

通常，Ti*k*Z 画出的图形都是不透明的。

pdfTeX 对透明度效果的支持最好。

49.2　为图形、路径、文字设定透明度

选项 `opacity=`⟨*value*⟩ 设置“不透明度”，⟨*value*⟩ 是 0 到 1 之间的数字，表示不透明的程度，若 ⟨*value*⟩ 大于 1，则当作 1 看待；若 ⟨*value*⟩ 小于 0，则当作 0 看待。

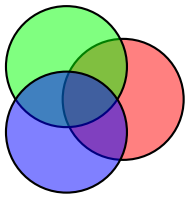
`/tikz/draw opacity=`⟨*value*⟩ (no default)

这个选项设置路径线条的“不透明度”，若 ⟨*value*⟩ 是 1，则线条完全不透明；若 ⟨*value*⟩ 是 0，则线条完全透明，即不可见。

```
\begin{tikzpicture}[line width=1ex]
  \draw (0,0) -- (3,1);
  \filldraw [fill=yellow!80!black,draw opacity=0.5] (1,0) rectangle
  ↪  (2,1);
\end{tikzpicture}
```

`/tikz/fill opacity=`⟨*value*⟩ (no default)

这个选项设置路径填充区域的“不透明度”，对路径上的文字标签、填充的颜色、颜色渐变、插入的外部图形都有效。



```
\begin{tikzpicture}[thick,fill opacity=0.5]
  \filldraw[fill=red] (0:.5cm) circle (8mm);
  \filldraw[fill=green] (120:.5cm) circle (8mm);
  \filldraw[fill=blue] (-120:.5cm) circle (8mm);
\end{tikzpicture}
```

```
\begin{tikzpicture}
  \fill[fill=red,fill opacity=0.5,] (0,0) rectangle (1,2)
    node[fill=cyan]{\Huge B};
\end{tikzpicture}
```

`/tikz/opacity=`⟨*value*⟩ (no default)

同时设置 `draw opacity=`⟨*value*⟩ 和 `fill opacity=`⟨*value*⟩.

`/tikz/text opacity=`⟨*value*⟩ (no default)

这个选项设置 node 文字标签的“不透明度”，其作用比 `fill opacity` 选项优先。

/tikz/transparent=⟨*value*⟩ (no default)

完全透明，不可见。

/tikz/ultra nearly transparent (style, no value)

这个 style 把透明度设为“几乎透明”。

/tikz/very nearly transparent (style, no value)

/tikz/nearly transparent (style, no value)

/tikz/semitransparent (style, no value)

/tikz/nearly opaque (style, no value)

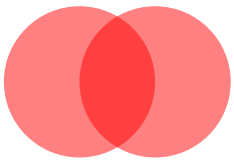
/tikz/very nearly opaque (style, no value)

/tikz/ultra nearly opaque (style, no value)

/tikz/opaque (style, no value)

完全不透明。

如果两个有某种不透明度的区域重叠，则重叠部分的不透明度会叠加。如果不希望出现叠加，可以在环境选项中使用 `transparency group` 选项，见后文 Transparency Groups 一节。



```
\begin{tikzpicture}[fill opacity=0.5]
  \fill[red] (0,0) circle (1);
  \fill[red] (1,0) circle (1);
\end{tikzpicture}
```

49.3 混色模式

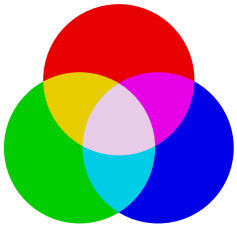
当两个或多个有某种不透明度的区域重叠时，重叠部分的颜色是混合颜色，有多种混色模式（blend mode）来决定重叠部分的颜色。这里用的混色模式是 PDF 参考文件（PDF Reference, six edition, §7.2.4）中的模式。

/tikz/blend mode=⟨*mode*⟩ (no default)

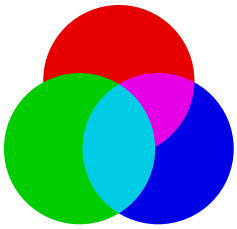
这个选项选定混色模式，⟨*mode*⟩ 是混色模式的名称。混色是 PDF 格式的高级功能，不同的预览器对颜色渲染（color render）的显示效果有所差别。

不同阅读器对于同一混色模式的显示效果可能不同，为了确保显示效果一致，最好这样做：如果这个选项用作环境选项，则这个环境必须是套嵌在某个外层环境内部的子环境，而且外层环境必须带有 `transparency group` 选项；如果这个选项用作某个绘图命令的选项，则这个命令所从属的环境必须带有 `transparency group` 选项。

如果这个选项用作环境选项，则对环境内的各个图形有效，即按混色模式画出图形的重叠部分。如果这个选项用作某个绘图命令的选项，则该命令绘出的图形与其它图形出现混色效果。



```
\tikz {
\begin{scope}[transparency group]
  \begin{scope}[blend mode=screen]
    \fill[red!90!black] ( 90:.6) circle (1);
    \fill[green!80!black] (210:.6) circle (1);
    \fill[blue!90!black] (330:.6) circle (1);
  \end{scope}
\end{scope}
}
```

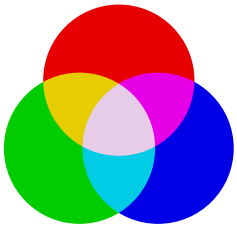


```
\tikz {
\begin{scope}[transparency group]
  \fill[red!90!black] ( 90:.6) circle (1);
  \fill[green!80!black] (210:.6) circle (1);
  \fill[blue!90!black,blend mode=screen]
    (330:.6) circle (1);
\end{scope}
}
```

/tikz/blend group=⟨*mode*⟩ (no default)

本选项会设置真值 \tikz@transparency@grouptrue.

把这个选项用作环境选项，它会使得当前环境是个“有透明度的环境”（transparency group），环境内的混色模式选定为 ⟨*mode*⟩.



```
\tikz [blend group=screen] {
  \fill[red!90!black] ( 90:.6) circle (1);
  \fill[green!80!black] (210:.6) circle (1);
  \fill[blue!90!black] (330:.6) circle (1);
}
```

不同的混色模式有不同的视觉效果，参考手册。

49.4 颜色淡入、淡出——fading

颜色的淡入、淡出——fading——指的是颜色透明度的平滑过渡，即 fading，soft masks，opacity masks，masks，soft clips.

参考 \pgfdeclarefading→P. 383.

49.4.1 创建 fading

创建一个 fading 的基本方法是使用灰度图（fading picture）。灰度图就是只有黑、白、灰三种颜色的图，即黑白图。颜色具有明度（也叫做亮度，luminosity）属性，颜色的明度使用黑色、白色、各种灰色来标示，白色的明度最高（光能量强），黑色的明度最低（光能量弱），灰色的明度居间。

把通常的绘图命令放入环境 {tikzfadingfrompicture} 中，对路径的不同部位规定不同的明度 (灰度，黑白度)，就作成一个灰度图。通常灰度图是不可见的。用 name=⟨*name*⟩ 选项给灰度图命名，然后在环境 {tikzfadingfrompicture} 之外，在正式的绘图环境中，将灰度图的名称作为某个绘图命令的选项，就把灰度图作为一个不可见的潜在图形引入绘图环境中。绘图命令定义的路径（图形）应当与灰度图有重叠，因为颜色的 fading 效果正是针对这个重叠部分的。灰度图的作用是这样的：假设 fading 图中的一个像素点 P 与 fading picture 中的像素点 P' 处于相同的坐标位置，则点 P' 的明度就是点 P 的不透明度；也就是说，若点 P' 是白色，则点 P 不透明；若点 P' 是黑色，则点 P 透明。这种“黑透白不透”的对应

关系有点反直觉，所以 TikZ 定义了一个名称为 transparent 的颜色，它实际上等效于黑色。颜色表达式 transparent!⟨*percentage*⟩ 用小数 ⟨*percentage*⟩ 来表示透明度，数值越大，越是透明。

如果路径（图形）上的像素点 P 不与灰度图上的任何像素点对应重合，则像素点 P 完全透明。为了避免路径（图形）与灰度图没有重合点的情况，TikZ 提供逻辑值选项 fit fading，并设定其初始值为 true，其作用是将灰度图做适当的平移和放缩：平移灰度图，使其边界盒子的中心与路径（图形）的边界盒子的中心重合；放缩灰度图，使其边界盒子能充分覆盖路径（图形）的边界盒子——这样使得二者的重合部分尽可能地大。

下面会使用 fading 库中定义的灰度图，需要载入这个库：\usetikzlibrary{fadings}.

\begin{tikzfadingfrompicture}[⟨*options*⟩]
⟨*environment content*⟩
\end{tikzfadingfrompicture}

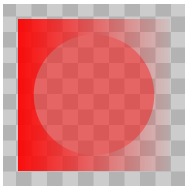
这个环境定义灰度图，但不可见，环境中的绘图命令是通常的绘图命令。

本环境定义的灰度图是全局有效的，各个灰度图的名称最好不要重复。

该环境可以带有命名选项：

/tikz/name=⟨*name*⟩ (no default)

这个选项作为环境 {tikzfadingfrompicture} 的选项，给该环境命名。命名后，可以用 path fading=⟨*name*⟩ 选项引用这个环境。



```
  % 定义灰度图
\begin{tikzfadingfrompicture}[name=fade right]
  \shade[left color=transparent!0,right color=transparent!100]
    (0,0) rectangle (2,2);
  \fill[transparent!50] (1,1) circle (0.7);
\end{tikzfadingfrompicture}
  % 下面在 {tikzpicture} 环境中画出 fading 图
\begin{tikzpicture}
  % 把背景设为 black!20 的棋盘
  \fill [black!20] (-1.2,-1.2) rectangle (1.2,1.2);
  \pattern [pattern=checkerboard,pattern color=black!30]
    (-1.2,-1.2) rectangle (1.2,1.2);
  % 使用 fill 命令画出 fading 图
  \fill [path fading=fade right,red] (-1,-1) rectangle (1,1);
\end{tikzpicture}
```

上面图形中，定义灰度图时用了 \shade 命令，使得灰度图的灰度平滑过渡。在最后的命令 \fill 中引用灰度图，这个命令定义的矩形路径与灰度图的一部分重合，fading 效果只在这一重合部分上显现。

```
  % 定义灰度图
\begin{tikzfadingfrompicture}[name=tikz]
  \node [text=transparent!30]
  {\scalebox{5}{\usefont{U}{yfrak}{m}{n}\selectfont TikZ}};
\end{tikzfadingfrompicture}
  % 下面在 {tikzpicture} 环境中画出 fading 图
\begin{tikzpicture}
  \fill [black!20] (-2,-1) rectangle (2,1);
  \pattern [pattern=checkerboard,pattern color=black!30]
    (-2,-1) rectangle (2,1);
  % 使用 shade 命令画出 fading 图
  \shade
  ↪  [path fading=tikz,fit fading=false,left color=blue,right color=red]
    (-2,-1) rectangle (2,1);
\end{tikzpicture}
```

\tikzfadingfrompicture[⟨*options*⟩]
⟨*environment contents*⟩
\endtikzfadingfrompicture

The plainTeX version of the environment.

\starttikzfadingfrompicture[⟨*options*⟩]

⟨*environment contents*⟩

\stoptikzfadingfrompicture

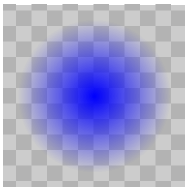
The ConTEXt version of the environment.

\tikzfading[⟨*options*⟩]

这个命令的 ⟨*options*⟩ 可以使用以下选项：

1. 用 name=⟨*name*⟩ 设置名称。这个名称是全局有效的，所用名称最好不要重复。
2. 用 shading 选项指定一种渐变模式。
3. 指定渐变模式后，再用 transparent!⟨*percentage*⟩ 值来指定灰度的变化。

由于在该命令中使用 shading=⟨*shading name*⟩ 选项，或者使用能决定某一种颜色渐变的选项，所以本命令实际上指定某一种颜色渐变，把颜色渐变转为灰度图；因为是颜色渐变，所以灰度是平滑变化的。当某个路径带有 path fading=⟨*name*⟩ 选项后，灰度图用于此路径，使得此路径产生 fading 效果。



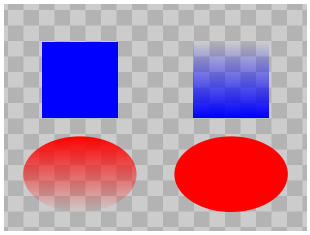
```
  % 规定灰度的变化
\tikzfading[name=fade out,
  inner color=transparent!0,
  outer color=transparent!100]
  % 下面在 {tikzpicture} 环境中画出 fading 图
\begin{tikzpicture}
  \fill [black!20] (-1.2,-1.2) rectangle (1.2,1.2);
  \path [pattern=checkerboard,pattern color=black!30]
    (-1.2,-1.2) rectangle (1.2,1.2);
  \fill [blue,path fading=fade out] (-1,-1) rectangle (1,1);
\end{tikzpicture}
```

49.4.2 创建 fading 路径

下面介绍使得一个路径具有 fading 效果的选项。

/tikz/path fading=⟨*name*⟩ (default scope's setting)

当一个路径带有这个选项后，该路径具有 fading 效果。⟨*name*⟩ 可以是环境 {tikzfadingfrompicture} 定义的灰度图名称，或命令 \tikzfading[⟨*options*⟩] 定义的名称，也可以是 fadings 库提供的预定义的灰度图名称。如果不写出 ⟨*name*⟩，就使用环境选项的设定。如果设置 path fading=none 则取消 fading 效果。注意，每个路径都会 reset 这个选项，也就是说，当给环境带上选项 path fading=⟨*name*⟩ 后，环境内的路径还要带上选项 path fading 才能具有 fading 效果。

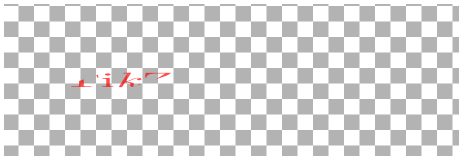


```
\begin{tikzpicture}[path fading=south] % 将 path fading 作为环境选项
  \fill [black!20] (0,0) rectangle (4,3);
  \pattern [pattern=checkerboard,pattern color=black!30] (0,0) rectangle
  ↪  (4,3);
  \fill [color=blue] (0.5,1.5) rectangle +(1,1);
  \fill [color=blue,path fading=north] (2.5,1.5) rectangle +(1,1);
  \fill [color=red,path fading] (1,0.75) ellipse (.75 and .5);
  ↪  % 用环境选项的值
  \fill [color=red] (3,0.75) ellipse (.75 and .5);
\end{tikzpicture}
```

```
\begin{tikzpicture}
  \fill [color=red,path fading=circle with fuzzy edge 10 percent]
    (0,0.5) ellipse (0.75 and 0.5);
  \fill [color=red,path fading=circle with fuzzy edge 20 percent]
    (0,-0.5) ellipse (0.75 and 0.5);
\end{tikzpicture}
```

/tikz/fit fading=⟨*boolean*⟩ (default true, initially true)

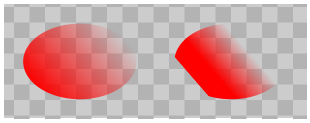
若这个选项的值为 true，则在制作 fading 时会平移灰度图，使得灰度图的中心与绘图命令定义的路径的中心重合（这里说的中心是它们各自的 bounding box 的中心），并且还会对灰度图做适当的放缩，并使得灰度图尽可能能充分地覆盖路径。若这个选项的值为 false，则平移灰度图，使得灰度图的边界盒子的中心与原点重合，然而并不对灰度图作放缩。



```
\begin{tikzfadingfrompicture}[name=Tikz]
  \node [text=transparent!30] {Ti\emph{k}Z};
\end{tikzfadingfrompicture}
\begin{tikzpicture}
  \pattern [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (6,2);
  \draw[path fading=Tikz,fit fading=true,red,line width=0.3cm] (0,0.5) -- (3,1.5);
  \shade[path fading=Tikz,fit fading=true,left color=blue,right color=red] (2,1) rectangle (5,2);
  ↪  % 有重合，文字扁
  \shade[path fading=Tikz,fit fading=false,] (5,0) rectangle (6,2); % 无重合，无文字
\end{tikzpicture}
```

/tikz/fading transform=⟨*transformation options*⟩ (no default)

这个选项值所设定的变换是针对灰度图的，先将这个选项指出的变换施加于灰度图，然后再制作 fading 图。

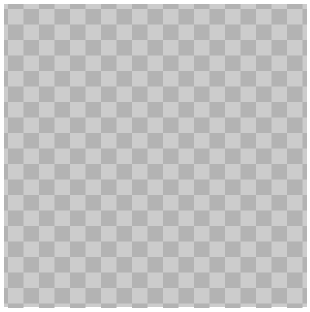


```
\begin{tikzpicture}[path fading=south]
  \fill [black!20] (0,0) rectangle (4,1.5);
  \path [pattern=checkerboard,pattern color=black!30]
    (0,0) rectangle (4,1.5);
  \fill [red,path fading,fading transform={rotate=120}]
    (1,0.75) ellipse (.75 and .5);
  \fill [red,path fading,fading transform={rotate=120,yscale=0.4}]
    (3,0.75) ellipse (.75 and .5);
\end{tikzpicture}
```

/tikz/fading angle=⟨*degrees*⟩ (no default)

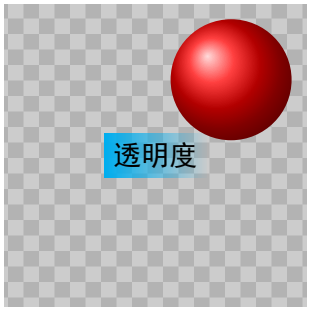
等价于 fading transform={rotate=⟨*degrees*⟩}.

任何东西都可以做出 fading 效果，包括颜色渐变。



```
\begin{tikzpicture}
% Checker board
  \fill [black!20] (0,0) rectangle (4,4);
  \path [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (4,4);
  \shade [ball color=blue,path fading=south] (2,2) circle (1.8);
\end{tikzpicture}
```

注意如果 node 带有 path fading 选项，则它的背景具有 fading 效果，而不是它的的文字具有 fading 效果。



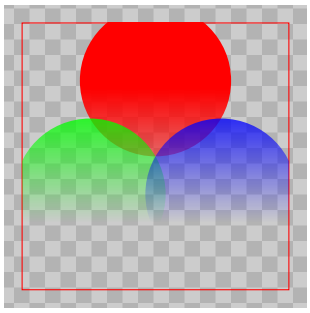

```
\tikzfading[name=fade inside,
  inner color=transparent!100,
  outer color=transparent!40]
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (4,4);
  \path [pattern=checkerboard,pattern color=black!30]
    (0,0) rectangle (4,4);
  \shade [ball color=red] (3,3) circle (0.8);
  \shade [ball color=white,path fading=fade inside]
    (2,2) circle (1.8);
  \node [fill=cyan,path fading=east] at(2,2) {\heiti 透明度};
\end{tikzpicture}
```

49.4.3 Fading a Scope

`/tikz/scope fading=`⟨*fading*⟩ (no default)

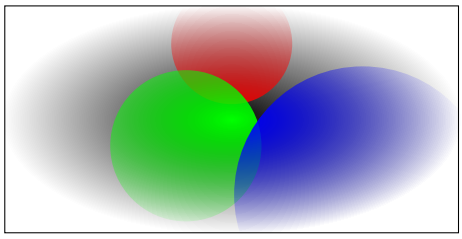
⟨*fading*⟩ 可以是环境 `{tikzfadingfrompicture}` 定义的灰度图名称，或命令 `\tikzfading[`⟨*options*⟩`]` 定义的名称，也可以是 fadings 库提供的预定义的灰度图名称。选项 `scope fading` 的作用类似于 `clip`，当一个路径带上这个选项后，此路径以及之后的各个路径具有 fading 效果，直到当前环境结束。

对于带有 `scope fading` 选项的路径来说，选项 `fit fading` 和 `fading transform` 默认是有效的，也就是说，带有 `scope fading` 选项的路径决定灰度图的尺寸、位置，从而决定了在图形的什么地方出现 fading 效果；如果给这个路径带上选项 `fit fading=false`，那么无论这个路径处于坐标系的什么位置，灰度图都被放到原点那里，灰度图的边界盒子的中心与原点重合，灰度图的尺寸保持它本身的尺寸。



```
\begin{tikzpicture}
  \fill [black!20] (-2,-2) rectangle (2,2);
  \pattern [pattern=checkerboard,pattern color=black!30]
    (-2,-2) rectangle (2,2);
  \draw [red] (-50bp,-50bp) rectangle (50bp,50bp);
  \path [scope fading=south,fit fading=false] (1,1);
  \fill[red] ( 90:1) circle (1);
  \fill[green] (210:1) circle (1);
  \fill[blue] (330:1) circle (1);
\end{tikzpicture}
```

上面例子中，1bp=1.00374pt. 命令 `\path` 引入了一个预定义的灰度图 `south`，还设置 `fit fading=false`，这样灰度图的尺寸就是固定的了。灰度图 `south` 的长度、宽度都是 `100bp`，中心在点 `(0,0)` 处，红色矩形就是灰度图 `south` 的轮廓。如果去掉命令 `\path` 中的选项 `fit fading=false`，那么灰度图 south 就会去匹配一个点，即只有位置、没有尺寸，所以不会出现任何 fading 效果。



```
\begin{tikzpicture}
  \tikzfading[name=fade out,inner color=transparent!0,outer color=transparent!100]
  \begin{scope}[overlay]
```

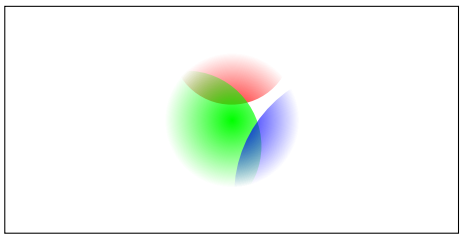
```
  \fill [scope fading=fade out,inner color=cyan,outer color=black]
    (-3,-1.5) rectangle (3,1.5);
  \fill[red] ( 90:1) circle (0.8);
  \fill[green] (210:0.7) circle (1);
  \fill[blue] (330:2) circle (1.7);
  \end{scope}
  \draw (-3,-1.5) rectangle (3,1.5);
\end{tikzpicture}
```

上面例子中，`inner color=transparent!0,outer color=transparent!100` 把名称为 radial 的渐变做成灰度图。渐变 radial 的定义见文件《tikz.code.tex》：

```
\tikzoption{inner color}{\pgfutil@colorlet{tikz@radial@inner}{#1}\def\tikz@shading
↪  {radial}\tikz@addmode{\tikz@mode@shadetrue}}%
\tikzoption{outer color}{\pgfutil@colorlet{tikz@radial@outer}{#1}\def\tikz@shading
↪  {radial}\tikz@addmode{\tikz@mode@shadetrue}}%
% 省略若干
\pgfdeclareradialshading[tikz@radial@inner,tikz@radial@outer]{radial}{
↪  \pgfpointorigin}{%
 color(0bp)=(tikz@radial@inner);
 color(25bp)=(tikz@radial@outer);
 color(50bp)=(tikz@radial@outer)}%

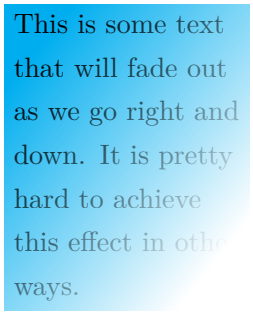
\pgfutil@colorlet{tikz@radial@inner}{gray}%
\pgfutil@colorlet{tikz@radial@outer}{white}%
```

猜一下上面定义的意思：渐变 radial 的形状是圆形，半径是 50bp；它的渐变颜色只有两种，一种位于圆心，一种围绕圆心；在半径 25bp 处开始渐变；默认的渐变颜色是中心灰、外围白。给上面例子中第一个 `\fill` 命令加选项 `fit fading=false` 就会让灰度图保持其原来的形状尺寸（半径 50bp），并且灰度图的中心位于原点：



```
\begin{tikzpicture}
  \tikzfading[name=fade out,inner color=transparent!0,outer color=transparent!100]
  \begin{scope}[overlay]
  \filldraw [scope fading=fade out,inner color=cyan,outer color=black,fit fading=false]
    (4,1) rectangle (5,1);
  \fill[red] ( 90:1) circle (0.8);
  \fill[green] (210:0.7) circle (1);
  \fill[blue] (330:2) circle (1.7);
  \end{scope}
  \draw (-3,-1.5) rectangle (3,1.5);
\end{tikzpicture}
```

如果 `scope fading` 用作 node 的选项，则 node 的背景和文字都具有 fading 效果，这与选项 `path fading` 不同。




```
\tikz \node [fill=cyan,scope fading=south,
    fading angle=45,text width=3cm]
{
This is some text that will fade out as we go right
and down. It is pretty hard to achieve this effect in
other ways.
};
```

49.5 Transparency Groups

在下面的图形中，右侧图形中的圆与斜线段都有不透明度，它们重叠部分的不透明度出现叠加，对于一个标志来说这显然不太合适。

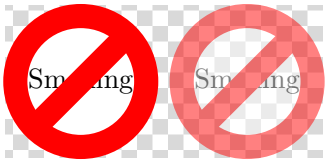
```
\begin{tikzpicture}
\node [forbidden sign,line width=2ex,draw=red,fill=white]
  at (0,0) {Smoking};
\node [opacity=.5,forbidden sign,line width=2ex,draw=red,fill=white]
  at (2.2,0) {Smoking};
\end{tikzpicture}
```

Transparency groups 可以解决这类问题。当环境带有 `transparency group` 选项后，此环境就成为一个 transparency group.

`/tikz/transparency group=`[⟨*options*⟩] (no default)

本选项会设置真值 `\tikz@transparency@grouptrue`.

这个选项只能用作环境选项。用它就不会出现不透明度叠加的情况。环境内的命令在绘图时，会忽略它之前诸命令的不透明度对该命令的影响—可能有数个命令都涉及同一个像素点，但是只有最后一个命令规定的不透明度对此像素点有效。

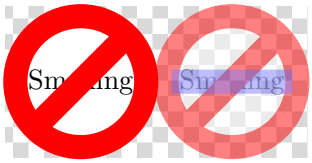


```
\begin{tikzpicture}
  \pattern[pattern=checkerboard,pattern color=black!15]
    (-1,-1) rectangle (3.2,1);
  \node [forbidden sign,line width=2ex,draw=red,fill=white]
    at (0,0) {Smoking};
  \begin{scope}[transparency group,opacity=.5]
    \node [forbidden sign,line width=2ex,draw=red,fill=white]
      at (2.2,0) {Smoking};
  \end{scope}
\end{tikzpicture}
```

观察下面的图形：

```
\begin{tikzpicture}
  \pattern[pattern=checkerboard,pattern color=black!15]
    (-1,-1) rectangle (3,1);
  \node [forbidden sign,line width=2ex,draw=red,fill=white]
    at (0,0) {Smoking};
  \begin{scope}[transparency group,opacity=.5]
    \node (s) [forbidden sign,line width=2ex,draw=red,fill=white]
      at (2,0) {Smoking};
    \draw [line width=2ex, blue] (1.2,0) -- (2.8,0);
  \end{scope}
\end{tikzpicture}
```

上面例子中，`{scope}` 环境选项中的 `opacity=.5` 对命令 `\draw` 有效，但是 `\draw` 的颜色 blue 完全取代了 `\node` 的颜色 red，并且完全覆盖了文字，没有出现“混色效果”。为了出现混色效果，需要给命令 `\draw` 加上选项 `opacity=.5`：



```
\begin{tikzpicture}
  \pattern[pattern=checkerboard,pattern color=black!15]
    (-1,-1) rectangle (3,1);
  \node [forbidden sign,line width=2ex,draw=red,fill=white]
    at (0,0) {Smoking};
  \begin{scope}[transparency group,opacity=.5]
    \node (s) [forbidden sign,line width=2ex,draw=red,fill=white]
      at (2,0) {Smoking};
    \draw [opacity=.5,line width=2ex, blue] (1.2,0) -- (2.8,0);
  \end{scope}
\end{tikzpicture}
```

观察下面图形：

```
\begin{tikzpicture}
  \pattern[pattern=checkerboard,pattern color=black!15]
    (-1,-1) rectangle (1,1);
  \begin{scope}[transparency group,opacity=.5]
    \node (s) [forbidden sign,line width=2ex,draw=red,fill=white]
    ↪  {Smoking};
    \draw [opacity=.25, line width=2ex, red] (-0.8,0) -- (1,0);
    \draw [opacity=1,line width=2ex, red] (-0.8,-.4) -- (1,-.4);
    \draw [line width=2ex, red] (-0.8,0.4) -- (1,0.4);
\end{scope}
\end{tikzpicture}
```

从上面例子看出，对于带有选项 transparency group 的环境，假设其环境选项中有 opacity=⟨*value 1*⟩ 选项，如果环境中的某个命令也带有选项 opacity=⟨*value 2*⟩，则该命令画出图形的不透明度好像是 ⟨*value 1*⟩ × ⟨*value 2*⟩，即两个不透明度值的乘积。

选项 transparency group=[⟨*options*⟩] 中的 ⟨*options*⟩ 可以是以下值：

knockout 打个比方，玻璃窗的一面覆盖了一层水汽，在这一面画上图形、写下文字可以去掉一部分水汽，通过图形、文字可以看到玻璃的另一面。

```
\begin{tikzpicture}
  \shade [left color=red,right color=blue] (-2,-1) rectangle (2,1);
  \begin{scope}[transparency group=knockout]
    \fill [white] (-1.9,-.9) rectangle (1.9,.9);
    \node [opacity=0,font=\fontfamily{ptm}\fontsize{45}{45}\bfseries]
      {Ti\emph{k}Z};
  \end{scope}
\end{tikzpicture}
```

上面例子中，\fill 命令创建“一层覆盖玻璃的水汽”，\node 命令在有水汽的一面写下文字，于是通过文字看到了玻璃的另一面，即 \shade 创建的颜色渐变。\node 命令能够“彻底擦除水汽”，它的选项 opacity=0 指的是它文字自己的不透明度为 0.

注意有的渲染器（renderer）不支持这个功能。

isolated=false 在默认下，transparency group 是被隔离的。将 ⟨*options*⟩ 设为 isolated=false 则取消隔离，详情参考 PDF Reference, six edition, §7.3.4.

TikZ 会自动计算 transparency group 的位置和尺寸，更新图形的边界盒子，如果绘图时使用了 overlay 或 transform canvas 选项，可能导致 TikZ 无法顺利计算坐标位置。

第五十章　装饰路径

50.1　Overview

用几个例子展示一下什么是装饰路径（decorated paths），为了能顺利编译涉及的各种装饰路径代码，首先调用各种相关的库：

- decorations
- decorations.pathmorphing
- decorations.pathreplacing
- decorations.markings
- decorations.footprints
- decorations.shapes
- decorations.text
- decorations.fractals

在调用任何一个以 decorations 为名称前缀的库时，`decorations` 库会被自动加载。

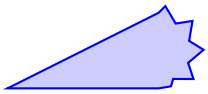
先画一个由直线段和圆弧构成的路径：

```
\tikz \fill [fill=blue!20,draw=blue,thick]
  (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```

然后“装饰”这个路径，给绘图命令添加 `decorate, decoration=zigzag` 选项，将直线段和圆弧换成 zigzag 线型：

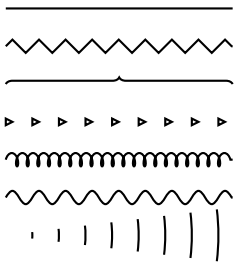
```
\tikz \fill [decorate,decoration={zigzag}]
  [fill=blue!20,draw=blue,thick]
  (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```

也可以保留直线段，仅把圆弧换成 zigzag 线型：



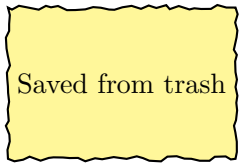
```
\tikz \fill [decoration={zigzag}]
  [fill=blue!20,draw=blue,thick]
 (0,0) -- (2,1) decorate { arc (90:-90:.5) } -- cycle;
```

装饰路径就是用具有装饰效果的线型或者标记来替换原有的直线或者曲线。有数个库（如前列出）分别提供多种不同的线型和标记以供装饰。



```
\begin{tikzpicture}[thick]
  \draw (0,3) -- (3,3);
  \draw[decorate,decoration=zigzag] (0,2.5) -- (3,2.5);
  \draw[decorate,decoration=brace] (0,2) -- (3,2);
  \draw[decorate,decoration=triangles] (0,1.5) -- (3,1.5);
  \draw[decorate,decoration={coil,segment length=4pt}]
    (0,1) -- (3,1);
  \draw[decorate,decoration={coil,aspect=0}]
    (0,.5) -- (3,.5);
  \draw[decorate,decoration={expanding waves,angle=7}]
    (0,0) -- (3,0);
\end{tikzpicture}
```

也可以装饰 node 的轮廓线条：



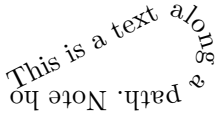

```
\begin{tikzpicture}
  \node [fill=yellow!50,draw,thick,minimum height=2cm, minimum width=3cm,
    decorate,
    decoration={random steps,segment length=3pt,amplitude=1pt}]
    {Saved from trash};
\end{tikzpicture}
```

有 3 种不同类型的路径装饰：

1. path morphing，即路径变体，这种装饰线型由程序库 decorations.pathmorphing 提供，例如 zigzag，bumps，coil，snake，saw 等线型。

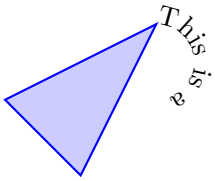
2. path replacing，路径替换，将某种特殊的符号或标记沿着路径放置，路径不再连续，故不能填充颜色。这种装饰由程序库 decorations.pathreplacing 和 decorations.shapes 提供，例如 brace，ticks，waves，crosses，triangles 等。

3. path removing，路径清除，将被装饰的路径清除，将文字或 node 沿着被装饰的路径放置。被装饰路径的选项（如 `draw`，`color=`，`fill` 等）对放置的文字或 node 没有作用。如果被装饰路径是主路径的一段子路径，在装饰完这段子路径后继续构建主路径，构建主路径时会忽略这一段子路径，也就是说，一旦某个路径（或子路径）被装饰，那么被装饰部分会被“丢掉”。




```
\tikz \fill [decorate,decoration={text along path,
    text=This is a text along a path. Note how the path is lost.}]
  [fill=blue!20,draw=blue,thick,red]
  (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```

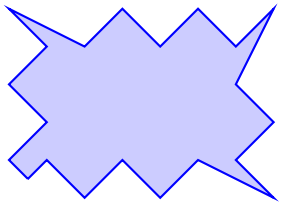
上面例子中的整个被装饰路径都被“丢掉”了。下面例子中，只有一段圆弧被丢掉了，主路径只是由线段构成：



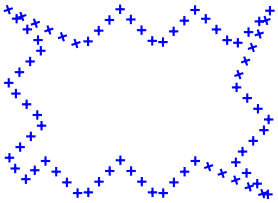

```
\tikz \fill[fill=blue!20,draw=blue,thick]
  (0,0) -- (2,1) decorate[decoration={text along path,
    text=This is a text along a path. Note how the path is lost.}]
  {arc (90:-90:.5)}
  -- (1,-1)--cycle;
```

装饰路径操作可以套嵌使用，形成迭代、叠加效果。

```
\tikz \fill [fill=blue!20,draw=blue,thick]
  (0,0) rectangle (3,2);
```



```
\tikz \fill [fill=blue!20,draw=blue,thick]
  decorate[decoration={zigzag,segment length=10mm,amplitude=2.5mm}]
  {(0,0) rectangle (3,2)};
```



```
\tikz \fill [fill=blue!20,draw=blue,thick]
  decorate[decoration={crosses,segment length=2mm}]
    {decorate[decoration={zigzag,segment length=10mm,amplitude=2.5mm}]
      {(0,0) rectangle (3,2)}
  };
```

复杂的装饰路径可能排版比较慢，也不太精确，这是因为 PGF 要做大量计算，而 TEX 并不太擅长计算。对于直线段的装饰会快一些。

TikZ Library decorations

```
\usetikzlibrary{decorations} % LaTeX and plain TeX
\usetikzlibrary[decorations] % ConTeXt
```

为了使用装饰路径功能，需要先载入这个库。这个库定义了基本的装饰操作、选项，但没有提供更多的装饰类型。其它的库提供多种装饰类型，并且都会调用 decorations 库。

为了清楚装饰路径的作用，需要区分几个概念。

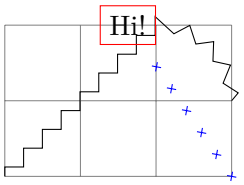
- 主路径，例如 \path, \draw, \node 创建的路径。
- 被装饰路径，即 decorate 的作用范围，可以是整个主路径，也可以是主路径的一部分。
- 子输入路径，子路径，这是两个没有严格区分的概念；有时“子路径”泛指主路径的一部分；有时“子路径”指的是由 line-to, curve-to 操作创建的路径；而“子输入路径”则指的是由 line-to, curve-to 操作创建的路径。在使用命令 \pgfdeclaredecoration →P. 404 时，装饰操作针对的是“子输入路径”。例如，circle 操作创建的圆由 4 段 curve-to 曲线构成，装饰 circle 圆时，针对每一段 curve-to 曲线作装饰。

50.2 用 decorate 操作装饰子路径

decorate 可以像 node 那样用在主路径中：

\path...decorate[⟨*options*⟩]{⟨*subpath*⟩}...;

操作 decorate 引起对 ⟨*subpath*⟩ 的装饰。⟨*subpath*⟩ 中可以有直线段，曲线，圆弧，椭圆弧，椭圆，圆，矩形，或已经装饰过的路径，等等。可以在 ⟨*subpath*⟩ 中使用 node，但 node 只是添加到 ⟨*subpath*⟩ 上的，不属于 ⟨*subpath*⟩，故 node 不被装饰。



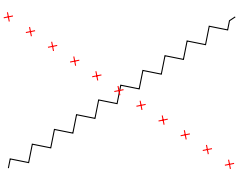

```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw decorate [decoration={name=zigzag}]
    {(0,0) -- (2,2) node (hi) [left,draw=red] {Hi!} arc(90:0:1)};
  \draw [blue] decorate [decoration={crosses}] {(3,0) -- (hi)};
\end{tikzpicture}
```

在 ⟨*options*⟩ 中可以使用以下选项。

/pgf/decoration=⟨*decoration options*⟩ (no default)

/tikz/decoration

本选项在 ⟨*decoration options*⟩ 中选定一种“装饰类型”，并可以用（与该装饰类型匹配的）选项设置装饰路径的外观。注意本选项（是个名词）并不直接引起装饰操作，引起装饰操作的是动词 decorate。

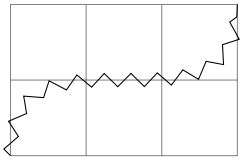


```
\begin{tikzpicture}[decoration=zigzag]
  \draw decorate {(0,0) -- (3,2)};
  \draw [red] decorate [decoration=crosses] {(0,2) -- (3,0)};
\end{tikzpicture}
```

用在 ⟨*decoration options*⟩ 中的 key 的路径都有前缀 /pgf/decoration/，不同装饰类型有不同的选项来调整其外观。

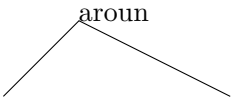
/pgf/decoration/name⟨*name*⟩ (no default, initially none)

这个 key 用在 decoration 之下时，用于指定装饰类型的名称，可以省略 name= 而只写出名称。如果令 name=none 则取消装饰。



```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw decorate [decoration={name=zigzag}]
    { (0,0) .. controls (0,2) and (3,0) .. (3,2) };
\end{tikzpicture}
```

下面例子中，使用装饰类型 text along path，选项 text 的参数是一串文字：




```
\begin{tikzpicture}[decoration={text along path,
  text=around and around and around and around we go}]
  \draw (0,0) -- (1,1) decorate { -- (2,1) } -- (3,0);
\end{tikzpicture}
```

在 ⟨*decoration options*⟩ 中可以套嵌 decorate 操作，构成迭代效果：

```
\begin{tikzpicture}[decoration=Koch
↪  snowflake,draw=blue,fill=blue!20,thick]
  \filldraw (0,0) -- ++(60:1) -- ++(-60:1) -- cycle ;
  \filldraw decorate{ (0,-1) -- ++(60:1) -- ++(-60:1) -- cycle };
  \filldraw decorate{ decorate{ (0,-2.5) -- ++(60:1) -- ++(-60:1)--
  ↪  cycle }};
\end{tikzpicture}
```

可以对一个连续路径的数个子路径分别做不同的装饰：

```
\tikz{
\draw decorate[decoration={name=zigzag}]{(0,0)--(1,0)}
      decorate[decoration={name=coil,aspect=0.6,amplitude=2mm}]
      ↪  {--(1,1)--(0,1)};
}
```

前面提到，如果被装饰路径是主路径的一段子路径，在装饰完这段子路径后继续构建主路径，但忽略被装饰路径。

50.3 装饰整个路径

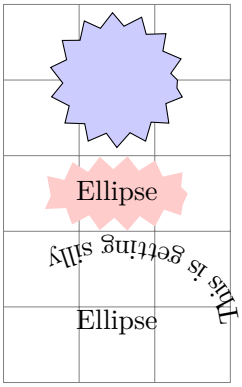
下面两个形式等效：

```
\path decorate[⟨options⟩]{⟨path⟩};
\path [decorate,⟨options⟩] ⟨path⟩;
```

第一句中 decorate 作为操作（动词）使用，第二句中 decorate 作为选项使用。

/tikz/decorate=⟨*boolean*⟩ (default true)

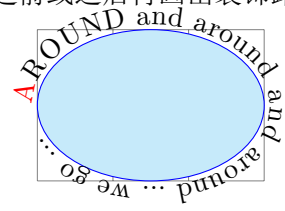
当某个路径带有这个选项时，对该路径启动装饰功能，但具体怎么装饰还得用选项 decoration 指定，否则没有装饰。使用该选项两次不意味着套嵌装饰操作，因为作为选项的 decorate 只是意味着逻辑值 true，这与作为操作（动词）的 decorate 不一样。



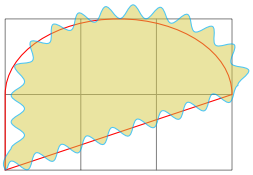

```
\begin{tikzpicture}[decoration=zigzag]
  \draw [help lines] (0,0) grid (3,5);
  \draw [fill=blue!20,decorate] (1.5,4) circle (0.8cm);
  \node at (1.5,2.5) [fill=red!20,ellipse,decorate] {Ellipse};
  \node at (1.5,0.8) [inner sep=6mm,fill=red!20,ellipse,
    decorate,decoration={text along path,text={This is getting silly}}]
  {Ellipse};
\end{tikzpicture}
```

上面例子中，装饰类型选项 `text along path` 清除了原来的被装饰路径，并使得文字沿着原来的被装饰路径排出，在默认下文字会排在路径的左侧。上面例子中，最后一个 node 的形状是 ellipse，其路径方向是逆时针的。

可以将装饰选项作为选项 `preaction` 或 `postaction` 的值添加到绘图命令选项中，这样在画出主路径之前或之后再画出装饰路径会，从而使被装饰路径仍然能得到显示：




```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\fill [draw=blue,fill=cyan!20,
  postaction={decorate,decoration={raise=2pt,text along path,
    text={{\color{red}A}ROUND and around and around ... we go ...}}}]
  (0,1) arc (180:-180:1.5cm and 1cm);
\end{tikzpicture}
```



```
\begin{tikzpicture}[decoration=snake]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate,fill=yellow!80!black,
      fill opacity=0.5,draw=cyan,draw opacity=0.7},red]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

50.4 调整装饰路径的外观

50.4.1 调整装饰路径与原被装饰路径的相对位置

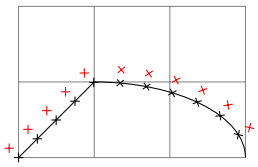
下面的选项只能用于 Ti*k*Z 中。

`/pgf/decoration/raise`=⟨*dimension*⟩ (no default, initially `0pt`)

这个选项使得装饰路径偏离原被装饰路径，偏离的距离是 ⟨*dimension*⟩，默认沿着路径方向“向左偏”。

如果同时给出 `raise` 和 `transform` 选项（见下文），则 `raise` 在 `transform` 之后起作用。

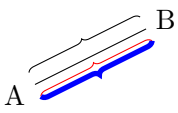
如果 ⟨*dimension*⟩ 是负值尺寸，则装饰路径沿着被装饰路径方向“向右偏”。



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw (0,0) -- (1,1) arc (90:0:2 and 1);
  \draw decorate [decoration=crosses] {(0,0)--(1,1) arc (90:0:2 and 1)
  ↪  };
  \draw[red] decorate [decoration={crosses,raise=5pt}] {(0,0) -- (1,1)
  ↪  arc (90:0:2 and 1)};
\end{tikzpicture}
```

`/pgf/decoration/mirror`=⟨*boolean*⟩ (no default)

将原被装饰路径作为“镜面”，将装饰路径从“镜面”的一侧变到另一侧。如果同时给出 `mirror`, `raise`, `transform` 选项（见下文），则 `mirror` 在最后起作用。




```
\begin{tikzpicture}
  \node (a) {A};
  \node (b) at (2,1) {B};
  \draw (a) -- (b);
  \draw[decorate,decoration={brace,raise=5pt}] (a) -- (b);
  \draw[decorate,decoration={brace,raise=-5pt},red] (a) -- (b);
  \draw[decorate,decoration={brace,raise=5pt,mirror},blue,line width=2pt]
    (a) -- (b);
\end{tikzpicture}
```

注意上面例子中，raise=-5pt 与 mirror，raise=5pt 不一样。

/pgf/decoration/transform=⟨*transformations*⟩ (no default)

这里的 ⟨*transformations*⟩ 是通常的 TikZ 变换，如 shift，rotate，变换是针对装饰路径的，该选项会在前面讲的选项 raise，mirror 之前起作用。

50.4.2 调整装饰路径的始端与终端的形态

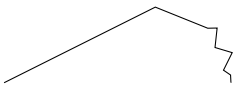
装饰路径从原被装饰路径的起点延续到终点，装饰路径的形态可能会使得原被装饰路径的起点和终点不容易辨认，也不够美观。这时候就需要调整装饰路径的始端与终端的形态，调整的方法不唯一，比较便利的是使用下面的选项，注意它们只能用于 decorations，不能用于 meta-decorations.

/pgf/decoration/pre=⟨*decoration*⟩ (no default, initially lineto)

⟨*decoration*⟩ 是装饰类型的名称，例如 lineto，curveto，moveto，zigaza，saw，coil 等等。这个选项使得装饰路径的“始端点”在被装饰路径的“起点”之后（仍在被装饰路径上），而这两点间的联系方式由 ⟨*decoration*⟩ 指定，两点间的距离由选项 pre length 规定。⟨*decoration*⟩ 的默认值是 lineto，即用直线段联系两点；值 moveto 使得两点间没有连线。

```
\tikz [decoration={zigzag,pre=crosses,pre length=1cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

/pgf/decoration/pre length=⟨*dimension*⟩ (no default, initially 0pt)



```
\tikz [decoration={zigzag,pre length=3cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

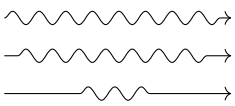
```
\tikz [decoration={zigzag,pre=curveto,pre length=3cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

/pgf/decorations/post=⟨*decoration*⟩ (no default, initially lineto)

作用类似 pre，不过针对的是装饰路径的终端点与原被装饰路径的终点。

/pgf/decoration/post length=⟨*dimension*⟩ (no default, initially 0pt)

类似 pre length.

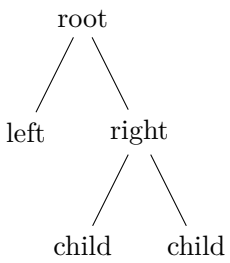


```
\begin{tikzpicture}[decoration=snake,
  line around/.style={decoration={pre length=#1,post length=#1}}]
  \draw[->,decorate] (0,0) -- ++(3,0);
  \draw[->,decorate,line around=5pt] (0,-5mm) -- ++(3,0);
  \draw[->,decorate,line around=1cm] (0,-1cm) -- ++(3,0);
\end{tikzpicture}
```

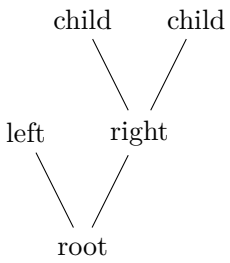
第五十一章　树

51.1　Child 操作简介

在 Ti*k*Z 中有两种自动绘制树的方法：使用 graph 操作（见 §19），使用 child 操作。下面是两个使用 child 操作的例子：




```
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```




```
\begin{tikzpicture}
\makeatletter
  \node (root) {root};
  \path [grow'=up]
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

上面例子中的 `child` 命令会在 node 路径中创建树的子节点，所创建的子节点是 node，它有自己的形状、各种锚位置。

注意，在用 node, child 操作创建树的句法中不能有空行。这个句法创建一个层级结构，每个 child 操作创建一个“子结构”。

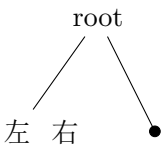
`\path ... child[`⟨*options*⟩`]` `foreach` ⟨*variables*⟩ `in {`⟨*values*⟩`}{`⟨*child path*⟩`}...;`

关于这个句法：

- `foreach` ⟨*variables*⟩ `in {`⟨*values*⟩`}` 是可选的。
- 当 Ti*k*Z 遇到第一个单词 `child` 时会触发内部命令 `\tikz@children`, 解析 tree 的句法。
- 在第一个 `child` 前应当事先创建一个 node 或 coordinate 作为根节点。如果根节点与第一个 `child` 命令不在一个路径内，那么最好为根节点命名，因为 Ti*k*Z 会用根节点的名称来为子节点命名。如果根节点与第一个 `child` 命令在一个路径内，那么可以不为根节点命名，此时 Ti*k*Z 会为根节点赋一个编号，将这个编号用作节点的名称。参考 `\tikz@fig@mustbenamed` → P. 842.
- `child` 命令创建分支内的第一个子节点时，会把之前最近创建的 node 或 coordinate 作为父节点。
- 在解析 `child` 命令时，符号代码会被固定，因此 `child` 命令的参数中不能含有抄录命令。
- Ti*k*Z 会统计、解析、保存遇到的各个 `child` 命令，然后创建 child 节点。Ti*k*Z 会计算各个子节点的位置，在创建一个子节点时，Ti*k*Z 会把坐标系的原点平移到该子节点的位置上，然后在这

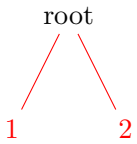
个坐标系中绘制该子节点，即绘制 ⟨*child path*⟩ 指定的图形。通常，⟨*child path*⟩ 只是一个 node 命令，创建一个 node 作为子节点。

- ⟨*child path*⟩ 中第一个 node 一个子节点被创建后，会在这个子节点与其父节点之间画边。



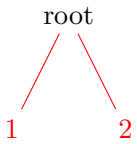

```
\begin{tikzpicture}
  \node (root){root}
    child {node [left]{左} node [right]{右}}
    child {[fill] circle (2pt)};
\end{tikzpicture}
```

`child` 命令中的 `foreach` 语句是可选的，例如




```
\tikz{
  \node {root} child [red] foreach \name in {1,2} {node {\name}};
}
```

等效于




```
\tikz{
  \node {root}
    child[red] {node {1}}
    child[red] {node {2}};
}
```

再如

```
node {root} child[\pos] foreach \name/\pos in {1/left,2/right} {node[\pos] {\name}
↪  }
```

等效于

```
node {root}
  child[left] {node[left] {1}}
  child[right] {node[right] {2}}
```

child 操作可以套嵌使用，如

```
\begin{tikzpicture}
  [level distance=4mm,level/.style={sibling distance=8mm/#1}]
  \coordinate
    child foreach \x in {0,1}
      {child foreach \y in {0,1}
        {child foreach \z in {0,1}}};
\end{tikzpicture}
```

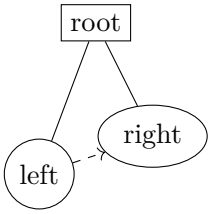
51.2 Child Paths and Child Nodes

如前述，Ti*k*Z 会计算各个子节点的位置，在创建一个子节点时，以该子节点的位置为原点创建一个坐标系（暂时称之为“子节点坐标系”），然后在这个坐标系中执行 ⟨*child path*⟩ 来绘制子节点；⟨*child path*⟩ 中的第一个 node （如果有的话）会被特殊对待，称之为 child node. 如果 ⟨*child path*⟩ 中没有使用 `node`，`coordinate` 命令，或者 ⟨*child path*⟩ 缺失（连花括号都没有），那么这个子节点就会被自动创建为一个形状是 `coordinate` 的空的子节点。例如

```
\node {x} child {node {y}} child;
```

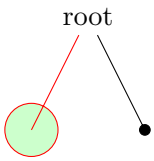
第二个 `child` 命令的 ⟨*child path*⟩ 缺失，故它创建的子节点是一个 `coordinate`.

⟨*child path*⟩ 中的 node 语句可以像通常的那样使用，可以为 node 命名，用选项对 node 做变换，改变其外观等。




```
\begin{tikzpicture}[sibling distance=15mm]
  \node[rectangle,draw] {root}
    child {node[circle,draw,yshift=-5mm] (left node) {left}}
    child {node[ellipse,draw] (right node) {right}};
  \draw[dashed,->] (left node) -- (right node);
\end{tikzpicture}
```

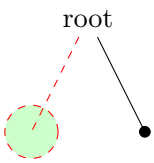
观察下面的例子：




```
\begin{tikzpicture}
  \node {root}
    child {[draw=red, fill=green, fill opacity=0.2] circle (10pt)}
    child {[fill] circle (2pt)};
\end{tikzpicture}
```

上面例子中，两个 child 命令的参数中都没有 node 命令，所以两个子节点都是 coordinate ——都没有内部尺寸，从根节点 root 到子节点的边直接连到子节点坐标系的原点位置；注意在子节点内有画圆的命令，画圆命令是在子节点坐标系中画圆的，为了方便，默认在画圆命令的前面使用 move-to (0,0) 操作，所以画出的圆心位于子节点坐标系的原点。

在 ⟨*child path*⟩ 的结尾处，TikZ 会自动使用一个特殊命令 edge from parent（见后文），你也可以手工添加这个命令并给这个命令带上选项，这样就可以设置从父节点到该子节点的边的外观，例如重画前面的图形




```
\begin{tikzpicture}
  \node {root}
    child {[draw=red, fill=green, fill opacity=0.2] circle (10pt)
            edge from parent[dashed]}
    child {[fill] circle (2pt)};
\end{tikzpicture}
```

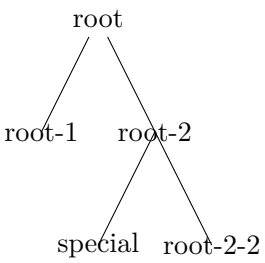
命令 edge from parent[dashed] 不仅影响了边，也影响了 circle 命令。

在用 node, child 操作创建树的句法中，node 操作的作用比较敏感，要注意把 node 操作用在正确的位置上，见 §21.6.

51.3 子节点的命名

一个子节点的 ⟨*child path*⟩ 中的 node 是可以被命名的，命名的方式与普通 node 的命名方式相同。子节点的 ⟨*child path*⟩ 中的第一个 node 的名称会被作为该子节点的名称。如果不为 ⟨*child path*⟩ 中的第一个 node 命名，那么 TikZ 会自动为它命名，命名规则是：如果“根节点”的名称是 ⟨*parent*⟩, 那么 ⟨*parent*⟩ 的第一个子节点的第一个 node 会被命名为 ⟨*parent*⟩-1，而第一个子节点的其它 node 不会被自动命名；⟨*parent*⟩ 的第二个子节点的第一个 node 会被命名为 ⟨*parent*⟩-2；依次类推。⟨*parent*⟩-1 的第一个子节点的第一个 node 会被命名为 ⟨*parent*⟩-1-1；⟨*parent*⟩-1 的第二个子节点的第一个 node 会被命名为 ⟨*parent*⟩-1-2；依次类推。

注意，如果只是把 ⟨*parent*⟩ 的第一个子节点命名为 ⟨*something*⟩, 那么 ⟨*parent*⟩ 的第二个子节点仍然会被自动命名为 ⟨*parent*⟩-2.

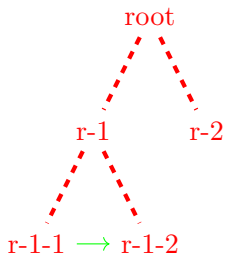



```
\begin{tikzpicture}[sibling distance=15mm]
  \node (root) {root}
    child
    child {
      child {coordinate (special)}
      child
  };
  \node at (root-1) {root-1};
  \node at (root-2) {root-2};
  \node at (special) {special};
  \node at (root-2-2) {root-2-2};
\end{tikzpicture}
```

51.4 为树或节点指定选项

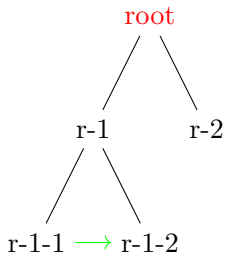
node 操作之后可以带有方括号选项，child 操作之前或之后可以带有方括号选项，子节点的 ⟨*child path*⟩ 中的 node 操作之后可以带有方括号选项，等等。树或节点的选项所处的位置不同，其作用范围也不同：

1. 针对整个树的选项要在根节点之前写出。



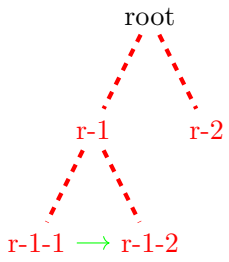

```
\begin{tikzpicture}
  {[ultra thick, dashed, red]
  \node (r){root}
    child { node{r-1}
              child {node{r-1-1}}
              child {node{r-1-2}}}
    child {node{r-2}};}
  \draw [->, green] (r-1-1)--(r-1-2);
\end{tikzpicture}
```

2. 针对根节点的选项要写在创建根节点的 node 操作之后，并且只对根节点有效。



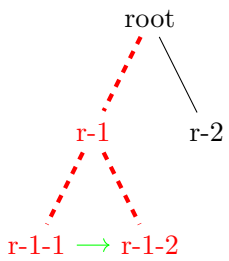

```
\begin{tikzpicture}
  \node [ultra thick, dashed, red] (r){root}
    child { node{r-1}
              child {node{r-1-1}}
              child {node{r-1-2}}}
    child {node{r-2}};
  \draw [->, green] (r-1-1)--(r-1-2);
\end{tikzpicture}
```

3. 针对所有子节点（由 child 操作创建的节点）的选项要在第一个 child 操作之前写出，并且对所有子节点和树的边有效。



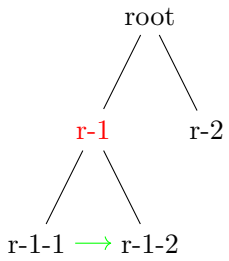

```
\begin{tikzpicture}
  \node (r){root}
    [ultra thick, dashed, red]
    child { node{r-1}
              child {node{r-1-1}}
              child {node{r-1-2}}}
    child {node{r-2}};
  \draw [->, green] (r-1-1)--(r-1-2);
\end{tikzpicture}
```

4. 针对某个分支的选项要写在产生分支的 child 操作之后，对该分支内的所有子节点以及与子节点相关的边有效。




```
\begin{tikzpicture}
  \node (r){root}
    child [ultra thick, dashed, red] { node{r-1}
      child {node{r-1-1}}
      child {node{r-1-2}}}
    child {node{r-2}};
  \draw [->, green] (r-1-1)--(r-1-2);
\end{tikzpicture}
```

5. 针对单个子节点的选项要写在创建该子节点的 ⟨*child path*⟩ 中的 node 操作之后，只针对该子节点有效，不针对与该节点相关的边。




```
\begin{tikzpicture}
  \node (r){root}
    child  {node [ultra thick, dashed, red] {r-1}
      child {node{r-1-1}}
      child {node{r-1-2}}}
    child {node{r-2}};
  \draw [->, green] (r-1-1)--(r-1-2);
\end{tikzpicture}
```

/tikz/every child (style, initially empty)

使用这个选项前应当先用手柄 `/.style`, `/.code` 定义这个选项，定义中不应当使用变量符号。这个样式为每个 child 子节点设置样式，此样式起作用的位置大致相当于：

`child[every child/.try,`⟨*options*⟩`]{...}`

先执行 `\tikzset{every child/.try,`⟨*options*⟩`}`, 再用命令 `\node` 创建节点标签。

/tikz/every child node (style, initially empty)

使用这个选项前应当先用手柄 `/.style`, `/.code` 定义这个选项，定义中不应当使用变量符号。这个选项针对的是 ⟨*child path*⟩ 中的第一个 node, 第一个 node 由命令：

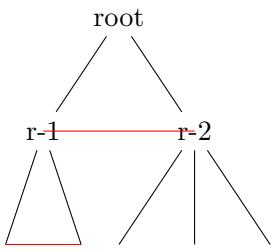
```
\node [name=\tikzparentnode-\the\tikznumberofcurrentchild,style=every child
↪  node][...]{...};
```

创建。这个样式会在 `every node` 设置的样式之后起作用。

/tikz/level=⟨*number*⟩ (no default, initially empty)

使用这个选项前应当先用手柄 `/.style`, `/.code` 定义这个选项，定义中应当使用一个变量符号 `#1`, 这个变量符号代表的是当前 level 的编号，第一个 level 的编号是 1. 每当创建 tree 的一个 level 之前都会执行这个选项。

观察下面的例子。

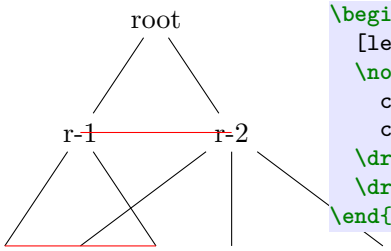



```
\begin{tikzpicture}[level/.style={sibling distance=20mm/#1}]
  \node (r){root}
    child { node{r-1} child child }
    child { node{r-2} child child child };
  \draw [red] (r-1.center)--++(20mm,0);
  \draw [red] (r-1-1.center)--++(20mm/2,0);
\end{tikzpicture}
```

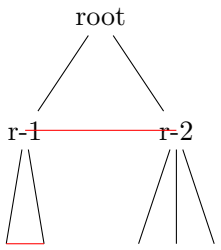
在上面例子中，样式选项 `level/.style` 设置了一个样式，其中包含一个变量 `#1`；对于根节点之下的第一层，程序会自动使用选项 `level=1`，即 `#1=1`, 使得第一层的相邻两个子节点的（中心）间距等于 `20mm/1`；对于根节点之下的第二层，程序会自动使用选项 `level=2`，即 `#1=2`, 使得第二层的相邻两个子节点的（中心）间距等于 `20mm/2`.

/tikz/level ⟨*number*⟩ (style, initially empty)

使用这个选项前应当先用手柄 /.style, /.code 定义这个选项，定义中不应当使用变量符号。在创建 tree 的第 ⟨*number*⟩ 个 level 之前会执行这个选项，对第 ⟨*number*⟩ 层以及之后的各层有效。




```
\begin{tikzpicture}
  [level 1/.style={sibling distance=20mm},]
  \node (r){root}
    child { node{r-1} child child }
    child { node{r-2} child child child };
  \draw [red] (r-1.center)--++(20mm,0);
  \draw [red] (r-1-1.center)--++(20mm,0);
\end{tikzpicture}
```




```
\begin{tikzpicture}
  [level 1/.style={sibling distance=20mm},
   level 2/.style={sibling distance=5mm}]
  \node (r){root}
    child { node{r-1} child child }
    child { node{r-2} child child child };
  \draw [red] (r-1.center)--++(20mm,0);
  \draw [red] (r-1-1.center)--++(5mm,0);
\end{tikzpicture}
```

51.5 子节点的位置

51.5.1 基本的处理流程

TikZ 在处理 tree 时，先保存整个树的结构，将所有节点组织成一个列表。事先创建一个 node 或 coordinate 作为根节点 ⟨*root*⟩, 用 \setbox 命令将根节点的所有子节点排布到盒子 \tikz@whichbox 中。

对于每一个父节点 ⟨*parent*⟩:

1. 从当前的父节点 ⟨*parent*⟩ 开始，记录由此生长的所有分支的个数，即 ⟨*parent*⟩ 的子节点的个数 (计数器 \tikznumberofchildren)，并保存由此生长的各个分支
2. 确定父节点 ⟨*parent*⟩ 之后的那一 level 的编号 (计数器 \tikztreelevel, 编号从 1 开始)，然后执行与这一 level 有关的选项
3. 用宏 \tikzparentnode 非全局地保存父节点 ⟨*parent*⟩ 的名称
4. 将当前父节点 ⟨*parent*⟩ 的第一、第二……分支的节点依次排布，在排布第 i 个分支的节点时：
   (a) 确定父节点 ⟨*parent*⟩ 的第 i 个子节点的编号 (计数器 \tikznumberofcurrentchild, 编号从 1 开始)，
   (b) 用 \setbox 命令定义盒子 \tikz@whichbox 的内容如下：
      i. 执行与 child 有关的选项
      ii. 执行“生长函数” \tikz@grow →P. 932, 这个函数一般包含 PGF 层的变换命令，对即将创建的子节点有影响，即决定该子节点的位置。
      iii. 用 \node 命令创建父节点 ⟨*parent*⟩ 的第 i 个子节点 ⟨*parent-i*⟩, 这个子节点接受变换命令，被放到预先计算好的位置上。
      iv. 用宏 \tikzchildnode 非全局地保存子节点 ⟨*parent-i*⟩ 的名称
      v. 然后设置一个花括号组，在组内将 ⟨*parent-i*⟩ 作为父节点 ⟨*parent*⟩, 重复以上步骤，即可完成第 i 个分支的节点排布。
5. 最后用命令 edge from parent 创建父子节点之间的边；在之前步骤排布节点时，有花括号组的多层套嵌，每一层组内都有一个被花括号组限制的 edge from parent 命令，按由内到外的层次执行

画边命令。

每个 child{node} 命令都会使用盒子，并在盒子内使用花括号组，所以 TikZ 在解析 tree 的句法时，会创建复杂的套嵌盒子，盒子内又有复杂的花括号组套嵌。例如，下面的结构：

```
child { node {root-1}}
child { node {root-2}
        child {node {root-2-1}}
        child {node {root-2-2}}
}
```

大体上是：

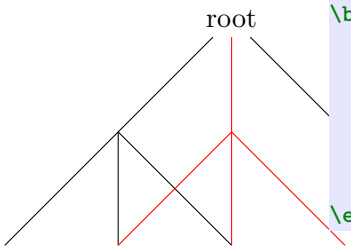
```
\setbox\tikz@whichbox=\hbox\bgroup
  \hbox\bgroup
    {
      ...
      \node {root-1};
      ...
      \path edge from parent;% 将 root-1 与其父节点连起来
    }
  \egroup
  \hbox\bgroup
    {
      ...
      \node {root-2};
      {
        \setbox\tikz@whichbox=\hbox\bgroup
          \hbox\bgroup
            {
              \setbox\tikz@whichbox=\hbox\bgroup
                \hbox\bgroup{
                    ...
                    \node {root-2-1};
                    ...
                    {\path edge from parent;}% 将 root-2-1 与其父节点连起来
                  }\egroup
                \hbox\bgroup{
                    ...
                    \node {root-2-2};
                    ...
                    {\path edge from parent;}% 将 root-2-2 与其父节点连起来
                  }\egroup
              \egroup
            }
          \egroup
        \egroup
        ⟨ 命令 edge from parent, 将 root-2 与其父节点连起来⟩
      }
    }
  \egroup
\egroup
```

在默认下有

```
\def\tikz@whichbox{\tikz@figbox}%
```

结束路径的分号“;”会导致执行 \tikz@finish →P. 759, 这个命令会释放盒子 \tikz@figbox 的内容。也就是说，tree 的子节点与 node, edge 路径一样，都是主路径的“附加物”。

注意在确定节点位置时不考虑节点的尺寸，因此节点之间可能会发生重叠。

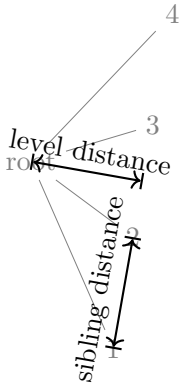



```
\begin{tikzpicture}
  \node {root}
    child { child child child
    }
    child [red] { child child child
    }
    child;
\end{tikzpicture}
```

树的层间距，一层内相邻两个节点之间的间距，树的生长方向，树的某个分支的生长方向都是可以调节的，也可以使用平移选项来调整单个节点的位置。使用这些调节方法可以避免节点发生重叠。

51.5.2 默认的生长函数

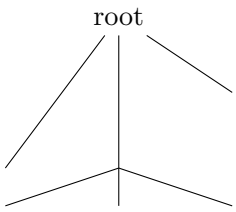
在默认之下，树的第一层节点位于水平直线 l_1 上，第二层节点位于 l_1 之下的水平直线 l_2 上，第三层节点位于 l_2 之下的水平直线 l_3 上……，从直线 l_1 到直线 l_2 到直线 l_3 的方向（竖直向下的方向）是树的生长方向。从根节点到直线 l_1 的距离，从直线 l_1 到直线 l_2 的距离……都叫作层间距（level distance）。一层之内相邻两个节点的间距叫作 sibling distance.



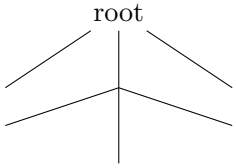

```
\begin{tikzpicture}[sibling distance=15mm, level distance=15mm]
  \path [help lines] node (root) {root}
    [grow=-10]
    child {node {1}}
    child {node {2}}
    child {node {3}}
    child {node {4}};
  \draw[|<->|,thick] (root-1.center) -- node[above,sloped]
    {sibling distance} (root-2.center);
  \draw[|<->|,thick] (root.center) -- node[above,sloped]
    {level distance} +(-10:\tikzleveldistance);
\end{tikzpicture}
```

/tikz/level distance=⟨*distance*⟩ (no default, initially 15mm)

本选项按照它的有效范围设置层间距。初始值为 15cm.




```
\begin{tikzpicture}
  \node {root} [level distance=20mm] % 为所有分支设置层间距
    child
    child { [level distance=5mm] % 为本分支设置层间距
      child child child
    }
    child[level distance=10mm]; % 为本分支设置层间距
\end{tikzpicture}
```




```
\begin{tikzpicture}
  [level 1/.style={level distance=10mm},
   level 2/.style={level distance=5mm}]
  \node {root}
    child
    child { child
            child[level distance=10mm]
            child
    }
    child;
\end{tikzpicture}
```

从以上例子看出，当用本选项为某个分支设置层间距时，凡是与本分支内的节点相关的层间距都被本选项影响到了。

本选项的定义是 (见《tikz.code.tex》):

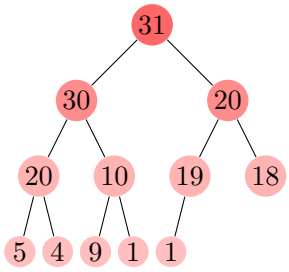
```
\newdimen\tikzleveldistance
\newdimen\tikzsiblingdistance
%......
\tikzoption{level distance}{\pgfmathsetlength\tikzleveldistance{#1}}%
\tikzoption{sibling distance}{\pgfmathsetlength\tikzsiblingdistance{#1}}%
%......
\tikzleveldistance=15mm%
\tikzsiblingdistance=15mm%
```

注意执行命令 `\pgfkeysvalueof{/tikz/level distance}` 不能得到此选项的值 15mm, 参考 \tikzoption→P. 676. 从定义看,在执行 `\tikzset{level distance={⟨expression⟩}}` 时,参数 ⟨*expression*⟩ 会被命令 \pgfmathsetlength 处理，处理结果带上单位 pt 后赋予尺寸寄存器 `\tikzleveldistance`, 此时可以利用这个寄存器值。参数 ⟨*expression*⟩ 可以是比较复杂的表达式。

/tikz/sibling distance=⟨*distance*⟩ (no default, initially 15mm)

参考上一选项。本选项按照它的有效范围设置一层之内相邻两个节点的间距。初始值为 15cm.

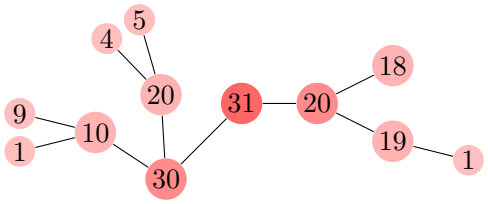
```
\begin{tikzpicture}
  [level distance=4mm,
   level 1/.style={sibling distance=8mm},
   level 2/.style={sibling distance=4mm},
   level 3/.style={sibling distance=2mm}]
  \coordinate
    child { child {child child} child {child child} }
    child { child {child child} child {child child} };
\end{tikzpicture}
```




```
\begin{tikzpicture}[level distance=10mm,
    every node/.style={fill=red!60,circle,inner sep=1pt},
    level 1/.style={sibling distance=20mm,nodes={fill=red!45}},
    level 2/.style={sibling distance=10mm,nodes={fill=red!30}},
    level 3/.style={sibling distance=5mm,nodes={fill=red!25}}]
  \node {31}
    child {node {30}
      child {node {20}
        child {node {5}} child {node {4}} }
      child {node {10}
        child {node {9}} child {node {1}} }
    }
    child {node {20}
      child {node {19}
        child {node {1}} child[missing] }
      child {node {18}}
    };
\end{tikzpicture}
```

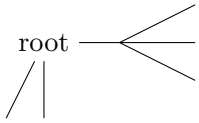
/tikz/grow=⟨*direction*⟩ (no default)

按照本选项的有效范围，本选项确定树或者某个分支的生长方向。⟨*direction*⟩ 可以是一个角度，也可以是单词: down, up, left, right, north, south, east, west, north east, north west, south east, south west.

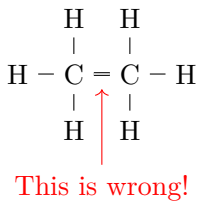



```
\begin{tikzpicture}[level distance=10mm,
    every node/.style={fill=red!60,circle,inner sep=1pt},
    level 1/.style={sibling distance=20mm,nodes={fill=red!45}},
    level 2/.style={sibling distance=10mm,nodes={fill=red!30}},
    level 3/.style={sibling distance=5mm,nodes={fill=red!25}}]
  \node {31}
    child {node {30} [grow=120]
      child {node {20}
        child {node {5}} child {node {4}} }
      child {node {10} [grow=180]
        child {node {9}} child {node {1}} }
    }
    child [grow=0]{node {20}
      child {node {19}
        child {node {1}} child[missing] }
      child {node {18}}
    };
\end{tikzpicture}
```

从上面例子中看出，当本选项用在某个 child 之前（`[grow=120] child`）时，对本选项之后的、本选项所在分支内的节点有效；当本选项用在某个 child 之后（`child [grow=0]`）时，对该 child 以及本选项之后的、本选项所在分支内的节点有效。

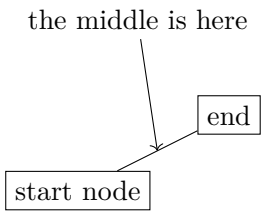



```
\begin{tikzpicture}[level distance=10mm,sibling distance=5mm]
  \node {root}
    [grow=down]
    child
    child
    child[grow=right] {
      child child child
    };
\end{tikzpicture}
```




```
\begin{tikzpicture}[level distance=2em]
  \node {C}
    child[grow=up] {node {H}}
    child[grow=left] {node {H}}
    child[grow=down] {node {H}}
    child[grow=right] {node {C}
        child[grow=up] {node {H}}
        child[grow=right] {node {H}}
        child[grow=down] {node {H}}
      edge from parent[double] coordinate (wrong)
    };
  \draw[<-,red] ([yshift=-2mm]wrong) -- +(0,-1)
    node[below]{This is wrong!};
\end{tikzpicture}
```

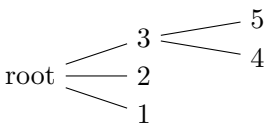
这个例子中的 `edge from parent[double] coordinate (wrong)` 创建的边是从左边的 C 到右边的 C 之间的边，这个边还附带一个 `coordinate (wrong)` 标签（即形状是 `coordinate` 的 node），这个标签在边的中点处，名称是 `(wrong)`.




```
\begin{tikzpicture}
  \node[rectangle,draw] (a) at (0,0) {start node};
  \node[rectangle,draw] (b) at (2,1) {end};
  \draw (a) -- (b)
    node[coordinate,midway] {}
      child[grow=100,<-] {node[above] {the middle is here}};
\end{tikzpicture}
```

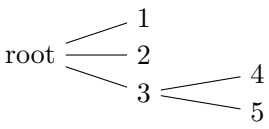
这个例子中的 \draw 命令在 (a), (b) 之间连线；连线的中点处有一个 node；在这个 node 之后使用一个 child 操作创建一个树的节点；这个 node 是父节点，child 创建子节点；选项 <- 给从父节点到子节点的边加箭头，且是在父节点处加箭头。

观察下面的例子：




```
\begin{tikzpicture}
  \node {root} [grow=right,sibling distance=5mm]
    child {node{1}}
    child {node{2}}
    child {node{3}
      child {node{4}}
      child {node{5}}
    };
\end{tikzpicture}
```

上面例子使用 grow=right, 同一层上的节点按代码次序自下而上排放，为了让同一层上的节点自上而下排放，可以用选项 yscale=-1, 对比：




```
\begin{tikzpicture}
  \node {root} [grow=right,sibling distance=5mm,yscale=-1]
    child {node{1}}
    child {node{2}}
    child {node{3}
      child {node{4}}
      child {node{5}}
    };
\end{tikzpicture}
```

本选项的定义是：

```
\tikzoption{grow}{\tikz@set@growth{#1}\edef\tikz@special@level{\the\tikztreelevel
↪  }}%
```

本选项指定命令 \tikz@grow 以及相关的变量。

《tikz.code.tex》中自己执行 \tikzset{grow=down}, 所以默认了向下生长的函数。

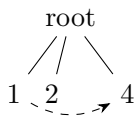
/tikz/grow'=⟨*direction*⟩ (no default)

这个选项决定的生长方向是 grow=⟨*direction*⟩ 的反方向。

51.5.3 缺失的节点

/tikz/missing=⟨*true or false*⟩ (default true)

当某个 child 带有这个选项后（child[missing]），这个 child 的内容、与之相关的边、以及以此节点为生长点的分支都不会被画出，它的名称（默认的节点名称或者手工设置的名称）也被看作是无效名称，但这个 child 仍然会被 Ti*k*Z 统计在内，因此它会占据一个节点位置。




```
\begin{tikzpicture}[level distance=10mm,sibling distance=5mm]
  \node (r) {root} [grow=down]
    child { node {1} }
    child { node {2} }
    child [missing] { node {3} child }
    child { node {4} };
  \draw [dashed, -Stealth] (r-1) to [bend right] (r-4);
\end{tikzpicture}
```

51.5.4 自定义生长函数

\tikz@grow

这是 TikZ 的内部命令，它就是所谓的“生长函数”。通常这个宏会被 let 为其他宏，它一般保存变换命令，在创建子节点 node 的命令 \node 之前被执行，使得子节点 node 被变换。

TikZ 本身会默认执行 \tikzset{grow=down}, 这导致

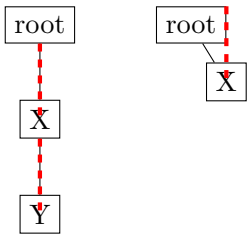
```
\let\tikz@grow=\tikz@grow@direction
```

也就是说，TikZ 默认的生长函数是 \tikz@grow@direction, 这个函数需要某些变量的配合。这个函数有可定制性，即可以使用选项 grow, grow', level distance, sibling distance 来修改这个函数用到的变量。

/tikz/growth parent anchor=⟨*anchor*⟩ (no default, initially center)

本选项影响层间距的计算方式。

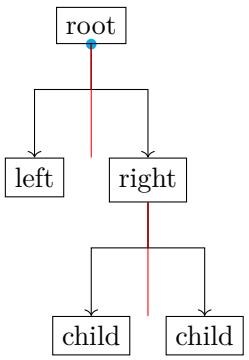
本选项的初始值 center 指的是父节点的 (P) 的中心位置（一个 anchor 位置）；在计算 (P) 的子节点 (P-1), (P-2), (P-3)... 的位置时，以父节点的锚位置 (P.center) 为参照点；例如，从父节点 (P) 到子节点 (P-1) 所在层的层间距要以 (P.center) 为起点来度量。将 ⟨*anchor*⟩ 换做其它锚位置后，作用是类似的。




```
\begin{tikzpicture}[level distance=1cm]
  \node [rectangle,draw] (a) at (0,0) {root}
    [growth parent anchor=south]
    child {node (X) [draw] {X} child {node (Y) [draw] {Y}}};
  \node [rectangle,draw] (b) at (2,0) {root}
    [growth parent anchor=north east] child {node [draw] {X}};
  \draw [red,ultra thick,dashed] (a.south) -- ++(0,-1);
  \draw [red,ultra thick,dashed] (X.south) -- ++(0,-1);
  \draw [red,ultra thick,dashed] (b.north east) -- ++(0,-1);
\end{tikzpicture}
```

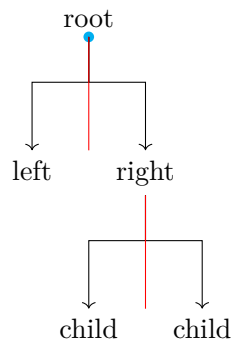
在这个例子中，从 (a.south) 到 (X.center) 的距离是选项 level distance=1cm 设置的 1cm；从 (X.south) 到 (Y.center) 的距离也是 1cm.

当希望以节点的边界来度量层间距时，可以配合使用选项 growth parent anchor 和样式 every child 或 every child node 或 every node, 例如配合 every node:



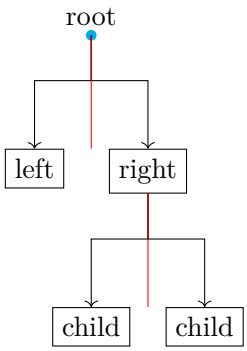

```
\begin{tikzpicture}[
    level distance=15mm, sibling distance=15mm,
    growth parent anchor=south,
    edge from parent/.style={draw,->},
    edge from parent path= {(\tikzparentnode.south)
        --++(-90:6mm) -| (\tikzchildnode.north)}]
  {[every node/.style={draw,anchor=north}]
  \fill[cyan] circle (2pt); % 注意原点位置与 (root) 的 south 重合
  \node (root)[anchor=south]{root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };}
  \draw [red](root.south)--++(-90:15mm);% 红线长 level distance=15mm
  \draw [red](root-2.south)--++(-90:15mm);
\end{tikzpicture}
```

配合 every child:




```
\begin{tikzpicture}[
    level distance=15mm, sibling distance=15mm,
    growth parent anchor=south,
    edge from parent/.style={draw,->},
    edge from parent path= {(\tikzparentnode.south)
        --++(-90:6mm) -| (\tikzchildnode.north)}]
  {[every child/.style={anchor=north,draw=black,}]
  \fill[cyan] circle (2pt); %
  \node (root)[anchor=south]{root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };}
  \draw [red](root.south)--++(-90:15mm);% 红线长 level distance=15mm
  \draw [red](root-2.south)--++(-90:15mm);
\end{tikzpicture}
```

注意上面代码中的 `draw=black` 没有起作用，下面配合 `every child node` 有作用：




```
\begin{tikzpicture}[
    level distance=15mm, sibling distance=15mm,
    growth parent anchor=south,
    edge from parent/.style={draw,->},
    edge from parent path= {(\tikzparentnode.south)
        --++(-90:6mm) -| (\tikzchildnode.north)}]
  {[every child node/.style={anchor=north,draw=black,}]
  \fill[cyan] circle (2pt); %
  \node (root)[anchor=south]{root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };}
  \draw [red](root.south)--++(-90:15mm);% 红线长 level distance=15mm
  \draw [red](root-2.south)--++(-90:15mm);
\end{tikzpicture}
```

本选项的定义是：

```
\tikzoption{growth parent anchor}{\def\tikz@growth@anchor{#1}}%
\def\tikz@growth@anchor{center}%
```

在命令 \tikz@children@collected →P. 942 中会执行

```
\ifx\tikz@grow\relax
\else%
  \pgftransformshift{\pgfpointanchor{\tikzparentnode}{\tikz@growth@anchor}}%
\fi%
```

所以，无论定义什么样的生长函数，这个选项总是起作用的。

/tikz/growth function=⟨*macro name*⟩ (no default, initially an internal function)

这是个底层选项，允许你自定义一个生长函数，初始的生长函数会使得树向下生长。

本选项的定义是：

```
\tikzoption{growth function}{\let\tikz@grow=#1}%
```

TikZ 用 \node 命令创建 ⟨*child path*⟩ 中的第一个 node 子节点前，先执行宏 \tikz@grow →P. 932，通常这个宏保存的是决定这个子节点位置的变换命令。

⟨*macro name*⟩ 是一个不带参数的宏，这个宏会被用于处理每个节点。这个宏应当具有这样的作用：调用这个宏后，当前坐标系的原点应当被平移到当前父节点的某个（指定的）锚位置上；然后在这个坐标系内确定该父节点的各个子节点的位置。也就是说，这个宏其实就是前面提到的“延伸模式”的一个具体实现。

在这个宏的定义中可以使用 \tikznumberofchildren 和 \tikznumberofcurrentchild，这是两个 TEX 计数器，前者的值是当前父节点的子节点的个数，后者的值是当前子节点的序号。
在这个宏的定义中可以使用坐标变换命令。
程序库 trees 定义了一些生长函数，见 §72. 例如文件《tikzlibrarytrees.code.tex》中有如下定义：

```
\tikzstyle{grow cyclic}=[growth function=\tikz@grow@circle]

\tikzset{sibling angle/.initial=20}

\def\tikz@grow@circle{%
  \pgftransformrotate{%
    (\pgfkeysvalueof{/tikz/sibling angle})*(-.5-.5*\tikznumberofchildren+
    ↪  \tikznumberofcurrentchild)}%
  \pgftransformxshift{\the\tikzleveldistance}%
}
```

51.6 从父节点到子节点的边

每个子节点都会与其父节点相连，边的方向是从父节点到子节点。
字符串 edge from parent 有 2 个意义：

- 类似 to, 画一个线条，将子节点与其父节点连起来。
- 可以用作一个选项，使用这个选项前应当先用手柄 /.style，/.code 定义这个选项，定义中不应当使用变量符号。

\path ... edge from parent[⟨*options*⟩] ⟨*node specification*⟩ ... ;

这是个画边的命令，在下文称这个命令为“edge from parent 命令”，它只能用在 ⟨*child path*⟩ 中。当 Ti*k*Z 在路径命令 \path 中遇到字符串 edge from parent 时，就会执行命令 \tikz@edgetoparent. 本命令所处的花括号组内的第一个 child 标签 (即第一个 node 语句创建的标签) 会被本命令“特别关注”，本命令会在“特别关注的 child 标签”与其父节点之间连线画边。也就是说，在

child {⟨*node specification*⟩ edge from parent}

中，把 ⟨*node specification*⟩ 设置的 node 标签看作子节点，与其上一级的父节点标签连线。而在

child {⟨*node specification 1*⟩ edge from parent [⟨*options*⟩] ⟨*node specification 2*⟩}

中，⟨*node specification 2*⟩ 作为 edge from parent 命令的参数，是边的标签 (默认这个标签位于边的中点处)。
如果 ⟨*child path*⟩ 中没有手工写出这个命令，Ti*k*Z 会自动添加这个命令来画边。

命令

\path ... edge from parent [⟨*options*⟩] ⟨*node specification*⟩}

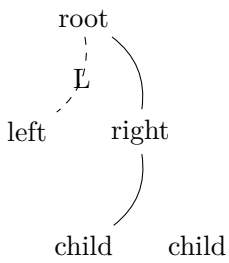
等价于

```
\path ...\tikz@scan@next@command⟨ 键 /tikz/edge from parent macro 保存的命令⟩{⟨options⟩}{
↪  ⟨node specification⟩}
```

在初始之下，键 /tikz/edge from parent macro 保存的命令是 \tikz@edge@from@parent@macro.

/tikz/edge from parent macro=⟨*macro*⟩

这个选项保存宏 ⟨*macro*⟩. 每当执行命令 edge from parent 时，这个选项保存的宏会被调用。如果要用本选项引入某个 ⟨*macro*⟩, 那么 ⟨*macro*⟩ 可以如下定义：




```
\def\mymacro#1#2{
  [style=edge from parent, #1]
  (\tikzparentnode\tikzparentanchor) to[bend left] #2 (
  ↪  \tikzchildnode\tikzchildanchor)
}
\begin{tikzpicture}
  \node {root}[edge from parent macro=\mymacro]
    child {node {left}edge from parent[dashed,] node{L}}
    child {node {right}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
    };
\end{tikzpicture}
```

如上面的例子所示：

- ⟨*macro*⟩ 的定义必须使用两个变量 `#1` 和 `#2`；第一个变量 `#1` 代表那些位于命令 `edge from parent` 之后的选项，即 ⟨*options*⟩；第二个变量 `#2` 代表命令 `edge from parent` 之后的 ⟨*node specifications*⟩ 部分，在 TikZ 读取 (处理、转存) 这个 ⟨*node specification*⟩ 后，才会执行 ⟨*macro*⟩ 画边。
- 最好不要在命令 `edge from parent [`⟨*options*⟩`]` 的 ⟨*options*⟩ 中使用本选项。

本选项的定义是：

```
\tikzset{edge from parent macro/.initial=\tikz@edge@from@parent@macro}%
\def\tikz@edge@from@parent@macro#1#2{
  [style=edge from parent, #1, /utils/exec=\tikz@node@is@a@labeltrue]
  ↪  \tikz@edge@to@parent@path #2}%
```

从这个定义看，选项 `edge from parent macro` 的初始值是命令 `\tikz@edge@from@parent@macro`，这个命令会引入选项 `edge from parent path` 保存的绘制边的命令，即 `\tikz@edge@to@parent@path`；如果打算把 `edge from parent macro` 的值定义为其他的 ⟨*macro*⟩，那么要注意这两个选项的配合。也可以仔细设计 ⟨*macro*⟩ 的定义，使之不依赖选项 `edge from parent path` 保存的命令。

`/tikz/edge from parent path=`⟨*path*⟩ (no default, initially code shown below)

这个选项可以重定义绘制边的代码。本选项的定义是

```
\tikzoption{edge from parent path}{\def\tikz@edge@to@parent@path{#1}}%
```

可见执行本选项导致重定义宏 `\tikz@edge@to@parent@path`:

```
\def\tikz@edge@to@parent@path{⟨path⟩}%
```

这个宏的最初定义是

```
\def\tikz@edge@to@parent@path{(\tikzparentnode\tikzparentanchor) -- (
↪  \tikzchildnode\tikzchildanchor)}%
```

也就是说，初始之下的 ⟨*path*⟩ 是

```
(\tikzparentnode\tikzparentanchor) -- (\tikzchildnode\tikzchildanchor)
```

默认下的 ⟨*path*⟩ 实际是

```
(\tikzparentnode) -- (\tikzchildnode)
```

因此默认下的 ⟨*path*⟩ 是从父节点的中心到子节点的中心的。

在参数 ⟨*path*⟩ 中可以使用下面的宏：

`\tikzchildnode`

这个宏保存的是当前节点 (看作是子节点) 的名称，如果不手工为这个节点命名，就使用内部默认的名称，也就是 ⟨父节点名称⟩`-1-2` 之类的名称。注意这个宏保存的是 node 的名称，也就是在

`child {`⟨*first node specification*⟩ ⟨*others*⟩`}`

中的第一个 node 语句 ⟨*first node specification*⟩ 所创建的 node 的名称，所以在 ⟨*first node specification*⟩ 中不能使用这个宏，可以用在 ⟨*others*⟩ 中。

\tikzparentnode

这个宏保存的是当前节点的父节点 (一个 node) 的名称，如果不手工为这个节点命名，就使用内部默认的名称。

\tikzchildanchor

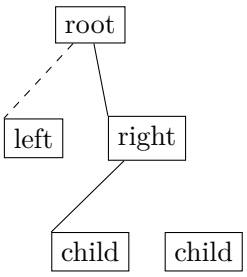
这个宏保存的是当前节点 (看作是子节点) 的锚位置，在默认下这宏是空的。可以重定义它：

```
\def\tikzchildanchor{.north}
```

所以 (\tikzchildnode\tikzchildanchor) 就是子节点的 node 坐标系中的一个点。

/tikz/child anchor=⟨*anchor*⟩ (no default, initially border)

这个选项把 \tikzchildanchor 设置为 .⟨*anchor*⟩. 本选项的初始值 border 会把 \tikzchildanchor 设置成空的。




```
\begin{tikzpicture}[every node/.style=draw]
  \node {root} [child anchor=north west]
    child {node {left} edge from parent[dashed]}
    child {node {right}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
  };
\end{tikzpicture}
```

注意本选项的定义是：

```
\tikzoption{child anchor}{\def\tikzchildanchor{.#1}
↪  \ifx\tikzchildanchor\tikz@border@text\let\tikzchildanchor\pgfutil@empty
↪  \fi}%
\def\tikz@border@text{.border}%
```

关于本选项的注意事项参考命令 \tikzoption→P. 676.

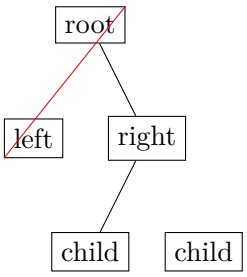
\tikzparentanchor

这个宏保存的是当前节点的父节点的锚位置，在默认下这宏是空的。可以重定义它：

```
\def\tikzparentanchor{.north}
```

/tikz/parent anchor=⟨*anchor*⟩ (no default, initially border)

这个选项把 \tikzparentnode 设置为 .⟨*anchor*⟩. 本选项的初始值 border 会把 \tikzparentnode 设置成空的。




```
\begin{tikzpicture}[every node/.style=draw]
  \node {root} []
    child {node {left}
      edge from parent[child anchor=south west,
        parent anchor=north east, red]}
    child {node {right}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
  };
\end{tikzpicture}
```

注意本选项的定义是：

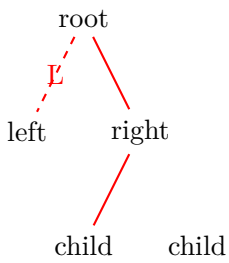
```
\tikzoption{parent anchor}{\def\tikzparentanchor{.#1}
↪  \ifx\tikzparentanchor\tikz@border@text
↪  \let\tikzparentanchor\pgfutil@empty\fi}%
```

```
\def\tikz@border@text{.border}%
```

关于本选项的注意事项参照上一选项。

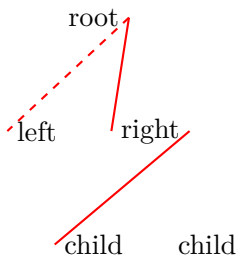
/tikz/edge from parent (style, initially draw)

使用这个选项前应当先用手柄 /.style, /.code 定义这个选项，定义中不应当使用变量符号。这个选项一般用作命令 edge from parent 的样式选项，针对子节点与父节点之间的边，例如前面的命令 \tikz@edge@from@parent@macro 利用了这个选项。



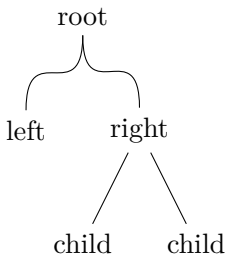

```
\begin{tikzpicture}
  [edge from parent/.style={draw,red,thick}]
  \node {root}
    child {node {left} edge from parent[dashed] node{L}}
    child {node {right}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
    };
\end{tikzpicture}
```

下面的例子中重定义了父节点和子节点的锚位置：

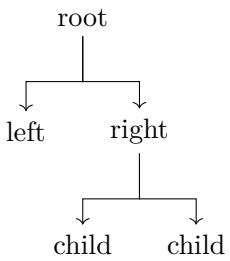



```
\begin{tikzpicture} [edge from parent/.style={draw,red,thick}]
  \def\tikzparentanchor{.east}
  \def\tikzchildanchor{.west}
  \node {root}
    child {node {left} edge from parent[dashed]}
    child {node {right}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
    };
\end{tikzpicture}
```

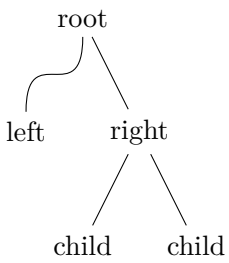
下面的例子中重定义了 ⟨*path*⟩：




```
\begin{tikzpicture}[level distance=15mm, sibling distance=15mm,
  my edge style/.style={edge from parent path=
    {(\tikzparentnode.south) .. controls +(0,-1) and +(0,1)
     .. (\tikzchildnode.north)}}]
  \node {root}
    child {node {left} edge from parent[my edge style]}
    child {node {right}
        child {node {child}}
        child {node {child}}
      edge from parent[my edge style]};
\end{tikzpicture}
```




```
\begin{tikzpicture}[level distance=15mm, sibling distance=15mm,
    edge from parent path= {[->](\tikzparentnode.south)
        --++(-90:6mm) -| (\tikzchildnode.north)}]
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```




```
\begin{tikzpicture}[level distance=15mm, sibling distance=15mm]
  \node {root}
    child {node {left}
      edge from parent[edge from parent path=
        {(\tikzparentnode.south) .. controls +(0,-1) and +(0,1)
         .. (\tikzchildnode.north)}]}
    child {node {right}
       child {node {child}}
       child {node {child}}
  };
\end{tikzpicture}
```

程序库 trees 定义了一些 `edge from parent path`.

51.7 注意的问题

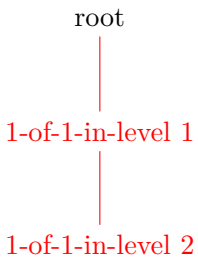
51.7.1 与子节点有关的计数器

以下计数器与根节点无关，只与子节点有关：

- `\tikztreelevel`, 当前 level 的编号，编号从 1 开始
- `\tikznumberofchildren`, 当前父节点之下的子节点的个数，即当前父节点之下的 level 内的节点个数
- `\tikznumberofcurrentchild`, 当前节点在“当前父节点之下的诸子节点中的”序号 (即当前父节点之下的 level 内的诸子节点中的序号)，序号从 1 开始

以上计数器可以用在：

◇ `/tikz/growth function`→P. 933=⟨*macro name*⟩ 的宏 ⟨*macro name*⟩ 的定义中
◇ `/tikz/edge from parent path`→P. 935=⟨*path*⟩ 的 ⟨*path*⟩ 的定义中
◇ `/tikz/edge from parent macro`→P. 934=⟨*macro*⟩ 的宏 ⟨*macro*⟩ 的定义中
◇ `child` [⟨*options 1*⟩] `{node` [⟨*options 2*⟩] `{`⟨*text*⟩`}}` 中
◇ 命令 `edge from parent` [⟨*options*⟩] ⟨*node specification*⟩ 中




```
\tikz{
  \node {root}
    child [red] {
      node {\the\tikznumberofcurrentchild-of-
      ↪  \the\tikznumberofchildren-in-level \the\tikztreelevel}
      child {
        node {\the\tikznumberofcurrentchild-of-
        ↪  \the\tikznumberofchildren-in-level \the\tikztreelevel}
      }
  };
}
```

51.7.2 节点名称

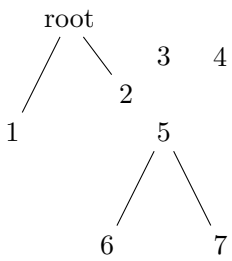
关于节点名称：

- 用户可以使用 (⟨*node name*⟩) 为各个节点命名
- 当前父节点的名称 (非全局地) 保存在宏 `\tikzparentnode` 中
- 之前创建的一个节点名称 (全局地) 保存在宏 `\tikz@last@fig@name` 中
- 用 `\node` 创建一个子节点后，这个子节点的名称就 (非全局地) 保存在宏 `\tikzchildnode` 中
- 如果用户没有使用 (⟨*node name*⟩) 为子节点命名，那么 TikZ 自动为子节点命名，即 (⟨*parent*⟩`-1-2-3`) 之类的名称

已创建的节点名称可以用在：

◇ /tikz/growth function→P. 933=⟨*macro name*⟩ 的宏 ⟨*macro name*⟩ 的定义中
◇ /tikz/edge from parent path→P. 935=⟨*path*⟩ 的 ⟨*path*⟩ 的定义中
◇ /tikz/edge from parent macro→P. 934=⟨*macro*⟩ 的宏 ⟨*macro*⟩ 的定义中
◇ child [⟨*options 1*⟩] {node [⟨*options 2*⟩] {⟨*text*⟩}} 中
◇ 命令 edge from parent [⟨*options*⟩] ⟨*node specification*⟩ 中

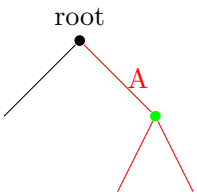
51.7.3 node 影响 child 的位置




```
\begin{tikzpicture}
\node (root){root}[level distance=15mm,]
  child{node{1}}
  child{node[shift={(0,5mm)}]{2}
    node(3)[at={([shift={(5mm,5mm)}]\tikzchildnode)}]{3}
    node[right=3mm of 3]{4}
    node[at={($(\tikzchildnode)+(5mm,-5mm)$)}]{5}
    child{node{6}}
    child{node{7}}
  };
\end{tikzpicture}
```

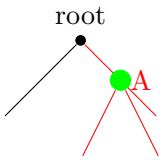
上面例子中，最后两个 child 以之前最近出现的 node 为父节点。

使用命令 edge from parent[⟨*options*⟩] ⟨*node specification*⟩ 时要注意它的位置，因为句式中的 ⟨*node specification*⟩ 会影响树的结构，比较下面几个例子：



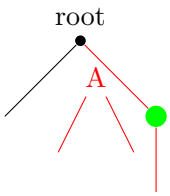

```
\begin{tikzpicture}
[level distance=10mm,level/.style={sibling distance=20mm/#1}]
\node [circle,inner sep=0pt,minimum size=4pt,fill,label=90:root]{}
  child {}
  child { node [circle,inner sep=0pt,minimum size=4pt,fill=green] {}
    child
    child
    edge from parent[red] node[right] {A}
  };
\end{tikzpicture}
```

上面结构是正常的，但下面的结构不是正常的：



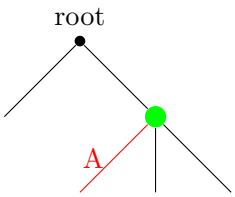

```
\begin{tikzpicture}
[level distance=10mm,level/.style={sibling distance=20mm/#1}]
\node [circle,inner sep=0pt,minimum size=4pt,fill,label=90:root]{}
  child {}
  child {
    edge from parent[red] node[right] {A}
    node [circle,inner sep=0pt,minimum size=8pt,fill=green] {}
    child
    child
  };
\end{tikzpicture}
```

在上面这个结构中，命令 edge from parent[red] 后的两个 node 都是边的标签，最后两个 child 以边的标签 node [circle,...] 为父节点。

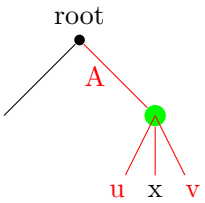



```
\begin{tikzpicture}
[level distance=10mm,level/.style={sibling distance=20mm/#1}]
\node [circle,inner sep=0pt,minimum size=4pt,fill,label=90:root]{}
  child {}
  child { node [circle,inner sep=0pt,minimum size=8pt,fill=green] {}
    child
    edge from parent[red] node[left] {A}
    child
    child
  };
\end{tikzpicture}
```

在上面这个结构中，命令 `edge from parent[red]` 针对的是 `node [circle,...]`, 而最后两个 child 却以边的标签 `node[left] {A}` 为父节点。使用花括号来纠正这个问题：




```
\begin{tikzpicture}
[level distance=10mm,level/.style={sibling distance=20mm/#1}]
\node [circle,inner sep=0pt,minimum size=4pt,fill,label=90:root]{}
  child {}
  child { node [circle,inner sep=0pt,minimum size=8pt,fill=green] {}
    child {edge from parent[red] node[left] {A}}
    child
    child
  };
\end{tikzpicture}
```




```
\begin{tikzpicture}
[level distance=10mm,level/.style={sibling distance=20mm/#1}]
\node [circle,inner sep=0pt,minimum size=4pt,fill,label=90:root]{}
  child {}
  child { node [circle,inner sep=0pt,minimum size=8pt,fill=green] {}
    child {node{x}}
      edge from parent[red] node[left] {A} node[shape=coordinate, shift=(
      ↪ \tikzchildnode)]{}
    child {node{u}}
    child {node{v}}
  };
\end{tikzpicture}
```

注意在上面例子中，使用选项 `shift=(\tikzchildnode)` 将标签平移到 `(\tikzchildnode)` 位置。

51.7.4 命令 edge from parent 的选项

假设命令 `edge from parent` 导致的画边命令是

`\path ...` ⟨*edge from parent path construction*⟩ `...;`

那么在

`child {... edge from parent[`⟨*options 1*⟩`] [`⟨*options 2*⟩`]}`

以及

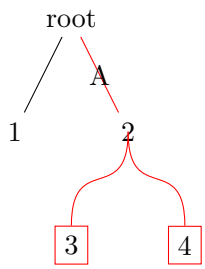
`child {... edge from parent[`⟨*options 1*⟩`]` ⟨*node specification*⟩ `[`⟨*options 2*⟩`]}`

中的 ⟨*options 1*⟩ 和 ⟨*options 2*⟩ 会这样起作用：

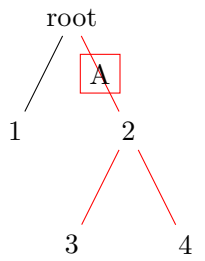
`\path` ⟨*options 1*⟩ ⟨*edge from parent path construction*⟩ ⟨*options 2*⟩`;`

所以在 ⟨*options 1*⟩ 与 ⟨*options 2*⟩ 中的某些选项可能不会都起作用。

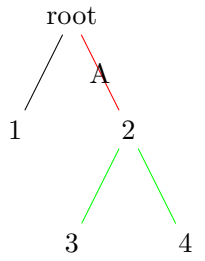
假设设置了选项 `edge from parent macro=`⟨*macro*⟩, 按前文的说明，宏 ⟨*macro*⟩ 需要两个参数。在执行命令 `edge from parent` 时，⟨*options 1*⟩ 成为 ⟨*macro*⟩ 的第一个参数，而 ⟨*options 2*⟩ 则是 ⟨*macro*⟩ 的第二个参数的一部分。在初始状态下，如果在 ⟨*options 2*⟩ 中使用选项 `edge from parent path` 修改了宏 `\tikz@edge@to@parent@path` 的定义 (这个宏保存构建路径的代码 ⟨*edge from parent path construction*⟩, 也就是说修改了构建路径的代码)，那么修改的结果并不能立即在当前画边的命令中体现出来，只对之后的画边命令有影响。




```
\tikz{
  \node {root}
    child {node{1}}
    child {node{2}
      edge from parent
        node{A}
        node[coordinate,shift=(\tikzchildnode)]{}
        [every node/.style=draw][draw=red]
        [edge from parent path=
          {(\tikzparentnode.south) .. controls +(0,-1) and +(0,1)
           .. (\tikzchildnode.north)}]
      child {node{3}}
      child {node{4}}
    };
  }
```




```
\tikz{
  \node {root}
    child {node{1}}
    child {node{2}
      child {node{3}}
      child {node{4}}
      edge from parent[draw=red,every node/.style=draw]node{A}
    };
  }
```




```
\tikz{
  \node {root}
    child {node{1}}
    child {node{2}
      child [draw=green]{node{3}}
      child [draw=green]{node{4}}
      edge from parent[draw=red]node{A}[every node/.style=draw]
    };
  }
```

51.8 TikZ 处理 tree 的过程

在路径命令 `\path... child ...;` 中遇到 `child` 时会导致 `\tikz@children`, 这个命令解析 tree 的句法。

`\tikz@children`

本命令：

1. 初始设置

```
\let\tikz@children@list=\pgfutil@empty%
\tikznumberofchildren=0\relax%
```

2. 调用 `\tikz@collect@child`

`\tikz@collect@child hild ...`

这个命令执行一个循环处理：

- 将生长于当前位置之后的所有分支都保存到宏 `\tikz@children@list` 中。例如，对于

```
\begin{tikzpicture}
  \node {root}
    child [⟨options 1⟩] {node {left}}% 第一个 child
    child [⟨options 2⟩] {node {right}
      child {node {child}}
```

```
      child {node {child}}
    };
\end{tikzpicture}
```

第一个 child 导致 \tikz@collect@child, 使得宏 \tikz@children@list 保存

```
\tikz@childnode[options 1]{node {left}}% 第一个分支
\tikz@childnode[options 2]{% 第二个分支
  node {right}
  child {node {child}}
  child {node {child}}
}
```

- 以上循环处理过程可以获得当前分支内的、当前位置所处的 level 内的元素数目 (分支数目)，这个数目由计数器 \tikznumberofchildren 记录。

3. 调用 \tikz@children@collected, 处理保存在宏 \tikz@children@list 中的内容。

\tikz@children@collected

本命令的定义是：

```
\long\def\tikz@children@collected{%
  \begingroup%
    \advance\tikztreelevel by 1\relax%
    \tikzgdeventgroupcallback{descendants}%
    \let\tikz@options=\pgfutil@empty%
    \tikz@clear@rdf@options%
    \let\tikz@transform=\pgfutil@empty%
    \tikzset{level/.try=\the\tikztreelevel,level \the\tikztreelevel/.try}%
    \tikz@transform%
    \let\tikz@transform=\relax%
    \let\tikzparentnode=\tikz@last@fig@name%
    \ifx\tikz@grow\relax\else%
      % Transform to center of node
      \pgftransformshift{\pgfpointanchor{\tikzparentnode}{
      ↪  \tikz@growth@anchor}}%
    \fi%
    \tikznumberofcurrentchild=0\relax%
    \tikz@children@list%
    \global\setbox\tikz@tempbox=\box\tikz@figbox%
    \global\setbox\tikz@tempbox@bg=\box\tikz@figbox@bg%
  \endgroup%
  \setbox\tikz@figbox=\box\tikz@tempbox%
  \setbox\tikz@figbox@bg=\box\tikz@tempbox@bg%
  \tikz@scan@next@command%
}%
```

这个命令：

- 创建 tree, 整个创建操作限制在一个 \begingroup 与 \endgroup 的组合中。
- 确定当前层的编号，由计数器 \tikztreelevel 记录。
- 创建 tree 前本命令会先作清理

```
\let\tikz@options=\pgfutil@empty%
\tikz@clear@rdf@options%
\let\tikz@transform=\pgfutil@empty%
```

然后执行选项 level=⟨当前层编号⟩, level ⟨当前层编号⟩，这两个选项的设置对当前以及之后的各个层都有效。

- 执行变换并清理

```
\tikz@transform%
\let\tikz@transform=\relax%
```

这里的变换只可能来自选项 level=⟨当前层编号⟩, level ⟨当前层编号⟩。

- 将之前最近出现的 node 作为父节点

```
\let\tikzparentnode=\tikz@last@fig@name
```

- 检查是否引入了生长函数

```
\ifx\tikz@grow\relax
\else%
  \pgftransformshift{\pgfpointanchor{\tikzparentnode}{
  ↪  \tikz@growth@anchor}}%
\fi%
```

- 初始化计数器

```
\tikznumberofcurrentchild=0\relax
```

- 执行 \tikz@children@list, 创建各个分支。

\tikz@childnode[⟨*options*⟩]{⟨*node child branch specification*⟩}

本命令的参数是

child [⟨*options*⟩]{⟨*node child branch specification*⟩}

中的内容，是生长于当前位置之后的一个分支。本命令的定义是：

```
\def\tikz@childnode[#1]#2{%
  \advance\tikznumberofcurrentchild by1\relax%
  {\tikzset{every child/.try,#1}\expandafter}%
  \iftikz@child@missing%
    \tikzgdeventcallback{node}{}%
  \else%
  \setbox\tikz@whichbox=\hbox\bgroup%
    \unhbox\tikz@whichbox%
    \hbox\bgroup\bgroup%
        \pgfinterruptpath%
          \pgfscope%
            \let\tikz@transform=\pgfutil@empty%
            \tikzset{every child/.try,#1}%
            \tikz@options%
            \tikz@transform%
            \let\tikz@transform=\relax%
            \tikz@grow%
            % Typeset node:
            \edef\tikz@parent@node@name{[name=\tikzparentnode-
            ↪  \the\tikznumberofcurrentchild,style=every child node]}%
            \def\tikz@child@node@text{[shape=coordinate]{}}
            \tikz@parse@child@node#2\pgf@stop%
            \expandafter\expandafter\expandafter\node
            \expandafter\tikz@parent@node@name
              \tikz@child@node@text
              \pgfextra{\global\let\tikz@childnode@name=\tikz@last@fig@name};%
            \let\tikzchildnode=\tikz@childnode@name%
            {%
              \def\tikz@edge@to@parent@needed{edge from parent}
              \ifx\tikz@child@node@rest\pgfutil@empty%
```

```
            \path edge from parent;%
          \else%
            \path \tikz@child@node@rest \tikz@edge@to@parent@needed;%
          \fi%
        }%
      \endpgfscope%
    \endpgfinterruptpath%
  \egroup\egroup%
 \egroup%
 \fi%
}%
```

本命令会：

- 把分支放在 \bgroup 与 \egroup 的组合中，然后将这个组合放到 \hbox 中，然后再添加到盒子 \tikz@whichbox 中。
- 本命令在创建分支前会执行样式 every child 以及选项 ⟨*options*⟩, 然后再执行 \tikz@options → P. 677.
- 宏 \tikz@grow 通常保存变换命令，用于决定子节点的位置。本命令执行变换：

  ```
  \tikz@transform%
  \let\tikz@transform=\relax%
  \tikz@grow
  ```

  然后用 \node 命令创建子节点，样式 every child node 用于这个 node, 并且是在样式 every node 之后起作用。
- 对于

  ```
  \tikz@childnode [⟨options 1⟩] {
    node [⟨options 2⟩] {⟨text⟩}
    child [⟨options 3⟩] {...}
    child [⟨options 4⟩] {...}
  }
  ```

  实际会执行

  ```
  \node [name=\tikzparentnode-\the\tikznumberofcurrentchild,style=every child
  ↪  node]
    {⟨text⟩}
    \pgfextra{\global\let\tikz@childnode@name=\tikz@last@fig@name};%
  \let\tikzchildnode=\tikz@childnode@name%
  {%
    \path child [⟨options 3⟩] {...}
          child [⟨options 4⟩] {...}
          edge from parent;%
  }%
  ```

  这就构成了一个循环操作。
- 条件判断

  ```
  \ifx\tikz@child@node@rest\pgfutil@empty
    \path edge from parent;
  ...
  ```

  的作用是：如果用户没有写出 edge from parent 命令，那么 Ti*k*Z 会添加这个命令。

第七部分

库

第五十二章　三维绘图库

TikZ Library 3d

```
\usetikzlibrary{3d} % LaTeX and plain TeX
\usetikzlibrary[3d] % ConTeXt
```

本程序库提供一些样式和选项，用于绘制简单的三维图形。

对于下面的各种坐标系统，都可以使用选项 /tikz/x^{→P. 859}, /tikz/y^{→P. 861}, /tikz/z^{→P. 861} 来设置各个坐标轴的单位向量。

52.1　坐标系统

52.1.1　Coordinate system xyz cylindrical

参考命令 \tikzdeclarecoordinatesystem^{→P. 691}, \pgfpointcylindrical^{→P. 254}.

定义是：

```
\tikzdeclarecoordinatesystem{xyz cylindrical}
{%
  \pgfset{/tikz/cs/.cd,angle=0,radius=0,z=0,#1}%
  \pgfpointcylindrical{\tikz@cs@angle}{\tikz@cs@xradius}{\tikz@cs@z}%
}%
```

下面的选项用于决定一个圆柱坐标点：

- 参考 /tikz/cs/radius^{→P. 688}.
  /tikz/cs/radius=⟨*number*⟩
  将 ⟨*number*⟩ 与 x 轴的单位向量相乘得到一个半轴向量，将 ⟨*number*⟩ 与 y 轴的单位向量相乘得到一个半轴向量，两个半轴向量确定一个椭圆，这个椭圆就是圆柱面与 xy 平面的交线。
  本选项的默认值是 0.
- 参考 /tikz/cs/angle^{→P. 947}.
  /tikz/cs/angle=⟨*degrees*⟩
  注意如果 x 轴单位向量与 y 轴单位向量的长度不一样，那么本选项指定的角度 ⟨*degrees*⟩ 就不是那么直观的。
  本选项的默认值是 0.
- 参考 /tikz/cs/z^{→P. 1153}.
  /tikz/cs/z=⟨*number*⟩
  将 ⟨*number*⟩ 与 z 轴的单位向量相乘，确定坐标点的 z 分量。
  本选项的默认值是 0.

```
\begin{tikzpicture}[->,x=2cm,y=0.5cm]
  \draw (0,0,0) -- (xyz cylindrical cs:radius=1);
  \draw [red](0,0,0) -- (xyz cylindrical cs:radius=1,angle=60);
  \draw (0,0,0) -- (xyz cylindrical cs:z=1);
\end{tikzpicture}
```

52.1.2 Coordinate system xyz spherical

参考命令 \tikzdeclarecoordinatesystem→P. 691, \pgfpointcylindrical→P. 254.

定义是：

```
\tikzdeclarecoordinatesystem{xyz spherical}
{%
  \pgfset{/tikz/cs/.cd,angle=0,radius=0,latitude=0,longitude=0,#1}%
  \pgfpointspherical{\tikz@cs@angle}{\tikz@cs@latitude}{\tikz@cs@xradius}%
}%
```

/tikz/cs/radius=⟨*number*⟩

将 ⟨*number*⟩ 与 x, y, z 轴的单位向量相乘，得到 3 个半轴向量，确定一个椭球。

本选项的默认值是 0.

/tikz/cs/latitude=⟨*degrees*⟩ (no default, initially 0)

本选项指定“纬度”。以 y, z 轴的单位向量为标架确定一个平面坐标系统，本选项的 ⟨*degrees*⟩ 就是这个坐标系统中的角度，以 y 轴的单位向量为度量角度的起始方向。

本选项的默认值是 0.

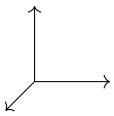
/tikz/cs/longitude=⟨*degrees*⟩ (no default, initially 0)

本选项指定“经度”。以 x, y 轴的单位向量为标架确定一个平面坐标系统，本选项的 ⟨*degrees*⟩ 就是这个坐标系统中的角度，以 y 轴的单位向量为度量角度的起始方向。

本选项等效于角度 /tikz/cs/angle, 默认值是 0.

/tikz/cs/angle=⟨*degrees*⟩ (no default, initially 0)

等效于经度 longitude.



```
\begin{tikzpicture}[->]
\draw (0,0,0) -- (xyz spherical cs:radius=1);
\draw (0,0,0) -- (xyz spherical cs:radius=1,latitude=90);
\draw (0,0,0) -- (xyz spherical cs:radius=1,longitude=90);
\end{tikzpicture}
```

52.2 坐标平面

如果需要在某个平面上绘图，可以使用下面的选项确定这个平面的标架，Ti*k*Z 会自动将绘图命令中的坐标转换到这个标架中，即以这个标架为参照系绘图。

52.2.1 转换到任意平面

为了确定一个三维空间中的一个平面，需要 3 个不共线点 P_0, P_1, P_2. 以 P_0 为原点、以向量 $\overrightarrow{P_0P_1}$ 为 x 轴的单位向量、以向量 $\overrightarrow{P_0P_2}$ 为 y 轴的单位向量构成一个标架，这个标架确定一个平面——即由向量 $\overrightarrow{P_0P_1}$, $\overrightarrow{P_0P_2}$ 张成的、经过点 P_0 的平面。点 P_0, P_1, P_2 由下面的选项指定：

/tikz/plane origin=⟨*point*⟩ (no default, initially (0,0))

本选项指定点 P_0, 用作平面标架的原点。初始值是 canvas 坐标系统的原点 \pgfpointorigin.

```
\tikzoption{plane origin}{\def\tikz@plane@origin{\tikz@scan@one@point
↪  \pgfutil@firstofone#1}}%
```

⟨*point*⟩ 是 TikZ 的坐标点，将来会被 \tikz@scan@one@point→P. 710 解析。⟨*point*⟩ 可以是二维点或三维点。如果 ⟨*point*⟩ 是三维点，那么它就属于 xyz 坐标系统。

/tikz/plane x=⟨*point*⟩ (no default, initially (1,0))

本选项指定点 P_1, 用于计算平面标架的 x 轴的单位向量。初始值是 xyz 坐标系统的 x 轴的单位点 \pgfpointxy{1}{0}.

```
\tikzoption{plane x}{\def\tikz@plane@x{\tikz@scan@one@point\pgfutil@firstofone#1}}
↪  %
```

⟨*point*⟩ 是 TikZ 的坐标点，将来会被 \tikz@scan@one@point→P. 710 解析。⟨*point*⟩ 可以是二维点或三维点。如果 ⟨*point*⟩ 是三维点，那么它就属于 xyz 坐标系统。

/tikz/plane y=⟨*point*⟩ (no default, initially (0,1))

本选项指定点 P_2, 用于计算平面标架的 y 轴的单位向量。初始值是 xyz 坐标系统的 y 轴的单位点 \pgfpointxy{0}{1}.

```
\tikzoption{plane y}{\def\tikz@plane@y{\tikz@scan@one@point\pgfutil@firstofone#1}}
↪  %
```

⟨*point*⟩ 是 TikZ 的坐标点，将来会被 \tikz@scan@one@point→P. 710 解析。⟨*point*⟩ 可以是二维点或三维点。如果 ⟨*point*⟩ 是三维点，那么它就属于 xyz 坐标系统。

/tikz/canvas is plane (no value)

用前面的选项指定三个点 P_0, P_1, P_2 ——平面标架后，再使用本选项将这个标架作为画布的标架，否则没有预期的效果。本选项必须写在前述 3 个选项之后。

```
\tikzoption{canvas is plane}[]{
  \tikz@canvas@is@plane
}%
```

本选项执行命令 \tikz@canvas@is@plane.

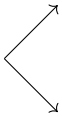
\tikz@canvas@is@plane

本命令的定义是：

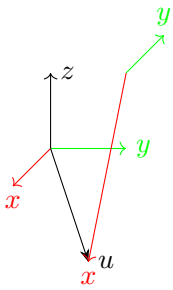
```
\def\tikz@canvas@is@plane{
  \pgf@process{\tikz@plane@x}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{\tikz@plane@y}%
  \pgf@xb=\pgf@x%
  \pgf@yb=\pgf@y%
  \pgf@process{\tikz@plane@origin}%
  \edef\pgf@marshal{\noexpand\tikz@addtransform{%
      \noexpand\pgftransformtriangle
      {\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}
      {\noexpand\pgfqpoint{\the\pgf@xa}{\the\pgf@ya}}
      {\noexpand\pgfqpoint{\the\pgf@xb}{\the\pgf@yb}}
      \noexpand\pgftransformscale{0.035146}%
      \noexpand\pgfsetxvec{\noexpand\pgfpoint{1cm}{0cm}}%
      \noexpand\pgfsetyvec{\noexpand\pgfpoint{0cm}{1cm}}%
      \noexpand\pgfsetzvec{\noexpand\pgfpoint{0cm}{0cm}}%
    }}%
  \pgf@marshal%
}%
```

本命令：

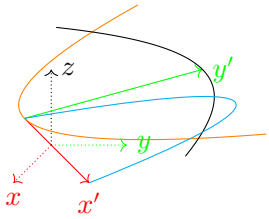
1. 用 \pgftransformtriangle →P. 266 设置 canvas 坐标系统，注意此时单位向量的长度对应 1pt，也就是说，(1pt,0) 对应向量 $\overrightarrow{P_0P_1}$，而 (0,1pt) 对应向量 $\overrightarrow{P_0P_2}$
2. 按比例 $\frac{1\text{pt}}{1\text{cm}} = \frac{1}{28.45274} = 0.035145999$ 调整 canvas 坐标系统的单位向量的长度所对应的长度单位——从 1pt 改为 1cm
3. 用 \pgfsetxvec →P. 253 等命令设置 xyz 坐标系统的单位向量，使之与 canvas 坐标系统的单位向量一样



```
\begin{tikzpicture}[->,
    plane x={(0.707,-0.707)}, plane y={(0.707,0.707)}, canvas is plane,]
  \draw (0,0) -- (1,0);
  \draw (0,0) -- (0,1);
\end{tikzpicture}
```

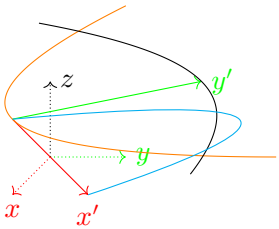



```
\begin{tikzpicture}[x=(-135:{0.5*sqrt(2)}),y=(0:1cm),z=(90:1cm)]
  \draw [->,red](0,0,0)--(1,0,0) node [below] {$x$};
  \draw [->,green](0,0,0)--(0,1,0) node [right] {$y$};
  \draw [->](0,0,0)--(0,0,1) node [right] {$z$};
  \draw [-Stealth](0,0,0)--(1,1,-1) node [right] {$u$};
  \begin{scope}[plane origin={(2,2,2)}, plane x={(1,1,-1)},plane y={(-1,1,1)
  ↪  },canvas is plane]
    \draw [->,red](0,0)--(1,0) node [below] {$x$};
    \draw [->,green](0,0)--(0,1) node [above] {$y$};
  \end{scope}
\end{tikzpicture}
```




```
\begin{tikzpicture}[x=(-135:{0.5*sqrt(2)}),y=(0:1cm),z=(90:1cm)]
  \draw [->,red,densely dotted](0,0,0)--(1,0,0) node [below] {$x$};
  \draw [->,green,densely dotted](0,0,0)--(0,1,0) node [right] {$y$};
  \draw [->,densely dotted](0,0,0)--(0,0,1) node [right] {$z$};
  \begin{scope}[plane origin={(-135:1)}, plane x={(1,1)},plane y={(-2,1)
  ↪  },canvas is plane]
    \draw [->,red](0,0)--(1,0) node [below] {$x'$};
    \draw [->,green](0,0)--(0,1) node [right] {$y'$};
    \draw [orange]plot [domain=-1:1, samples=200] (\x,\x*\x);
    \draw plot [domain=-1:1, samples=200] (\x,{cos(\x r)});
    \draw [cyan](0,0) parabola bend(0.5,1) (1,0);
  \end{scope}
\end{tikzpicture}
```

上面例子中，选项 plane origin={(-135:1)}, plane x={(1,1)},plane y={(-2,1)} 指定的二维坐标点都是 xyz 坐标系 x=(-135:{0.5*sqrt(2)}),y=(0:1cm) 中的点。




```
\begin{tikzpicture}[x=(-135:{0.5*sqrt(2)}),y=(0:1cm),z=(90:1cm)]
  \draw [->,red,densely dotted](0,0,0)--(1,0,0) node [below] {$x$};
  \draw [->,green,densely dotted](0,0,0)--(0,1,0) node [right] {$y$};
  \draw [->,densely dotted](0,0,0)--(0,0,1) node [right] {$z$};
  \begin{scope}[plane origin={(1,0,1)}, plane x={(1,1,0)},plane y={(-2,1,0)
  ↪  },canvas is plane]
    \draw [->,red](0,0)--(1,0) node [below] {$x'$};
    \draw [->,green](0,0)--(0,1) node [right] {$y'$};
    \draw [orange]plot [domain=-1:1, samples=200] (\x,\x*\x);
    \draw plot [domain=-1:1, samples=200] (\x,{cos(\x r)});
    \draw [cyan](0,0) parabola bend(0.5,1) (1,0);
  \end{scope}
\end{tikzpicture}
```

52.2.2 预定义的平面

/tikz/canvas is xy plane at z=⟨*z value*⟩ (no default)

本选项的定义是：

```
\tikzoption{canvas is xy plane at z}[]{%
  \def\tikz@plane@origin{\pgfpointxyz{0}{0}{#1}}%
  \def\tikz@plane@x{\pgfpointxyz{1}{0}{#1}}%
  \def\tikz@plane@y{\pgfpointxyz{0}{1}{#1}}%
  \tikz@canvas@is@plane
}%
```

本选项决定一个平面标架：以当前 xyz 坐标系的 (0,0,⟨*z value*⟩) 为原点，以当前 xyz 坐标系的 (1,0,⟨*z value*⟩) 为 x 轴单位向量，以当前 xyz 坐标系的 (0,1,⟨*z value*⟩) 为 y 轴单位向量。

/tikz/canvas is yx plane at z=⟨*z value*⟩ (no default)

本选项的定义是：

```
\tikzoption{canvas is yx plane at z}[]{%
  \def\tikz@plane@origin{\pgfpointxyz{0}{0}{#1}}%
  \def\tikz@plane@x{\pgfpointxyz{0}{1}{#1}}%
  \def\tikz@plane@y{\pgfpointxyz{1}{0}{#1}}%
  \tikz@canvas@is@plane
}%
```

本选项决定一个平面标架：以当前 xyz 坐标系的 (0,0,⟨*z value*⟩) 为原点，以当前 xyz 坐标系的 (0,1,⟨*z value*⟩) 为 x 轴单位向量，以当前 xyz 坐标系的 (1,0,⟨*z value*⟩) 为 y 轴单位向量。

/tikz/canvas is xz plane at y=⟨*y value*⟩ (no default)

本选项的定义是：

```
\tikzoption{canvas is xz plane at y}[]{%
  \def\tikz@plane@origin{\pgfpointxyz{0}{#1}{0}}%
  \def\tikz@plane@x{\pgfpointxyz{1}{#1}{0}}%
  \def\tikz@plane@y{\pgfpointxyz{0}{#1}{1}}%
  \tikz@canvas@is@plane
}%
```

/tikz/canvas is zx plane at y=⟨*y value*⟩ (no default)

本选项的定义是：

```
\tikzoption{canvas is zx plane at y}[]{%
  \def\tikz@plane@origin{\pgfpointxyz{0}{#1}{0}}%
  \def\tikz@plane@x{\pgfpointxyz{0}{#1}{1}}%
  \def\tikz@plane@y{\pgfpointxyz{1}{#1}{0}}%
  \tikz@canvas@is@plane
}%
```

/tikz/canvas is yz plane at x=⟨*x value*⟩ (no default)

本选项的定义是：

```
\tikzoption{canvas is yz plane at x}[]{%
  \def\tikz@plane@origin{\pgfpointxyz{#1}{0}{0}}%
  \def\tikz@plane@x{\pgfpointxyz{#1}{1}{0}}%
  \def\tikz@plane@y{\pgfpointxyz{#1}{0}{1}}%
  \tikz@canvas@is@plane
}%
```

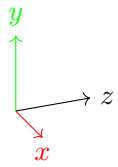
/tikz/canvas is zy plane at x=⟨*x value*⟩ (no default)

本选项的定义是：

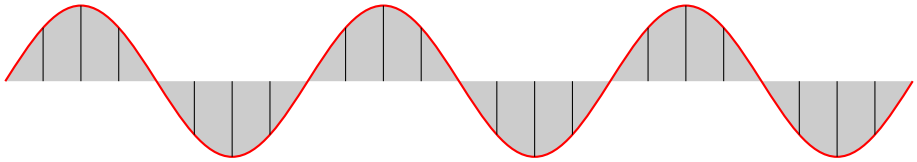
```
\tikzoption{canvas is zy plane at x}[]{%
  \def\tikz@plane@origin{\pgfpointxyz{#1}{0}{0}}%
  \def\tikz@plane@x{\pgfpointxyz{#1}{0}{1}}%
```

```
  \def\tikz@plane@y{\pgfpointxyz{#1}{1}{0}}%
  \tikz@canvas@is@plane
}%
```

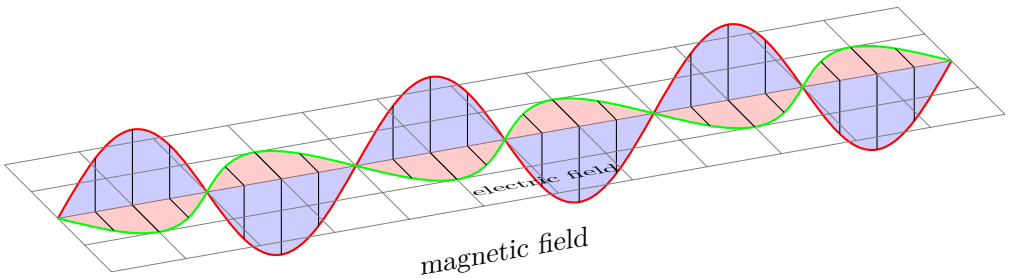
52.3 例子




```
\begin{tikzpicture}[z={(10:10mm)},x={(-45:5mm)}]
  \draw [->,red](0,0,0)--(1,0,0) node [below] {$x$};
  \draw [->,green](0,0,0)--(0,1,0) node [above] {$y$};
  \draw [->](0,0,0)--(0,0,1) node [right] {$z$};
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[draw=red, fill,thick,fill opacity=.2]
      (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0)
            sin (5,1) cos (6,0) sin (7,-1) cos (8,0)
            sin (9,1) cos (10,0)sin (11,-1)cos (12,0);
  \foreach \shift in {0,4,8}
    {
      \begin{scope}[xshift=\shift cm,thin]
        \draw (.5,0) -- (0.5,0 |- 45:1cm);
        \draw (1,0) -- (1,1);
        \draw (1.5,0) -- (1.5,0 |- 45:1cm);
        \draw (2.5,0) -- (2.5,0 |- -45:1cm);
        \draw (3,0) -- (3,-1);
        \draw (3.5,0) -- (3.5,0 |- -45:1cm);
      \end{scope}
    }
\end{tikzpicture}
```




```
\begin{tikzpicture}[z={(10:10mm)},x={(-45:5mm)}]
  \def\wave#1{
    \draw[#1,fill,thick,fill opacity=.2]
      (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0)
            sin (5,1) cos (6,0) sin (7,-1) cos (8,0)
            sin (9,1) cos (10,0)sin (11,-1)cos (12,0);
    \foreach \shift in {0,4,8}
    {
      \begin{scope}[xshift=\shift cm,thin]
        \draw (.5,0) -- (0.5,0 |- 45:1cm);
        \draw (1,0) -- (1,1);
        \draw (1.5,0) -- (1.5,0 |- 45:1cm);
        \draw (2.5,0) -- (2.5,0 |- -45:1cm);
        \draw (3,0) -- (3,-1);
        \draw (3.5,0) -- (3.5,0 |- -45:1cm);
```

```
      \end{scope}
    }
  }
  \begin{scope}[canvas is zy plane at x=0,fill=blue]
    \wave{draw=red}
    \node at (6,-1.5) [transform shape] {magnetic field};
  \end{scope}
  \begin{scope}[canvas is zx plane at y=0,fill=red]
    \draw[help lines] (0,-2) grid (12,2);
    \wave{draw=green}
    \node at (6,1.5) [rotate=180,xscale=-1,transform shape] {electric field};
  \end{scope}
\end{tikzpicture}
```

第五十三章　angles 库

TikZ Library angles

```
\usetikzlibrary{angles} % LaTeX and plain TeX
\usetikzlibrary[angles] % ConTeXt
```

这个库定义一个名称为 angle 的 pic type 用来标识“角”；定义一个名称为 right angle 的 pic type 用来标识“直角”。

angle

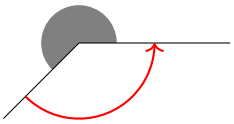
angle=⟨*A*⟩--⟨*B*⟩--⟨*C*⟩

angle 是个 pic type，其中的 ⟨*A*⟩, ⟨*B*⟩, ⟨*C*⟩ 必须是 node 名称或者 coordinate 名称，不能直接使用坐标数值；其中 ⟨*B*⟩ 用作角的顶点，⟨*A*⟩--⟨*B*⟩ 指示角的始边，⟨*B*⟩--⟨*C*⟩ 指示角的终边，从始边沿着逆时针方向到终边创建一个圆弧路径，这个圆弧就是 angle 对“角”的标识。这个圆弧默认是不画出的，当 pic 操作带有 draw 选项时会画出这个圆弧。当 pic 带有 fill 选项时，圆弧与角的起止边构成的区域会被填充。

注意这里 ⟨*A*⟩, ⟨*B*⟩, ⟨*C*⟩ 是不带圆括号的名称，而且在格式“⟨*A*⟩--⟨*B*⟩--⟨*C*⟩”中不要随意加空格。

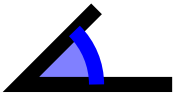
/tikz/angle radius=⟨*dimension*⟩ (no default, initially 5mm)

这个选项用于设置指示圆弧的半径。



```
\tikz \draw (2,0) coordinate (A) -- (0,0) coordinate (B)
  -- (-1,-1) coordinate (C)
    pic [fill=black!50] {angle = A--B--C}
    pic [draw,->,red,thick,angle radius=1cm] {angle = C--B--A};
```

在一个绘图句子中，如果名称 ⟨*A*⟩, ⟨*B*⟩, ⟨*C*⟩ 是已经明确规定的，那么也可以只写出 angle, 略去 = ⟨*A*⟩--⟨*B*⟩--⟨*C*⟩.

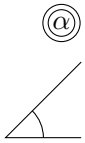


```
\tikz \draw [line width=2mm]
  (2,0) coordinate (A) -- (0,0) coordinate (B) -- (1,1) coordinate (C)
    pic [draw=blue, fill=blue!50, angle radius=1cm] {angle};
```

当用选项 pic text 添加标签时，或使用引用句法（双引号）添加标签时，标签 node 会被放在指示圆弧附近，且位于角平分线上。标签 node 没有自己的名称，它会继承 pic 的名称。标签与角的顶点的距离由下面的选项调节：

/tikz/angle eccentricity=⟨*factor*⟩ (no default, initially 0.6)

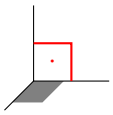
将 ⟨*factor*⟩ 与半径 angle radius 相乘，乘积就是标签与角的顶点的距离。



```
\tikz {
  \draw (1,0) coordinate (A) -- (0,0) coordinate (B)
    -- (1,1) coordinate (C)
      pic (alpha) ["$\alpha$"{above=1cm}, draw, angle
      ↪  eccentricity=1.6] {angle};
  \draw (alpha) circle [radius=5pt] circle [radius=7pt];
}
```

`right angle=`⟨*A*⟩`--`⟨*B*⟩`--`⟨*C*⟩

这个 pic type 的用法类似 `angle`，它画一个直角标记。



```
\tikz
  \draw (1,0,0) coordinate (A) -- (0,0,0) coordinate (B)
    -- (0,0,1) coordinate (C)
    (B) -- (0,1,0) coordinate (D)
    pic [fill=gray,angle radius=4mm] {right angle = A--B--C}
    pic [draw,red,thick,angle eccentricity=.5,pic text=$\cdot$]
      {right angle = A--B--D};
```

第五十四章　动画

TikZ Library animations

```
\usetikzlibrary{animations} % LaTeX and plain TeX
\usetikzlibrary[animations] % ConTeXt
```

载入这个库后可以用 TikZ 制作动画。

54.1　Introduction

有的宏包制作动画的机制是这样的：首先制作一些图形，然后把这些图形排好次序，并按次序快速播放这些图形，形成动画效果。TikZ 的 animations 库不是这样工作的。目前，TikZ 输出的动画文件其实是个 svg 格式的文件，文件内容只是对动画的描述。制作这个 svg 文件时只涉及 svg 格式的语法，不涉及关于动画的数学计算，因此 TikZ 能以较快的速度生成这个 svg 文件。用专门读取 svg 文件的阅读器打开这个文件，阅读器会自己进行相关的计算并展示动画。也就是说，TikZ 只是提出了“设计要求”，把“要求”变成现实的繁重工作则留给了 svg 阅读器。这种机制的优点是不会拖慢 TEX 的运行速度。

动画涉及几个概念：对象（object），属性（attribute），时间线（timeline）。如果把一个 node 看作是对象，则它的文字可以看作是它的属性，让它的文字发生变化可以形成动画。如果把文字看作是对象，则文字的颜色可以看作是它的属性，让文字的颜色发生变化也可以形成动画。这样看起来似乎对象与属性的区分是相对的，但实际上，什么是对象，什么是属性，都是人为规定的，二者的区分是绝对的。时间线就是规定在什么时刻，让哪个对象的何种属性呈现什么状态。时间线可以很简单，也可以很复杂。复杂的时间线就像一场舞台剧，在一定的时间内展示丰富的舞台内容。时间线通常都有自己的“开始时刻”和“结束时刻”，时间线开始的时刻可以规定为 `0s` （即 0 秒时刻），也可以规定为 `-5s`, `2s` 等。时间线结束的时刻可以规定为“无穷”（即时间线不结束）。

下文说到“时间线”时，可能指的是某一个时间线，也可能指的是某一组时间线。

使用 TikZ 创建动画要注意以下几点：

1. 动画需要指定一个输出格式，目前只能是 svg 格式。
2. 动画针对的对象必须是尚未创建的，已经创建的对象不能形成动画。
3. 考虑 `\draw (a)--(b);`，其中 `(a)` 与 `(b)` 是两个 node. 当 `(a)` 运动时，`(a)(b)` 之间的连线却不能如所期望地那样运动。不过把“`(a)`，`(b)`，连线”三者作为一个整体来运动是没有问题的。如果出现了这样的问题就需要采用某些额外的代码，画某些线条来得到希望的效果。
4. 动画能自动计算图形的边界盒子，不过当对 object 施加旋转、放缩、倾斜变换时，或在同一时刻对同一个 object 做数个平移变换时，对边界盒子的计算可能失效。

下面的图形构件可以看作是对象：

1. node，由 `\node`，`\graph` 创建。node 的多个部分都可以形成动画，例如可以把 node 的背景路径的颜色做成动画，文字内容的颜色等。
2. 图形环境，如 `{scope}` 环境，`{tikzpicture}` 环境，命令 `\scopes`, `\tikz` 创建的图形。
3. View boxes, 由 view 库创建的盒子。
4. path, 由 `\path`, `\draw` 等命令创建的路径，node 的形状路径都可以形成动画。

对象的哪些要素可以看作是它的属性完全是人为的设计，参考后文。（例如 `:fill`, `:draw`, `:text`, `:color`, `:opacity` 等）

时间线必须包括：时刻、对象、属性的值。例如，在时刻 5s 把属性 `:xshift` 的值规定为 0mm，在时刻 10s 把属性 `:xshift` 的值规定为 10mm；这样规定后，属性 `:xshift` 在 0s 到 10s 之间的值使用插值计算的办法计算出来，也就是说（假设使用线性插值），在时刻 7.5s 属性 `:xshift` 的值是 5mm，在时刻 9s 属性 `:xshift` 的值是 8mm. 对属性值的插值计算并非简单，例如，有的属性值是 true 或 false，这就不好插值。有时候需要非线性插值。

动画是从 moment zero 开始的，如果没有特别的设置，moment zero 就是图形展示出来的时刻。也可以让 moment zero 对应到某个事件（event），例如可以把鼠标点击的时刻作为 moment zero，也可以把当前动画的 moment zero 规定为另外某个动画开始（或结束）的时刻。这个 moment zero 也与时间线有关，但未必就对应时间线的时刻 0s.

使用 Ti*k*Z 制作动画时可以如下编译：参考手册 §10.2.4, 执行 `lualatex --output-format=dvi` ⟨*tex file name*⟩ 得到 dvi 文件，然后再运行命令 `dvisvgm` ⟨*dvi file name*⟩ 得到 svg 文件。

也可以执行 `xelatex --no-pdf` ⟨*tex file name*⟩ 得到 xdv 文件，然后再运行命令 `dvisvgm` ⟨*xdv file name*⟩ 得到 svg 文件。参考 https://zhuanlan.zhihu.com/p/54877220

下文在展示例子时使用了命令 `\onetcbox`, 这是利用 tcolorbox 宏包定义的一个简单命令：

```
\newtcbox{\onetcbox}[1]{
  enhanced,  frame hidden,interior hidden,colbacktitle=white,coltitle=black,
  size=fbox,fonttitle=\small,halign title=center,nobeforeafter,title={#1}
}
```

54.2 创建动画

54.2.1 Animate 选项

选项 `animate` 用于创建时间线。

`/tikz/animate`=⟨*animation specification*⟩ (no default)

所有用来创建动画的那些选项都要放在 ⟨*animation specification*⟩ 中。可以连续多次使用本选项。如果对同一对象的同一属性使用两个 `animate` 选项，那么就会有两个时间线，如果这两个时间线之间有冲突，就会按照某些规则来决定哪个时间线有效。在能够使用 Ti*k*Z 选项的地方都可以使用本选项，例如本选项可以作为 `{scope}` 环境选项，node 的选项，命令 `\tikz` 的选项。

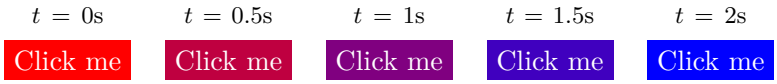
⟨*animation specification*⟩ 实际是 Ti*k*Z 的键值对（key-value pairs）列表，其中的选项都有路径前缀 `/tikz/animate/`, 不过使用选项的句法有点特别。

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s

Click me Click me Click me Click me Click me

```
\tikz \node [fill, text = white, animate = {
      myself:fill = {0s = "red", 2s = "blue", begin on = click }}] {Click me};
```

上面例子中，选项 animate 用作 node 的选项，其中的名称 myself 是个特殊名称，它指的就是 node 本身，这样就使得 node 具有动画效果；myself:fill 指的是 node 的填充色属性 fill；选项 0s = "red" 规定属性 fill 在时刻 0s 的值是 red；选项 2s = "blue" 规定属性 fill 在时刻 2s 的值是 blue；选项 begin on = click 规定当鼠标点击 node 时就启动动画。




```
\tikz [animate = {a node:fill = {0s = "red", 2s = "blue",begin on = click}}]
      \node (a node) [fill, text = white] {Click me};
```

上面例子中，在命令 \tikz 的选项中设置 animate, 其中 a node 是动画对象的名称，在之后的 \node 命令中把 node 的名称设为 a node, 就使得 node 具有动画效果。

可以自定义前缀为 /tikz/animate/ 的样式：

```
\tikzset{
  animate/shake/.style = {myself:xshift = { begin on=click,
    0s = "0mm", 50ms = "#1", 150ms = "-#1", 250ms = "#1", 300ms = "0mm" }}}
\tikz \node [fill = blue!20, draw=blue, very thick, circle,
  animate = {shake = 1mm}] {Shake};
\tikz \node [fill = blue!20, draw=blue, very thick, circle,
  animate = {shake = 2mm}] {SHAKE};
```

可以在 animate 中使用选项 name=⟨*name*⟩ 来为这个动画命名，然后在其它的动画中引用 ⟨*name*⟩ 的开始时刻或结束时刻。

在 animate 中设置动画对象时，用的是对象的“名称”，注意这个“名称”只是一个记号，它所代表的对象必须是尚未被创建的对象，也就是说，在写完动画代码后，具有此“名称”的对象才应当被创建。给动画对象命名时使用选项 /tikz/name→ P. 908.

54.2.2 时间线条目

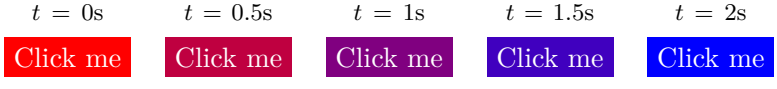
animate 中的选项实际上用于指定时间线。一个时间线必须（至少）包含以下 5 个要素：

- object, 此选项用于指定一个对象。
- attribute, 此选项用于指定一个属性。
- id, 此选项指定当前时间线的 id, 本选项有默认值，如无必要可以不写出本选项。
- time, 此选项指定时刻。
- value, 此选项设置（由选项 attribute 指定的）属性的值。

当设置以上 5 个要素后，就可以再使用选项 entry 了：

/tikz/animate/entry (no value)

每当在 animate 中使用本选项时，Ti*k*Z 就会检查 entry 之前的 5 个选项 object, attribute, id, time, value 是否都已经设置好了；如果其中有一个选项未设置好，那什么也不会发生（注意选项 id 有默认值，可以不写出 id 选项）；如果这 5 个选项都已经设置好了，就创建一个 time-value pair. 在 entry 之前也可以使用其它的以 /tikz/animate/options/ 为前缀的选项（如 begin on），这种选项也会影响 time-value pair.



```
\tikz [animate = {
  object = node, attribute = fill, time = 0s, value = red, entry,
  object = node, attribute = fill, time = 2s, value = blue, entry,
  object = node, attribute = fill, begin on = click, entry}]
  \node (node) [fill, text=white] { Click me };
```

上面代码可写成更简洁的形式：

```
\tikz [animate = {
  object = node, attribute = fill, time = 0s, value = red, entry,
                                   time = 2s, value = blue, entry,
                                   begin on = click, entry}]
  \node (node) [fill, text=white] { Click me };
```

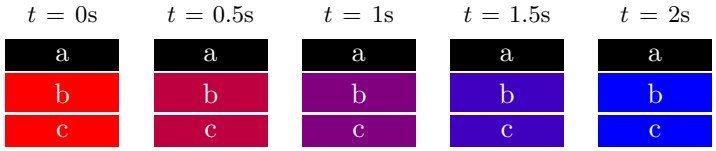
上面代码中有 3 个 entry, 所以设置了 3 个时间线，这 3 个时间线的 object，attribute 值相同，所以只在第一个时间线中设置选项 object = node, attribute = fill, 这两个选项设置会自动延续到第 2、第 3 个时间线中。第 2 个时间线的 time，value 值与第 3 个的也相同，所以在第 3 个时间线中也无需写出这两个选项。这里使用选项 id 的默认值，也无需写出。

注意，在一个 animate 内设置时间线时，各个时间线的时刻必须是递增（non-decreasing order）的。

54.2.3 指定对象

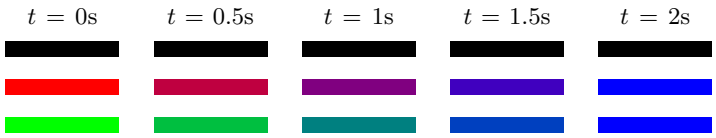
/tikz/animate/object=⟨*list of object*⟩ (no default)

在 ⟨*list of object*⟩ 中列出对象名称,相邻两个对象名称之间用逗号分隔,对象名称的形式是 ⟨*object*⟩.⟨*type*⟩, 所列出的各个对象名称都会被加入到相应的时间线中。⟨*list of object*⟩ 中列出的对象名称应当是尚未被创建的对象。




```
\tikz [animate = { object = {b,c}, :fill = {0s = "red", 2s = "blue", begin on =
↪  click }}]
{
  \node (a) [fill, text = white, minimum width=1.5cm] at (0,1cm) {a};
  \node (b) [fill, text = white, minimum width=1.5cm] at (0,5mm) {b};
  \node (c) [fill, text = white, minimum width=1.5cm] at (0,0mm) {c}; }
```

上面例子中，时间线包含名称为 b，c 的 node 对象，而名称为 a 的 node 没有动画效果。下面例子中把 \scope 创建的图形作为时间线的对象：




```
\tikz [animate ={ object = b, :fill = {0s = "red", 2s = "blue", begin on = click
↪  },
         object = c, :fill = {0s = "green", 2s = "blue", begin on = click } }]
{
  \scoped [name = a, yshift=1cm] \fill (0,0) rectangle (1.5cm,2mm);
  \scoped [name = b, yshift=5mm] \fill (0,0) rectangle (1.5cm,2mm);
  \scoped [name = c, yshift=0mm] \fill (0,0) rectangle (1.5cm,2mm); }
```

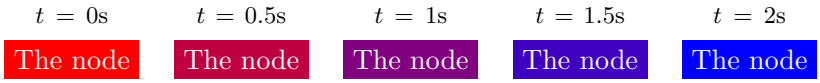
⟨*object*⟩ 可以是特殊名称 myself, 当某个对象带有 animate={myself...} 选项时，这个动画就是针对此对象本身的，其它对象不能被命名为 myself.

有的对象有多个组成部分，每个组成部分也都可以作为一个对象，当只需要把某个部分添加到时间线中时，可以使用 ⟨*object*⟩.⟨*type*⟩ 形式的名称，其中的 ⟨*type*⟩ 用于指定把哪个部分加入到时间线中。

54.2.4 指定属性

/tikz/animate/attribute=⟨*list of attribute*⟩ (no default)

⟨*list of attribute*⟩ 是属性名称的列表，所列出的属性会被加入到相应的时间线中。

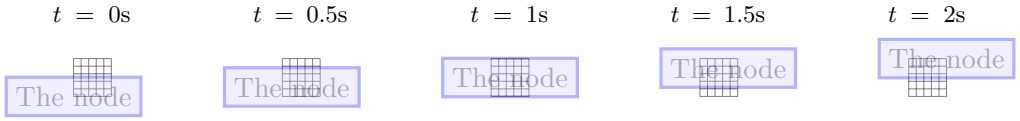



```
\tikz [animate = {attribute = fill, n: = { 0s = "red", 2s = "blue", begin on =
↪  click } }]
  \node (n) [fill, text = white] {The node};
```

54.2.5 指定 ID

/tikz/animate/id=⟨*id*⟩ (no default, initially `default`)

如果在某个相同时刻两个时间线对同一对象的同一属性分别指定了不同的值，那么就需要给这两个时间线分别规定 `id`, 例如




```
\tikz [animate = {
  id = 1, n:shift = { 0s = "{(0,0)}", 2s = "{(0,5mm)}", begin on = click },
  id = 2, n:shift = { 0s = "{(0,0)}", 2s = "{(5mm,0)}", begin on = click }
}]{
  \draw[help lines] (0,0) grid[step=1mm] (0.5,0.5);
  \node (n) [opacity=0.3, fill = blue!20, draw=blue, very thick] {The node};}
```

如果不使用 `id` 选项，那么……（试了一下，没有动画）

如果这两个时间线相互冲突，那么 Ti*k*Z 按如下规则来决定哪个时间线是“有效的”：

1. 如果在当前时刻没有动画，即在当前时刻所有动画都已结束或尚未开始，或根本没有动画，就把选项 /tikz/animate/base →P. 976 的值作为属性的当前值。如果没有设置 `base` 的值，那么属性的值就依照环境的设置，此时没有动画。
2. 如果当前时刻有数个动画，那么最近开始的动画有效。
3. 如果当前时刻有数个动画并且这些动画开始于同一时刻，那么代码最近的动画有效。

注意这些规则对画布变换无效，因为画布变换总是有效的，其效果是“累积”的。

54.2.6 指定时刻

/tikz/animate/time=⟨*time*⟩`later` (no default)

这里的单词 `later` 是可选的。

Time Parsing. ⟨*time*⟩ 会被命令 `\pgfparsetime` 解析，这个命令类似 Ti*k*Z 的数学解析器，它会把结果解释为时间（以“秒 s”为单位），例如 `2+3` 就是“5 seconds”；`2*(2.1)` 就是“4.2 seconds”；带长度单位的 `1in` 会被解释为“72.27 seconds”（因为 1in = 72.26999pt）。书写 ⟨*time*⟩ 时注意以下几点：

- 如果 ⟨*time*⟩ 中只有数字，那么 TikZ 会自动在数字后面加时间单位 `s`, 也就是说，写下 `5s` 与写下 `5` 是等效的。
- 如果 ⟨*time*⟩ 中使用 `ms`, 则 `ms` 被解释为“毫秒”，例如 `2ms` 等于 0.002s.
- 如果 ⟨*time*⟩ 中使用 `min`, 则 `min` 被解释为“分钟”。
- 如果 ⟨*time*⟩ 中使用 `h`, 则 `min` 被解释为“小时”。
- 如果 ⟨*time*⟩ 中使用冒号“`:`”, 例如 `1:20`, 则被解释为“1 分 20 秒”，即 `80s`；`01:00:00` 被解释为“1 小时”。
- 不会把 ⟨*time*⟩ 中的数字解释为八进制数。

使用冒号指定时刻时要注意避免歧义，例如 `01:20 = "0"` 可能被这样解释：名称为 `01` 的对象的属性 `20` 的值是 `"0"`——这是无效的；而 `time = 1:20, "0"` 就会得到正常的解释。

Relative Times. 当在 ⟨*time*⟩ 后面使用单词 `later` 时，⟨*time*⟩ 就变成了“相对”时间，它所确定的时刻是前一个时刻之后经过 ⟨*time*⟩ 时间的时刻。

下面的时刻设置

```
\tikz \node :fill = { begin on = click,
  0s = "white",
  500ms later = "red",
  500ms later = "green",
  500ms later = "blue"}
  [fill=blue!20, draw=blue, very thick, circle] {Click me};
```

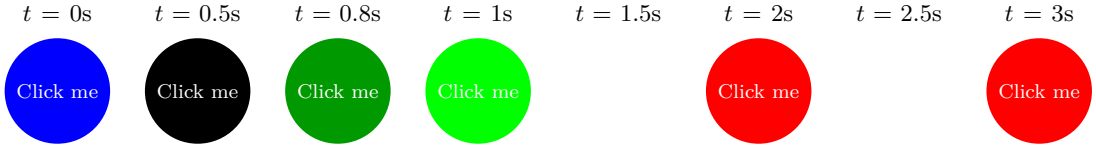
等效于

```
\tikz \node :fill = { begin on = click,
  0s = "white",
  500ms = "red",
  1s = "green",
  1.5s = "blue"}
  [fill=blue!20, draw=blue, very thick, circle] {Click me};
```

Fork Times. 由选项 `time=`⟨*time*⟩ 设置的 ⟨*time*⟩ 并不直接作为时间线中的时刻，把 ⟨*time*⟩ 加上当前的 fork time 后确定的时刻才是时间线中的时刻。

`/tikz/animate/fork=`⟨*t*⟩ (default `0s later`)

本选项把 local scope 中的 fork time 设置为 ⟨*t*⟩，并把当前时刻设置为 `0s`；在这个 local scope 中，写出的时刻如 `2s` 指的是在 ⟨*t*⟩ 之后 2s 的时刻。




```
\tikz [animate/highlight/.style = {
  scope = { fork=#1, :fill={ 0s = "green", 0.5s = "white", 1s="red"} }}]
  \node [animate = { myself: = { :fill = { 0.5s = "black", begin on = click },
      highlight = 1s, highlight = 2s } },
    fill=blue, text=white, very thick, circle, font=\footnotesize] { Click me };
```

上面例子中定义的样式 `/tikz/animate/highlight` 包含一个 `/tikz/animate/scope` →P. 962 选项，`scope` 中的 fork time 是 `#1`. 当 `highlight = 1s` 时，`0s = "green"` 实际上指的是 `1s + 0s` 时刻的值为 `"green"`, `0.5s = "white"` 实际上指的是 `1s + 0.5s` 时刻的值为 `"white"`, `1s="red"` 实际上指的是

1s + 1s 时刻的值为 "red". 当 highlight = 2s 时，0s = "green" 实际上指的是 2s + 0s 时刻的值为 "green". 不过上面图形中显示的是 2s − ε (很接近但还不到 2s) 的时刻的画面 (红色)。

Remembering and Resuming Times. 有时候需要在一个动画过程中的某个时点开启另一个动画，可以使用 later 和 fork time 来实现这一点，使用下面的选项也很方便。

/tikz/animate/remember=⟨*macroname*⟩ (no default)

本选项创建宏 ⟨*macroname*⟩, 并把当前的时刻（由最近的选项 time 决定的时刻，是经过 fork time 换算的时刻）保存在宏 ⟨*macroname*⟩ 中，这个宏是全局有效的。

```
time = 2s,
fork = 2s later, % fork time is now 4s
time = 1s, % local time is 1s, absolute time is 5s (1s + fork time)
time = 1s later, % local time is 2s, absolute time is 6s (2s + fork time)
remember = \mytime % \mytime is now 6s
```

/tikz/animate/resume=⟨*absolute time*⟩ (no default)

⟨*absolute time*⟩ 会被 \pgfparsetime 解析，把解析的结果减去当前的 fork time 作为当前的时刻。若 ⟨*absolute time*⟩ 是选项 remember 定义的宏，则本选项的作用就是返回使用选项 remember 时的时刻（绝对时刻，不是相对于 fork time 的时刻）。

```
fork = 4s,
time = 1s,
remember = \mytime % \mytime 是 5s
fork = 2s, % fork time 是 2s, local time 是 0s
resume = \mytime % local time 是 3s, 绝对时刻是 5s
```

```
scope = {
  fork,
  time = 1s later,
  ...
  remember = \forka
},
scope = {
  fork,
  time = 5s later,
  ...
  remember = \forkb
},
scope = {
  fork,
  time = 2s later,
  ...
  remember = \forkc
},
resume = {max(\forka,\forkb,\forkc)} % "join" the three forks
```

54.2.7 属性的值

/tikz/animate/value=⟨*value*⟩ (no default)

本选项设置当前时间线中的属性的值。⟨*value*⟩ 的书写格式比较多样，不同的属性对应不同的 ⟨*value*⟩ 格式。一般而言，attribute 与 ⟨*value*⟩ 的对应跟 key 与 value 的对应是一致的。例如对应属性 opacity

的 ⟨*value*⟩ 应当是个表达式，表达式的计算结果在 0 到 1 之间；对应属性 color 的 ⟨*value*⟩ 应当是颜色表达式。注意，如果 ⟨*value*⟩ 中含有逗号，则要用花括号把整个 ⟨*value*⟩ 括起来。

⟨*value*⟩ 可以是特殊值 current value, 这个值是在时间线开始时的属性值。例如

```
animate = { obj:color = { 0s = "current value", 2s = "white" } }
```

使用特殊值 current value 时要注意以下几点：

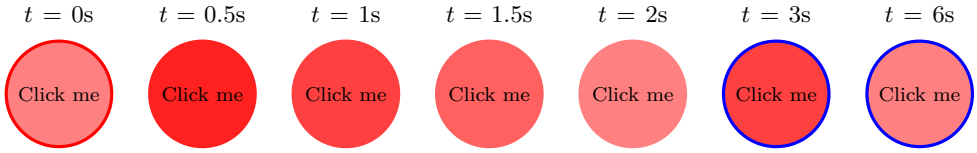
- 如果一个时间线中使用 current value，那么 current value 只能用作第一个时刻的属性值。
- 如果一个时间线中使用 current value，那么这个时间线中只能设置两个时刻。
- 如果一个时间线中使用 current value，那么这个时间线就不支持快照操作（snapshot, 参考选项 /tikz/make snapshot of →P. 980）。

54.2.8 Scopes

当同时设置多个时间线时可以使用下面的选项。

/tikz/animate/scope=⟨*options*⟩ (no default)

⟨*options*⟩ 中的选项会被放入一个 TEX scope 中来执行，这些选项也只在 scope 之内有效。在 scope 之后设置的相对时刻，如 1s later, 所相对的也是 scope 之前的时刻。




```
\tikz \node [animate = { myself: = { begin on = click,
    scope = { attribute = fill, repeats = 3, 0s = "red", 2s = "red!50" },
    scope = { attribute = draw, 0s = "red", 2s = "red!50" }
  }},
  fill=blue!20, draw=blue, very thick, circle, font=\footnotesize] {Click me};
```

如果想延续 scope 结束时的时刻，可以使用下面的选项。

/tikz/animate/sync=⟨*options*⟩ (no default)

这个选项相当于 scope={⟨*options*⟩, remember=\temp}, resume=\temp 其中的 \temp 是内部宏，这样 scope 之内的时刻就会延续到 scope 之后。

54.3 各种句法

在选项 animate=⟨*animation specification*⟩ 之内设置对象、属性值、时刻时，除了使用 ⟨*key*⟩=⟨*value*⟩ 这种键值对的形式外，还可以使用其它比较灵活的句法。

54.3.1 指定对象和属性的句法

设置对象及其属性的句法可以是如下形式：

⟨*object name(s)*⟩:⟨*attribute(s)*⟩={⟨*options*⟩}

或者

⟨*object name(s)*⟩:⟨*attribute(s)*⟩_⟨*id*⟩={⟨*options*⟩}

以上句法形式等效于

sync={ object=⟨*object*⟩, attribute=⟨*attribute*⟩, id=⟨*id*⟩, ⟨*options*⟩, entry }

上面使用冒号的句法比较灵活，例如假设要用 2 个名称为 mynode，othernode 的对象，针对它们的属性 opacity，color 设置动画，下面的设置：

```
animate = {
  mynode:opacity = { 0s = "1", 5s = "0" },
  mynode:color = { 0s = "red", 5s = "blue" },
  othernode:opacity = { 0s = "1", 5s = "0" },
}
```

等效于：

```
animate = {
  mynode: = {
    :opacity = { 0s = "1", 5s = "0" },
    :color = { 0s = "red", 5s = "blue" }
  },
  othernode:opacity = { 0s = "1", 5s = "0" },
}
```

也等效于：

```
animate = {
  :opacity = {
    mynode: = { 0s = "1", 5s = "0" },
    othernode: = { 0s = "1", 5s = "0" }
  },
  mynode:color = { 0s = "red", 5s = "blue" }
}
```

也等效于：

```
animate = {
  {mynode,othernode}:opacity = { 0s = "1", 5s = "0" },
  mynode:color = { 0s = "red", 5s = "blue" }
}
```

对于 Tikz 来说，myself 是个预定义的特殊的对象名称，当某个对象中使用 myself 这个特殊名称后，所设置的动画就是针对该对象的。不能把 myself 作为 node 或环境的名称从而使之具有动画效果。

```
\begin{scope} [animate = {
  myself: = { % Animate the attribute of the scope
    :opacity = { ... },
    :xshift = { ... }
  }
}]
...
\end{scope}
```

54.3.2 关于 myself 的动画

若要让 node, \tikz, \scope, {tikzpicture}, {scope} 生成的图形具有动画效果，可以如下：

1. 在 node 操作后面（在 ⟨*node specification*⟩ 中），node 的内容之前，写下

   :⟨*some attribute*⟩ = {⟨*options*⟩}

   这等效于

[animate = { myself: = { :⟨*some attribute*⟩ = {⟨*options*⟩} } }]

在 node 操作后面可以多次使用这个句法创建多个时间线。

2. 紧跟在 \tikz, \scope, {tikzpicture}, {scope} 后面写下

:⟨*some attribute*⟩ = {⟨*options*⟩}

就会把

[animate = { myself: = { :⟨*some attribute*⟩ = {⟨*options*⟩} } }]

添加到其选项中。可以连续多次使用这个句法创建多个时间线。当遇到开方括号“[”时，对这种句法的解析就会结束。

```
\tikz \node
  :fill opacity = { 0s="1", 2s="0", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Here!};
```

```
\tikz \node
  :fill opacity = { 0s="1", 2s="0", begin on=click }
  :rotate = { 0s="0", 2s="90", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Here!};
```

也可以这样写：

```
\tikz \node
  :fill opacity = { 0s="1", 2s="0", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle]
  :rotate = { 0s="0", 2s="90", begin on=click } {Here!};
```

```
\tikz :fill opacity = { 0s="1", 2s="0", begin on=click }
      :rotate = { 0s="0", 2s="90", begin on=click }
      [ultra thick]
  \node [fill = blue!20, draw = blue, circle] {Here!};
```

但不能这样写：

```
\tikz :fill opacity = { 0s="1", 2s="0", begin on=click }
      [ultra thick]
      :rotate = { 0s="0", 2s="90", begin on=click }
  \node [fill = blue!20, draw = blue, circle] {Here!};
```

因为方括号后面的 :rotate 是无效的。

54.3.3 关于时刻的句法

可以把 time=⟨*time*⟩ 和 value=⟨*value*⟩ 这两个选项合并为一个 time-value pair，即

⟨*time*⟩="⟨*value*⟩"

其中必须使用双引号把 ⟨*value*⟩ 限制起来。当设置多个 time-value pair 时，必须按照时序来写，时间序列必须是（非严格）递增的。

也可以使用下面的句法

⟨*time*⟩=⟨*options*⟩

其中的 ⟨*time*⟩ 应当满足：

1. ⟨*time*⟩ 不能是一个 key, 不能包含冒号，不能以引号开头。
2. ⟨*time*⟩ 可以由数字，正号“+”，负号“-”，点号“.”，或圆括号开头。

⟨*time*⟩=⟨*options*⟩ 会被转换为：

sync = { time = ⟨*time*⟩, ⟨*options*⟩, entry }

54.3.4 引号与属性值

下面的句法

"⟨*value*⟩"base = ⟨*options*⟩

会被转换为

sync = { value = ⟨*value*⟩, ⟨*options*⟩, entry }

若其中的 ⟨*value*⟩ 中含有逗号，则必须用花括号把 ⟨*value*⟩ 括起来。

例如 1s = "red" 会被转换为

```
sync = { time = 1s, sync = { value = red, entry }, entry }
```

注意上面一行代码中的第二个 entry 不能创建一个 time-value pair，因为这个 entry 缺少 ⟨*value*⟩ 项目。

54.3.5 时间表

在创建动画的时间线时，有两种比较有用的思路：

1. 先写出对象，再写属性，再写 time-value pairs，例如：

```
animate = {
  obj:color = {
    0s = "red",
    2s = "blue",
    1s later = "green",
    1s later = "green!50!black",
    10s = "black"
  }
}
```

```
animate = {
  obj: = {
    :color = { 0s = "red", 2s = "green" },
    :opacity = { 0s = "1", 2s = "0" }
  }
}
```

2. 先写时刻，再写对象或属性，例如：

```
animate = {
  0s = {
    obj:color = "red",
    obj:opacity = "1"
  },
  2s = {
    obj:color = "green",
    obj:opacity = "0"
  }
}
```

```
animate = {
  obj: = {
    0s = {
      :color = "red",
      :opacity = "1"
    },
    2s = {
```

```
      :color = "green",
      :opacity = "0"
    }
  }
}
```

```
animate = {
  0s = {
    obj: = {
      :color = "red",
      :opacity = "1"
    },
    main node: = {
      :color = "black"
    }
  },
  2s = {
    obj: = {
      :color = "green",
      :opacity = "0"
    },
    main node: = {
      :color = "white"
    }
  }
}
```

54.4 可用于动画的属性

下面是能用于创建动画的属性，属性名称本身是不带冒号“:”的，但为了容易识别，下文在属性名称前加了冒号。

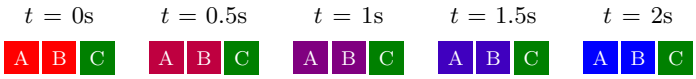
1. :dash phase
2. :dash pattern
3. :dash
4. :draw opacity
5. :draw
6. :fill opacity
7. :fill
8. :line width
9. :opacity
10. :position
11. :path
12. :rotate

13. :scale
14. :stage
15. :text opacity
16. :text
17. :translate
18. :view
19. :visible
20. :xscale
21. :xshift
22. :xskew
23. :xslant
24. :yscale
25. :yshift
26. :yskew
27. :yslant

54.4.1 Animating Color, Opacity, and Visibility

Animation attribute :fill, :draw

这是填充色属性和线条颜色属性。

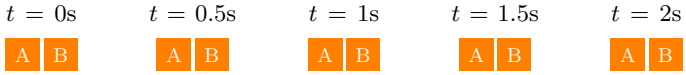



```
\tikz :fill = {0s = "red", 2s = "blue", begin on = click}
    [text = white, fill = orange, font=\footnotesize] {
  \node [fill] at (0mm,0) {A};
  \node [fill] at (5mm,0) {B};
  \node [fill = green!50!black ] at (1cm,0) {C};
}
```

上面代码中的第 3 个 \node 没有动画效果，因为它的选项 fill = green!50!black 覆盖了时间线的填充色设置。

Animation attribute :text

这个属性只能用于 node，它针对 node 的文字内容的颜色，只有把它写在 ⟨*node specification*⟩ 中才有效。



```
\tikz [my anim/.style={ animate = {
        myself:text = {0s = "red", 2s = "blue", begin on = click}}},
      text = white, fill = orange, font=\footnotesize] {
  \node [fill, my anim] at (0,0) {A};
  \node [fill, my anim] at (0.5,0) {B};
}
```

当这个属性用作环境选项时无效。

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s


```
\tikz [animate = {myself:text = {0s = "red", 2s = "blue", begin on = click}},
      text = white, fill = orange, font=\footnotesize] {
  \node [fill] at (0,0) {A};
  \node [fill] at (0.5,0) {B};
}
```

上面代码中的属性 :text 用作 \tikz 的选项，它对 \node 命令无效，node 的文字颜色总是决定于 text = white.

Animation attribute :color

color 并非一个单独的属性，实际上它会同时设置 draw, fill, text 这 3 个属性的值。

Animation attribute :opacity, :fill opacity, :stroke opacity

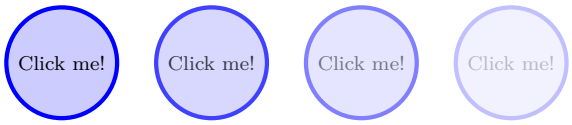
:fill opacity 是填充色的不透明度属性，:stroke opacity 是线条颜色的不透明度属性。

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s


```
\tikz \node :fill opacity = { 0s="1", 2s="0", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle, font=\footnotesize] {Click
  ↪ me!};
```

属性 :opacity 会直接设置整个图形的不透明度（而不是分别设置:fill opacity 与:stroke opacity）。如果能得到驱动的支持，属性 opacity 会把图形作为 transparency group 来处理。

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s




```
\tikz \node :opacity = { 0s="1", 2s="0", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle, font=\footnotesize] {Click
  ↪ me!};
```

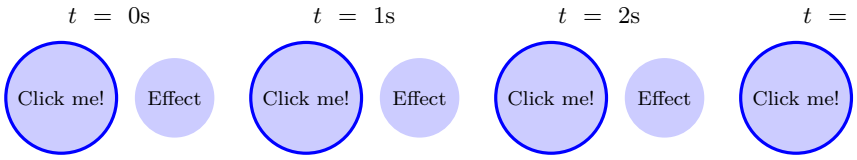
Animation attribute :visible, :stage

属性 :visible 的值是 true 或 false. 注意不可见对象与不透明度为 0 的对象是不同的：不可见对象是不会被渲染出来的，对鼠标的点击也不会做出反应；而不透明度为 0 的对象能对鼠标的点击做出反应。

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s

```
\tikz :visible = {begin on=click, 0s="false", 2s="false"}
  \node (node) [fill = blue!20, draw = blue, very thick, circle, font=
  ↪  \footnotesize] {Click me!};
```

属性 :stage 类似 :visible，不过属性 :stage 的默认值被设为 base="false".

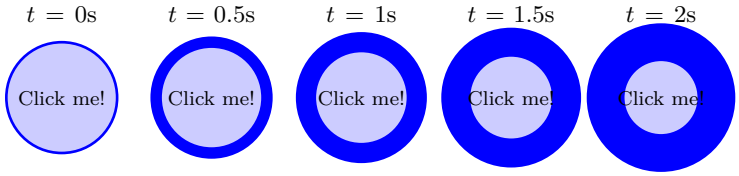



```
\tikz [font=\footnotesize, animate = {example:stage = {
                  begin on = {click, of next=node},
                  0s="true", 2s="true" }}] {
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \node at (1.5,0) (example) [fill = blue!20, circle] {Effect};
}
```

54.4.2 Animating Paths and their Rendering

Animation attribute :line width

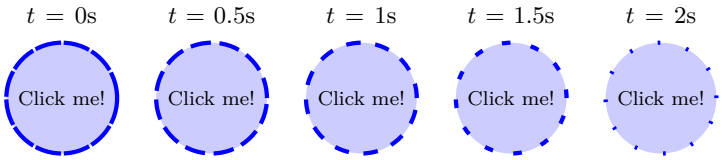
这个属性针对路径的线宽。注意此属性的值必须是带长度单位的数字或表达式，不能使用“thin, thick”这种样式（style）. 注意“线宽”是图形的状态参数，不属于路径本身，一般不会计入图形的边界盒子内。




```
\tikz \node :line width = { 0s="1pt", 2s="5mm", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle, font=\footnotesize] {Click
  ↪  me!};
```

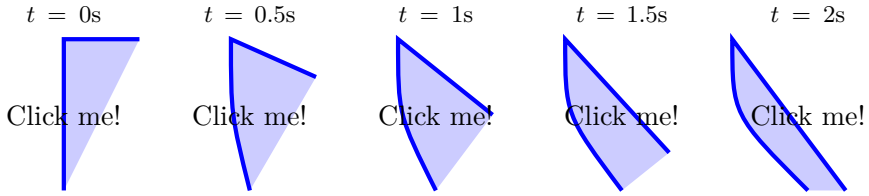
Animation attribute :dash, :dash phase, :dash pattern

属性 :dash 的值是用 on 和 off 构造的。




```
\tikz \node :dash = { 0s="on 10pt off 1pt phase 0pt",
                      2s="on 1pt off 10pt phase 0pt", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle, font=\footnotesize] {Click
  ↪  me!};
```

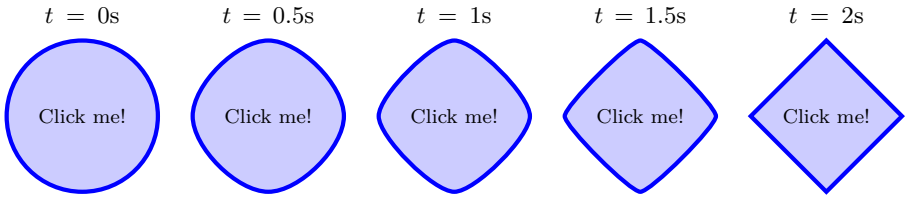
Animation attribute :path



```
\tikz \node :path = {
    0s = "{(0,-1) .. controls (0,0) and (0,0) .. (0,1) -- (1,1)}",
    2s = "{(0,-1) .. controls (-1,0) and (-1,0) .. (-1,1) -- (.5,-1)}", begin
    ↪  on=click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

关于属性 :path 的值，即各种路径，要注意以下几点：

- 这些路径会被放入一个受到保护的 scope 中来执行，尽量减少其副作用，最好不要对这些路径作“fancy things”.
- 这些路径的结构应当是类似的，即它们应当是使用相同的命令、操作构造的，它们之间的区别仅仅是坐标点的不同。例如，若在 1s 时的路径是 rectangle，那么在 2s 时的路径就不能是 circle. 不过画圆的操作 circle 和控制曲线可以同时作为属性 path 的值：




```
\tikz \node :path = {begin on=click,
    0s = "{(0,0) circle [radius=1cm]}",
    2s = "{(0,0) (1,0) .. controls +(0,0) and +(0,0) .. (0,1)
                       .. controls +(0,0) and +(0,0) .. (-1,0)
                       .. controls +(0,0) and +(0,0) .. (0,-1)
                       .. controls +(0,0) and +(0,0) .. (1,0)
                       -- cycle (0,0)}"}
  [fill = blue!20, draw = blue, ultra thick, circle, font=\footnotesize]
  ↪  {Click me!};
```

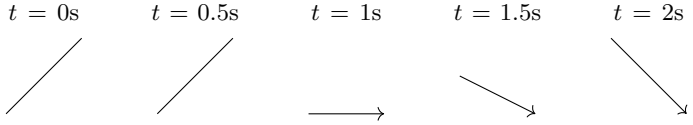
- 若路径被做成动画并且要在该路径上添加箭头，就必须使用选项 /tikz/animate/arrows 来设置箭头，不能使用通常的办法加箭头。

/tikz/animate/arrows=⟨*arrow spec*⟩ (no default)

这个选项只能用在属性 :path 中，用来指定箭头类型。本选项的作用是：把箭头作为一个 marker 放到路径的开端或末端，并且将箭头旋转，使之能指向路径的切线方向——这种机制与通常的添加箭头的选项 /tikz/arrows 并不一样，不过同样要求路径必须是开路径（不能是闭合路径），也要求路径必须有足够的长度。⟨*arrow spec*⟩ 是通常的指定箭头的句法，但是不支持正常箭头的 bend 选项。下面的代码会导致错误：

```
\draw :path = { 1s = "{(0,0) -- (1,0)}", 2s = "{(0,1) -- (1,0)}" }
  [->] (0,0) -- (1,0);
```

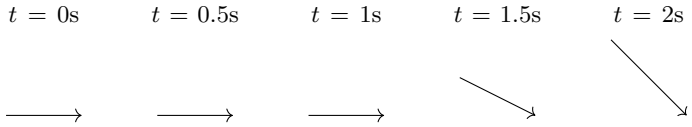
因为这个路径有动画效果，但却使用通常的方法给路径加箭头，应当改为如下用法：



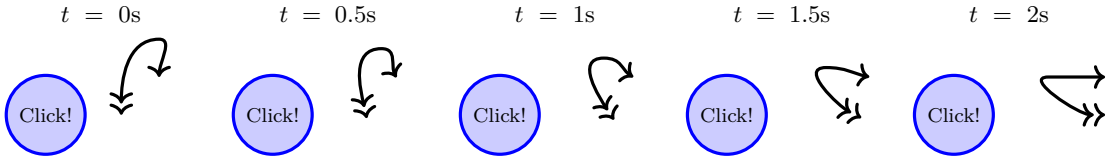

```
\tikz \draw :path = { 1s = "{(0,0) -- (1,0)}", 2s = "{(0,1) -- (1,0)}", arrows =
↪  -> }
  (0,0)--(1,1);
```

这种加箭头的办法有一个缺点：当没有动画运行的时候，路径上就没有箭头显示出来。解决这个问题

的办法是使用选项 base 将某个带箭头的路径做成“基础路径”（“base” path）：




```
\tikz \draw :path = { 1s = "{(0,0) -- (1,0)}" base, 2s = "{(0,1) -- (1,0)}",
↪  arrows = -> };
```




```
\tikz [very thick] {
  \node (node) at (-1,0)
    [fill = blue!20, draw = blue, very thick, circle, font=\footnotesize] {Click!
    ↪  };
  \draw :path = {
    0s = "{(0,0) to[out=90, in=180] (.5,1) to[out=0, in=90] (.5,.5)}" base,
    2s = "{(1,0) to[out=180, in=180] (.25,.5) to[out=0, in=180] (1,.5)}",
    arrows = .<-, begin on = {click, of=node} }; }
```

/tikz/animate/shorten <=⟨*dimension*⟩ (no default)

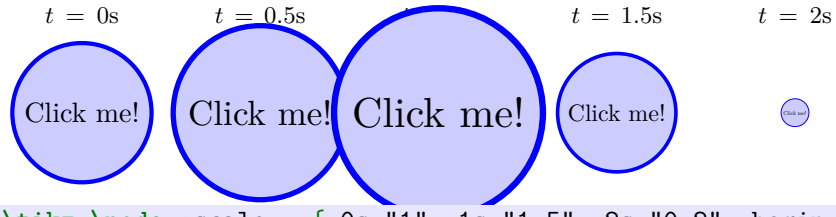
/tikz/animate/shorten >=⟨*dimension*⟩ (no default)

这两个选项只能用在属性 :path 中。

54.4.3 动态变换: Relative Transformations

下面所说的属性能实施变换，变换的效果是累计的。有的属性所做的变换能影响图形边界盒子的计算（动画过程被限制在边界盒子内），有的属性所做的变换则不影响图形边界盒子的计算。

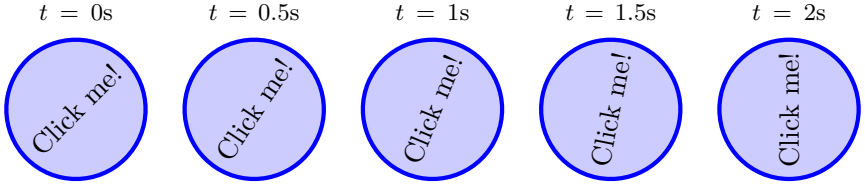
Animation attribute :scale, :xscale, :yscale




```
\tikz \node :scale = { 0s="1", 1s="1.5", 2s="0.2", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

上面图形中，图形的边界盒子由正常的 node 决定，由属性 :scale 导致的 node 尺寸变化是画布变换，不会对图形的边界盒子产生影响。

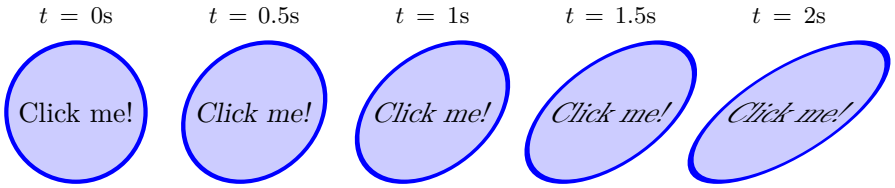
Animation attribute :rotate



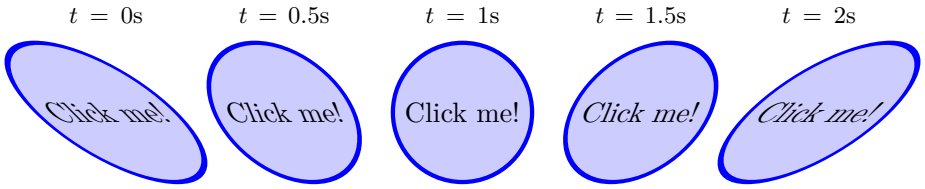

```
\tikz \node :rotate = { 0s="45", 2s="90", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

注意没有 rotate around 这个属性，不过可以使用 /tikz/animate/options/origin →P. 974 选项来改变旋转的“原点”。

Animation attribute :xskew, :yskew, :xslant, :yslant

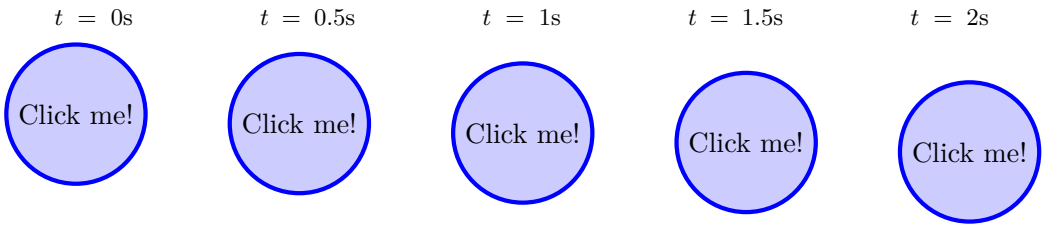



```
\tikz \node :xskew = { 0s="0", 2s="45", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

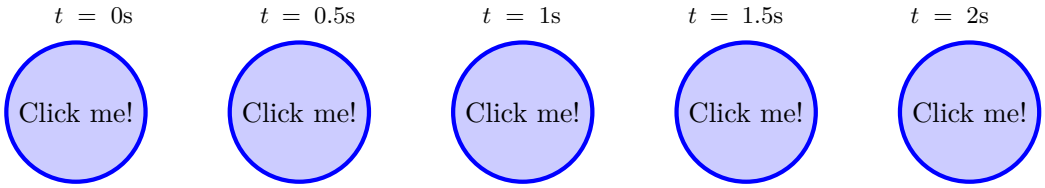



```
\tikz \node :xslant = { 0s="-1", 2s="1", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

Animation attribute :xshift, :yshift



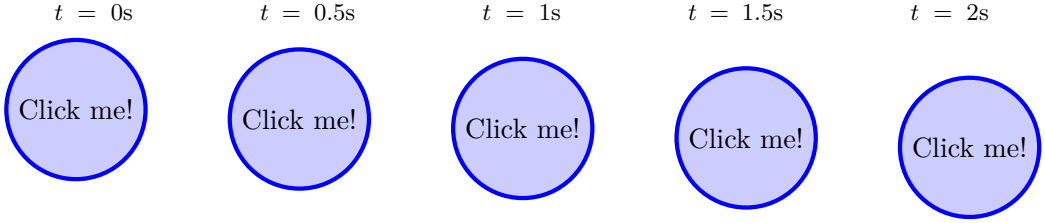

```
\tikz \node :shift = { 0s="{(0,0)}", 2s="{(5mm,-5mm)}",begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```




```
\tikz \node :xshift = { 0s="0pt", 2s="5mm", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

Animation attribute :shift

下面是使用 :shift 的一种方式：

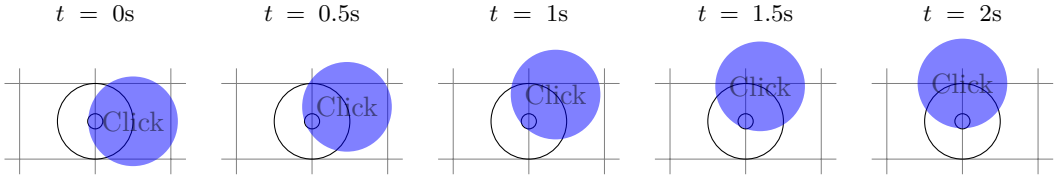



```
\tikz \node :shift = { 0s = "{(0,0)}", 2s = "{(5mm,-5mm)}", begin on = click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

另一种使用 :shift 的方式是配合选项 along：

/tikz/animate/options/along={⟨path⟩} ⟨sloped or uoright⟩ in ⟨time⟩ (no default)

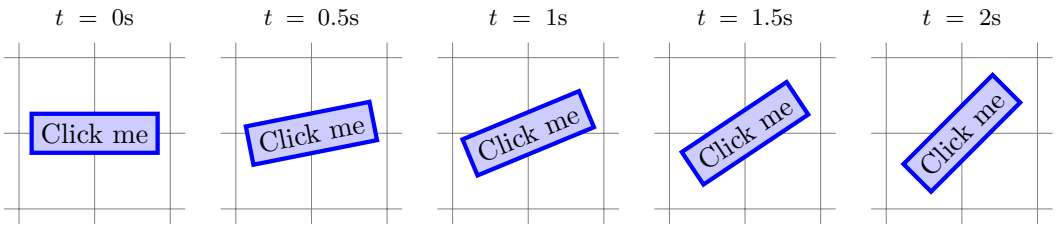
这个选项用在属性 :shift 或 :position 中，可以使得对象沿着 ⟨path⟩ 移动。使用本选项后，编写的属性值就不能再是坐标，而应该是“小数”，数值 0 代表 ⟨path⟩ 的起点，数值 1 代表 ⟨path⟩ 的终点。⟨path⟩ 必须放入花括号中，关键词 sloped 或 upright 是必选的，而 in⟨time⟩ 是可选的。如果写出可选项 in 8s，则它等效于 0s="0", 8s="1"，也就是说，在从 0s 到 8s 的时间内，对象沿着路径，从路径的起点运动到路径的终点。




```
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm];
  \draw (1,0.5) circle[radius=5mm];
  \node :shift = {
      along = {(0,0) circle[radius=5mm]} upright,
      0s="0", 2s=".25", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```

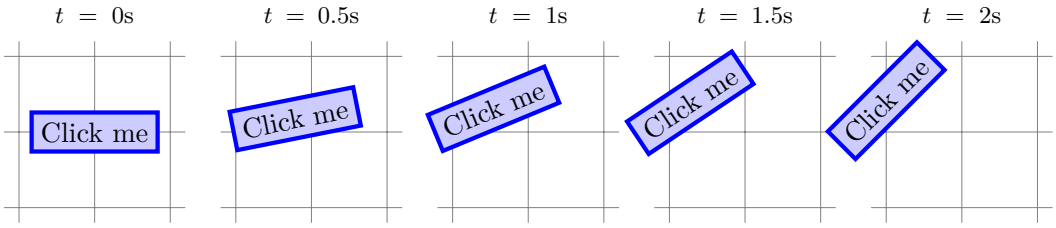
上面例子中，node 的圆心位于坐标点 (1,0.5)，以这个点为原点的坐标系是 animation coordinate system，选项 along 规定的路径 {(0,0) circle[radius=5mm]} 就是 animation coordinate system 中的路径，node 的中心沿着这个路径移动形成动画效果。

前面提到了 animation coordinate system，对于像 :shift 或:scale 这样的变换，需要一个参照系，即 animation coordinate system，这个参照系默认为对象的本地坐标系（local coordinate system of the to-be-animated object）.




```
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :rotate = { 0s="0", 2s="45", begin on=click}
    at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
}
```

上面例子中，node 的旋转中心是 node 的中心点 (1,1)，以 (1,1) 为原点的 animation coordinate system 是 node 的本地坐标系，是旋转变换的参照系。对比下面的例子：



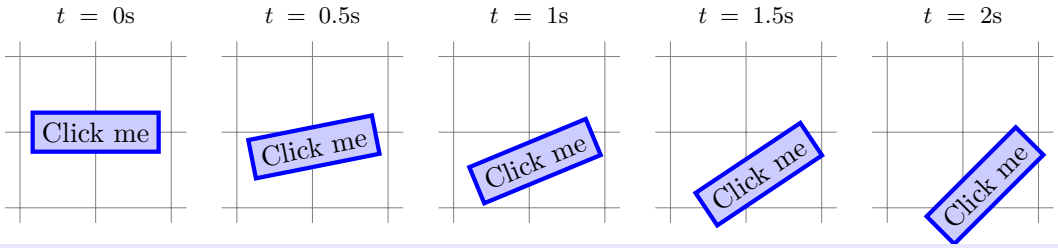
```
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \scoped :rotate = { 0s="0", 2s="45", begin on={click, of next=n} }
    \node (n) at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
}
```

上面例子中，动画的对象是个 scope, 所以旋转变换的参照系 animation coordinate system 就是 scope 的坐标系，而不是 node 自己的坐标系。

可以用下面的选项平移 animation coordinate system.

/tikz/animate/options/origin=⟨*coordinate*⟩ (no default)

本选项以 ⟨*coordinate*⟩ 为平移向量，来平移 animation coordinate system.




```
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :rotate = { 0s="0", 2s="45", begin on=click, origin = {(1,0)}}
    at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
}
```

上面例子中，animation coordinate system 的原点本来位于图形坐标系的 (1,1) 点处，选项 origin = {(1,0)} 将 animation coordinate system 的原点平移到 (2,1)，所以旋转变换的中心就是 (2,1).

下面的选项能够对 animation coordinate system 做更复杂的变换。

/tikz/animate/options/transform=⟨*transformation keys*⟩ (no default)

这个选项对 animation coordinate system 做变换。⟨*transformation keys*⟩ 中可以使用 shift, rotate, reset cm, cm 等选项来对 animation coordinate system 做变换。例如 origin=⟨*c*⟩ 等效于 transform = {shift = ⟨*c*⟩}.

这个选项只影响动画过程，并不影响对象本身的尺寸和形状。

54.4.4 Animating Transformations: Positioning

假设在图形环境的坐标系中 node 的中心在 (1,1), 若要（在图形环境的坐标系中）将 node 平移到 (2,1) 再平移到 (2,0), 则需要如下设置：

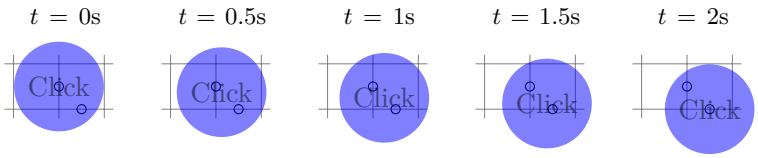
```
\node :shift = { 0s="{(1,1)}", 2s="{(1,0)}", 3s="{(1,-1)}", begin on=click }
  at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
```

其中的坐标 (1,0), (1,-1) 是把图形环境的坐标系中的 (2,1) 和 (2,0) 转换到 animation coordinate system 后的坐标，这里需要进行坐标变换。如果要省去这种坐标变换，可以使用属性 :position 把上面的代码改为：

```
\node :position = { 0s="{(1,1)}", 2s="{(2,1)}", 3s="{(2,0)}", begin on=click }
  at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
```

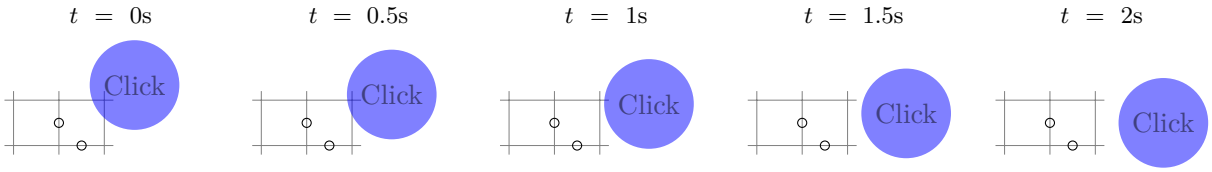
Animation attribute :position

对比下面的例子。



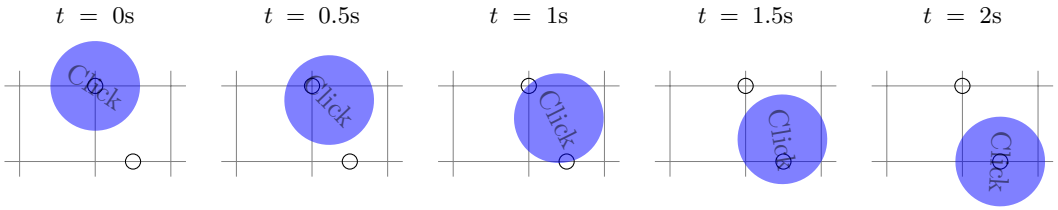

```
\tikz [scale=0.6]{
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm] (1.5,0) circle [radius=1mm];
  \node :position = { 0s="{(1,.5)}", 2s="{(1.5,0)}", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```

上面例子使用属性 :position, 仅仅把这个属性替换为 :shift 则是下面的效果:




```
\tikz [scale=0.6]{
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm] (1.5,0) circle [radius=1mm];
  \node :shift = { 0s="{(1,.5)}", 2s="{(1.5,0)}", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```

在属性 :position 中可以使用选项 along.




```
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,1) circle [radius=1mm] (1.5,0) circle [radius=1mm];
  \node :position = {
      along = {(1,1) to[bend left] (1.5,0)} sloped in 2s, begin on = click }
    at (1,1) [fill = blue, opacity=.5, circle] {Click};
}
```

上面例子中，由于选项 along 处于属性 :position 中，所以路径 {(1,1) to[bend left] (1.5,0)} 是图形环境坐标系中的路径，而不是 node 的本地坐标系中的路径。

54.4.5 Animating Transformations: Views

Animation attribute :view

这个属性实施画布变换，参考 views 库。

54.5 调控时间线

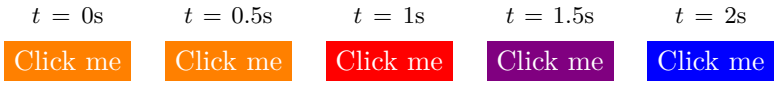
54.5.1 Before and After the Timeline: Value Filling

一个动画通常存在于某个时间区间 $[t_1, t_2]$ 内，在此时间区间外对象的外观与动画无关。但下面的选项能超出此区间起作用：

`/tikz/animate/base=`⟨*options*⟩ (no default)

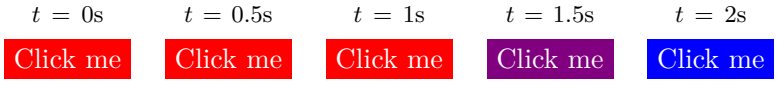
当时间线尚未启动时或已经结束后，对象的属性值就由本选项规定。

可以使用 `base="`⟨*value*⟩`"` 这种格式：



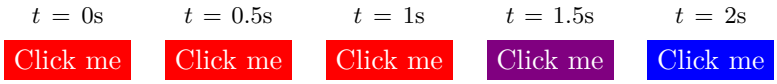

```
\tikz \node [fill = green, text = white] :fill =
    { 1s = "red", 2s = "blue", base = "orange", begin on = click }
  {Click me};
```

可以使用 `"`⟨*value*⟩`"=base` 这种格式：




```
\tikz \node [fill = green, text = white] :fill =
    { 1s = {"red" = base}, 2s = "blue", begin on = click }
  {Click me};
```

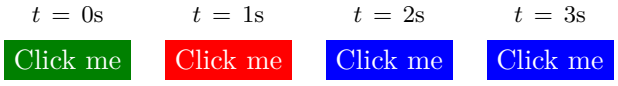
也可以使用 `"`⟨*value*⟩`" base` 这种格式：




```
\tikz \node [fill = green, text = white] :fill =
    { 1s = "red" base, 2s = "blue", begin on = click }
  {Click me};
```

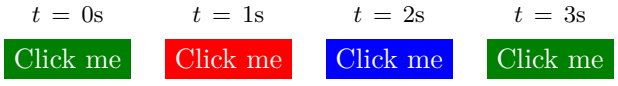
`/tikz/animate/options/forever` (no value)

这个选项使得时间线的延续时间是“永远”，这会把对象的属性冻结在某个状态。




```
\tikz \node :fill = { 1s="red", 2s="blue", forever, begin on=click}
  [fill = green!50!black, text = white] {Click me};
```

对比：




```
\tikz \node [fill = green!50!black, text = white]
    :fill = { 1s = "red", 2s = "blue", begin on = click }
  {Click me};
```

`/tikz/animate/options/freeze` (no value)

这是 `/tikz/animate/options/forever` 的别名。

54.5.2 Beginning and Ending Timelines

下面的选项对动画的开始时刻与结束时刻有较强的控制力，但是当执行 snapshots 操作时，这些选项无效，参考 `/tikz/make snapshot of` → P. 980.

/tikz/animate/options/begin=⟨*time*⟩ (no default)

把图形展示出来的时刻记为 t_0, 本选项把时刻 $t_0 + \langle time \rangle$ 作为动画开始的时刻。可以多次使用本选项，在各个 begin 选项规定的时刻，动画都会“重启”（不管都动画是否已经开始或结束）。如果不给出 begin 选项，那么实际效果等效于使用选项 begin=0s.

本选项的 ⟨*time*⟩ 可以是负值时刻。当执行 snapshots 操作时，本选项无效，参考 /tikz/make snapshot of → P. 980.

/tikz/animate/options/end=⟨*time*⟩ (no default)

把图形展示出来的时刻记为 t_0, 本选项把时刻 $t_0 + \langle time \rangle$ 作为动画停止的时刻（不管动画进行到哪个步骤，都被终止）。当执行 snapshots 操作时，本选项无效，参考 /tikz/make snapshot of → P. 980.

/tikz/animate/options/begin on=⟨*options*⟩ (no default)

⟨*options*⟩ 中的选项会被冠以路径 /pgf/animation/events/ 来执行。本选项把 ⟨*options*⟩ 指定的事件所发生的时刻作为开启动画的“moment 0s”. 例如 begin on = {click, of = X}, 其意思是当用鼠标点击对象 X （这是个“事件”）时就开启动画。

/pgf/animation/events/of=⟨*id*⟩.⟨*type*⟩ (no default)

这里的 ⟨*id*⟩ 是图形中某个对象的“id”（名称），⟨*id*⟩ 所引用的应该是已经被创建的对象，即之前已经创建的名称为 ⟨*id*⟩ 的对象。

```
\tikz [very thick] {
  \node (X) at (1,1.2) [fill = blue!20, draw = blue, circle] {1};
  \node (X) at (1,0.4) [fill = orange!20, draw = orange, circle] {2};
  \node (node) :rotate = {0s="0", 2s="90", begin on = {click, of = X}}
    [fill = red!20, draw = red, rectangle] {Anim};
  \node (X) at (1,-0.4) [fill = blue!20, draw = blue, circle] {3};
  \node (X) at (1,-1.2) [fill = blue!20, draw = blue, circle] {4}; }
```

上面代码中的 of = X 指的是第二个 node.

/pgf/animation/events/of next=⟨*id*⟩.⟨*type*⟩ (no default)

本选项类似 of, 不过这里的 ⟨*id*⟩ 是之后将要被创建的对象的名称。

```
\tikz [very thick] {
  \node (X) at (1,1.2) [fill = blue!20, draw = blue, circle] {1};
  \node (X) at (1,0.4) [fill = blue!20, draw = blue, circle] {2};
  \node (node) :rotate = {0s="0", 2s="90", begin on = {click, of next = X}}
    [fill = red!20, draw = red, rectangle] {Anim};
  \node (X) at (1,-0.4) [fill = orange!20, draw = orange, circle] {3};
  \node (X) at (1,-1.2) [fill = blue!20, draw = blue, circle] {4}; }
```

上面代码中的 of next = X 指的是第四个 node.

/pgf/animation/events/event=⟨*event name*⟩ (no default)

⟨*event name*⟩ 是事件的名称、类型。“plain svg”支持的事件类型有：click, focusin, focusout, mousedown, mouseup, mouseover, mousemove, mouseout, begin, end, 与这些事件类型对应的“简写形式”如下：

/pgf/animate/events/click (no value)

这是 event=click 的简写。这个事件指的是“鼠标点击某个对象”，或与之等效的事件。

/pgf/animation/events/mouse down (no value)

这是 event=mousedown 的简写。这个事件指的是“在某个对象上面按下鼠标按钮”。

/pgf/animation/events/mouse up (no value)

这是 event=mouseup 的简写。这个事件指的是“在某个对象上面弹起鼠标按钮”。

/pgf/animation/events/mouse over (no value)

这是 event=mouseover 的简写。这个事件指的是“鼠标在某个对象上面悬停”。

/pgf/animation/events/mouse move (no value)

这是 event=mousemove 的简写。这个事件指的是“鼠标在某个对象上面滑过”。

/pgf/animation/events/mouse out (no value)

这是 event=mouseout 的简写。这个事件指的是“鼠标从某个对象上面移走”。

/pgf/animation/events/begin (no value)

这是 event=begin 的简写。这个事件指的是“其他某个动画开始”。

```
\tikz \node [animate = {
    myself:rotate = { 0s="0", 2s="90", begin on = {begin, of next=anim}},
    myself:xshift = { 0s="0mm", 2s="5mm", begin on = {click}, name=anim}
  },
  fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

上面例子中，第二个时间线中使用选项 name=anim 指定了此时间线的名称，在第一个时间线中使用 of next=anim 引用了第二个时间线。

/pgf/animation/events/end (no value)

这是 event=end 的简写。这个事件指的是“其他某个动画结束”。

/pgf/animation/events/focus in (no value)

这是 event=focusin 的简写。这个事件指的是“动画对象获得焦点”，即对象获得光标。

/pgf/animation/events/focus out (no value)

这是 event=focusout 的简写。

/pgf/animation/events/repeat=⟨*number*⟩ (no default)

这个事件指的是：当某个重复播放的动画被重复 ⟨*number*⟩ 次。

/pgf/animation/events/key=⟨*key*⟩ (no default)

这个事件指的是：键盘上的键 ⟨*key*⟩ 被按下。

/pgf/animation/events/delay=⟨*time*⟩ (no default)

当事件发生时，本选项不会让动画立即开始，而是延迟 ⟨*time*⟩ 再开始。

/tikz/animate/options/restart=⟨*choice*⟩ (default true)

⟨*choice*⟩ 有以下选择：

- true, 每当事件发生时，动画都会被重新开始，不管动画是否正在进行。
- false, 一旦动画开始，就不会再重新开始。
- never, 等效于 false.
- when not active, 当事件发生并且动画不在进行中时，动画会被重新开始。

/tikz/animate/options/end on=⟨*options*⟩ (no default)

类似 begin on, 只是针对动画的“停止”。

54.5.3 Repeating Timelines and Accumulation

/tikz/animate/options/repeats=⟨*specification*⟩ (no default)

这个选项决定时间线是否重复播放，以及如何重复播放。⟨*specification*⟩ 由两部分构成，第一部分可以是：

- 留空，那么时间线会被不断重复。

```
\tikz \node :rotate = { 0s = "0", 2s = "90", repeats, begin on = click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

- 一个数字 ⟨*number*⟩，例如 2 或 3.25, 那么时间线会被重复播放 ⟨*number*⟩ 次。
- 一段文字“for ⟨*time*⟩”, 例如 for 2s, 那么时间线会被重复播放，但重复播放的时间不超过 ⟨*time*⟩.

⟨*specification*⟩ 的第二部分可以是：

- 留空，那么每当时间线被重复时，其状况与前一次执行时间线时的状况无异。
- 文字 accumulating, 那么之前时间线结束时的状态会被作为之后（将重复执行的）时间线的初始状态。例如，假设某个对象位于原点，一个时间线的效果是把此对象向右移动 1cm, 如果不设置文字 accumulating, 那么当这个时间线重复播放时，这个对象就总是在 (0,0) 与 (1cm,0) 这两个位置之间摆动；如果设置文字 accumulating, 那么当这个时间线重复播放 5 次后，这个对象就会移动到 (5cm,0).

/tikz/animate/options/repeat=⟨*specification*⟩ (no default)

这是 /tikz/animate/options/repeats 的别名。

54.5.4 Smoothing and Jumping Timelines

在设置时间线时，用到的只是一个或数个“time-value pair”，这是离散的值。为了形成连续的动画效果，需要在离散的值之间作插值计算。插值计算的方式不止一种，除了线性插值外，使用 time-attribute curve 也是一种插值计算方式。假设以时间为横轴，某个属性的值（假设为纯数字）为纵轴做成一个坐标系，那么下面的曲线：

```
(0s,50) .. controls (5s,50) and (9s,100) .. (10s,100)
```

就是一个 time-attribute curve, 曲线上的每一个点都对应一个“time-value pair”，其中的第一个 control point, 即 (5s,50) 称为“exit control”，第二个 control point, 即 (9s,100) 称为“entry control”. 实际上，time-attribute curve 是通过一个映射来计算的，对于上面的例子来说，规定对应关系：

(0s,50)→(0,0), (10s,100)→(1,1), (5s,50)→(0.5,0), (9s,100)→(0.9,1)

就把 time-attribute curve 映射为：

```
(0,0) .. controls (0.5,0) and (0.9,1) .. (1,1)
```

/tikz/animate/options/exit control={⟨*time fraction*⟩}{⟨*value fraction*⟩} (no default)

本选项指定 time-attribute curve 的第一个 control point 点“exit control”, 例如上面例子中的 time-attribute curve 可以设置为：

```
exit control={0.5}{0},
entry control={0.9}{1},
0s = "50",
10s = "100"
```

注意其中 exit control, entry control 的值是“映射值”（纯数字）。

当使用这种方式定义 time-attribute curve 时，exit control 与 entry control 最好是完全不同的两个“time-value pair”，即二者的时刻不同、值也不同。

/tikz/animate/options/entry control={⟨*time fraction*⟩}{⟨*value fraction*⟩} (no default)

本选项指定 time-attribute curve 的第二个 control point 点“entry control”.

/tikz/animate/options/ease in={⟨*fraction*⟩} (default 0.5)

这是 entry control={1-⟨*fraction*⟩}{1} 的简写。

/tikz/animate/options/ease out={⟨*fraction*⟩} (default 0.5)

这是 exit control={⟨*fraction*⟩}{1} 的简写。

/tikz/animate/options/ease={⟨*fraction*⟩} (default 0.5)

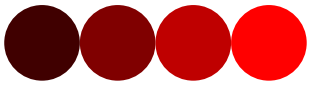
这是 ease in={⟨*fraction*⟩}, ease out={⟨*fraction*⟩} 的简写。

/tikz/animate/options/stay (no value)

/tikz/animate/options/jump (no value)

54.6 Snapshots

Tikz 可以获得动画过程在某些个时刻的“状态”，并把这些“状态”插入到 PDF 文件中，这种操作类似拍“快照”，通过这些快照可以大致了解动画过程。



```
\foreach \t in {0.5, 1, 1.5, 2}
  \tikz [make snapshot of = \t]
    \fill :fill = {0s="black", 2s="red"} (0,0) circle [radius = 5mm];
```

/tikz/make snapshot of=⟨*time*⟩ (no default)

当在 TEX scope 中使用本选项后，scope 中的动画命令不再向输出中添加动画代码。Tikz 会计算动画过程在时刻 ⟨*time*⟩ 的状态（包括各种命令、属性值等），并把这个状态插入到输出中，从而得到动画过程在时刻 ⟨*time*⟩ 的画面。

```
\tikz [make snapshot of = 1s] {
  \fill :fill = { 0s = "black", 2s = "white" } (0,0) rectangle ++(1,1);
  \fill :fill = { 1s = "black", 3s = "white" } (2,0) rectangle ++(1,1);
}
```

动画开始的时刻是其“moment zero”，快照时刻 ⟨*time*⟩ 是“moment zero”之后 ⟨*time*⟩ 的时刻。有的动画需要用户触发某个事件才会开始，例如由选项 begin on={click} 规定的事件，当执行拍快照操作时，这种“触发特性”会失效。

下面的选项对拍快照的时刻具有调整作用：

/tikz/animate/options/begin snapshot=⟨*start time*⟩ (no default)

假设动画原来的“moment zero”是 t_0 时刻，原来的快照时刻是 ⟨*time*⟩，那么本选项会让 Tikz 把 $t_0 + \langle start\ time \rangle$ 假定为动画的“moment zero”时刻，这使得快照时刻实际上变成 $\langle time \rangle - t_0$.

```
\tikz [make snapshot of = 1s] {
  \fill :fill = { 0s = "black", 2s = "white", begin snapshot = 1s }
  ↪  (0,0) rectangle ++(1,1);
  \fill :fill = { 1s = "black", 3s = "white" } (2,0) rectangle
  ↪  ++(1,1);
}
```

对于快照的计算完全由 Tikz 完成，限于 TEX 的计算能力，对快照的计算可能不是非常精确、快速，另外还要注意以下几点：

- 前面已提到，那些由用户触发事件才开始的动画不支持快照操作，当对这种动画使用快照选项时，设置事件的选项（如 begin, begin on, end, end on）会被自动忽略。

- 特殊值 current value 不能为 Tikz 计算，故不能把它用于快照计算。
- 目前，快照计算不支持具有 accumulating 效果的 /tikz/animate/options/repeats → P. 978.

如果快照时刻 ⟨*time*⟩ 被空置，则取消拍快照操作，动画代码按正常的方式生成动画。

/tikz/make snapshot after=⟨*time*⟩ (no default)

类似 make snapshot of, 只是这里的 ⟨*time*⟩ 会被解释为 ⟨*time*⟩ + ϵ. 假设时间线 l 结束于时刻 t, 此时其中某个属性的值是 v, 这个 v 是属于时间线 l 的，那么选项 make snapshot of=t 得到的快照就使用属性值 v；但因为时间线 l 结束于时刻 t, 所以同在时刻 t 这个属性要变成其它的值 v', 使用本选项后，此时的快照使用属性值 v', 即时刻 t 被延迟了很小的一段时间 ϵ，所得快照中的属性值实际是时刻 $t+\epsilon$ 的值。

```
\tikz [make snapshot of = 2s]
  \fill :fill = { 0s = "green", 2s = "red" } (0,0) rectangle ++(1,1);
\tikz [make snapshot after = 2s]
  \fill :fill = { 0s = "green", 2s = "red" } (0,0) rectangle ++(1,1);
```

/tikz/make snapshot if necessary=⟨*time*⟩ (default 0s)

本选项的作用是：如果输出格式不支持动画，就获取动画在时刻 ⟨*time*⟩ 的快照；如果输出格式支持动画（如 SVG），则按正常方式生成动画，本选项不起作用。

手册的导言区做了如下设置：

```
\tikzset{make snapshot if necessary}
```

所以在手册的 PDF 格式版本中，每个动画示例都会展示在 0s 时刻的快照；而在手册的 SVG 格式版本中，动画都能正常创建。

54.7 一个例子

```
\begin{tikzpicture}
  \def\forktimespec{-15s}
  \def\timeofonecycle{16s}
  \begin{scope}[minimum size=1.4em,above,animate={:opacity={
    sync={n1:={0s="0",1s="0.5",2s="0",base="0",16s="0",repeats},name=a1},
    sync={fork=\forktimespec later,n2:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a2},
    sync={fork=\forktimespec later,n3:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a3},
    sync={fork=\forktimespec later,n4:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a4},
    sync={fork=\forktimespec later,n5:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a5},
    sync={fork=\forktimespec later,n6:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a6},
    sync={fork=\forktimespec later,n7:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a7},
    sync={fork=\forktimespec later,n8:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a8},
    sync={fork=\forktimespec later,n9:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a9},
    sync={fork=\forktimespec later,n10:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a10},
    sync={fork=\forktimespec later,n11:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a11},
```

```
    sync={fork=\forktimespec later,n12:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a12},
    sync={fork=\forktimespec later,n13:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a13},
    sync={fork=\forktimespec later,n14:={0="0",1="0.5",2="0",base="0",
    ↪  \timeofonecycle="0",repeats},name=a14}
}}]
    \node (n1)at(0*1.4em+0.7em,0){劝};
    \node (n2)at(1*1.4em+0.7em,0){君};
    \node (n3)at(2*1.4em+0.7em,0){更};
    \node (n4)at(3*1.4em+0.7em,0){进};
    \node (n5)at(4*1.4em+0.7em,0){一};
    \node (n6)at(5*1.4em+0.7em,0){杯};
    \node (n7)at(6*1.4em+0.7em,0){酒};
    \node (n8)at(0*1.4em+0.7em,0){西};
    \node (n9)at(1*1.4em+0.7em,0){出};
    \node (n10)at(2*1.4em+0.7em,0){阳};
    \node (n11)at(3*1.4em+0.7em,0){关};
    \node (n12)at(4*1.4em+0.7em,0){无};
    \node (n13)at(5*1.4em+0.7em,0){故};
    \node (n14)at(6*1.4em+0.7em,0){人};
  \end{scope}
\end{tikzpicture}
```

上面代码把汉字做成动画，代码的形式很整齐，但编辑起来有点繁琐。下面使用 parser 模块 (\usepgfmodule{parser}) 修改上面的代码：

```
\newcount\charcountone
\newcount\charcounttwo
\def\baocunnode{}
\def\baocunsync{}
\pgfparserdef{Char list}{all}。{\pgfparserswitch{final}}%
\pgfparserdef{Char list}{all}，{\charcountone0}%
\pgfparserdefunknown{Char list}{all}{%
  \advance\charcountone 1%
  \advance\charcounttwo 1%
  \edef\CharCountOne{\the\charcountone}
  \edef\CharCountTwo{\the\charcounttwo}
  \edef\charletter{\pgfparserletter}
  \ifnum \CharCountTwo=1
    \edef\baocunnode{node (n1)at(1*1.4em-1.4em+0.7em,0){\charletter}}
    \edef\baocunsync{sync={n1:={0s="0",0.5s="0.8",1s="0",base="0",30s="0",repeats
    ↪  },name=a1}}
  \else
    \edef\baocunsync{\baocunsync,sync={fork=-29s later,n\CharCountTwo:=
    ↪  {0="0",1="0.8",2="0",base="0",30s="0",repeats},name=a\CharCountTwo}}
    \edef\baocunnode{\baocunnode node (n\CharCountTwo)at(
    ↪  \CharCountOne*1.4em-1.4em+0.7em,0){\charletter}}
  \fi}%
\pgfparserset{Char list/silent=true}%
\pgfparserparse{Char list}渭城朝雨浥轻尘，客舍青青柳色新，劝君更进一杯酒，西出阳关无故人。

\edef\aaaa{animate={:opacity={\baocunsync}}}
\def\bbbb{\begin{scope}[minimum size=1.4em,above,}
\begin{tikzpicture}
\expandafter\bbbb\aaaa]
  \path \baocunnode;
```

```
\end{scope}
\end{tikzpicture}
```

这样每个汉字的显现时间是 1s, 每隔 30s 重复一次诗句。

第五十五章　Decoration 库

55.1　公共选项

Decoration 库提供的一些选项对很多装饰类型都有效，是公共选项，这些选项由 decoration 模块直接定义。有的选项只针对某个装饰类型有效，这种选项由相应的装饰程序库定义。下面介绍公共选项，注意有的公共选项的值保存在 TEX 寄存器中，有的公共选项的值保存在宏中。

注意这些选项（key）的前缀都是 `/pgf/decoration/`，因此都可用在 `decoration={`⟨*options*⟩`}` 中。

`/pgf/decoration/amplitude=`⟨*dimension*⟩ (no default, initially `2.5pt`)

这个选项设置装饰路径的“振幅”（amplitude），例如，装饰类型 zigzag 是沿着被装饰路径放置的“之字形”装饰路径，之字形装饰路径的典型形式是 ⩘，这是个振动形式，其振幅是它的高度（从最下端到最上端）的一半。

本选项通过重设 TEX 寄存器 `\pgfdecorationsegmentamplitude` 的值来发挥作用，可以直接修改这个寄存器的值来调节装饰路径的振幅。

`/pgf/decoration/meta-amplitude=`⟨*dimension*⟩ (no default, initially `2.5pt`)

这个选项针对 meta-decoration 的振幅，本选项设置 TEX 宏 `\pgfmetadecorationsegmentamplitude` 的值。

`/pgf/decoration/segment length=`⟨*dimension*⟩ (no default, initially `10pt`)

有的装饰路径由很多小线段构成（如 zigzag），本选项设置每个小线段的长度。本选项设置 TEX 寄存器 `\pgfdecorationsegmentlength` 的值

`/pgf/decoration/meta-segment length=`⟨*dimension*⟩ (no default, initially `1cm`)

这个选项针对 meta-decoration 的小线段的长度。本选项设置 TEX 宏 `\pgfmetadecorationsegmentlength` 的值。

`/pgf/decoration/angle=`⟨*angle*⟩ (no default, initially `45`)

有的装饰类型具有“角度”属性，例如，装饰类型 `wave`，它由数个小圆弧组成，小圆弧有自己的角度。本选项调整这种角度。本选项设置 TEX 宏 `\pgfdecorationsegmentangle` 的值。

`/pgf/decoration/aspect=`⟨*factor*⟩ (no default, initially `0.5`)

有的装饰类型具有“宽高比例”属性，例如，装饰类型 `brace` 是个大括号，它有宽高比。本选项调整这种“宽高比例”。本选项设置 TEX 宏 `\pgfdecorationsegmentaspect` 的值。

`/pgf/decoration/start radius=`⟨*dimension*⟩ (no default, initially `2.5pt`)

本选项的值直接保存在它自己这里。

`/pgf/decoration/end radius=`⟨*dimension*⟩ (no default, initially `2.5pt`)

本选项的值直接保存在它自己这里。

/pgf/decoration/radius=⟨*dimension*⟩ (style, no default)

同时设置 start radius, end radius 为 ⟨*dimension*⟩。

/pgf/decoration/path has corners=⟨*boolean*⟩ (no default, initially false)

本选项决定装饰路径是否采用圆角。如果装饰路径本身是有“尖角”的，那么设置本选项值为 true 可能会改善装饰路径的外观，但如果装饰路径本身没有尖角，或者组成装饰路径的线段太短，那么设置本选项值为 true 可能会出现意外状况。本选项设置 TEX-if \ifpgfdecoratepathhascorners 的值。

下面以装饰类型 zigzag 和 straight zigzag 为例，看一下选项 amplitude, meta-amplitude, segment length 是如何起作用的。

在库文件《pgflibrarydecorations.pathmorphing.code》中对装饰类型 zigzag 的定义如下：

```
\pgfdeclaredecoration{zigzag}{up from center}{
  \state{up from center}[width=+.5\pgfdecorationsegmentlength, next state=big down]
  {
    \pgfpathlineto{\pgfqpoint{.25\pgfdecorationsegmentlength}{
    ↪  \pgfdecorationsegmentamplitude}}
  }
  \state{big down}[switch if less than=+.5\pgfdecorationsegmentlength to center
  ↪  finish,
                  width=+.5\pgfdecorationsegmentlength,
                  next state=big up]
  {
    \pgfpathlineto{\pgfqpoint{.25\pgfdecorationsegmentlength}{-
    ↪  \pgfdecorationsegmentamplitude}}
  }
  \state{big up}[switch if less than=+.5\pgfdecorationsegmentlength to center finish,
                 width=+.5\pgfdecorationsegmentlength,
                 next state=big down]
  {
    \pgfpathlineto{\pgfqpoint{.25\pgfdecorationsegmentlength}{
    ↪  \pgfdecorationsegmentamplitude}}
  }
  \state{center finish}[width=0pt, next state=final]{
    \pgfpathlineto{\pgfpointorigin}
  }
  \state{final}
  {
    \pgfpathlineto{\pgfpointdecoratedpathlast}
  }
}
```

以上定义代码规定了 5 个状态，即 up from center, big down, big up, center finish, final. 在用 zigzag 装饰路径时，可能出现以下几种状态组合：

- final
- up from center → final
- up from center → big down → final
- up from center → big down → center finish → final
- up from center → big down → big up → final
- up from center → big down → big up → big down → final
- ……

其中的典型组合是 up from center → big down → big up → final，在选项的初始值下这个组合画出的图形相当于

```
\tikz \draw (0,0) -- (2.5pt,2.5pt)--(7.5pt,-2.5pt)--(10pt,0pt);
```

从定义代码看，这个图形的宽度就是 \pgfdecorationsegmentlength 的值，即选项 segment length 的值；这个图形的高度是 \pgfdecorationsegmentamplitude 的值的 2 倍，即“振幅”是选项 amplitude 的值。

文件《pgflibrarydecorations.pathmorphing.code》中对装饰路径 straight zigzag 的定义如下：

```
\pgfdeclaremetadecoration{straight zigzag}{line to}{
  \state{line to}[width=\pgfmetadecorationsegmentlength, next state=zigzag]
  {
    \decoration{curveto}
  }
  \state{zigzag}[width=\pgfmetadecorationsegmentlength, next state=line to]
  {
    \decoration{zigzag}
  }
  \state{final}
  {
    \decoration{curveto}
  }
}
```

从上面的定义代码看，straight zigzag 类型的典型形式由三段构成，第一段是 curveto，第二段是 zigzag，第三段是 curveto，这三段的宽度都是 \pgfmetadecorationsegmentlength 的值，即选项 meta-segment length 的值。

55.2 修饰路径的装饰类型

TikZ Library decorations.pathmorphing

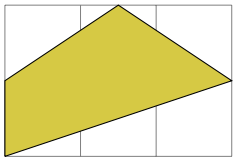
```
\usepgflibrary{decorations.pathmorphing} % LaTeX and plain TeX and pure pgf
\usepgflibrary[decorations.pathmorphing] % ConTeXt and pure pgf
\usetikzlibrary{decorations.pathmorphing} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.pathmorphing] % ConTeXt when using TikZ
```

这种装饰类型会改换被装饰路径的外观，但不会改变被装饰路径的子路径的个数，也不改变被装饰路径的连续性。

55.2.1 由直线段构成的装饰路径

Decoration lineto

这个装饰类型实际上是 decoration 模块定义的，它用直线段代替被装饰路径，无论被装饰路径是曲线还是直线。



```
\begin{tikzpicture}[decoration=lineto]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

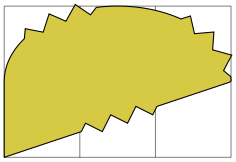
Decoration straight zigzag

这个装饰类型由三段已定义的装饰路径构成：曲线装饰路径、之字形装饰路径、曲线装饰路径，因此是 meta-decoration 类型的。这个装饰类型中的之字形装饰路径有自己的振幅，它是沿着被装饰路径放置的，它的走势随着被装饰路径的弯曲而弯曲。

amplitude 这个选项确定本装饰路径的 zigzag 部分的振幅。

segment length 这个选项确定构成本装饰路径的 zigzag 部分的一个周期的宽度。

meta-segment length 这个选项确定的长度是构成本装饰路径的各段的宽度（见前面的定义代码）。



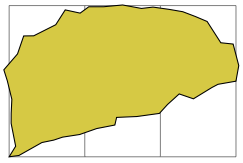
```
\begin{tikzpicture}[decoration={straight zigzag,
    meta-segment length=1.1cm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration random steps

这个装饰类型由许多前后相接的直线段构成，原本每个直线段的端点都应该位于被装饰路径上，不过本装饰类型会使得直线段的端点随机地偏离被装饰路径。这个偏离包括水平方向的偏离 h 和竖直方向的偏离 v，$h, v \in [-d, d]$，这里 d 由选项 amplitude=d 指定。

segment length 本选项确定构成本装饰路径的单个小线段的基本长度。

amplitude 本选项的作用如前述。



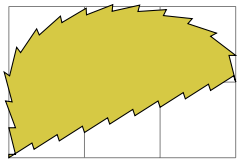
```
\begin{tikzpicture}
    [decoration={random steps,segment length=2mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration saw

这个装饰路径是锯齿形状的。

amplitude 本选项确定锯齿的振幅。

segment length 本选项确定构成锯齿路径的一个锯齿的宽度。



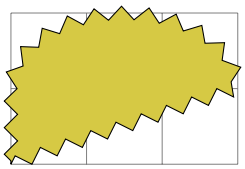
```
\begin{tikzpicture}[decoration=saw]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration zigzag

这个装饰路径是之字形路径。

amplitude 本选项确定之字形路径的振幅。

segment length 本选项确定之字形路径的一个周期的宽度。



```
\begin{tikzpicture}[decoration=zigzag]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

55.2.2 由曲线构成的装饰路径

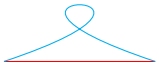
Decoration bent

这个装饰路径由弯曲线条构成。设 aspect=t，当前子输入路径的未装饰部分的长度是 r，amplitude=a，这个装饰类型就是控制曲线：

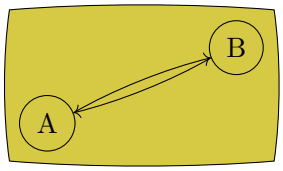
⟨*current point*⟩..controls (t*r, a) and ((1-t)*r, a)..(r,0)

amplitude 这个选项的值越大，装饰线条的就越是弯曲。若 amplitude=0 则没有弯曲，等效于装饰类型 lineto.

aspect 这个选项的值影响控制曲线的两个支撑点在 x 轴方向的位置。



```
\begin{tikzpicture}
    [decoration={bent,aspect=1.5,amplitude=10mm}]
  \draw [red](0,0)--(2,0);
  \draw [decorate,cyan](0,0)--(2,0);
\end{tikzpicture}
```



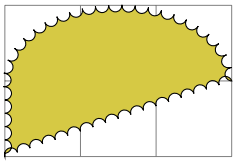

```
\begin{tikzpicture}[decoration={bent,aspect=.3}]
  \draw [decorate,fill=yellow!80!black]
    (0,0) rectangle (3.5,2);
  \node[circle,draw] (A) at (.5,.5) {A};
  \node[circle,draw] (B) at (3,1.5) {B};
  \draw[->,decorate] (A) -- (B);
  \draw[->,decorate] (B) -- (A);
\end{tikzpicture}
```

Decoration bumps

这个装饰类型的构成元素是前后相连的两个“半椭圆弧”。

amplitude 本选项的值确定半椭圆弧的“拱高”。

segment length 本选项的值是两个半椭圆弧的宽度。



```
\begin{tikzpicture}[decoration=bumps]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

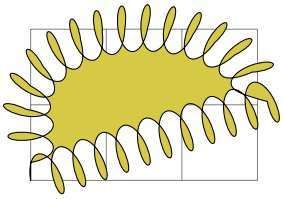
Decoration coil

这个装饰类型的构成元素是“螺线圈”。

amplitude 这个选项的值是螺线圈的振幅。

segment length 这个选项的值大约是，螺线转一圈时起止点之间的直线距离。

aspect 这个选项的值可以调节螺线圈的立体感，一般是它的值越大越有立体感，但如果它的值过大就会导致装饰路径“走样”。如果它的值是 0，则螺线圈近似平面上的正弦曲线。



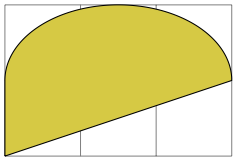
```
\begin{tikzpicture}[decoration={coil,aspect=0.4,
    segment length=3mm,amplitude=3mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration curveto

这个装饰类型是 decoration 模块定义的。文件《pgfmoduledecorations.code》中对装饰类型 curveto 的定义如下：

```
\pgfdeclaredecoration{curveto}{initial}{
  \state{initial}[width=\pgfdecoratedinputsegmentlength/100]
  {
    \pgfpathlineto{\pgfpointorigin}
  }
  \state{final}{\pgfpathlineto{\pgfpointdecoratedpathlast}}
}
```

从这个定义看出，curveto 实际上是用“折线段”来代替原来的被装饰路径。用来替换当前子输入路径的折线段的每个小线段的长度，约是当前子输入路径长度的 $\frac{1}{100}$. 如果原来的路径是直线段，则替换后的外观还是直线段。如果原来的路径是曲线，而且长度不是过长，外观还是近似曲线的。



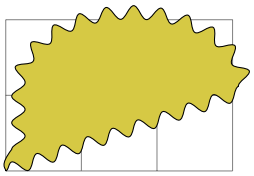
```
\begin{tikzpicture}[decoration=curveto]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration snake

这个装饰类型主要是由类似正弦曲线的曲线段构成的。

amplitude 这个选项的值决定振幅。

segment length 这个选项的值决定一个周期的宽度。



```
\begin{tikzpicture}[decoration=snake]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

55.3 替换路径的装饰类型

TikZ Library decorations.pathreplacing

```
\usepgflibrary{decorations.pathreplacing} % LaTeX and plain TeX and pure pgf
\usepgflibrary[decorations.pathreplacing] % ConTeXt and pure pgf
\usetikzlibrary{decorations.pathreplacing}
↪  % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.pathreplacing] % ConTeXt when using TikZ
```

这种装饰类型会严重改变被装饰路径的连续性，改变被装饰路径的子路径个数，所以当填充装饰路径时，与填充被装饰路径的效果很不一样。

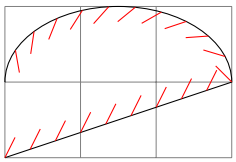
Decoration border

这个装饰类型会沿着被装饰路径画出一些小线段，这些小线段与被装饰路径之间有某个角度，这样被装饰路径就“被标记”了。

segment length 这个选项的值确定相邻两个小线段的间距。

amplitude 这个选项的值确定小线段的长度。

angle 这个选项的值确定小线段与被装饰路径之间的夹角。



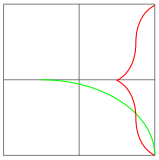
```
\begin{tikzpicture}[decoration={border,amplitude=3mm}]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate,draw,red}]
    (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration brace

这个装饰类型是一个括号，并且只有一个括号，括号的起点位于被装饰路径的起点，括号的方向是被装饰路径在起点处的切线方向；括号的跨度是被装饰路径的总长度。所以当被装饰路径是直线段时，本装饰类型的效果较好。如果被装饰路径是个半圆，那么括号就处于半圆的一侧了。

amplitude 这个选项的值确定括号的“拱高”。

aspect 这个选项的值影响括号尖点的位置，最好保持默认值 0.5.



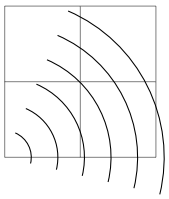
```
\begin{tikzpicture}[decoration={brace,amplitude=5mm}]
  \draw [help lines] grid (-2,2);
  \draw [postaction={decorate,draw,red}][green]
    (0,0) arc (0:90:1.5 and 1);
\end{tikzpicture}
```

Decoration expanding waves

这个装饰类型是“逐渐扩散的波形”，装饰片段是圆弧，沿着被装饰路径摆放一些圆弧，圆弧的尺寸越来越大，用以模仿“波动”。

segment length 这个选项的值确定相邻两个圆弧的间距。

angle 这个选项的值是圆弧角度值的一半。



```
\begin{tikzpicture}[decoration={expanding waves,angle=40}]
  \draw [help lines] grid (2,2);
  \draw [decorate] (0,0) -- (2,1);
\end{tikzpicture}
```

Decoration moveto

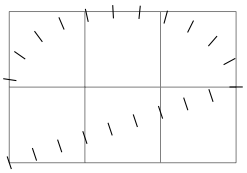
这个装饰类型是 decoration 模块定义的，它直接跳到被装饰路径的终点，常用在选项 pre=moveto 或 post=moveto 中。

Decoration ticks

这个装饰类型是——沿着被装饰路径添加“刻度线”。

segment length 这个选项值确定相邻两个刻度线的间距。

amplitude 这个选项值确定刻度线的长度。



```
\begin{tikzpicture}[decoration=ticks]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

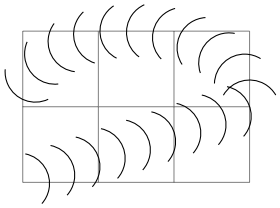
Decoration waves

这个装饰类型类似 expanding waves，都是由圆弧构成的，只是这个装饰类型中的圆弧尺寸保持不变。

segment length 这个选项值确定相邻两个圆弧的间距。

angle 这个选项的值是圆弧角度值的一半。

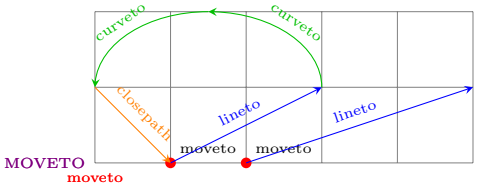
radius 这个选项的值是圆弧的半径。



```
\begin{tikzpicture}[decoration={waves,radius=4mm,angle=60}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration show path construction

被装饰路径可能由数种不同类型的“子输入路径”构成，例如被装饰路径可能含有 moveto, lineto, curveto, closepath 操作，不同操作构建不同类型的子输入路径。这个装饰类型可以针对各种类型的子输入路径分别进行装饰。



```
\begin{tikzpicture}[>=stealth, every node/.style={midway, sloped, font=\tiny},
  decoration={show path construction,
    moveto code={
      \node [left,text=violet] at(current subpath start) {MOVETO};
      \fill [red] (\tikzinputsegmentfirst) circle (2pt)
        node [fill=none, below] {moveto};
      \pgftext[at={\pgfpointdecoratedinputsegmentfirst},bottom,left] {\tikz\node{\tiny moveto
      ↪  };}},
    lineto code={
      \draw [blue,->] (\tikzinputsegmentfirst) -- (\tikzinputsegmentlast)
        node [above] {lineto};},
    curveto code={
      \draw [green!75!black,->] (\tikzinputsegmentfirst) .. controls (
      ↪  \tikzinputsegmentsupporta) and (\tikzinputsegmentsupportb) ..(
      ↪  \tikzinputsegmentlast) node [above] {curveto};},
    closepath code={
      \draw [orange,->] (\tikzinputsegmentfirst) -- (\tikzinputsegmentlast)
        node [above] {closepath};}
  }]
  \draw [help lines] (0,0) grid (5,2);
  \path [decorate] (1,0) -- (3,1) arc (0:180:1.5 and 1) --cycle (2,0) -- (5,1);
\end{tikzpicture}
```

使用下面的选项来分别为各个类型的子输入路径设置装饰路径。

`/pgf/decoration/moveto code=`⟨*code*⟩ (no default, initially `{}`)

这个选项的 ⟨*code*⟩ 针对的是 moveto 操作的“落脚点”。如前面的例子所示，使用命令 `\node` 或者 `node` 操作给 moveto 操作的“落脚点”加 node 时，所加的 node 只能位于原点附近，即所加的 node 的锚定点只能是原点。要想使得所加的 node 跟随 moveto 操作的“落脚点”，应当使用命令 `\pgftext`.

`/pgf/decoration/lineto code=`⟨*code*⟩ (no default, initially `{}`)

这个选项的 ⟨*code*⟩ 针对的是 lineto 操作。

`/pgf/decoration/curveto code=`⟨*code*⟩ (no default, initially `{}`)

这个选项的 ⟨*code*⟩ 针对的是 curveto 操作。

`/pgf/decoration/closepath code=`⟨*code*⟩ (no default, initially `{}`)

这个选项的 ⟨*code*⟩ 针对的是 closepath 操作。

在以上选项的 ⟨*code*⟩ 中，可以使用下面的宏来引用需要的点。

`\pgfpointdecoratedinputsegmentfirst`

这个宏保存的是当前子输入路径的第一个“构造点”。这个宏用在 PGF 命令中。

`\pgfpointdecoratedinputsegmentlast`

这个宏保存的是当前子输入路径的最后一个“构造点”。这个宏用在 PGF 命令中。

`\pgfpointdecoratedinputsegmentsupporta`

这个宏保存的是由 curveto 操作构建的子输入路径的第一个支撑点。这个宏用在 PGF 命令中。

`\pgfpointdecoratedinputsegmentsupportb`

这个宏保存的是由 curveto 操作构建的子输入路径的第二个支撑点。这个宏用在 PGF 命令中。

`\tikzinputsegmentfirst`

这个宏保存的是当前子输入路径的第一个“构造点”。这个宏用在 Ti*k*Z 命令中。

\tikzinputsegmentlast

这个宏保存的是当前子输入路径的最后一个“构造点”。这个宏用在 Ti*k*Z 命令中。

\tikzinputsegmentsupporta

这个宏保存的是由 curveto 操作构建的子输入路径的第一个支撑点。这个宏用在 Ti*k*Z 命令中。

\tikzinputsegmentsupportb

这个宏保存的是由 curveto 操作构建的子输入路径的第二个支撑点。这个宏用在 Ti*k*Z 命令中。

55.4 标记装饰

这种装饰类型是沿着被装饰路径放置标记符号。由于历史的原因，有 3 个不同程序库提供多种标记装饰类型。后文逐次介绍这三个程序库。

55.5 自选标记装饰

55.5.1 程序库 decorations.markings

Ti*k*Z Library decorations.markings

```
\usepgflibrary{decorations.markings} % LaTeX and plain TeX and pure pgf
\usepgflibrary[decorations.markings] % ConTeXt and pure pgf
\usetikzlibrary{decorations.markings} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.markings] % ConTeXt when using TikZ
```

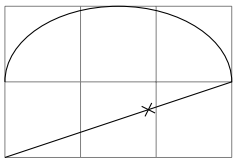
这个程序库提供 markings 装饰类型，允许你自己选择或定义一种标记（mark）类型来装饰路径。

Decoration markings

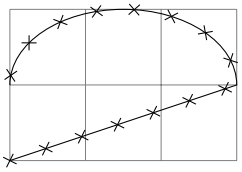
这个装饰类型允许你自己定义一个标记（mark），也就是说，你可以用绘图代码自己画一个标记，用于装饰路径。绘制标记的代码被放入一个局部域中来执行。例如，可以用下面的代码

```
\draw (-2pt,-2pt) -- (2pt,2pt);
\draw (2pt,-2pt) -- (-2pt,2pt);
```

定义一个叉号来作为标记。放置标记时，使用某个选项来确定被装饰路径上的一系列点：P_1，P_2 ……标记就放在这些点上。在点 P_i 放置标记时，PGF 会开启一个 TEX 分组，将绘制标记的代码放入这个 TEX 分组中执行。绘制标记的代码需要在一个坐标系内实现，这个坐标系类似路径在点 P_i 处的“自然坐标系”，Ti*k*Z 会使用（顶层的）坐标变换使得这个坐标系的原点位于点 P_i 处（平移），其 x 轴沿着路径在点 P_i 处的切线方向，y 轴与 x 轴成右手系。因此如果绘制标记的代码中含有 node 并且其选项中有 transform shape，那么该 node 就会接受这个（顶层的）坐标变换。装饰过程会破坏原来的被装饰路径，你可以将标记装饰选项作为 postaction 的值，从而在装饰过程结束后还能显示被装饰路径。下面是个例子。



```
\begin{tikzpicture}[decoration={
    markings,% 选定装饰类型 markings
    mark=at position 2cm with % 指定标记的位置，绘制标记的代码
      {\draw (-2pt,-2pt) -- (2pt,2pt);
        \draw (2pt,-2pt) -- (-2pt,2pt);}}]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0)--(3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```



```
\begin{tikzpicture}[decoration={
    markings,% 选定装饰类型 markings
    mark=between positions 0 and 1 step 5mm with % 标记位置，标记代码
      {\draw (-2pt,-2pt) -- (2pt,2pt);
        \draw (2pt,-2pt) -- (-2pt,2pt);}}]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0)--(3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

装饰类型 markings 对应下面的选项。

/pgf/decoration/mark=at position ⟨*pos*⟩ with ⟨*code*⟩ (no default)

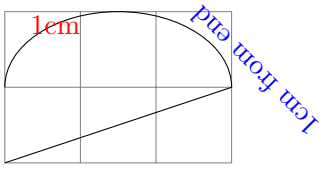
如上面的例子所示，这里 ⟨*code*⟩ 就是绘制标记的代码，其中可以使用 node 操作。⟨*pos*⟩ 对应一个长度，用于决定标记在路径上的位置，即沿着被装饰路径行进 ⟨*pos*⟩ 所达到的点。

⟨*pos*⟩ 会被命令 \pgf@lib@dec@parsenum 处理，此命令先用 \pgfmathparse 解析 ⟨*pos*⟩，再检查 ⟨*pos*⟩ 中是否有长度单位（参考 \ifpgfmathunitsdeclared→P. 113），记“被装饰路径的总长度”为 L（即 \pgfdecoratedpathlength→P. 409 的值，带长度单位）：

- 如果 ⟨*pos*⟩ 是负的尺寸（带单位），那么确定的长度是 $L + \langle pos \rangle$；
- 如果 ⟨*pos*⟩ 是非负的尺寸（带单位），那么确定的长度是 ⟨*pos*⟩；
- 如果 ⟨*pos*⟩ 是负的数值（不带单位），那么确定的长度是 $L + \langle pos \rangle \times L$；
- 如果 ⟨*pos*⟩ 是正的数值（不带单位），那么确定的长度是 $\langle pos \rangle \times L$；
- 所计算出来的长度保存到宏 \pgf@lib@dec@computed@width 中。

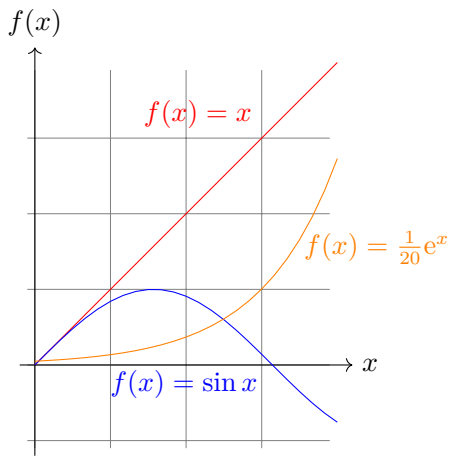
如果 ⟨*pos*⟩ 是绝对值过大的数字或数字运算表达式，例如 1.2，会导致标记位置超出被装饰路径，此时没有标记画出。

使用这个选项一次只能决定一个标记位置，你可以多次使用这个选项添加多个标记。假如多次使用这个选项，设第 i 次使用该选项确定的位置是点 P_i，第 $i+1$ 次使用该选项确定的位置是点 P_{i+1}，那么从点 P_i 到点 P_{i+1} 的方向最好“总是”沿着被装饰路径的行进方向（或反方向），否则可能造成混乱。下面例子中的位置参数“6cm，1cm，-4cm”不是单调数列，于是出现了混乱：




```
\begin{tikzpicture}[decoration={
  markings,% switch on markings
  mark=at position 6cm with \node[red]{1cm};,
  mark=at position 1cm with \node[green]{mid};,
  mark=at position -4cm with {\node[blue,transform shape]{1cm from end
  ↪  };}}
]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

下面的例子展示了如何用 markings 装饰类型给路径加标签。




```
\begin{tikzpicture}[domain=0:4,label/.style={postaction={
    decorate,
    decoration={
    markings,
    mark=at position .75 with \node #1;}}}]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);
  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};
  \draw[red,label={[above left]{$f(x)=x$}}] plot (\x,\x);
  \draw[blue,label={[below left]{$f(x)=\sin x$}}] plot (\x,{sin(\x r)});
  \draw[orange,label={[right]{$f(x)= \frac{1}{20} \mathrm e^x$}}] plot (\x,{0.05*exp(\x)});
\end{tikzpicture}
```

/pgf/decoration/mark=between positions ⟨*start pos*⟩ and ⟨*end pos*⟩ step ⟨*stepping*⟩ with ⟨*code*⟩ (no default)

这个选项可以确定被装饰路径上的数个位置，这些位置用来放置标记。这里 ⟨*start pos*⟩，⟨*end pos*⟩，⟨*stepping*⟩ 这 3 个参数的格式类似前面选项的 ⟨*pos*⟩.

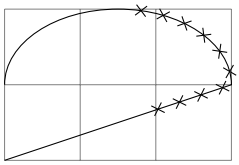
参数 ⟨*start pos*⟩ 对应某个尺寸，沿着被装饰路径行进这个尺寸，到达点 P_1, 这是装饰的起点；参数 ⟨*end pos*⟩ 也对应某个尺寸，也指定被装饰路径上的点 P_n; 从 P_1 到 P_n 这一部分路径是需要被装饰的。⟨*stepping*⟩ 指定相邻两个标记的间距。

所以被装饰路径上的点 P_1, P_2, $\cdots$, P_n 是应该放置标记的位置，其中 P_{i-1} 与 P_i 之间的间距是 ⟨*stepping*⟩; 但是 P_{n-1} 与 P_n 之间的间距必然是小于等于 ⟨*stepping*⟩ 的——如果是“小于”，则不用标记来装饰点 P_n, 停止装饰。

参数 ⟨*start pos*⟩，⟨*end pos*⟩，⟨*stepping*⟩ 都会被命令 \pgf@lib@dec@parsenum 处理，此命令会调用 \pgfmathparse 解析这 3 个参数。例如对 ⟨*stepping*⟩ 的处理是：先用 \pgfmathparse 解析 ⟨*stepping*⟩，再检查 ⟨*stepping*⟩ 中是否有长度单位（参考 \ifpgfmathunitsdeclared→P. 113），记“被装饰路径的总长度”为 L（即 \pgfdecoratedpathlength→P. 409 的值，带长度单位）：

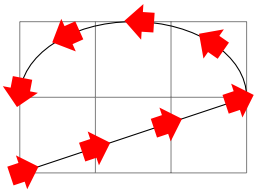
- 如果 ⟨*stepping*⟩ 是负的尺寸（带单位），那么相邻两个标记的间距是 $L + ⟨stepping⟩$;
- 如果 ⟨*stepping*⟩ 是非负的尺寸（带单位），那么相邻两个标记的间距是 ⟨*stepping*⟩;
- 如果 ⟨*stepping*⟩ 是负的数值（不带单位），那么相邻两个标记的间距是 $L + ⟨stepping⟩ \times L$;
- 如果 ⟨*stepping*⟩ 是正的数值（不带单位），那么相邻两个标记的间距是 $⟨stepping⟩ \times L$;
- 所计算出来的尺寸保存到宏 \pgf@lib@dec@computed@width 中。

如果 ⟨*start pos*⟩ 对应的尺寸大于 ⟨*end pos*⟩ 对应的尺寸，那么就没有装饰（不设置选项 /pgf/decoration/mark connection node→P. 997 的情况下），或者只有一个线段（设置选项 /pgf/decoration/mark connection node→P. 997 的情况下）。



```
\begin{tikzpicture}[decoration={markings,
  mark=between positions 0.3 and 0.7 step 3mm with
    { \draw (-2pt,-2pt) -- (2pt,2pt);
      \draw (2pt,-2pt) -- (-2pt,2pt); }} ]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0)--(3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

下面的例子中使用 shapes.arrow 库的形状 `single arrow` 来装饰路径：

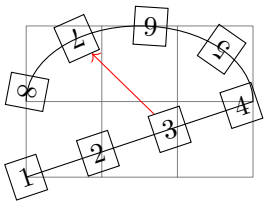


```
\begin{tikzpicture}[decoration={markings,
  mark=between positions 0 and 1 step 1cm with
    {\node [single arrow,fill=red,
      single arrow head extend=3pt, transform shape] {};}}]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0)--(3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

在前两个选项的 ⟨*code*⟩ 中可以使用以下两个选项，下面两个选项都是“只读”的，可以利用其选项值，但最好不要试图改变其选项值。

`/pgf/decoration/mark info/sequence number` (no value)

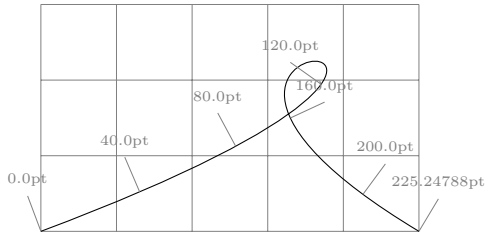
当用前两个 `mark` 选项为装饰路径设置一个或数个标记时，TikZ 会按照这些标记被添加的次序为它们编号。添加的第一个标记的编号是 1，添加的第二个标记的编号是 2……当前标记的编号就保存在这个 key 中，可以使用命令 `\pgfkeysvalueof{/pgf/decoration/mark info/sequence number}` 来引用或者输出当前标记的编号。




```
\begin{tikzpicture}[decoration={markings, mark=between positions 0 and 1 step 1cm with {
    \node [draw,name=mark-\pgfkeysvalueof{/pgf/decoration/mark info/sequence number},
           transform shape]
      {\pgfkeysvalueof{/pgf/decoration/mark info/sequence number}};}}]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
  \draw [red,->] (mark-3) -- (mark-7);
\end{tikzpicture}
```

`/pgf/decoration/mark info/distance from start` (no value)

沿着被装饰路径，从被装饰路径的起点到当前标记位置的长度保存在这个 key 中，这个长度的单位是 pt.




```
\begin{tikzpicture}[decoration={markings,
  mark=between positions 0 and 1 step 40pt with
  {\draw [help lines] (0,0) -- (0,0.5)
    node[above,font=\tiny]{\pgfkeysvalueof{/pgf/decoration/mark info/distance from start}};},
  mark=at position -0.1pt with
  {\draw [help lines] (0,0) -- (0,0.5)
```

```
    node[above,font=\tiny]{\pgfkeysvalueof{/pgf/decoration/mark info/distance from start}};}}
]
  \draw [help lines] grid (5,3);
  \draw [postaction={decorate}] (0,0) .. controls (8,3) and (0,3) .. (5,0) ;
\end{tikzpicture}
```

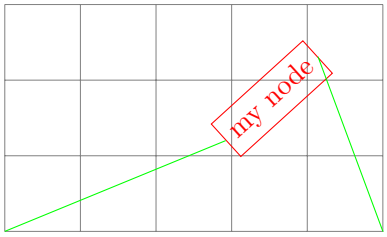
在选项 decoration 的值中可以使用以下选项。

/pgf/decoration/reset marks (no value)

使用 markings 类型时，可以多次使用 mark 来设置标记装饰，为了不让各个 mark 的设置相互干扰，每当执行完 ⟨*code*⟩ 后，装饰过程就会被自动重置（reset），本选项的作用就是实现这种重置。

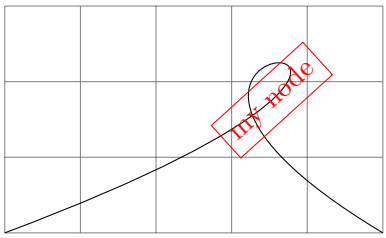
/pgf/decoration/mark connection node=⟨*node name*⟩ (no default, initially empty)

在下面的例子中使用选项 mark connection node=my node 指定了一个 node 名称，又在 mark 选项的 ⟨*code*⟩ 中设置了一个名称为 my node 的 node，这样得到的装饰路径就是“直线段—my node—直线段”，即“起点—lineto—my node 的某个边界点—moveto—my node 的某个边界点—lineto—终点”。




```
\begin{tikzpicture}[decoration={markings,
    mark connection node=my node,
    mark=at position .5 with
    {\node [draw,red,transform shape] (my node) {my node};}}]
  \draw [help lines] grid (5,3);
  \draw [decorate,green] (0,0) .. controls (8,3) and (0,3) .. (5,0);
\end{tikzpicture}
```

但如果装饰选项 decorate 换成 postaction={decorate}，就没有直线段，只有“my node”：

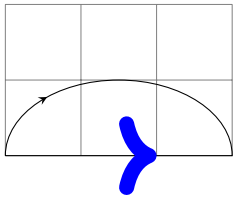



```
\begin{tikzpicture}[decoration={markings,
    mark connection node=my node,
    mark=at position .5 with
    {\node [draw,red,transform shape] (my node) {my node};}}]
  \draw [help lines] grid (5,3);
  \draw [postaction={decorate,green}] (0,0) .. controls (8,3) and (0,3) .. (5,0);
\end{tikzpicture}
```

在 mark 选项的 ⟨*code*⟩ 中可以使用以下两个箭头命令，并且这两个箭头命令只能用在此处：

\arrow[⟨*options*⟩]{⟨*arrow end tip*⟩}

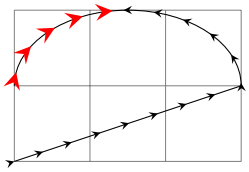
其中 ⟨*arrow end tip*⟩ 是箭头类型的名称，⟨*options*⟩ 是箭头选项。在 ⟨*code*⟩ 的坐标系中，箭头的尖点位于坐标系原点，箭头方向指向右侧，添加箭头后，箭头的尖点位于指定的标记位置点上，箭头方向沿着路径的切线方向。在使用 tikz 的环境、命令时，选项 ⟨*options*⟩ 才有效。箭头和 ⟨*options*⟩ 都会被放入一个 scope 中来执行。



```
\begin{tikzpicture}[decoration={markings,
  mark=at position 2cm with {\arrow[blue,line width=2mm]{>}},
  mark=at position -1cm with {\arrowreversed[black]{stealth}}}
]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,0) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

\arrowreversed[⟨*options*⟩]{⟨*arrow end tip*⟩}

本命令与上一命令类似，本命令添加的是一个反向的箭头。



```
\begin{tikzpicture}[decoration={markings,
  mark=between positions 0 and .75 step 4mm with {\arrow{stealth}},
  mark=between positions .75 and 1 step 4mm
    with {\arrowreversed[red,scale=2]{stealth}}}
]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

55.5.2 脚印标记

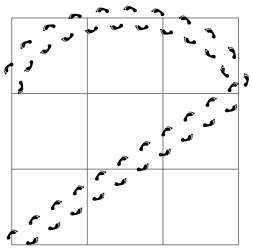
TikZ Library decorations.footprints

```
\usepgflibrary{decorations.footprints} % LaTeX and plain TeX and pure pgf
\usepgflibrary[decorations.footprints] % ConTeXt and pure pgf
\usetikzlibrary{decorations.footprints} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.footprints] % ConTeXt when using TikZ
```

这个库提供“脚印”装饰类型—一串沿着被装饰路径放置的脚印，就像沿着路径走过一样。

Decoration footprints

这个选项指定脚印装饰类型。



```
\begin{tikzpicture}[decoration={footprints,
    foot length=5pt,stride length=10pt}]
  \draw [help lines] grid (3,3);
  \fill [decorate] (0,0) -- (3,2) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

下面是调整脚印装饰外观的选项。

/pgf/decoration/foot length=⟨*dimension*⟩ (initially 10pt)

这个选项值调节脚印的长度，但不改变两个脚印之间的步长。

```
\begin{tikzpicture}[decoration={footprints,
    foot length=30pt}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/stride length=⟨*dimension*⟩ (initially 30pt)

这个选项值调节“步长”，即前后两个脚印的间距。

```
\begin{tikzpicture}[decoration={footprints,
    stride length=50pt}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot sep=⟨*dimension*⟩ (initially 4pt)

这个选项值调节左右脚印的横向间距。

```
\begin{tikzpicture}[decoration={footprints,
    foot sep=30pt}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot angle=⟨*angle*⟩ (initially 10)

这个选项值调节脚印的角度，如果这个选项的值是 60，就有点“外八字脚”。

```
\begin{tikzpicture}[decoration={footprints,
    foot angle=60}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot of=⟨*name*⟩ (initially human)

这个选项的值 ⟨*name*⟩ 用来选择脚印类型，初始值的“人类”的脚印，可用的 ⟨*name*⟩ 是：

- gnome, 矮人
- bird, 鸟
- felis silvestris, 猫
- human, 人类

```
\begin{tikzpicture}[decoration={footprints,
    foot of=felis silvestris}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

55.5.3 形状装饰

TikZ Library decorations.shapes

```
\usepgflibrary{decorations.shapes} % LaTeX and plain TeX and pure pgf
\usepgflibrary[decorations.shapes] % ConTeXt and pure pgf
\usetikzlibrary{decorations.shapes} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.shapes] % ConTeXt when using TikZ
```

shape 是一种复杂路径或图形。预定义的 shape 只有 coordinate, rectangle, circle 三种，在 shapes.geometric, shapes.symbols, shapes.callouts, shapes.misc, shapes.arrows, shapes.multipart 等库中定义了很多 shape，也可以用 \pgfdeclareshape → P. 436 自定义一种形状。程序库 decorations.shapes 允许使用各种已定义的“形状”（shape）来装饰路径，由于历史的原因保留这个库，不过使用 markings 库更好。

库 decorations.shapes 提供以下选项。

/pgf/decoration/shape width=⟨*dimension*⟩ (no default, initially 2.5pt)

这个选项设置形状的宽度，包括 start width 和 end width, 选项 shape start width, shape end width 等都可以改写本选项的设置。

/pgf/decoration/shape height=⟨*dimension*⟩ (no default, initially 2.5pt)

类似 /pgf/decoration/shape width, 这个选项设置形状的高度。

/pgf/decoration/shape size=⟨*dimension*⟩ (no default)

这个选项同时设置形状的宽度、高度。

关于以上选项的详细信息可参考程序库的代码。

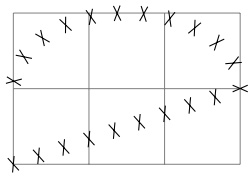
Decoration crosses

这个选项使用叉号 crosses 来替换被装饰路径。下面的选项可以调节叉号的外观。

segment length 这个选项值确定相邻两个叉号的中心点的距离。

shape height 这个选项值确定叉号的高度。

shape width 这个选项值确定叉号的宽度。



```
\begin{tikzpicture}[decoration={crosses,shape height=2mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

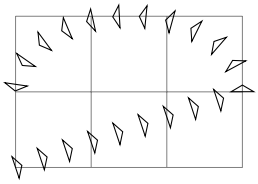
Decoration triangles

这个选项使用三角形状 triangles 来替换被装饰路径。下面的选项可以调节这个形状的外观。

segment length 这个选项值确定相邻两个三角的间距。

shape height 这个选项值确定三角的高度（与路径垂直的边的长度）。

shape width 这个选项值确定三角的宽度。

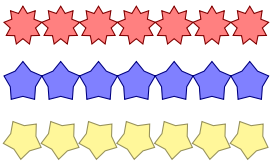


```
\begin{tikzpicture}[decoration={triangles,shape height=3mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

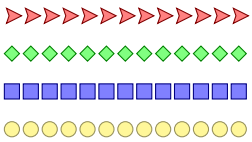
Decoration shape backgrounds

这个装饰类型允许使用“shape”来替换被装饰路径。“shapae”是用命令 \pgfdeclareshape 定义的形状。注意，shape 不同于 node，用来替换被装饰路径的是 shape, 装饰过程不创建 node, 因此用作装饰的 shape 中不能使用文字，不能为 shape 命名，也不能引用它们；另外，如果一个 shape 的尺寸强烈地依赖文字盒子（如 arrow shapes），那么这个 shape 也不能用作装饰。若不希望遇到这些限制，可以改用 markings 库。

有的形状有属于自己的选项来调节其外观，例如库 shapes.geometric 提供的形状 star，其外观受到选项 star points，star point height 的影响。



```
\tikzset{
    paint/.style={draw=#1!50!black, fill=#1!50},
    my star/.style={decorate,decoration={shape backgrounds,shape=star}, star points=#1}
}
\begin{tikzpicture}[decoration={shape sep=.5cm, shape size=.5cm}]
  \draw [my star=9, paint=red] (0,1.5) -- (3,1.5);
  \draw [my star=5, paint=blue] (0,.75) -- (3,.75);
  \draw [my star=5, paint=yellow, shape border rotate=30] (0,0) -- (3,0);
\end{tikzpicture}
```



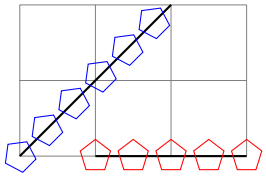
```
\tikzset{paint/.style={ draw=#1!50!black, fill=#1!50 },
    decorate with/.style={decorate,decoration={shape backgrounds,shape=#1,shape size=2mm}}}
\begin{tikzpicture}
  \draw [decorate with=dart, paint=red] (0,1.5) -- (3,1.5);
  \draw [decorate with=diamond, paint=green] (0,1) -- (3,1);
  \draw [decorate with=rectangle, paint=blue] (0,0.5) -- (3,0.5);
  \draw [decorate with=circle, paint=yellow] (0,0) -- (3,0);
\end{tikzpicture}
```

使用这个装饰类型时，下面的选项能影响装饰 shape 的外观、位置。

/pgf/decoration/anchor=⟨*anchor*⟩ (no default, initially `center`)

PGF 会根据有关选项自动确定被装饰路径上的点来放置 shape，这些点就是 shape 的锚定点。这个选项会把 shape 的锚位置 ⟨*anchor*⟩ 放在这些锚定点上。初始之下，shape 的锚位置 `center` 位于这些锚定点上。

在被装饰路径的起点上会放置一个 shape, 如果 shape 之间的间距合适，那么在被装饰路径的终点上也会有一个 shape.



```
\begin{tikzpicture}[decoration={
    shape backgrounds,shape=regular polygon,shape size=4mm}]
  \draw [help lines] grid (3,2);
  \draw [thick] (0,0) -- (2,2) (1,0) -- (3,0);
  \draw [red, decorate, decoration={shape sep=.5cm}] (1,0) -- (3,0);
  \draw [blue, decorate, decoration={shape sep=.5cm}] (0,0) --
  ↪  (2,2);
\end{tikzpicture}
```

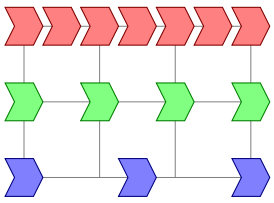
用作装饰的 shape 没有文字，当把装饰 shape 放到路径上之后，它有自己的默认尺寸。针对 shape 的变换有效，但是像 `inner sep`, `minimum size` 这种针对 node 的选项对装饰 shape 无效。

/pgf/decoration/shape=⟨*shape name*⟩ (no default, initially `circle`)

这个选项确定用哪种 shape 来做装饰。

/pgf/decoration/shape sep=⟨*spacing*⟩ (no default, initially `.25cm, between centers`)

这个选项确定相邻两个装饰 shape 的间距，初始之下使用 shape 中心之间的间距，即 `shape sep={.25cm, between centers}`，也可以换成形状边界间距，即 `shape sep={.25cm, between borders}`.

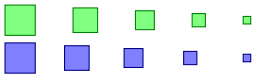


```
\begin{tikzpicture}[
    decoration={shape backgrounds,shape size=.5cm,shape=signal},
    signal from=west, signal to=east,
    paint/.style={decorate, draw=#1!50!black, fill=#1!50}]
  \draw [help lines] grid (3,2);
  \draw [paint=red, decoration={shape sep=.5cm}] (0,2) -- (3,2);
  \draw [paint=green, decoration={shape sep={1cm, between centers}}]
  ↪  (0,1) -- (3,1);
  \draw [paint=blue, decoration={shape sep={1cm, between borders}}]
  ↪  (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/shape evenly spread=⟨*number*⟩

⟨ 或可带有⟩, `by centers` 或 `by borders` (default `by centers`)

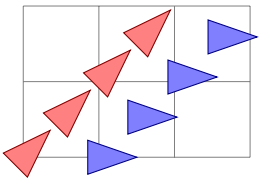
这个选项能取消（overrides）选项 shape sep 的设置。本选项在被装饰路径上放置 ⟨*number*⟩ 个装饰 shape, 并使它们均匀分布。如果 ⟨*number*⟩ 小于 1，那么没有 shape；如果 ⟨*number*⟩ 等于 1，那么一个 shape 被放在路径中间。还有两个可选的关键词 by centers 和 by borders, 这两个参数确定两个相邻 shape 路径的间距的计算方式，即“中心到中心”和“边界到边界”，默认 by centers.



```
\tikzset{
  paint/.style={draw=#1!50!black, fill=#1!50},
  spreading/.style={decorate,decoration={shape backgrounds,
    shape=rectangle, shape start size=4mm,shape end size=1mm,shape
    ↪  evenly spread={#1}}}
}
\begin{tikzpicture}
  \fill [paint=green,spreading={5, by borders}, decoration={shape
  ↪  scaled}] (0,2) -- (3,2);
  \fill [paint=blue,spreading={5, by centers}, decoration={shape
  ↪  scaled}] (0,1.5) -- (3,1.5);
\end{tikzpicture}
```

/pgf/decoration/shape sloped=⟨*boolean*⟩ (no default, initially true)

在默认下，绘制装饰路径的坐标系类似 turn 坐标系统，因此用于装饰的 shape 路径会随着被装饰路径的切线变化而旋转。如果本选项设置为 shape sloped=false，则 shape 路径不会出现这种旋转。本选项通过设置 TEX-if \ifpgfshapedecorationsloped 来起作用。



```
\tikzset{
  paint/.style={draw=#1!50!black, fill=#1!50}
}
\begin{tikzpicture}[decoration={shape backgrounds,
  shape width=.65cm, shape height=.45cm,
  shape=isosceles triangle, shape sep=.75cm}]
  \draw [help lines] grid (3,2);
  \draw [paint=red,decorate] (0,0) -- (2,2);
  \draw [paint=blue,decorate,decoration={shape sloped=false}] (1,0) --
  ↪  (3,2);
\end{tikzpicture}
```

/pgf/decoration/shape start width=⟨*length*⟩ (no default, initially 2.5pt)

当沿着被装饰路径放置装饰 shape 时，本选项设置第一个 shape 的宽度为 ⟨*length*⟩，它能够改写选项 shape width 的设置。

/pgf/decoration/shape start height=⟨*length*⟩ (no default, initially 2.5pt)

当沿着被装饰路径放置装饰 shape 时，本选项设置第一个 shape 的高度为 ⟨*length*⟩，它能够改写选项 shape height 的设置。

/pgf/decoration/shape start size=⟨*length*⟩ (style, no default)

当沿着被装饰路径放置装饰 shape 时，本选项设置第一个 shape 的尺寸为 ⟨*length*⟩，即它同时设置第一个 shape 的宽度和高度。

/pgf/decoration/shape end width=⟨*length*⟩ (no default, initially 2.5pt)

当沿着被装饰路径放置装饰 shape 时，如果被装饰路径的终点处有一个 shape，那么这个 shape 的宽度就是本选项指定的 ⟨*length*⟩.

/pgf/decoration/shape end height=⟨*length*⟩ (no default, initially 2.5pt)

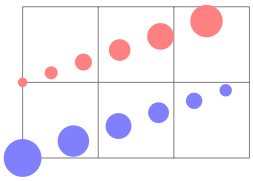
类似 shape end width, 只是针对高度。

/pgf/decoration/shape end size=⟨*length*⟩ (no default)

本选项同时设置 shape end width 和 shape end height.

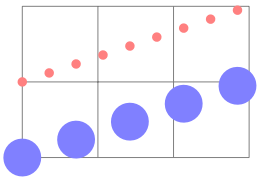
/pgf/decoration/shape scaled=⟨*boolean*⟩ (no default, initially false)

当本选项的值为 true 时，其作用是，例如，假设沿着被装饰路径放置了 4 个 shape，第一个 shape 的尺寸是 1mm，第 4 个 shape 的尺寸是 4mm，那么这 4 个 shape 的尺寸就是 1mm, 2mm, 3mm, 4mm. 所以本选项通常配合 shape start width, shape end width 等选项使用。



```
\tikzset{
  bigger/.style={decoration={shape start size=.125cm, shape end
  ↪  size=.5cm}},
  smaller/.style={decoration={shape start size=.5cm, shape end
  ↪  size=.125cm}},
  decoration={shape backgrounds, shape sep={.25cm, between borders
  ↪  },shape scaled}
}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \fill [decorate, bigger, red!50] (0,1) -- (3,2);
  \fill [decorate, smaller, blue!50] (0,0) -- (3,1);
\end{tikzpicture}
```

把上面例子中的 shape scaled 去掉后就是下面的结果：



```
\tikzset{
  bigger/.style={decoration={shape start size=.125cm, shape end
  ↪  size=.5cm}},
  smaller/.style={decoration={shape start size=.5cm, shape end
  ↪  size=.125cm}},
  decoration={shape backgrounds, shape sep={.25cm, between borders}}
}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \fill [decorate, bigger, red!50] (0,1) -- (3,2);
  \fill [decorate, smaller, blue!50] (0,0) -- (3,1);
\end{tikzpicture}
```

55.6 文字装饰

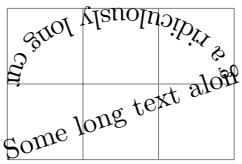
TikZ Library decorations.text

```
\usepgflibrary{decorations.text} % LaTeX and plain TeX and pure pgf
\usepgflibrary[decorations.text] % ConTeXt and pure pgf
\usetikzlibrary{decorations.text} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.text] % ConTeXt when using TikZ
```

载入这个库后可以使用文字来装饰路径。这个库提供两种文字装饰类型：text along path 和 text effects along path.

55.6.1 装饰类型 text along path

在默认下，沿着被装饰路径的方向，从被装饰路径的起点开始，文字会被放置在的路径左侧。放置文字后，被装饰路径会被丢弃。下面是个例子。




```
\begin{tikzpicture}[decoration={text along path,
    text={Some long text along a ridiculously long curve that}}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

关于这个装饰类型要注意以下几点：

- 每个文字字符都被放入一个单独的 \hbox 中。
- 默认把字符的基线中点放在被装饰路径上，可以使用变换选项来改变文字与被装饰路径的相对位置。
- 文字符号沿着被装饰路径放置，文字符号之间可能重叠。
- 文字可以是数学模式下的文字，数学模式的上下标要用花括号括起来，例如 {^y_z}；各种算符，例如 \times，\cdot 也要用花括号括起来。如果数学式子太复杂可能会影响装饰效果。
- 在子输入路径的边界位置上文字位置可能出现偏差，此时需要手工纠正。

这个装饰类型有以下选项可用。

/pgf/decoration/text=⟨*text*⟩ (no default, initially empty)

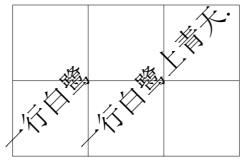
这个选项引入用作装饰的文字。这里 ⟨*text*⟩ 是需要沿着被装饰路径放置的文字。文字中多余的空格会被忽略，因此需要适当使用命令 \␣ 或者 \space. 也可以使用字体命令，如 \it, \bf 等设置文字字体，也可以使用颜色命令 \color 设置文字颜色。注意，一旦把多个字符放入一个盒子或 TEX 分组内，这些字符就不再随着被装饰路径的弯曲而旋转。


```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate,decoration={text along path,
      text={a big {\color{green}green} juicy apple.}}]
    (0,0) .. controls (0,2) and (3,0) .. (3,2);
\end{tikzpicture}
```


```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate,decoration={text along path,
      text={a big green juicy {\bf apple}.}}]
    (0,0) .. controls (0,2) and (3,0) .. (3,2);
\end{tikzpicture}
```

通常，用于装饰的文字 ⟨*text*⟩ 保存在宏 \pgfdecorationtext 中。如果被装饰路径太短而文字太多，文字超出被装饰路径的端点，那么有的文字就不能显示出来，这些不能显示出来的文字保存在宏 \pgfdecorationrestoftext 中。

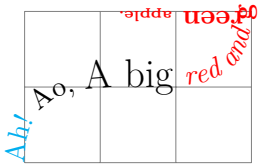
下面例子中，第一个被装饰路径长度是 $\sqrt{2}$，装饰文字过多，所以文字没有全部显示。第二个被装饰路径使用了选项 scale=2 把路径长度变为原来的 2 倍，于是装饰文字都显示出来了：




```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate,decoration={text along path,
      text={一行白鹭上青天.}}]
    (0,0)--(1,1);
  \path [scale=2, decorate,decoration={text along path,
      text={一行白鹭上青天.}}]
    (0.5,0)--(1.5,1);
\end{tikzpicture}
```

/pgf/decoration/text format delimiters={⟨*before*⟩}{⟨*after*⟩} (no default, initially {|}{})

这个选项设置定界符，用于界定命令作用范围，如果 ⟨*after*⟩ 是空的，那么就把 ⟨*before*⟩ 同时用作开定界符和闭定界符。初始之下把“|”作为开定界符和闭定界符。与定界符配合使用的还有符号“+”，如下面的例子所示。




```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\path [decorate,
    decoration={text along path, text format delimiters={[]{[]},
    text={[\color{cyan}]Ah![] Ao, [\Large]A big
      [\color{red}\it]red and [+\bf]green [+\tiny]apple.}}]
  (0,0) .. controls (0,2) and (3,0) .. (3,2)--(0,2);
\end{tikzpicture}
```

上面例子中，将“[”和“]”分别作为开定界符和闭定界符。定界符的用法是：

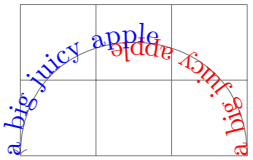
- 定界符的符号类别是 11 或 12；
- 定界符要成对使用；
- 在一对定界符内使用那些设置文字外观的命令（字体、字号、文字颜色等），定界符不能套嵌使用；
- 把某些命令放在一对定界符内，这些命令就对之后的文字起作用；
- 使用一对内容为空（即不含任何命令）的定界符将文字还原为默认状态；
- 加号“+”的作用是把之前一对定界符内的命令添加到“+”所在位置，因此“+”要放在定界符内。

/pgf/decoration/text color=⟨*color*⟩ (no default, initially `black`)

本选项设置文字颜色。

/pgf/decoration/reverse path=⟨*boolean*⟩ (no default, initially `false`)

这个选项使得被装饰路径的方向变成原方向的反方向，因此文字会沿着反方向放置在被装饰路径的另一侧。



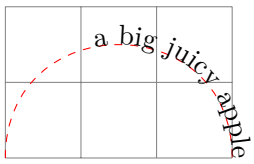

```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [gray, ->]
    [postaction={decorate, decoration={text along path,
     text={a big juicy apple}, text color=red}}]
    [postaction={decorate, decoration={text along path,
     text={a big juicy apple}, text color=blue, reverse path}}]
  (3,0) .. controls (3,2) and (0,2) .. (0,0);
\end{tikzpicture}
```

/pgf/decoration/text align={⟨*alignment options*⟩} (no default)

这个选项会给 ⟨*alignment options*⟩ 加上前缀 `/pgf/decoration/text align/` 来执行，因此 ⟨*alignment options*⟩ 中的选项可以是，例如（下文介绍的），`left`，`align=left`，`center`，`fit to path=true` 等。

/pgf/decoration/text align/align=⟨*alignment*⟩ (no default, initially `left`)

⟨*alignment*⟩ 可以是 `left`，`right`，`center` 三者之一。左对齐 `left` 一般指的与路径的起点对齐。




```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [red, dashed]
    [postaction={decorate, decoration={text along path,
     text={a big juicy apple}, text align={align=right}}}]
  (0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

/pgf/decoration/text align/left (style, no value)

本选项等效于 `/pgf/decoration/text align/align=left`.

/pgf/decoration/text align/right (style, no value)

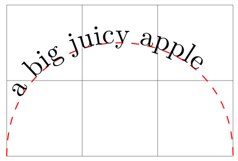
本选项等效于 `/pgf/decoration/text align/align=right`.

/pgf/decoration/text align/center (style, no value)

本选项等效于 /pgf/decoration/text align/align=center.

/pgf/decoration/text align/left indent=⟨*length*⟩ (no default, initially 0pt)

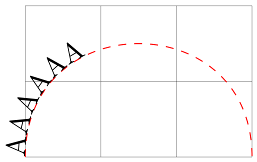
本选项使得文字左侧有一段（沿着路径）长度为 ⟨*length*⟩ 的空白（类似文字的缩进），其效果相当于改变了的被装饰路径的起点位置，即在被装饰路径的开始处裁掉一段路径。




```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [red, dashed]
    [postaction={decorate, decoration={text along path,
     text={a big juicy apple}, text align={left indent=.8cm}}}]
  (0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

/pgf/decoration/text align/right indent=⟨*length*⟩ (no default, initially 0pt)

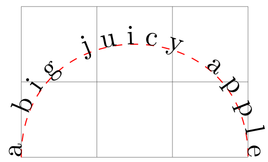
本选项规使得文字右侧有一段（沿着路径）长度为 ⟨*length*⟩ 的空白（类似文字的缩进），其效果相当于改变了的被装饰路径的终点位置，即在被装饰路径的结尾处裁掉一段路径。




```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [red, dashed]
    [postaction={decorate, decoration={text along path,
     text={AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA},
     text align={right indent=3cm}}}]
  (0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

/pgf/decoration/text align/fit to path=⟨*boolean*⟩ (no default, initially false)

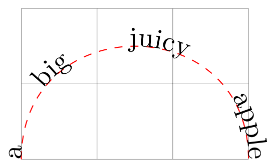
这个选项使得各个文字字符（包括有效空格）在被装饰路径上均匀分布。但如果文字太多而被装饰路径太短，以致于文字超出被装饰路径的端点，那么本选项无效。




```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [red, dashed]
    [postaction={decorate, decoration={text along path,
     text={a big juicy apple},
     text align=fit to path}}]
  (0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

/pgf/decoration/text align/fit to path stretching spaces=⟨*boolean*⟩(no default, initially false)

这个选项的作用类似前一个选项，只不过单个单词的各个字母之间的间距不会被改变，改变的是单词之间的空白长度，注意由 \space 生成的空格会被拉长或压缩，但由 \␣ 生成的空格长度不会被改变。




```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [red, dashed]
    [postaction={decorate, decoration={text along path,
     text={a big juicy apple},
     text align={fit to path stretching spaces}}}]
  (0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

55.6.2 装饰类型 text effects along path

这个文字装饰类型与 text along path 类似，不过在这个文字装饰类型中，每个字符都是作为 TikZ 的 node 添加到被装饰路径上的，故每个字符都是一个小“图形”，关于 node 的各种选项，例如 text,

scale, opacity, fill, draw, shift 等都是可用的，能产生多种“effects”. 这种装饰类型只能用在 tikz 的环境、命令中。

注意区分“字母符号”和“非字母符号”，字母符号构成单词，而非字母符号有其他作用，例如空格可以分隔单词。

与 text along path 不同的是，定界符在 text effects along path 类型中无效，但是有其它选项能调整装饰文字外观。能用于这个装饰类型、调整装饰文字外观的选项很多，注意有的选项的前缀是 /tikz/，这种前缀的选项是 tikz 的选项。

本装饰类型的编译过程可能会慢一些。


```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!}, text align=center,
    text effects/.cd,
    character count=\i, character total=\n,
    characters={evaluate={\c=\i/\n*100;}, text along path,
      text=red!\c!orange},
    character widths={text along path, xslant=0, yscale=1}}]
  \path [postaction={decorate}, preaction={decorate,
     text effects={characters/.append={yscale=-1.5,
       opacity=0.5, text=gray, xslant=(\i/\n-0.5)*3}}}]
    (0,0) .. controls ++(2,1) and ++(-2,-1) .. (3,4);
\end{tikzpicture}
```

上面例子中的选项 evaluate={...} 是数学程序库定义的选项。

下面介绍能用于这个装饰类型的选项。

/pgf/decoration/text=⟨*text*⟩ (no default)

这个选项设置用于装饰的文字。⟨*text*⟩ 中的字符可以用花括号括起来，⟨*text*⟩ 中的命令在编译时才会展开。注意 ⟨*text*⟩ 中不能使用定界符。

/pgf/decoration/text align=⟨*align*⟩ (no default)

这个选项确定文字的对齐方式，⟨*align*⟩ 可以是 left, right, center 之一。

/tikz/text effects={⟨*options*⟩} (no default)

⟨*options*⟩ 是某些能用于本装饰类型的选项,这些选项会被冠以前缀 /pgf/decoration/text effects/ 来执行。注意本选项的路径前缀是 /tikz/,不是 /pgf/decoration/, 所以本选项不能用在 decoration 中。

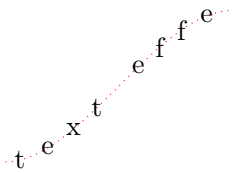
/pgf/decoration/text effects/every character (style, no value)

这是个样式（style），用法如 every character/.style=⟨*options*⟩，⟨*options*⟩ 中的选项应当都是 TikZ 的能用于 node 的各种选项，会用在各个装饰字符 node 的开头，所以本样式之后的选项可以修改本样式的设置。例如以下两个样式

```
every letter/.style={fill=green}, every character/.style={fill=red}
```

样式 every character 先把所有字符 node 的填充色设为红色（尽管这个样式在后），样式 every letter 又把所有字母字符 node 的填充色由红色改为绿色（尽管这个样式在前）。

初始之下本样式为空，文字 node 只是沿着路径放置，本身并没有什么“effects”，比如，文字 node 不会随着被装饰路径的弯曲而旋转。



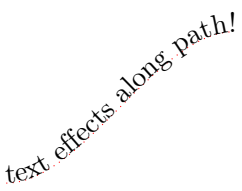

```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!}}]
  \path [draw=red, dotted, postaction={decorate}]
    (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\end{tikzpicture}
```

长 大
河 漠
落 孤
日 烟
圆 直

```
\tikz{
  \draw[decorate,
    decoration={text effects along path, text={大漠孤烟直长河落日圆}},
    text effects={text along path,characters={draw=none,rotate=90,
      inner sep=0pt,minimum size=1em}}]
  (0,0)--+(-90:5em) (-1.5em,0)--+(-90:5em);}
```

/pgf/decoration/text effects/text along path (style, no value)

这个选项会自动设置 tikz 选项 `transform shape, anchor=baseline, inner sep=0pt`，使得文字 node 随着被装饰路径的弯曲而旋转，效果就像装饰类型 `text along path` 那样，但仍然不能使用定界符。




```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!}}]
  \path [draw=red, dotted, postaction={decorate},
     text effects={text along path}]
    (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\end{tikzpicture}
```

/pgf/decoration/text effects/characters={⟨*effects*⟩} (no default)

这个选项是 `every character` 的简洁形式。

/pgf/decoration/text effects/character⟨*number*⟩ (style, no value)

这是个样式，用法如 `character` ⟨*number*⟩`/.style=`⟨*options*⟩，其中 ⟨*number*⟩ 是个正整数，代表第 ⟨*number*⟩ 个字符；⟨*options*⟩ 是针对第 ⟨*number*⟩ 个字符的 tikz 选项。

/pgf/decoration/text effects/every letter (style, no value)

这是个样式，它设置的选项针对所有的“字母符号”node.

/pgf/decoration/text effects/letter⟨*number*⟩ (style, no value)

这是个样式，它设置的选项针对每个单词的第 ⟨*number*⟩ 个“字母符号”node.

/pgf/decoration/text effects/every first letter (style, no value)

这是个样式，它设置的选项针对每个单词的第一个“字母符号”node.

/pgf/decoration/text effects/every last letter (style, no value)

这是个样式，它设置的选项针对每个单词的最后一个“字母符号”node.

/pgf/decoration/text effects/every word (style, no value)

这是个样式，它设置的选项针对每个单词的每个“字母符号”node.

/pgf/decoration/text effects/word⟨*number*⟩ (style, no value)

这是个样式，它设置的选项针对第 ⟨*number*⟩ 个单词的各个“字母符号”node.

/pgf/decoration/text effects/word ⟨*m*⟩ `letter` ⟨*n*⟩ (style, no value)

这是个样式，它设置的选项针对第 ⟨*m*⟩ 个单词的第 ⟨*n*⟩ 个“字母符号”node.

/pgf/decoration/text effects/every word separator (style, no value)

这是个样式，它设置的选项针对单词之间的分隔符。

/pgf/decoration/text effects/word separator=⟨*character*⟩ (no default, initially `space`)

这个选项设置单词分隔符，初始之下，单词分隔符是空格。这里用作分隔符的 ⟨*character*⟩ 必须是单个字符，例如 `a` 或 `-`.

/pgf/decoration/text effects/every character width (style, no value)

这是个样式，它设置所有字符 node 的宽度。本样式设置的选项应当是各种能够影响 node 宽度的 tikz 选项，例如 inner xsep, text width, minimum width 等。

/pgf/decoration/text effects/character widths=⟨*effects*⟩ (no default)

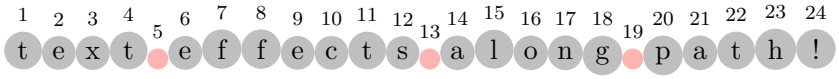
这个选项是样式 every character width 的简洁形式。

下面几个选项涉及装饰字符 node 的编号、个数统计，它们都不能用在前面介绍的各个样式（style）中。利用装饰字符 node 的编号可以引用它们。

/pgf/decoration/text effects/character count=⟨*macro*⟩ (no default)

这个选项将字符 node 的编号保存在宏 ⟨*macro*⟩ 中，字符 node 的编号从 1 开始。⟨*macro*⟩ 只是相应字符 node 的编号，不是字符 node 的名称。如果在 characters 中使用选项 name=⟨*macro*⟩, 那么就把字符 node 命名为 (⟨*macro*⟩).

注意这个选项不能用在前面介绍的各个样式（style）中。



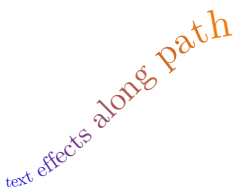

```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text,
      character count=\i, every word separator/.style={fill=red!30},
      characters={text along path, shape=circle, fill=gray!50}}]
  \path [decorate, text effects={characters/.append={label=above:\footnotesize\i}}] (0,0);
\end{tikzpicture}
```

上面例子中，共有 24 个字符 node，其中包括 3 个空格（用于分隔单词），一个叹号。选项 path from text 的作用见后文。宏 \i 保存字符 node 的编号，整个命令相当于使用了 \foreach 语句

```
\foreach \i in {1,...,24}
......
```

/pgf/decoration/text effects/character total=⟨*macro*⟩ (no default)

这个选项将字符 node 的总个数保存在宏 ⟨*macro*⟩ 中。注意这个选项不能用在前面介绍的各个样式（style）中。

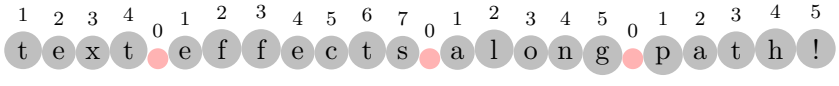



```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      character count=\i, character total=\n,
      characters={text along path, evaluate={\c=\i/\n*100;},
      text=orange!\c!blue, scale=\i/\n+0.5}}]
  \path [decorate]
    (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\end{tikzpicture}
```

上面代码中的选项 evaluate 是数学程序库中的选项。

/pgf/decoration/text effects/letter count=⟨*macro*⟩ (no default)

这个选项将字母字符 node 的编号保存在宏 ⟨*macro*⟩ 中。当使用这个选项后，字母字符 node 的编号从 1 开始，用作单词分隔符号的 node 的编号则统一编为 0. 注意这个选项不能用在前面介绍的各个样式（style）中。



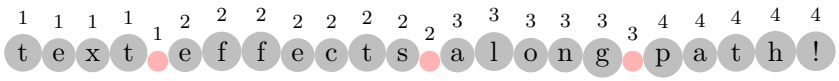
```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text, letter count=\i, every word separator/.style={fill=red!30},
      characters={text along path, shape=circle, fill=gray!50}}]
  \path [decorate, text effects={characters/.append={label=above:\footnotesize\i}}] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text/effetcs/letter total=⟨*macro*⟩ (no default)

这个选项将字母字符 node 的总个数保存在宏 ⟨*macro*⟩ 中。当使用这个选项后，用作单词分隔符号的 node 的编号统一编为 0. 注意这个选项不能用在前面介绍的各个样式（style）中。

/pgf/decoration/text effects/word count=⟨*macro*⟩ (no default)

这个选项将单词的编号保存在宏 ⟨*macro*⟩ 中，编号从 1 开始。例如，假设第一个单词是 `text`，这个单词有 4 个字母符号，那么这 4 个字母 node 的编号都是 1，换句话说，单词 `text` 由 4 个编号都是 1 的字母符号 node 组成。当使用这个选项后，用作单词分隔符号的 node 的编号与它前面的单词的编号相同。如果整个文字以单词分隔符号开头，那么这个单词分隔符 node 的编号是 0. 注意这个选项不能用在前面介绍的各个样式（style）中。




```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text, word count=\i, every word separator/.style={fill=red!30},
      characters={text along path, shape=circle, fill=gray!50}}]
  \path [decorate, text effects={characters/.append={label=above:\footnotesize\i}}] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/word total=⟨*macro*⟩ (no default)

这个选项将单词的总数保存在宏 ⟨*macro*⟩ 中。注意这个选项不能用在前面介绍的各个样式（style）中。

/pgf/decoration/text effects/style characters={⟨*characters*⟩}with{⟨*effects*⟩} (no default)

在用文字做装饰时，装饰的文字中可能含有某个（某些）符号，例如，假设装饰文字是 `text-title`，其中含有字母符号 `t` 以及符号 `-`，那么可以用本选项对这些符号 `t`，`-` 做某种特别设置。

这里 ⟨*characters*⟩ 是将要被设置的某个或某些符号，这些符号依次列出，之间不需要（用逗号或空格）分隔。⟨*effects*⟩ 是所期望的外观设置（即 tikz 选项设置）。

Falsches Üben von Xylophonmusik quält jeden größeren Zwerg

```
\begin{tikzpicture}[decoration={text effects along path,
    text={Falsches {\"U}ben von Xylophonmusik qu{\"a}lt jeden gr{\"o}{\ss}eren Zwerg},
    text effects/.cd,
      path from text,
      style characters=aeiou{\"U}{\"a}{\"o} with {text=red},
      characters={text along path}}]
  \path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/path from text=⟨*true or false*⟩ (default `true`)

当被装饰路径只含有一个点时 P，将本选项的值设为 `true`，那么 PGF 会计算装饰文字的总宽度 d，并且假设一个水平向右的直线段 $|PQ| = d$，把 PQ 当作被装饰路径，因此装饰文字就沿着水平向右的方向显示。

text effects along path!

```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text,
      character count=\i, character total=\n,
      characters={text along path, scale=\i/\n+0.5}}]
  \path [decorate] (0,0);
\end{tikzpicture}
```

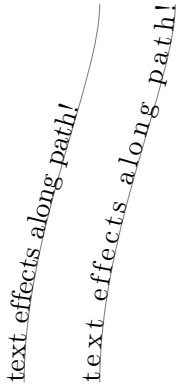
/pgf/decoration/text effects/path from text angle=⟨*angle*⟩ (no default)

这个选项与前一个选项 `path from text` 配合使用，本选项会使得假想的被装饰直线段 PQ 围绕起点 P 旋转 ⟨*angle*⟩ 角度。


```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text, path from text angle=60,
      character count=\i, character total=\n,
      characters={text along path, scale=\i/\n+0.5}}]
  \path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/fit text to path=⟨*true or false*⟩ (default `true`)

这个选项会使得装饰字符 node 在被装饰路径上均匀分布。




```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/every character/.style={text along path}}]
  \path [draw=gray, postaction={decorate}, rotate=90]
    (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
  \path [draw=gray, postaction={decorate}, rotate=90,
      yshift=-1cm, text effects={fit text to path}]
    (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\end{tikzpicture}
```

/pgf/decoration/text effects/scale text to path=⟨*true or false*⟩ (default `true`)

这个选项会根据被装饰路径的长度对装饰字符 node 做放缩，使得整个被装饰路径从头到尾全被装饰起来。


```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/every character/.style={text along path}}]
  \path [draw=gray, postaction={decorate}, rotate=90,
      yshift=0.8cm, text effects={scale text to path}]
    (0,0) .. controls ++(1,0) and ++(-0.5,0) .. (2.5,-0.5);
  \path [draw=gray, postaction={decorate}, rotate=90]
    (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
  \path [draw=gray, postaction={decorate}, rotate=90,
      yshift=-0.8cm, text effects={scale text to path}]
    (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\end{tikzpicture}
```

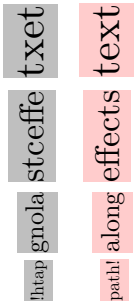
/pgf/decoration/text effects/reverse text (no value)

这个选项使得各个字符 node 按照倒序排布，如果原来的装饰文字是“从左向右”阅读的，那么使用本选项后，装饰文字是“从右向左”阅读的，不过 PGF 先将各个字符 node 倒序排布后再对它们做处理，因此各个字符 node 的编号仍然是“从左向右”的，选项的作用也是“从左向右”的。注意，装饰文字开头的‘soft’spaces 会被忽略。


```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text, path from text angle=60,
      character count=\i, character total=\n,
      characters={text along path, scale=\i/\n+0.5}}]
  \path [decorate, text effects={reverse text}] (0,0);
  \path [red, decorate, decoration={reverse path},
    text effects={characters/.append={scale=-1}}] (1,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/group letters (no value)

这个选项把连续的数个字母 node （即一个单词）看作一个“分组”，并把一个“分组”作为一个字符 node 来处理。如果同时使用选项 reverse text 和 group letters，那么一定要注意二者的次序。



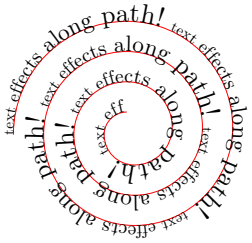

```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text, path from text angle=90,
      every word separator/.style={fill=none},
      character count=\i, character total=\n,
      characters={text along path, fill=gray!50, scale=\i/\n+0.5}}]
  \path [decorate, text effects={reverse text, group letters}] (0,0);
  \path [decorate, text effects={group letters, reverse text,
    characters/.append={fill=red!20}}] (1,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/repeat text=⟨*times*⟩ (default 将路径完整装饰所需的次数)

/pgf/decoration/text effects/repeat text=⟨*times*⟩

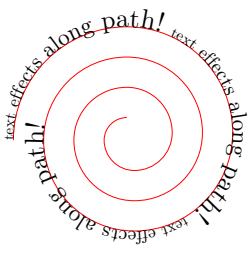
如果被装饰路径过长，而装饰文字过短，那么装饰文字就只能装饰一部分路径，此时可使用这个选项，让装饰文字沿着被装饰路径重复。如果使用选项 repeat text，那么装饰文字会不断重复直到把被装饰路径全部装饰完毕。如果使用选项 repeat text=⟨*times*⟩，那么在首次添加完毕装饰文字后，再重复添加 ⟨*times*⟩ 次，故装饰文字总共被添加 ⟨*times*⟩ + 1 次。在重复装饰文字时，字符、字母、单词的编号都会重新编排，针对装饰文字的各种选项也会重新计算。

下面的例子设置选项 repeat text:




```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!\ },
    text effects/.cd,
      repeat text,
      character count=\m, character total=\n,
      characters={text along path, scale=0.5+\m/\n/2}}]
  \path [draw=red, ultra thin, postaction=decorate]
    (180:2) \foreach \a in {0,...,12}{ arc (180-\a*90:90-\a*90:1.5-
    ↪  \a/10) };
\end{tikzpicture}
```

将上面的例子修改为选项 repeat text=2:



```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!\ },
    text effects/.cd,
      repeat text=2,
      character count=\m, character total=\n,
      characters={text along path, scale=0.5+\m/\n/2}}]
  \path [draw=red, ultra thin, postaction=decorate]
    (180:2) \foreach \a in {0,...,12}{ arc (180-\a*90:90-\a*90:1.5-
    ↪  \a/10) };
\end{tikzpicture}
```

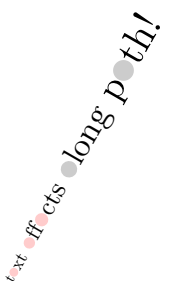
/pgf/decoration/text effects/character command=⟨*macro*⟩ (no default)

本选项所针对的对象可能是一个字符，或者数个字符，或者针对各个单词，等等。这里 ⟨*macro*⟩ 是个 TEX 宏，这个宏至多可以带有一个参数。假如本选项针对各个单词，那么在输出任何一个单词时，都会把该单词作为宏 ⟨*macro*⟩ 的操作对象，如下面的例子所示。

```
\def\mycommand#1{#1$_\n$}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text, path from text angle=60, group letters,
      word count=\n,
      every word/.style={character command=\mycommand},
      characters={text along path}}]
  \path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/replace characters=⟨*characters*⟩with{⟨*code*⟩} (no default)

这里 ⟨*characters*⟩ 是一个或一串符号，其中相邻两个字符之间不必用空格或其它符号分隔。本选项的 ⟨*characters*⟩ 声明了某些字符，一旦装饰文字中出现了这些被声明的字符，就使用 ⟨*code*⟩ 代替这些被声明的字符。⟨*code*⟩ 可以是绘图命令、字符、数学公式。



```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text, path from text angle=60,
      replace characters=e with {\fill [red!20] (0,1mm) circle
      ↪  [radius=1mm];},
      replace characters=a with {\fill [black!20] (0,1mm) circle
      ↪  [radius=1mm];},
      character count=\i, character total=\n,
      characters={text along path, scale=\i/\n+0.5}}]
  \path [decorate] (0,0);
\end{tikzpicture}
```

55.7 分形装饰

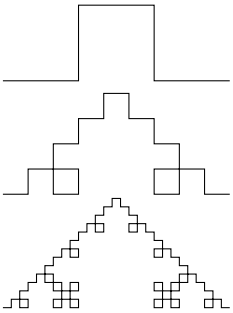
TikZ Library decorations.fractals

```
\usepgflibrary{decorations.fractals} % LaTeX and plain TeX and pure pgf
\usepgflibrary[decorations.fractals] % ConTeXt and pure pgf
\usetikzlibrary{decorations.fractals} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.fractals] % ConTeXt when using TikZ
```

这个程序库提供几种分形装饰类型，这些装饰类型主要用于被装饰路径是直线形的情况，而且要“套嵌装饰”才能显示出效果。

Decoration Koch curve type 1

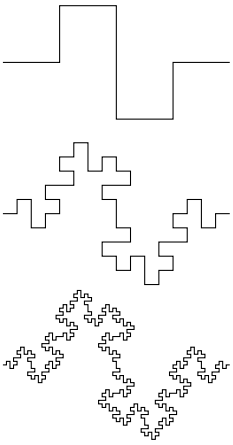
这个装饰类型是“第 1 种 Koch 曲线”，这种曲线的基本迭代方式是：将任何直线段“_____”替换为折线段“_⌐¬_”，其 Hausdorff 维数是 $\frac{\log 5}{\log 3}$.



```
\begin{tikzpicture}[decoration=Koch curve type 1]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-1.5) -- (3,-1.5) }};
  \draw decorate{ decorate{ decorate{ (0,-3) -- (3,-3) }}};
\end{tikzpicture}
```

Decoration Koch curve type 2

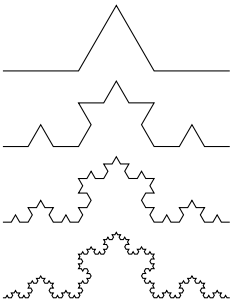
这个装饰类型是“第 2 种 Koch 曲线”，这种曲线的基本迭代方式是：将任何直线段“_____”替换为折线段“⌐¬_”，其 Hausdorff 维数是 $\frac{3}{2}$.



```
\begin{tikzpicture}[decoration=Koch curve type 2]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-2) -- (3,-2) }};
  \draw decorate{ decorate{ decorate{ (0,-4) -- (3,-4) }}};
\end{tikzpicture}
```

Decoration Koch snowflake

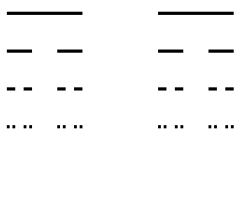
这个装饰类型的基本迭代方式是：将任何直线段“_____”替换为折线段“_/_”，其 Hausdorff 维数是 $\frac{\log 4}{\log 3}$.



```
\begin{tikzpicture}[decoration=Koch snowflake]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-1) -- (3,-1) }};
  \draw decorate{ decorate{ decorate{ (0,-2) -- (3,-2) }}};
  \draw decorate{ decorate{ decorate{ decorate{
    (0,-3) -- (3,-3) }}}};
\end{tikzpicture}
```

Decoration Cantor set

这个装饰类型的 Hausdorff 维数是 $\frac{\log 2}{\log 3}$.



```
\begin{tikzpicture}[decoration=Cantor set,very thick]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-.5) -- (3,-.5) }};
  \draw decorate{ decorate{ decorate{ (0,-1) -- (3,-1) }}};
  \draw decorate{ decorate{ decorate{ decorate{
    (0,-1.5) -- (3,-1.5) }}}};
\end{tikzpicture}
```

第五十六章　fadings 库

fadings 库定义了几种常用的灰度图，这些灰度图的宽度、高度都是 100bp.

east 矩形，右侧透明，左侧不透明。其定义是：

```
\pgfdeclarehorizontalshading{pgf@lib@fade@east}{100bp}
{color(0bp)=(pgftransparent!0); color(25bp)=(pgftransparent!0);
 color(75bp)=(pgftransparent!100); color(100bp)=(pgftransparent!100)}%
% 省略若干
\pgfdeclarefading{east}{\pgfuseshading{pgf@lib@fade@east}}%
```

可见它有 1/4 的部分是完全透明的。

west 矩形，左侧透明，右侧不透明。

north 矩形，上部透明，下部不透明。

south 矩形，下部透明，上部不透明。

circle with fuzzy edge 10 percent 圆形，其定义是：

```
\pgfdeclareradialshading{tikz@lib@fade@circle@10}{\pgfpointorigin}{
  color(0pt)=(pgftransparent!0); color(22.5bp)=(pgftransparent!0);
  color(25bp)=(pgftransparent!100); color(50bp)=(pgftransparent!100)}%
\pgfdeclarefading{circle with fuzzy edge 10 percent}{\pgfuseshading
↪  {tikz@lib@fade@circle@10}}%
```

其中的意思是：圆心在原点，距离圆心不大于 22.5bp 的地方的透明度都是 0；距离圆心大于等于 25bp 的地方的透明度都是 1；也就是说，只有 22.5bp 之外的地方的透明度有变化，所以“fuzzy edge 10 percent”指的是 $(25-22.5)/25=10\%$.

circle with fuzzy edge 15 percent 圆形，“fuzzy edge 15 percent”指的是 $(25-21.5)/25=15\%$.

circle with fuzzy edge 20 percent 圆形，“fuzzy edge 20 percent”指的是 $(25-20)/25=20\%$.

fuzzy ring 15 percent 环形，“fuzzy ring 15 percent”指的是 $(25-21.5)/25=15\%$.

第五十七章　fit 程序库

TikZ Library fit

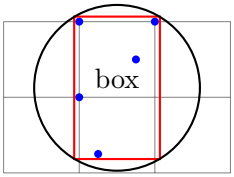
```
\usetikzlibrary{fit} % LaTeX and plain TeX
\usetikzlibrary[fit] % ConTeXt
```

这个程序库提供一种方法，能够把已创建的一个或数个坐标点，或 nodes 放入另一个 node 中（为了方便称这个 node 为 fit node）。

/tikz/fit=⟨*coordinates or nodes*⟩ (no default)

这个选项只能用作 node 的选项。⟨*coordinates or nodes*⟩ 是由坐标点或 node 名称组成的列表，列表项之间不用逗号分隔，可以用空格分隔。使用本选项后，程序会创建一个 fit node，将 ⟨*coordinates or nodes*⟩ 中列出的坐标点和 nodes 包含在内。程序首先计算一个尺寸尽量小的盒子，将列出的坐标点以及各 nodes 的锚位置 `east`，`west`，`north`，`south` 包含在盒子内，这个盒子就是被创建的 fit node 的文字盒子。

注意如果 ⟨*coordinates or nodes*⟩ 中含有坐标点且该坐标点包含逗号，要用花括号把整个 ⟨*coordinates or nodes*⟩ 括起来。




```
\begin{tikzpicture}[inner sep=0pt,thick,
    dot/.style={fill=blue,circle,minimum size=3pt}]
  \draw[help lines] (0,0) grid (3,2);
  \node[dot] (a) at (1,1) {};
  \node[dot] (b) at (2,2) {};
  \node[dot] (c) at (1,2) {};
  \node[dot] (d) at (1.25,0.25) {};
  \node[dot] (e) at (1.75,1.5) {};
  \node[draw=red, fit={(1,1) (b) (c) (d) (e)}] {box};
  \node[draw,circle,fit=(a) (b) (c) (d) (e)] {};
\end{tikzpicture}
```

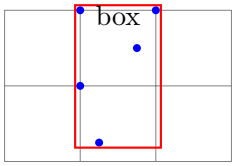
/tikz/every fit (style, initially empty)

这是个样式，它针对所有的 fit node.

关于 fit node 注意以下几点：

1. fit node 的文字盒子包含 ⟨*coordinates or nodes*⟩ 中列出的坐标点，以及列出的 nodes 的锚位置 `east`，`west`，`north`，`south`（而不是整个 node）。
2. 可以用选项 `text width` 来调节 fit node 的文字盒子的宽度。
3. fit node 的文字盒子使用对齐方式 `align=center`，这个对齐方式是固定不变的，因此不能使用选项 `align` 来改变 fit node 的文字盒子的对齐方式。
4. 可以给 fit node 使用 `at` 选项来确定它的锚定点。
5. fit node 的锚位置 `center` 位于它的锚定点上。
6. 程序根据 fit node 的文字盒子的内容来确定 fit node 的宽度和高度。

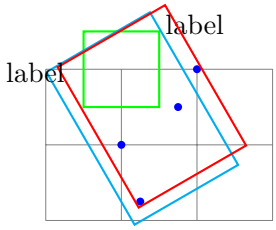
由于 fit node 的文字盒子的对齐方式是固定不变的，因此要想调整其文字盒子里的文字内容的位置就需要变通的方法，如下面的例子所示，另作一个包含文字的 node 添加到图形中：




```
\begin{tikzpicture}[inner sep=0pt,thick,
    dot/.style={fill=blue,circle,minimum size=3pt}]
  \draw[help lines] (0,0) grid (3,2);
  \node[dot] (a) at (1,1) {};
  \node[dot] (b) at (2,2) {};
  \node[dot] (c) at (1,2) {};
  \node[dot] (d) at (1.25,0.25) {};
  \node[dot] (e) at (1.75,1.5) {};
  \node[draw=red,fit=(a) (b) (c) (d) (e)] (fit) {};
  \node[below] at (fit.north) {box};
\end{tikzpicture}
```

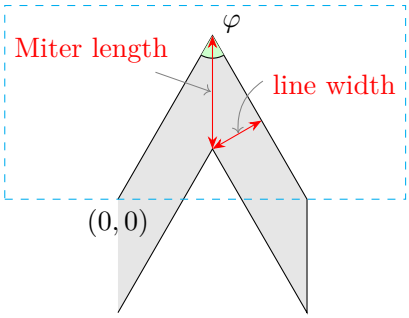
/tikz/rotate fit=⟨*angle*⟩ (no default, initially 0)

这个选项将 fit node 旋转 ⟨*angle*⟩ 角度，它的副作用是会把 /tikz/rotate 的值也设成 ⟨*angle*⟩. 注意对于 fit node 来说，选项 rotate fit 与 rotate 的作用是不同的。rotate fit 作用后的结果仍然是 fit 的，但 rotate 的作用结果未必还是 fit 的。




```
\begin{tikzpicture}[inner sep=0pt,thick, dot/.style={fill=blue,circle,minimum size=3pt}]
  \draw[help lines] (0,0) grid (3,2);
  \node[dot] (a) at (1,1) {};
  \node[dot] (b) at (2,2) {};
  \node[minimum size=1cm,draw=green] (c) at (1,2) {};
  \node[dot] (d) at (1.25,0.25) {};
  \node[dot] (e) at (1.75,1.5) {};
  \node[draw=cyan, rotate=30, fit=(a) (b) (c) (d) (e), label={above right:label}] {};
  \node[draw=red, rotate fit=30, fit=(a) (b) (c) (d) (e), label={above left:label}] {};
\end{tikzpicture}
```

有时候会遇到这种情况：画出图形后才发现图形中包含了某些个不需要的东西，实际上仅仅需要已画出图形的某一部分。例如，画出下面的图形后：




```
\tikz{
  \tikzmath{
    \jiao=60; \jing=2.5; \miterlen=1.5;
    coordinate \b,\c;
    \b=(\jiao:\jing);
    \c=($(\jiao:\jing)+(-\jiao:\jing)$);
  }
  \coordinate (a) at (0,0);
  \coordinate (b) at (\b);
  \coordinate (c) at (\c);
  \foreach \pyd in {a,b,c}
    \coordinate (\pyd') at ($(\pyd)+(0,-\miterlen)$);
  \filldraw [fill=gray!20](a)--(b)--(c)--(c')--(b')--(a');
```

```
  \pic ["$\varphi$"{name=phi,right}, draw, fill=green!20, angle radius=3mm, angle eccentricity=-0.5]
  ↪  {angle=a--b--c};
  \draw [Stealth-Stealth,red](b)--node[inner sep=0pt,pin={[name=ml,pin distance=0.5cm,pin edge={<-
  ↪  }]150:Miter length}]{}(b');
  \draw [Stealth-Stealth,red](b')--node[inner sep=0pt,pin={[name=lw,pin distance=0.5cm,pin edge=
  ↪  {<-,bend left}]50:line width}]{}($(b')+(90-\jiao:{\miterlen*sin(0.5*\jiao)})$);
  \node[below]{$(0,0)$};
  \node (fit node)[draw=cyan,dashed,inner sep=0pt,fit=(a)(ml)(phi)(lw)]{};
}
```

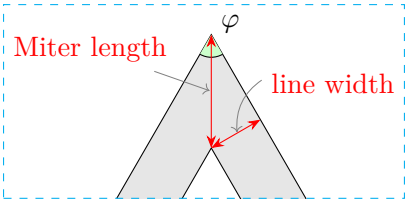
可能会觉得实际上只需要虚线框内的部分，即 (fit node) 内的那一部分。这个情况下首先想到的当然是使用 clip 命令，但是 (fit node) 是由最后一个 node 命令创建的，给这个 node 命令带上 clip 选项并不能达到目的。此时可以变通一下，先把上面的绘图代码保存到某个 key 中：

```
\pgfkeys{/caotu/.code={
  \tikzmath{
    \jiao=60; \jing=2.5; \miterlen=1.5;
    coordinate \b,\c;
    \b=(\jiao:\jing);
    \c=($(\jiao:\jing)+(-\jiao:\jing)$);
  }
  \coordinate (a) at (0,0);
  \coordinate (b) at (\b);
  \coordinate (c) at (\c);
  \foreach \pyd in {a,b,c}
    \coordinate (\pyd') at ($(\pyd)+(0,-\miterlen)$);
  \filldraw [fill=gray!20](a)--(b)--(c)--(c')--(b')--(a');
  \pic ["$\varphi$"{name=phi,right}, draw, fill=green!20, angle radius=3mm, angle
  ↪  eccentricity=-0.5] {angle=a--b--c};
  \draw [Stealth-Stealth,red](b)--node[inner sep=0pt,pin={[name=ml,pin
  ↪  distance=0.5cm,pin edge={<-}]150:Miter length}]{}(b');
  \draw [Stealth-Stealth,red](b')--node[inner sep=0pt,pin={[name=lw,pin
  ↪  distance=0.5cm,pin edge={<-,bend left}]50:line width}]{}($(b')+(90-\jiao:{
  ↪  \miterlen*sin(0.5*\jiao)})$);
  \node[below]{$(0,0)$};
  \node (fit node)[draw=cyan,dashed,inner sep=0pt,fit=(a)(ml)(phi)(lw)]{};
}}
```

然后执行：

```
\tikz{
  \makeatletter
    \pgfsys@begininvisible
      \pgfkeys{/caotu}
    \pgfsys@endinvisible
  \makeatother
  \pgfresetboundingbox
  \clip (fit node.south west) rectangle (fit node.north east);
  \pgfkeys{/caotu}
}%
```

就得到



上面代码中使用命令 \pgfsys@begininvisible→P. 216 和 \pgfsys@endinvisible→P. 216, 这两个命令之间的绘图命令所画出来的图形是“不可见的”（不可见的原因是被一个画布变换平移到了很远的地方，被画布变换的内容不会被 PGF 计入 bounding box 之内）。命令 \pgfresetboundingbox 使得程序“忘记”已经计算出的图形的边界盒子，并从当下开始重新计算边界盒子。上面例子中，把绘图代码放在键 /caotu 中，使用命令 \pgfkeys 将图形画了两遍，第一遍画的是不可见的图形，然后用命令 \pgfresetboundingbox 重设边界盒子并设置剪切路径；第二遍画的是可见的图形，但受到剪切，结果就是虚线框内的部分。

第五十八章　graphs 库

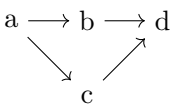
58.1　Overview

“图”是由顶点和顶点之间的边构成的图形。顶点就是 node，边就是 node 之间的连线。Ti*k*Z 的程序库 graphs 提供一些命令和选项来绘制图。

```
\usetikzlibrary{graphs} % LaTeX and plain TeX
\usetikzlibrary[graphs] % ConTeXt
```

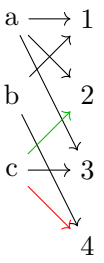
用 Ti*k*Z 的 \node 命令和 edge 算子也可以绘制图，只是需要手工确定顶点之间的相对位置，需要一定的手工计算量。graphs 程序库提供简单的算法来自动确定顶点之间的相对位置，省去了手工计算的麻烦，但是只能绘制比较简单的图。要想绘制复杂的图，参考手册的 graph drawing algorithms.

由于使用了自动化的算法，graphs 程序库的句法与通常的 Ti*k*Z 句法不同，下面是几个例子。

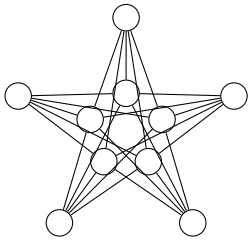



```
\tikz \graph { a -> {b, c} -> d };
```

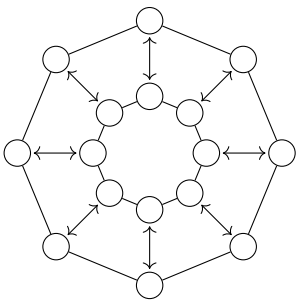
下面例子中的算子 subgraph I_nm 需要调用 graphs.standard 程序库：



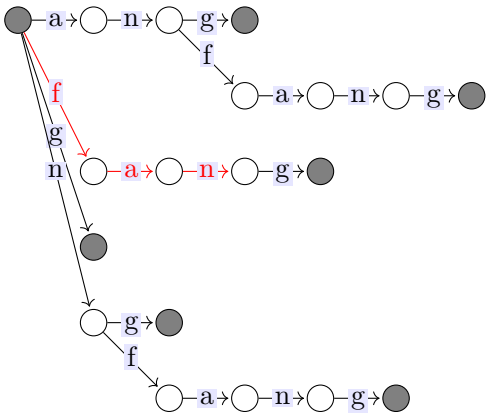

```
\tikz \graph {
  subgraph I_nm [V={a, b, c}, W={1,...,4}];
    a -> { 1, 2, 3 };
    b -> { 1, 4 };
    c -> { 2 [>green!75!black], 3, 4 [>red]}
};
```



```
\tikz
\graph [nodes={draw, circle}, clockwise, radius=.5cm,
    empty nodes, n=5] {
  subgraph I_n [name=inner] --[complete bipartite]
  subgraph I_n [name=outer]
};
```



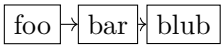
```
\tikz
\graph [nodes={draw, circle}, clockwise, radius=.75cm,
    empty nodes, n=8] {
  subgraph C_n [name=inner] <->[shorten <=1pt, shorten >=1pt]
  subgraph C_n [name=outer]
};
```




```
\tikz [>={To[sep]}, rotate=90, xscale=-1,
      mark/.style={fill=black!50}, mark/.default=]
  \graph [trie, simple, nodes={circle,draw},
          edges={nodes={
            inner sep=1pt, anchor=mid,
            fill=examcodebgcolor}},
          put node text on incoming edges]
  {
    root[mark] -> {
      a -> n -> {
        g [mark],
        f -> a -> n -> g [mark]
      },
      f -> a -> n -> g [mark],
      g[mark],
      n -> {
        g[mark],
        f -> a -> n -> g[mark]
      }
    },
    { [edges=red]
      root -> f -> a -> n
    }
  };
```

58.2 基本概念

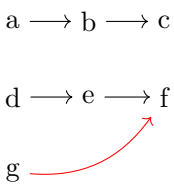
58.2.1 顶点，链




```
\tikz [every node/.style = draw]
  \graph { foo -> bar -> blub };
```

如上例所示，在命令 `\graph{...}` 内绘制顶点链，`foo -> bar -> blub` 生成一个顶点链，其中有 3 个 node. 文字 `foo`，`bar`，`blue` 有多个作用：创建 node，作为相应 node 的名称，也用作 node 内部的文字。符号 `->` 在 node 之间画箭头，符号 `->` 有“算子”的特点，它后面可以带有选项来设置边的外观样式。

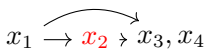
在一个`\graph{...}` 内可以绘制多个链，链与链之间用逗号或分号分隔。如果重复使用某个文字（即 node 的名称），那么默认之下，这个 node 就会被再次连接。




```
\tikz \graph {
  a -> b -> c,
  d -> e -> f;
  g ->[red,bend right] f
};
```

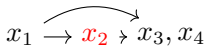
node 名称具有“算子”的特性，它可以创建 node，也可以在它后面设置方括号选项，方括号里的选项的路径可以是 /tikz/graphs 或者 /tikz.

如果想把 node 的内容与名称区分开，分别设置，可以给 node 名称使用 as=⟨*content text*⟩ 选项，也可以在名称后使用一个斜线“/”来引出 node 的内容：



```
\tikz \graph {
  x1/$x_1$ -> x2 [as=$x_2$, red] -> x34/{$x_3,x_4$};
  x1 -> [bend left] x34;
};
```

可以使用双引号句法来创建 node, 例如：



```
\tikz \graph {
  "$x_1$" -> "$x_2$"[red] -> "$x_3,x_4$";
  "$x_1$" ->[bend left] "$x_3,x_4$";
};
```

这个例子中，"x_1" 是 node 名称，也作为引用句法生成 node 的内容，即数学模式下的符号。

58.2.2 顶点组

如果一个组中的各个链都只有一个顶点，这个组就是顶点组。

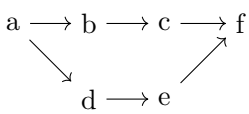
58.2.3 链组

如果一个组内有某些链，那么这个组就是一个“链组”（chain group）或简称之为一个“组”。\graph 命令后面被花括号包裹的参数是一个链组，再例如

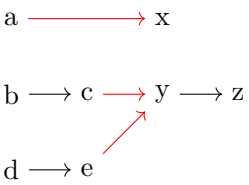
```
{a} 或者
{ a -> b } 或者
{a, b -> { c, d} } 等等
```

一个链组本身可以用作一个链，也可以把一个链组用作一个顶点。

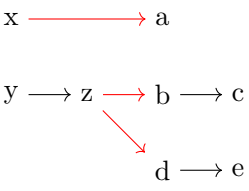
把一个链组与另一个链组连接时，默认使用选项 matching and star 决定的规则画边，这个规则会尽量按照顶点次序来连接顶点。



```
\tikz \graph {
  a -> {
    b -> c,
    d -> e
  } -> f
};
```

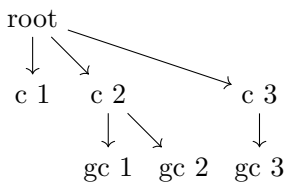


```
\tikz \graph {
  {a ,
   b -> c,
   d -> e} ->[red]
  {x,
   y->z}
};
```



```
\tikz \graph {
  {x,
   y->z} ->[red]
  {a ,
   b -> c,
   d -> e}
};
```

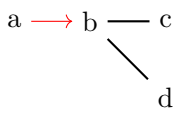
用链组可以构造“树”：




```
\tikz \graph [grow down,branch right=1cm]
{
  root -> {c 1,
           c 2 -> {gc 1,
                   gc 2},
           c 3 -> {gc 3}}
};
```

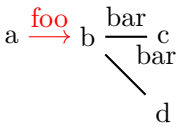
58.2.4 边的外观及其标签

顶点、链组之间的连线用符号 `->`（带箭头）或 `--` 画出（不带箭头），连线符号后面可以带有方括号选项来设置边的外观，方括号里的选项的路径可以是 `/tikz/graphs` 或者 `/tikz`.



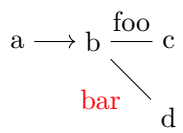

```
\tikz \graph {
  a ->[red] b --[thick] {c, d};
};
```

给边带上引用句法选项（§17.10.4）可以给边加标签：




```
\tikz \graph {
  a ->[red, "foo"] b --[thick, "bar"] {c, d};
};
```

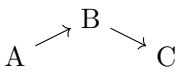
上面例子中，顶点 `b` 后面的 `--` 会在 `b`，`c` 和 `b`，`d` 之间分别画线，如果要对 `b`，`c` 和 `b`，`d` 之间的连线分别设置，使这两条边有不同的外观，则需要在 `c` 和 `d` 之后分别使用选项设置，这要用到符号“`>`”，例如：




```
\tikz \graph {
  a -> b -- {c [> "foo"], d [> "bar"{',red}]};
};
```

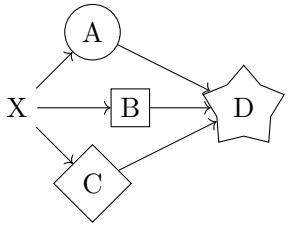
58.2.5 顶点集

观察两个例子。



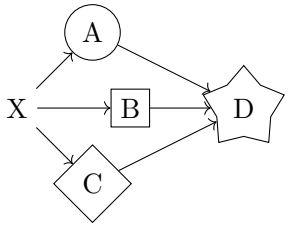

```
\tikz {
  \node (a) at (0,0) {A};
  \node (b) at (1,.5) {B};
  \node (c) at (2,0) {C};
  \graph { (a) -> (b) -> (c) };
}
```

这个例子中，先创建 3 个 node 并为它们命名，然后用命令 `\graph` 引用它们的名称，在它们之间画线，做成一个图。




```
\tikz [new set=my nodes] {
  \node [set=my nodes, circle, draw] at (1,1) {A};
  \node [set=my nodes, rectangle, draw] at (1.5,0) {B};
  \node [set=my nodes, diamond, draw] at (1,-1) {C};
  \node (d) [star, draw] at (3,0) {D};
  \graph { X -> (my nodes) -> (d) };
}
```

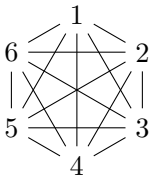
在这个例子中，先创建 4 个 node，前 3 个 node 都带上选项 `set=my nodes`，此选项把这 3 个 node 做成一个名称为 `my nodes` 的“顶点集合”，然后在命令 `\graph` 中引用它的名称。实际效果相当于：




```
\tikz [new set=my nodes] {
  \node [name=a, circle, draw] at (1,1) {A};
  \node [name=b, rectangle, draw] at (1.5,0) {B};
  \node [name=c, diamond, draw] at (1,-1) {C};
  \node (d) [star, draw] at (3,0) {D};
  \graph { X -> {(a),(b),(c)} -> (d) };
}
```

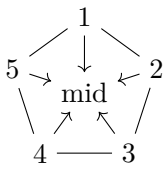
58.2.6 图宏

图宏是个“算子”,它可以带有选项,按照其选项的设置自动生成某个样式的图。在程序库 graphs.standard 中预定义了几种图宏。




```
\tikz \graph { subgraph K_n [n=6, clockwise] };
```

这个例子中的 `subgraph K_n` 是个图宏，其选项设置 6 个顶点，顶点按顺时针方向排成一圈。



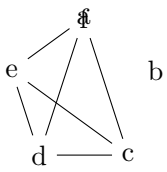

```
\tikz \graph { subgraph C_n [n=5, clockwise] -> mid };
```

58.2.7 子图

见 /tikz/graphs/declare → P. 1119.

58.2.8 在顶点之间连线的规则

有的图包含多个边，手工画边有点麻烦。程序库预定义了数个选项，每个选项都定义了一个连线规则，可以在指定的顶点之间自动画边（这些选项在后文介绍）。例如，在一个链组（用花括号括起来的数个链）的开端处（开花括号 { 之后）使用选项 [clique=⟨*color*⟩]，并且在本链组的某些个 node 后面使用选项 [⟨*color*⟩]，那么这些 node 就被标记了，这些被标记的 node 构成一个集合，并且在此集合的任意两个 node 之间画线。




```
\tikz \graph [clockwise, n=5] {
  a, b, {[clique]c, d, e, f}
};
```

这个例子中，选项 clockwise 将顶点按顺时针方向排成一圈，选项 n=5 选定圆圈上的 5 个位置来放置顶点，但顶点有 6 个，所以最后一个顶点 f 与第一个顶点 a 重叠。在链组 {[clique]c, d, e, f} 的开端处使用了选项 clique，将此链组内的顶点两两连线。


```
\tikz \graph [color class=red,clockwise, n=5] {
  a, b, {[clique=red,->]c, d -> e[red,> green], f[red,> purple]}
};
```

这个例子中，先用选项 color class=red 声明一个“颜色类别”，类别名称 red 只是一个标记符号，并不真地画出红色。选项 clique=red 只对属于类别 red 的顶点有效，其中只有顶点 e，f 属于该类别，因此所画的边只与 e，f 有关。

58.2.9 注意句法格式

关于命令 \graph[⟨*graph options*⟩]{⟨*group specification*⟩}; 的句法格式：

- ⟨*graph options*⟩ 中的选项的路径应当是 /tikz/graphs.
- 包裹 ⟨*group specification*⟩ 的花括号最好不要省略。
- 顶点可以是用圆括号包裹的 node 名称，或顶点集合名称，圆括号代表“引用指定的顶点”。
- 相邻的 2 个链之间以分号或逗号为分隔标志。
- 如果出现连续的分号或逗号，则意味着由空的顶点构成的空链。
- 在 ⟨*group specification*⟩ 中，用分号或者逗号作为一个链的结束标志，但分号或者逗号不属于链本身；在内部处理中，分号会被替换为逗号。
- 在 ⟨*group specification*⟩ 中使用套嵌的花括号组来明确链的层次。注意花括号包裹的应当是一个顶点 (不以分号或逗号结束，将被看作花括号组内的一个链)，或者是某些 (个) 链 (相邻 2 个链之间以分号或逗号为分隔标志)。
- 在一个链中，可以把一个组用作一个顶点，此时这个组内应当是一个顶点 (不以分号或逗号结束，将被看作花括号组内的一个链)，或者某些 (个) 链 (相邻 2 个链之间以分号或逗号为分隔标志)。
- 在一个链中，相邻两个顶点之间必须使用“边”相连，边应当是 <->，<-，->，--，-!- 这 5 种之一。
- 顶点 node 后方括号里的选项 (键) 的路径可以是 /tikz/graphs 或者 /tikz.
- 边后方括号里的选项 (键) 的路径可以是 /tikz/graphs 或者 /tikz.

58.3 \graph 的处理流程

\graph[⟨*graph options*⟩]{⟨*group specification*⟩};

在 tikzpicture 环境的开头会定义

```
\def\graph{\path graph}%
```

由于 \tikz@lib@graphs@normal@main 的定义有前缀 \long, 所以在 ⟨*group specification*⟩ 可以含有 \par.

在路径命令 \path ... graph ...; 的处理过程中，当 \tikz@scan@next@command → P. 757 遇到 graph 时会导致执行 \tikz@graph, \tikz@lib@graph@parser. 在《tikz.code.tex》中有如下的初始定义：

```
\def\tikz@graph aph{\tikz@lib@graph@parser}%

\def\tikz@lib@graph@parser{\pgfutil@ifnextchar[\tikz@graph@error{\tikz@graph@error[]}}
↪  %]%
\def\tikz@graph@error[#1]#2{%
```

```
  \tikzerror{You need to say \string\usetikzlibrary{graphs} in order to use the graph
  ↪ syntax}%
  \tikz@lib@graph@parser@done%
}%

\def\tikz@lib@graph@parser@done{%
  \tikz@scan@next@command%
}%
```

文件《tikzlibrarygraphs.code.tex》会重定义 \tikz@lib@graph@parser, \tikz@lib@graph@parser@done.

命令 \graph[⟨graph options⟩]{⟨group specification⟩}; 导致的处理是:首先由 \tikz@lib@graph@parser@ 读取 ⟨graph options⟩, 然后由 \tikz@lib@graphs@normal@main 读取 ⟨group specification⟩, 再对 ⟨group specification⟩ 作进一步处理。整个处理过程如下:

```
\setbox\tikz@whichbox=\hbox\bgroup%
  \unhbox\tikz@whichbox%
  \hbox\bgroup
    \bgroup%
      \pgfinterruptpath%
        \scope[graphs/.cd,@graph drawing setup/.try,@operators=,every graph/.try,
        ↪ ⟨graph options⟩]% 选项
          \begingroup%
%             \iftikz@graph@quick%
%               \expandafter\tikz@lib@graphs@parse@quick@graph%
%             \else%
%               \expandafter\tikz@lib@graphs@normal@main%
%             \fi%
% 假设 \iftikz@graph@quick 的真值是 false
% 执行 \tikz@lib@graphs@normal@main
%\tikz@lib@graphs@normal@main{⟨group specification⟩} 以下
            \pgfkeysgetvalue{/tikz/graphs/@operators}\tikz@lib@graph@outer@operators%
            \let\tikz@lib@graph@options\pgfutil@empty%
            \let\tikz@lib@graph@node@list\pgfutil@empty%
            \pgfkeyssetvalue{/tikz/graphs/placement/depth}{0}%
            \pgfkeyssetvalue{/tikz/graphs/placement/width}{0}%
            \pgfkeyssetvalue{/tikz/graphs/placement/level}{0}%
%%\tikz@lib@graph@start@hint@group 以下
            \pgfkeyssetvalue{/tikz/graphs/placement/local depth}{0}%
            \pgfkeyssetvalue{/tikz/graphs/placement/local width}{0}%
            \pgfkeyssetvalue{/tikz/graphs/placement/chain count}{0}%
            \pgfkeyssetvalue{/tikz/graphs/placement/element count}{0}%
%%\tikz@lib@graph@start@hint@group 以上
              \tikz@lib@graph@parse@group →P. 1032{⟨group specification⟩}% 这一行
%%\tikz@lib@graph@end@hint@group 以下
            \xdef\tikz@lib@graph@group@depth{\pgfkeysvalueof
            ↪ {/tikz/graphs/placement/local depth}}
            \xdef\tikz@lib@graph@group@width{\pgfkeysvalueof
            ↪ {/tikz/graphs/placement/local width}}
%%\tikz@lib@graph@end@hint@group 以上
            \tikz@lib@graph@outer@operators%
            \let\tikz@lg@do=\tikz@lib@graph@cleanup%
            \tikz@lib@graph@node@list%
%%\tikz@lib@graph@main@done 以下
          \endgroup%
        \endscope%
      \endpgfinterruptpath%
```

```
    \egroup
  \egroup%
\egroup% 结束 \tikz@whichbox
\tikz@lib@graph@parser@done%
%%\tikz@lib@graph@main@done 以上
%\tikz@lib@graphs@normal@main{⟨group specification⟩} 以上
```

本命令将 ⟨*group specification*⟩ 的解析结果放到一个组中，再把这个组放到一个水平盒子中，再把这个水平盒子添加到盒子 `\tikz@whichbox` 中。可见命令 `\graph` 创建的图类似 node, 是路径的“附加物”，在路径结束命令 `\tikz@finish` 那里会向输入流中插入盒子 `\tikz@whichbox` 的内容。

在命令 `\tikz@lib@graph@parse@group{`⟨*group specification*⟩`}` 的处理过程中，⟨*group specification*⟩ 中的组的套嵌结构决定了以下处理原则：

- 按组的套嵌结构，先读取靠近外层的链或者组。
- 按组的套嵌结构，如果两个链处于同一层次，则先处理在前的链。
- 按组的套嵌结构，先处理内层的组，结束内层组后，再回到外层组继续作处理。

命令 `\tikz@lib@graph@parse@group` → P. 1032 的处理过程大体上是：

1. 读入的 ⟨*group specification*⟩ 可能是这样带选项的形式：

   ```
   [⟨group options⟩]...
   ```

   此时执行选项

   ```
   \tikzgraphsset{
     @operators=,
     every group/.try,
     @extra group options,
     @extra group options/.style=,%
     ⟨group options⟩}%
   ```

2. 调整 ⟨*group specification*⟩ 中的符号，例如去掉其中的 `\par`, 添加花括号，用逗号替换分号。最后，在所得结果的末尾加上逗号，留给下一步作处理。

   参考 `\tikz@lib@graph@par`.

3. 以逗号为标志，读取一个链 ⟨*a chain*⟩, 见 `\tikz@lib@graph@main@parser@cont@normal` → P. 1034.

4. 解析链 ⟨*a chain*⟩ 中的顶点、边。

5. 如果某个顶点是一个组，就再次调用 `\tikz@lib@graph@parse@group` → P. 1032 处理这个组的内容。所以命令 `\tikz@lib@graph@parse@group` 是一个循环处理。

   见 `\tikz@lib@graph@parse@one`, `\tikz@lib@graph@scope`.

6. 按类似方式逐个处理所有的链。

/tikz/graphs/every graph

这个键是没有定义的，如果想用这个键，应当提前定义它，例如

```
\tikzgraphsset{every graph/.style=...} 或者
\tikzgraphsset{every graph/.append style=...} 或者
\tikzgraphsset{every graph/.code=...} 或者
\tikzgraphsset{every graph/.append code=...} 或者其他
```

例如，对于

```
{
  {
    a -> b,
    c -> d,
  } -> e,
  f -> g,
},
```

其处理是：

1. 先读取最外层的链

   ```
   {
     {
       a -> b,
       c -> d,
     } -> e,
     f -> g,
   }
   ```

   这个链只有一个作为顶点的链组
2. 进入组中读取靠近最外层的链

   ```
   {
     a -> b,
     c -> d,
   } -> e
   ```

   这个链的第一个顶点是一个链组
3. 进入组中读取并解析一个链

   ```
   a -> b
   ```

4. 然后读取并解析链

   ```
   c -> d
   ```

5. 然后读取并解析

   ```
   -> e
   ```

6. 然后读取并解析

   ```
   f -> g
   ```

在解析 ⟨*group specification*⟩ 的过程中：

- 命令 \tikz@lib@graph@main@parser@cont@normal → P. 1034 提取一个链。
- 命令 \tikz@lib@graph@parse@one → P. 1037 提取一个顶点。
- 解析 \pgf@stop@eogroup 时就意味着 ⟨*group specification*⟩ 的结束，此时执行 \tikz@lib@graph@graph@group@done.
- 解析 \pgf@stop@eodashes 时就意味着一个链的结束，此时执行 \tikz@lib@graph@graph@done, 再解析之后的链;如果之后没有链,那就应当遇到记号 \pgf@stop@eogroup, 这意味着 ⟨*group specification*⟩ 的结束。
- 解析 \pgf@stop 时就意味着一个顶点 node 的结束，排布这个顶点后，再解析这个顶点后面的边；如果这个顶点后面没有边，那就应当遇到记号 \pgf@stop@eodashes, 这意味着当前链的结束。
- 解析边的命令是
  - 首先由 \tikz@lib@graph@back@arrow 检查是否反向边 <- 或者双向边 <->
  - 如果不是，再由 \tikz@lib@graph@no@back@arrow 检查是否其他边，包括 ->, --, -!-

– 如果边不是以上类型就报错。

解析边之后，再调用 \tikz@lib@graph@parse@one 解析之后的顶点。

58.4 对组的处理

在 \graph[⟨graph options⟩]{⟨group specification⟩}; 中，{⟨group specification⟩} 可能是多个组的多重套嵌，按照由外而内，由前到后的读取次序来处理这些组。首先由 \tikz@lib@graphs@normal@main 读入 ⟨group specification⟩, 然后把 ⟨group specification⟩ 交给 \tikz@lib@graph@parse@group 处理。此后，每当遇到一个组时，由命令 \tikz@lib@graph@parse@group 读取这个组并处理之。

命令 \tikz@lib@graph@parse@group → P. 1032{[⟨graph options⟩]⟨group content⟩} 的处理过程是：

1. 整个地读取 [⟨graph options⟩]⟨group content⟩.

   由于命令 \tikz@lib@graph@parse@group → P. 1032 是能处理一个参数的函数,所以 {[⟨graph options⟩]⟨group content⟩} 外围的花括号用于界定参数，会被忽略。

2. 清空

```
\let\tikz@lib@graph@group@qa\pgfutil@empty%
\let\tikz@lib@graph@group@q\pgfutil@empty%
\let\tikz@lib@graph@group@c\pgfutil@empty%
\let\tikz@lib@graph@group@cont\pgfutil@empty%
\let\tikz@lib@graph@group@conta\pgfutil@empty%
```

   后续步骤对 ⟨group content⟩ 的符号作处理，用以上宏临时保存一些记号。

3. 清空

```
\let\tikz@lib@graph@parse@extras\pgfutil@empty
```

   此后，可以用键 /tikz/graphs/parse → P. 1035 重定义宏 \tikz@lib@graph@parse@extras.

4. 执行以下键值对

```
\tikzgraphsset{
  @operators=,
  every group/.try,
  @extra group options,
  @extra group options/.style=,%
  ⟨graph options⟩}%
```

5. 执行

```
\expandafter\tikz@lib@graph@par\tikz@lib@graph@parse@extras\par\pgf@stop@eogroup%
```

   意思是，将宏 \tikz@lib@graph@parse@extras 展开一次后的内容放到 ⟨group content⟩ 的前面，将 \par\pgf@stop@eogroup 放到 ⟨group content⟩ 的后面，构成一个新的记号序列，结构如下：

   ⟨ 展开一次的 *\tikz@lib@graph@parse@extras*⟩⟨group content⟩\par\pgf@stop@eogroup

   然后由命令 \tikz@lib@graph@par 处理这个记号序列：

   - 逐个去掉其中的 \par
   - 如果在其中使用了指定顶点的“双引号句法”，那么将各个配对的双引号之内的内容加上花括号，例如
     – "a" 变成 "{a}"

– `"a""b""c"` 变成 `"{a}""{b}""{c}"`

添加花括号是为了在后面的处理中，将引号的全部内容变成“一整个的”参数。

- 把其中的 `...[`⟨*something*⟩`]...` 变成 `...[{`⟨*something*⟩`}]...`
- 把其中的分号替换为逗号 (在用户输入中，分号与逗号都可以作为链、链组的结束标志)

经过以上处理后，得到记号序列

⟨*modified group content*⟩`\pgf@stop@eogroup`

例如：

a -> ”name”/”text”„

macro:->a -> ”{name}”/”{text}”,

```
{
  \makeatletter
  \let\tikz@lib@graph@group@qa\pgfutil@empty%
  \let\tikz@lib@graph@group@q\pgfutil@empty%
  \let\tikz@lib@graph@group@c\pgfutil@empty%
  \let\tikz@lib@graph@group@cont\pgfutil@empty%
  \let\tikz@lib@graph@group@conta\pgfutil@empty%
  \let\tikz@lib@graph@main@parser\relax
  \let\pgf@stop@eogroup\relax
  \tikz@lib@graph@par a -> "name"/"text";\par\pgf@stop@eogroup\par
  \meaning\tikz@lib@graph@group@conta
  \makeatother
}
```

6. 然后再执行

`\tikz@lib@graph@main@parser`→P. 1033⟨*modified group content*⟩`,\pgf@stop@eogroup`

注意，上面命令在 ⟨*modified group content*⟩ 后面添加了一个额外的逗号，这意味着，在用户输入的 ⟨*group specification*⟩ 中，任何一个组内的最后一个链的后面，可以没有分号或逗号。

上面命令会分情况处理：

- 如果 ⟨*modified group content*⟩ 是以下形式

  `{`⟨*a group*⟩`}`⟨*following*⟩

  即以一个花括号组 `{`⟨*a group*⟩`}` 开头，则导致

  ```
  \begingroup% 这个开启组的命令将在 \tikz@lib@graph@graph@done 那里结束
    \pgfkeyssetvalue{/tikz/graphs/placement/local depth}{0}%
    \pgfkeyssetvalue{/tikz/graphs/placement/local width}{0}%
    \let\tikz@lib@graph@stored@actions\pgfutil@empty%
    \let\tikz@lib@graph@node@list\pgfutil@empty% reset
    \tikz@lib@graph@check@quotes\tikz@lib@graph@main@parser@cont@normal{{
    ↪  ⟨a group⟩}}⟨following⟩,\pgf@stop@eogroup
  ```

- 如果 ⟨*modified group content*⟩ 不以一个花括号组开头，则导致

  ```
  \begingroup% 这个开启组的命令将在 \tikz@lib@graph@graph@done 那里结束
    \pgfkeyssetvalue{/tikz/graphs/placement/local depth}{0}%
    \pgfkeyssetvalue{/tikz/graphs/placement/local width}{0}%
    \let\tikz@lib@graph@stored@actions\pgfutil@empty%
    \let\tikz@lib@graph@node@list\pgfutil@empty% reset
    \tikz@lib@graph@check@quotes\tikz@lib@graph@main@parser@cont@normal
    ↪  ⟨modified group content⟩,\pgf@stop@eogroup
  ```

上面代码中，命令 `\tikz@lib@graph@main@parser@cont@normal`→P. 1034 之后的一个项目应当是一个顶点，这个顶点可能是一个花括号组。

上面代码中，\begingroup 开始的组将在 \tikz@lib@graph@graph@done→P. 1034 那里结束。

7. 继续上面的步骤，命令 \tikz@lib@graph@check@quotes→P. 1033 会检查 \tikz@lib@graph@main@parser@cont@normal→P. 1034 之后的第一个记号：
   - 如果这个记号不是双引号，就让 \tikz@lib@graph@main@parser@cont@normal→P. 1034 处理之后的记号。
   - 如果这个记号是双引号，就意味着这是一个指定顶点的双引号句法，于是处理这个双引号句法中的符号，处理结果放回到记号序列中，再让 \tikz@lib@graph@main@parser@cont@normal→P. 1034 处理记号序列。

   也就是说，会导致

```
\tikz@lib@graph@main@parser@cont@normal{{⟨a group⟩}}⟨following⟩,\pgf@stop@eogroup
或者
\tikz@lib@graph@main@parser@cont@normal⟨被处理的双引号句法⟩...,\pgf@stop@eogroup
或者，假如也不是双引号句法
\tikz@lib@graph@main@parser@cont@normal⟨modified group content⟩,\pgf@stop@eogroup
```

8. 继续上面的步骤，命令 \tikz@lib@graph@main@parser@cont@normal→P. 1034 会读取一个链，处理链中的顶点、边。

   一个顶点被处理完毕后，会遇到记号 \pgf@stop, 此时再处理顶点之后的边。一个边被处理完毕后，就再处理之后的顶点。

   在不涉及 graphdrawing 库时，每到当解析完毕一个新顶点后，就会计算这个顶点的位置，创建这个顶点并平移到计算出来的位置上。每当处理完毕一个边后的顶点时，就会分析是否创建这个边：在多重图的情况下，会立即创建这个边；在简单图的情况下，只是保存创建这个边的代码，待到适当的地方再调出所保存的代码，创建这个边。

   在一个链中也可能有作为顶点的组，遇到这种组时就执行 \tikz@lib@graph@scope→P. 1038, 此命令会调用 \tikz@lib@graph@parse@group 处理这个作为顶点的组，所以命令 \tikz@lib@graph@parse@group 是能对套嵌组作循环处理的命令。

9. 当链中的最后一个顶点被处理后，会遇到记号 \pgf@stop@eodashes, 此时执行 \tikz@lib@graph@graph@done→P. 这会导致再次执行 \tikz@lib@graph@main@parser→P. 1033, 即开始步骤 6, 继续处理之后的链。

10. 当处理完毕所有的链后，会遇到记号 \pgf@stop@eogroup, 此时执行 \tikz@lib@graph@graph@group@done→P. 1035 结束对整个组的处理。

命令 \tikz@lib@graph@parse@group 与 \tikz@lib@graph@graph@group@done→P. 1035 的组合并不会引入额外的组来限制处理过程；而 \tikz@lib@graph@scope→P. 1038 则会设置组来限制对“作为顶点的链组”的处理。

当命令 \tikz@lib@graph@parse@group 与 \tikz@lib@graph@scope→P. 1038 套嵌时，最外层必定是 \tikz@lib@graph@parse@group.

命令 6 与 \tikz@lib@graph@graph@done→P. 1034 的组合会设置组来限制对链的处理。

\tikz@lib@graph@parse@group{⟨*group specification*⟩}

通常本命令以花括号为定界标志，读取一个组的内容。组的形式应当是：

```
{[⟨graph options⟩]⟨group content⟩}
```

其中 ⟨*graph options*⟩ 中的键的路径应当是 /tikz/graph.

本命令的定义是：

```
\long\def\tikz@lib@graph@parse@group#1{
  \let\tikz@lib@graph@group@qa\pgfutil@empty%
  \let\tikz@lib@graph@group@q\pgfutil@empty%
  \let\tikz@lib@graph@group@c\pgfutil@empty%
  \let\tikz@lib@graph@group@cont\pgfutil@empty%
  \let\tikz@lib@graph@group@conta\pgfutil@empty%
  \tikz@lib@graph@group@check#1\par\pgf@stop@eogroup%
}%
```

注意本命令在 ⟨*group specification*⟩ 后面附加了 `\par\pgf@stop@eogroup`.
与本命令对应的组结束命令是 `\tikz@lib@graph@graph@group@done` →P. 1035.

`\tikz@lib@graph@group@check`[⟨*options*⟩]⟨*group specification*⟩

本命令的定义是：

```
\def\tikz@lib@graph@group@check{%
  \pgfutil@ifnextchar[\tikz@lib@graph@group@opt{\tikz@lib@graph@group@opt[]}%]
}%
```

`\tikz@lib@graph@group@opt`[⟨*options*⟩]⟨*group specification*⟩

本命令的定义是：

```
\def\tikz@lib@graph@group@opt[#1]{%
  \let\tikz@lib@graph@parse@extras\pgfutil@empty%
  \tikzgraphsset{
    @operators=,
    every group/.try,
    @extra group options,
    @extra group options/.style=,%
    #1}%
  \expandafter\tikz@lib@graph@par\tikz@lib@graph@parse@extras%
}%
```

宏 `\tikz@lib@graph@parse@extras` 由键 `/tikz/graphs/parse` 重定义。

`\tikz@lib@graph@main@parser`

本命令的定义是：

```
\def\tikz@lib@graph@main@parser{%
  \begingroup%
    \pgfkeyssetvalue{/tikz/graphs/placement/local depth}{0}%
    \pgfkeyssetvalue{/tikz/graphs/placement/local width}{0}%
    \let\tikz@lib@graph@stored@actions\pgfutil@empty%
    \let\tikz@lib@graph@node@list\pgfutil@empty% reset
    \tikz@lib@graph@main@parser@start%
}%
```

上面定义中的 `\begingroup` 将在 `\tikz@lib@graph@graph@done` →P. 1034 那里结束。

`\tikz@lib@graph@check@quotes`⟨*a token*⟩⟨*something*⟩

本命令的处理是：

1. 保存记号 ⟨*a token*⟩
2. 检查 ⟨*something*⟩ 是否以双引号开头，
   - 如果 ⟨*something*⟩ 不以双引号开头，那么本命令就直接导致 ⟨*a token*⟩⟨*something*⟩
   - 如果 ⟨*something*⟩ 以双引号开头，那么其形式应当是 "⟨*string*⟩", 但不应当是 ""⟨*string*⟩"", 并且在 ⟨*string*⟩ 中不可以出现奇数个连续的双引号 (包括 1 个双引号的情况)；此时本命令的处理是：

(a) 把 "⟨*string*⟩" 外围的一对双引号去掉，只对 ⟨*string*⟩ 进行操作。

(b) 把 ⟨*string*⟩ 中的连续的 $2n$ 个双引号替换为 n 个双引号，将所得的 ⟨*string*⟩ 保存到记号寄存器 \pgfkeys@temptoks 中。

(c) 复制 \pgfkeys@temptoks 的内容，将复制内容中的特殊符号的类代码改为 13(活动符)，并将这些活动符定义为其自身的 Unicode 名称。例如

```
\def${@DOLLAR SIGN@}
```

然后将复制内容中的特殊符号展开，这样就把复制内容中的特殊符号作了替换，所得到的结果被全局地保存在宏 \tikzlibgraphreplaced 中。

(d) 检查 ⟨*something*⟩ 后面是否有反斜线 /,

– 如果 ⟨*something*⟩ 后面没有反斜线 /, 则

```
\endgroup⟨a token⟩⟨ 展开的\tikzlibgraphreplaced⟩/
↪  ⟨ 展开的\the\pgfkeys@temptoks⟩
```

– 如果 ⟨*something*⟩ 后面有 2 个反斜线 //, 则有等价操作：吃掉这 2 个反斜线，然后插入

```
\endgroup⟨a token⟩⟨ 展开的\tikzlibgraphreplaced⟩/
↪  ⟨ 展开的\the\pgfkeys@temptoks⟩//
```

– 如果 ⟨*something*⟩ 后面有 1 个反斜线 /, 则吃掉这个反斜线，然后插入

```
\endgroup⟨a token⟩⟨ 展开的\tikzlibgraphreplaced⟩/
```

其中的 \endgroup 所对应的 \begingroup 由命令 \tikz@lib@graph@main@parser→P. 1033 设置，这个组合的作用是限制某些符号类代码、宏、计数器、记号寄存器的定义范围。

注意命令 \tikz@lib@graph@check@quotes→P. 1033 只处理顶点 node 的名称部分，而不会处理顶点 node 的内容部分。

\tikz@lib@graph@main@parser@cont@normal⟨*a chain*⟩,

本命令的定义是：

```
\def\tikz@lib@graph@main@parser@cont@normal#1,{%
  \tikz@lib@graph@parse@one#1-\pgf@stop@eodashes%
}%
```

本命令以逗号为参数定界标志，读取之后的一个链 ⟨*a chain*⟩ 并解析之。注意本命令在 ⟨*a chain*⟩ 后面添加了记号“-\pgf@stop@eodashes”, 当记号 \pgf@stop@eodashes 被解析到时，就意味着完成了对 ⟨*a chain*⟩ 的处理。

由于本命令会把一个逗号作为参数定界符号吃掉，所以，在用户输入的 ⟨*group specification*⟩ 中，如果出现了连续的分号或者逗号，那么意味着“空链”。

\tikz@lib@graph@graph@done\pgf@stop@eodashes

当解析完毕一个链后，即解析到 \pgf@stop@eodashes 时，执行本命令。

本命令的定义是：

```
\def\tikz@lib@graph@graph@done\pgf@stop@eodashes{%
    % Get local depth and width outside
    \xdef\tikz@lib@graph@chain@depth{\pgfkeysvalueof{/tikz/graphs/placement/local
    ↪  depth}}
    \xdef\tikz@lib@graph@chain@width{\pgfkeysvalueof{/tikz/graphs/placement/local
    ↪  width}}
    % Get node list outside...
    \expandafter%
```

```
  \endgroup%
  \expandafter\expandafter\expandafter\def%
  \expandafter\expandafter\expandafter\tikz@lib@graph@node@list%
  \expandafter\expandafter\expandafter{\expandafter\tikz@lib@graph@node@list
  ↪  \tikz@lib@graph@node@list}%
  % Compute new local depth and width of group...
  \tikz@lib@graph@placement@after@chain@update
  %
  \pgfutil@ifnextchar\pgf@stop@eogroup%
  \tikz@lib@graph@graph@group@done%
  \tikz@lib@graph@main@parser%
}%
```

可见本命令做一些收尾工作，然后调用 \tikz@lib@graph@main@parser →P. 1033 再解析下一个链。

上面定义中的 \endgroup 结束由 \tikz@lib@graph@main@parser →P. 1033 开启的组，也就是说，每个链都会被放到一个 \begingroup 与 \endgroup 的组合中作处理。在这个组合结束前，会保存在这个组合中得到的 \tikz@lib@graph@node@list.

\tikz@lib@graph@graph@group@done

本命令的定义是：

```
\def\tikz@lib@graph@graph@group@done\pgf@stop@eogroup{%
  \pgfkeysvalueof{/tikz/graphs/@operators}%
}%
```

与本命令对应的组开始命令是 \tikz@lib@graph@parse@group →P. 1032.

/tikz/graphs/every group (no default)

见 \tikz@lib@graph@group@opt →P. 1033.

每当解析过程进入一个组

```
{[⟨current graph options⟩]
  ⟨current group specification⟩
}
```

后，先执行这个键，再执行选项 ⟨*current graph options*⟩，再处理 ⟨*current group specification*⟩. 这个键对当前组以及套嵌在内的组都有效。

这个键原本是没有定义的，如果希望利用这个键，应当提前定义它，例如

```
\tikzgraphsset{every group/.style=...} 或者
\tikzgraphsset{every group/.append style=...} 或者
\tikzgraphsset{every group/.code=...} 或者
\tikzgraphsset{every group/.append code=...} 或者其他
```

/tikz/graphs/parse={⟨*extra group specification*⟩} (no default)

这个键将 ⟨*extra group specification*⟩ 添加到宏 \tikz@lib@graph@parse@extras 中 (右侧)。

这个键的定义是：

```
\tikzgraphsset{
  parse/.code={\expandafter\def\expandafter\tikz@lib@graph@parse@extras
  ↪  \expandafter{\tikz@lib@graph@parse@extras#1}},
}%
```

本选项的用处见命令 \tikz@lib@graph@group@opt →P. 1033.

每当解析过程进入一个组

```
{[⟨current graph options⟩]
  ⟨current group specification⟩
}
```

后，⟨*extra group specification*⟩ 会被放到 ⟨*current group specification*⟩ 的前面，被命令 `\tikz@lib@graph@par` 处理，即

```
\expandafter\tikz@lib@graph@par\tikz@lib@graph@parse@extras⟨current group specification⟩
↪  %
```

所以 ⟨*extra group specification*⟩ 应当是某些顶点，链，组。

注意命令 `\tikz@lib@graph@group@opt` → P. 1033 会先清空宏 `\tikz@lib@graph@parse@extras`, 因此选项 `parse` 的作用仅限于当前组，对套嵌在内的组无效，并且应当放在 ⟨*current graph options*⟩ 中。

58.5 解析顶点 node 的名称，内容，选项

用户写出的顶点可以是以下形式：

- 一个花括号组
- 用 `\foreach` 命令创建顶点
- 用圆括号括起来的名称
  - `(`⟨*a set name*⟩`)[`⟨*options*⟩`]`
    ⟨*a set name*⟩ 是 (已创建的) 一个顶点集合名称
  - `(`⟨*a node name*⟩`)[`⟨*options*⟩`]`
    ⟨*a node name*⟩ 是 (已创建的) 一个顶点 node 的“全名”(见 `/tikz/graphs/name` → P. 1051, `/tikz/graphs/numbe` `nodes` → P. 1052)
  - `({`⟨*a list of set or node name*⟩`})[`⟨*options*⟩`]`
    ⟨*a list of set or node name*⟩ 是由 (已创建的) 顶点集合名称，或者顶点 node 的“全名”(见 `/tikz/graphs/name` → P. 1051, `/tikz/graphs/number nodes` → P. 1052) 构成的列表 (用逗号分隔)，注意这个列表要用花括号包裹起来，这个列表会被 `\foreach` 命令处理

  圆括号代表“引用”，即引用圆括号中的名称所指代的顶点或顶点集合。圆括号之后的 `[`⟨*options*⟩`]` 是可选的，实际上此处的 ⟨*options*⟩ 会被命令 `\tikz@lib@graph@node@opt@normal` → P. 1046 忽略，没有任何作用。
- 不以双引号、圆括号括开头的一串符号 ⟨*string*⟩`[`⟨*options*⟩`]`, 此时 ⟨*string*⟩ 既是 node 的名称，也是 node 的内容
- 不以双引号、圆括号括开头，但含有斜线的一串符号
  - ⟨*name*⟩`/`⟨*content*⟩
  - ⟨*name*⟩`/`⟨*content*⟩`[`⟨*options*⟩`]`
  - ⟨*name*⟩`/"`⟨*content*⟩`"[`⟨*options*⟩`]`
  - `/`⟨*content*⟩`[`⟨*options*⟩`]`

  此时 ⟨*name*⟩ 是 node 的名称，⟨*content*⟩ 是 node 的内容，⟨*options*⟩ 是 node 的选项
- 以双引号开头的“引号句法”，其形式为：
  - `""/"`⟨*text content*⟩`"`
  - `""/`⟨*text content*⟩
  - `"`⟨*name*⟩`"/"`⟨*text content*⟩`"`
  - `"`⟨*name*⟩`"/`⟨*text content*⟩
  - `"`⟨*name*⟩`"/"`⟨*text content*⟩`"[`⟨*options*⟩`]`
  - `"`⟨*name*⟩`"`, 等效于 `"`⟨*name*⟩`"/"`⟨*name*⟩`"`

- "⟨*name*⟩"[⟨*graph options*⟩]
- "⟨*name*⟩"__"⟨*text content*⟩", 等效于 "⟨*name*⟩"/"⟨*text content*⟩"
- "⟨*name*⟩"__"⟨*text content*⟩"[⟨*options*⟩]

注意：

1. ⟨*name*⟩ 是顶点 node 的名称，⟨*text content*⟩ 是顶点 node 的内容，⟨*options*⟩ 中列出的选项的路径可以是 /tikz/graphs 或 /tikz.
2. 在双引号句法的开头和结尾只使用 1 个双引号，不能写成 ""..."" 或 ""..."/""..."".
3. 如果需要在 ⟨*name*⟩ 或 ⟨*text content*⟩ 中使用双引号，那么应当使用连续的 2 个或偶数个双引号，并且要匹配，不能使用连续的奇数个双引号。例如可以写 "a""b""c".
4. 在 ⟨*name*⟩ 和 ⟨*text content*⟩ 中可以使用特殊符号 $, &, ^,~, _, |, [,], (,), /, ., -, ,, +, *, ', !, ", :, ;, <, =, >, ?, `, %, #, \, @, {, }. 当在 ⟨*text content*⟩ 中使用这些符号时，可能需要在符号前加上反斜线，例如“\$”.

58.5.1 初步的处理

对顶点的初步处理指的是：

- \tikz@lib@graph@par 的处理，参考步骤 5.
- \tikz@lib@graph@check@quotes → P. 1033 的处理，参考步骤 7. 经过这一步的处理后，顶点一般不会以双引号开头。

58.5.2 进一步的处理

\tikz@lib@graph@main@parser@cont@normal → P. 1034.

经过初步处理后，顶点的形式可能是：

第 1 类 \foreach 句法。

第 2 类 一个组。

第 3 类 以开圆括号 (开头的引用形式，例如

- (⟨*a set name*⟩)
- (⟨*a node name*⟩)
- ({⟨*a list of set or node name*⟩})

圆括号代表的是“引用”。

第 4 类 其他类型，

- 含有斜线 / 的字符串，如 ⟨*name*⟩/⟨*content*⟩[⟨*options*⟩], 或 ⟨*name*⟩/"⟨*content*⟩"[⟨*options*⟩].
- 不含有斜线 / 的字符串，如 ⟨*string*⟩[⟨*options*⟩].

一般情况下，以上 4 类形式都不会以双引号开头。

\tikz@lib@graph@parse@one⟨*a node type*⟩...-\pgf@stop@eodashes

这个命令的后面应当是链上的一个顶点，本命令导致这个顶点被读取：

- 如果这个顶点是一个组 {⟨*a group as node*⟩}, 则由 \tikz@lib@graph@scope → P. 1038 处理它，这个命令会调用 \tikz@lib@graph@parse@group → P. 1032.
- 如果这个顶点不是一个组那么

  \tikz@lib@graph@node → P. 1038⟨*a node type*⟩...-\pgf@stop@eodashes

本命令的定义是：

```
\def\tikz@lib@graph@parse@one{%
  \pgfutil@ifnextchar\bgroup\tikz@lib@graph@scope\tikz@lib@graph@node%
```

```
}%
```

58.5.2.1 如果顶点是一个组

假如顶点是一个组 {⟨*a group as node*⟩}, 那么由 \tikz@lib@graph@scope 处理它。

\tikz@lib@graph@scope{⟨*a group as node*⟩}...-\pgf@stop@eodashes

本命令的定义是：

```
\def\tikz@lib@graph@scope#1{
  \begingroup%
    \tikzgdeventgroupcallback{array}%
    \let\tikz@lib@graph@node@list\pgfutil@empty%
    \tikz@lib@graph@start@hint@group%
      \tikz@lib@graph@parse@group{#1}%
    \tikz@lib@graph@end@hint@group%
    \expandafter%
  \endgroup%
  \expandafter\expandafter\expandafter\def%
  \expandafter\expandafter\expandafter\tikz@lib@graph@node@list%
  \expandafter\expandafter\expandafter{\expandafter\tikz@lib@graph@node@list
  ↪  \tikz@lib@graph@node@list}%
  \tikz@lib@graph@hint@aftergroup%
  \tikz@lib@graph@stored@actions%
  \pgfutil@ifnextchar-{\tikz@lib@graph@scope@minus}{%
    \pgfutil@ifnextchar<{\tikz@lib@graph@scope@less}{%
      \tikzerror{One of the arrow types <-, --, ->, -!-, or <-> expected}%
    }%
  }%
}%
```

58.5.2.2 如果顶点不是一个组

不是组的顶点由 \tikz@lib@graph@node, \tikz@lib@graph@node@normal →P. 1039 处理。

\tikz@lib@graph@node⟨*an unbraced node*⟩...-\pgf@stop@eodashes

本命令的定义是：

```
\def\tikz@lib@graph@node{\tikz@lib@graph@check@quotes\tikz@lib@graph@node@normal}%
```

本命令处理一个顶点 node, 顶点的名称是经过 \tikz@lib@graph@check@quotes →P. 1033 处理的，按本命令的定义，本命令之后的顶点名称可以是：

- (⟨*a set name*⟩)
- (⟨*a node name*⟩)
- ({⟨*a list of set or node name*⟩})
- 一个 \foreach 语句
- 其他不被花括号包裹的顶点形式

本命令调用 \tikz@lib@graph@check@quotes →P. 1033 检查 ⟨*an unbraced node*⟩ 的第一个记号，通常这个记号不会是双引号，所以这个检查一般没什么作用，只是起到“保险”作用，即再次确保顶点名称的外围没有双引号。

所以本命令导致：

\tikz@lib@graph@node@normal →P. 1039⟨*an unbraced node*⟩...-\pgf@stop@eodashes

\tikz@lib@graph@node@normal⟨*an unbraced node*⟩...-\pgf@stop@eodashes

本命令的定义是：

```
\def\tikz@lib@graph@node@normal{%
  \pgfutil@ifnextchar({\tikz@lib@graph@node@dressed}{\tikz@lib@graph@node@naked}%
}%
```

本命令检查之后的第一个记号是否开圆括号 (, 也就是检查用户给出的顶点 ⟨*an unbraced node*⟩ 是否以圆括号开头：

- 如果以开圆括号开头，就是 (⟨*listed name*⟩), 此时导致

  \tikz@lib@graph@node@naked{(⟨*listed name*⟩)}...-\pgf@stop@eodashes

  添加花括号的作用是在后续处理中把 (⟨*listed name*⟩) 作为一整个的参数。
- 如果是不以开圆括号开头的顶点 ⟨*an unbraced unparenthesised node*⟩, 则导致

  \tikz@lib@graph@node@naked⟨*an unbraced unparenthesised node*⟩...-\pgf@stop@eodashes

\tikz@lib@graph@node@naked⟨*node spec*⟩...-\pgf@stop@eodashes

本命令的定义是：

```
\def\tikz@lib@graph@node@naked#1-{%
  % Detect trailing <
  \tikz@lib@graph@@node#1<\pgf@stop%
}%
```

本命令以 - 为参数定界符号读取一个 node. 因为在记号序列 ⟨*a chain*⟩- 中，任何一个 node 之后必有符号 -, 所以本命令读取一个 node.

本命令导致：

- 如果顶点是 (⟨*listed name*⟩), 根据此顶点之后有没有边、边的类型，可以有以下情况：
  - (⟨*listed name*⟩)--, 导致

    \tikz@lib@graph@@node(⟨*listed name*⟩)<\pgf@stop-...-\pgf@stop@eodashes

  - (⟨*listed name*⟩)->, 导致

    \tikz@lib@graph@@node(⟨*listed name*⟩)<\pgf@stop>...-\pgf@stop@eodashes

  - (⟨*listed name*⟩)-!-, 导致

    \tikz@lib@graph@@node(⟨*listed name*⟩)<\pgf@stop!-...-\pgf@stop@eodashes

  - (⟨*listed name*⟩)<-, 导致

    \tikz@lib@graph@@node(⟨*listed name*⟩)<<\pgf@stop...-\pgf@stop@eodashes

  - (⟨*listed name*⟩)<->, 导致

    \tikz@lib@graph@@node(⟨*listed name*⟩)<<\pgf@stop>...-\pgf@stop@eodashes

  - 如果之后没有边，即 (⟨*listed name*⟩) 是当前链的最后一个顶点，导致

    \tikz@lib@graph@@node(⟨*listed name*⟩)<\pgf@stop\pgf@stop@eodashes

  在以上各个情况中，如果 (⟨*listed name*⟩) 后面有 2 个 <, 那么它后面应当是 <- 或者 <->, 这在后面会导致 \tikz@lib@graph@back@arrow; 否则导致 \tikz@lib@graph@no@back@arrow, 开始对边的解析过程。
- 如果顶点是不以开圆括号开头的 ⟨*an unbraced unparenthesised node*⟩, 根据此顶点之后有没有边、边的类型，也有类似上面的结果。

\tikz@lib@graph@@node⟨*an unbraced node*⟩<...\pgf@stop...\pgf@stop@eodashes

本命令的定义是：

```
\def\tikz@lib@graph@@node#1<#2\pgf@stop%
{
  %
  % #1 will be a node (not a group)
  %
  % Syntax: node name [options]
  %
  % Grab node name
  \tikz@lib@graph@grab@name#1\pgf@stop%
  \tikz@lib@graph@stored@actions%
  \pgfutil@ifnextchar\pgf@stop@eodashes{%
    \tikz@lib@graph@graph@done%
  }{%
    %
    % Now, get arrow kind
    %
    \def\pgf@test{#2}%
    \ifx\pgf@test\pgfutil@empty%
      \expandafter\tikz@lib@graph@no@back@arrow%
    \else%
      \expandafter\tikz@lib@graph@back@arrow%
    \fi%
  }%
}%
```

其中：

9 行 \tikz@lib@graph@grab@name 会完成对顶点的解析、创建或重复利用。

10 行 此处执行宏 \tikz@lib@graph@stored@actions.

在 \tikz@lib@graph@main@parser →P. 1033 那里，在读取一个链之前，就

```
\let\tikz@lib@graph@stored@actions\pgfutil@empty%
```

然后在解析边的过程中会重定义宏 \tikz@lib@graph@stored@actions.

见 \tikz@lib@graph@after@arrow@opt →P. 1056.

12 行 解析到 \pgf@stop@eodashes 时意味着当前链的结束,此时执行 \tikz@lib@graph@graph@done →P. 1034.

17-22 行 参考 \tikz@lib@graph@node@naked →P. 1039.

\tikz@lib@graph@grab@name

本命令的定义是：

```
\def\tikz@lib@graph@grab@name{%
  \pgfutil@ifnextchar\foreach\tikz@lib@graph@do@foreach
  ↪  \tikz@lib@graph@parse@node@text%
}%
```

58.5.2.3 不以圆括号开头的非 \foreach 顶点

不以圆括号开头的非 \foreach 顶点的形式可以是：

- 含有斜线 / 的字符串，如 ⟨*name*⟩/⟨*content*⟩[⟨*options*⟩], 或 ⟨*name*⟩/"⟨*content*⟩"[⟨*options*⟩].
- 不含有斜线 / 的字符串，如 ⟨*string*⟩[⟨*options*⟩].

对于 ⟨*name*⟩/"⟨*content*⟩"[⟨*options*⟩] 这一情况的处理是：

```
\tikz@lib@graph@parse@node@text⟨name⟩/"⟨content⟩"[⟨options⟩]\pgf@stop
\tikz@lib@graph@stored@actions%
\pgfutil@ifnextchar\pgf@stop@eodashes{%
  \tikz@lib@graph@graph@done%
}{%
    %
    % Now, get arrow kind
    %
  \def\pgf@test{⟨ 此处是符号 < 或者什么也没有⟩}%
% 见 \tikz@lib@graph@node@naked → P. 1039
% 如果是符号 <, 就说明边的类型是 <- 或 <->
  \ifx\pgf@test\pgfutil@empty%
    \expandafter\tikz@lib@graph@no@back@arrow%
  \else%
    \expandafter\tikz@lib@graph@back@arrow%
  \fi%
}%
...-\pgf@stop@eodashes
```

具体步骤是：

1. 执行

   ```
   \tikz@lib@graph@parse@node@text⟨name⟩/"⟨content⟩"[⟨options⟩]\pgf@stop
   ```

   导致

   ```
   \tikz@lib@graph@fake@nodefalse
   \let\tikz@lib@graph@node@parsed\tikz@lib@graph@node@opt@normal%
   \def\tikz@lib@graph@empty@node@parsed{\tikzgdeventcallback{node}{}}%
   \tikz@lib@parse@normal@node⟨name⟩/"⟨content⟩"[⟨options⟩][\pgf@stop%
   ```

2. 执行 \tikz@lib@parse@normal@node⟨name⟩/"⟨content⟩"[⟨options⟩][\pgf@stop 导致

   ```
   \tikz@lib@parse@node@with@slash⟨name⟩/"⟨content⟩"\pgf@stop
   \tikz@lib@graph@handle@node@cont%
   ⟨options⟩][\pgf@stop
   ```

   也就是

   ```
   \pgfkeys@spdef\tikz@lib@graph@name@only{⟨name⟩}%
   \ifx\tikz@lib@graph@name@only\pgfutil@empty%
     \global\advance\tikz@fig@count by1\relax
     \edef\tikz@lib@graph@name@only{tikz@f@\the\tikz@fig@count}%
   \fi%
   \edef\tikz@lib@graph@name{\tikz@lib@graph@path\tikz@lib@graph@name@only}%
   \iftikz@handle@active@nodes%
     \def\tikz@lib@graph@node@text{\scantokens{⟨content⟩}}%
   \else
     \def\tikz@lib@graph@node@text{⟨content⟩}%
   \fi%
   % 以上 \tikz@lib@parse@node@with@slash
   \iftikzgraphsautonumbernodes%
     \tikz@lib@do@autonumber%
   \fi%
   \iftikz@lib@graph@fresh@node%
     \tikz@lg@if@local@node{\tikz@lib@graph@name}{\tikz@lg@find@fresh@name}{}%
   \fi%
   \let\tikzgraphnodeas\tikzgraphnodeas@default%
   \pgfutil@ifnextchar\pgf@stop{%
   ```

```
  \ifx\tikz@lib@graph@name@only\pgfutil@empty%
    \expandafter\tikz@lib@graph@node@empty@done%
  \else
    \expandafter\tikz@lib@graph@node@opt\expandafter[\expandafter]\expandafter[%
  \fi%
}{\tikz@lib@graph@node@opt[}%
% 以上 \tikz@lib@graph@handle@node@cont
⟨options⟩][\pgf@stop
```

3. 假设 \tikz@lib@graph@name@only 不是空的,执行 \tikz@lib@graph@node@opt[⟨options⟩][\pgf@stop 导致

```
\tikz@lib@graph@node@parsed{⟨options⟩}%
也就是
\tikz@lib@graph@node@opt@normal{⟨options⟩}%
```

4. 然后

```
\tikz@lib@graph@stored@actions%
\pgfutil@ifnextchar\pgf@stop@eodashes{%
  \tikz@lib@graph@graph@done%
}{%
    %
    % Now, get arrow kind
    %
  \def\pgf@test{⟨ 此处是符号 < 或者什么也没有⟩}%
% 见 \tikz@lib@graph@node@naked P. 1039
% 如果是符号 <, 就说明边的类型是 <- 或 <->
  \ifx\pgf@test\pgfutil@empty%
    \expandafter\tikz@lib@graph@no@back@arrow%
  \else%
    \expandafter\tikz@lib@graph@back@arrow%
  \fi%
}%
...-\pgf@stop@eodashes
```

58.5.2.4 如果顶点以圆括号开头

这一情况的处理是:

```
\tikz@lib@graph@parse@node@text(⟨listed name⟩)\pgf@stop
\tikz@lib@graph@stored@actions%
\pgfutil@ifnextchar\pgf@stop@eodashes{%
  \tikz@lib@graph@graph@done%
}{%
    %
    % Now, get arrow kind
    %
  \def\pgf@test{⟨ 此处是符号 < 或者什么也没有⟩}%
% 见 \tikz@lib@graph@node@naked P. 1039
% 如果是符号 <, 就说明边的类型是 <- 或 <->
  \ifx\pgf@test\pgfutil@empty%
    \expandafter\tikz@lib@graph@no@back@arrow%
  \else%
    \expandafter\tikz@lib@graph@back@arrow%
  \fi%
}%
```

```
...-\pgf@stop@eodashes
```

具体步骤是：

1. 执行

   ```
   \tikz@lib@graph@parse@node@text(⟨listed name⟩)\pgf@stop
   ```

   导致

   ```
   \tikz@lib@graph@fake@nodefalse
   \let\tikz@lib@graph@node@parsed\tikz@lib@graph@node@opt@normal%
   \def\tikz@lib@graph@empty@node@parsed{\tikzgdeventcallback{node}{}}%
   \tikz@lib@parse@normal@node(⟨listed name⟩)[\pgf@stop%
   ```

2. 执行 `\tikz@lib@parse@normal@node(⟨listed name⟩)[\pgf@stop` 导致

   ```
   \edef\pgf@marshal{\noexpand\pgfkeys@spdef\noexpand\tikz@lib@graph@name@only{(
   ↪  ⟨listed name⟩)}}%
   \pgf@marshal%
   \let\tikz@lib@graph@node@text\tikz@lib@graph@name@only%
   \edef\tikz@lib@graph@name{\tikz@lib@graph@path\tikz@lib@graph@name@only}%
   \tikz@lib@graph@handle@node@cont%
   \pgf@stop
   ```

3. 执行 `\tikz@lib@graph@handle@node@cont` 导致

   ```
   \iftikzgraphsautonumbernodes%
     \tikz@lib@do@autonumber%
   \fi%
   \iftikz@lib@graph@fresh@node%
     \tikz@lg@if@local@node{\tikz@lib@graph@name}{\tikz@lg@find@fresh@name}{}%
   \fi%
   \let\tikzgraphnodeas\tikzgraphnodeas@default%
   \pgfutil@ifnextchar\pgf@stop{%
     \ifx\tikz@lib@graph@name@only\pgfutil@empty%
       \expandafter\tikz@lib@graph@node@empty@done%
     \else
       \expandafter\tikz@lib@graph@node@opt\expandafter[\expandafter]\expandafter[%
     \fi%
   }{\tikz@lib@graph@node@opt[}%
   \pgf@stop
   ```

4. 执行 `\tikz@lib@graph@node@opt[][\pgf@stop` 导致

   ```
   \tikz@lib@graph@node@parsed{}%
   也就是
   \tikz@lib@graph@node@opt@normal{}%
   ```

5. 执行 `\tikz@lib@graph@node@opt@normal` → P. 1046`{}` 导致

   ```
   \def\tikz@lib@graph@use@list{⟨listed name⟩}\tikz@lib@graph@use@listtrue
   % Ok, make a list of the nodes stored in #1:
   \let\tikz@lg@temp\pgfutil@empty%
   \foreach \tikz@lg@node@name in \tikz@lib@graph@use@list {\expandafter
   ↪  \tikz@lib@graph@handle@use\expandafter{\tikz@lg@node@name}}
   % Ok, now add the nodes to the node list
   \expandafter\expandafter\expandafter\def%
     \expandafter\expandafter\expandafter\tikz@lib@graph@node@list%
       \expandafter\expandafter\expandafter{%
         \expandafter\tikz@lib@graph@node@list\tikz@lg@temp}%
   ```

```
% Then color and initialize them:
\let\tikz@lg@do\tikz@lib@graph@do@use%
\tikz@lg@temp%
```

这一步有 2 个结果：一是扩充 \tikz@lib@graph@node@list 的内容，而是恢复 ⟨*listed name*⟩ 所指代的那些顶点的初始颜色类属性。

参考 \tikz@lib@graph@handle@use→P. 1049, \tikz@lib@graph@do@use→P. 1050.

6. 然后

```
\tikz@lib@graph@stored@actions%
\pgfutil@ifnextchar\pgf@stop@eodashes{%
  \tikz@lib@graph@graph@done%
}{%
    %
    % Now, get arrow kind
    %
  \def\pgf@test{⟨ 此处是符号 < 或者什么也没有⟩}%
% 见 \tikz@lib@graph@node@naked→P. 1039
% 如果是符号 <, 就说明边的类型是 <- 或 <->
  \ifx\pgf@test\pgfutil@empty%
    \expandafter\tikz@lib@graph@no@back@arrow%
  \else%
    \expandafter\tikz@lib@graph@back@arrow%
  \fi%
}%
...-\pgf@stop@eodashes
```

\tikz@lib@graph@parse@node@text⟨*node spec*⟩\pgf@stop

本命令的定义是：

```
\def\tikz@lib@graph@parse@node@text#1\pgf@stop{%
  %
  % Ok, first test whether #1 contains "//"
  %
  \pgfutil@in@{//}{#1 }
  \ifpgfutil@in@%
    % Ok, a layout node:
    \tikz@lib@parse@layout@node#1\pgf@stop%
  \else%
    \tikz@lib@graph@fake@nodefalse
    \let\tikz@lib@graph@node@parsed\tikz@lib@graph@node@opt@normal%
    \def\tikz@lib@graph@empty@node@parsed{\tikzgdeventcallback{node}{}}%
    \tikz@lib@parse@normal@node#1[\pgf@stop%
  \fi%
}%
```

如果 ⟨*node spec*⟩ 中含有连续的 2 个斜线 //, 那么就意味着使用 graphdrawing 库的排布方法，否则使用 graphs 库的排布方法。

\tikz@lib@parse@normal@node⟨*name and(or) content*⟩[

本命令提取顶点 node 的名称和内容:

1. 检查 ⟨*name and(or) content*⟩ 中是否含有斜线 /,
   - 如果含有斜线 /, 则顶点的名称与内容是分别提供的，

     \tikz@lib@parse@node@with@slash⟨*name and(or) content*⟩\pgf@stop%

     \tikz@lib@parse@node@with@slash⟨*name*⟩/⟨*content*⟩\pgf@stop

本命令的定义形式是：

```
\def\tikz@lib@parse@node@with@slash#1/{...}
```

可见本命令读取顶点名称 ⟨*name*⟩ 和斜线 /.

本命令的处理是：

(a) 定义宏 \tikz@lib@graph@name@only 保存 ⟨*name*⟩

(b) 如果宏 \tikz@lib@graph@name@only 是空的 (等于 \pgfutil@empty), 就把计数器 \tikz@fig@count 的值全局地加 1, 把

tikz@f@⟨*\the\tikz@fig@count* 的值⟩

保存到宏 \tikz@lib@graph@name@only 中，计数器 \tikz@fig@count 的值用作顶点 node 的编号。

(c) 定义宏 \tikz@lib@graph@name 保存顶点 node 的全名，即

```
\edef\tikz@lib@graph@name{\tikz@lib@graph@path
↪  \tikz@lib@graph@name@only}%
```

其中的 \tikz@lib@graph@path 在初始之下等于 \pgfutil@empty, 这个宏由选项 /tikz/graphs/name →P. 1051 重定义。

(d) 读取后面的作为顶点 node 的内容的记号 ⟨*content*⟩, 保存在宏 \tikz@lib@graph@node@text 中。

如果 ⟨*content*⟩ 是以双引号开头的形式："⟨*cont*⟩", 那么就忽略双引号，只保存双引号引起来的内容 ⟨*cont*⟩. 同时也会检查双引号是否活动符，如果是，就定义

```
\def\tikz@lib@graph@node@text{\scantokens{⟨cont⟩}}%
```

\tikz@lib@parse@node@with@slash 的处理过程会涉及 \iftikz@handle@active@nodes, 这个 TEX-if 在《tikz.code.tex》中被声明，参考 babel 库，键 /tikz/handle active characters in code, handle active characters in nodes.

- 如果不含有斜线 /, 再检查 ⟨*name and(or) content*⟩ 中是否含有连续的 2 个下划线 __,
  - 如果含有 __, 则顶点的名称与内容是分别提供的，

```
\tikz@lib@parse@node@with@doubleunder⟨name and(or) content⟩\pgf@stop%
```

\tikz@lib@parse@node@with@doubleunder⟨*name*⟩__

本命令的定义是：

```
\def\tikz@lib@parse@node@with@doubleunder#1__{
↪  \tikz@lib@parse@node@with@slash{#1}/}%
```

本命令把 __ 换成 /, 并调用 \tikz@lib@parse@node@with@slash →P. 1044.

  - 如果不含有 __, 则顶点的名称、内容的文本都是一样的：
    - 定义宏 \tikz@lib@graph@name@only 保存 ⟨*name and(or) content*⟩, 作为顶点 node 的名称。
    - let 宏 \tikz@lib@graph@node@text 等于 \tikz@lib@graph@name@only, 作为顶点 node 的内容。
    - 定义宏 \tikz@lib@graph@name 保存顶点 node 的全名，即

```
\edef\tikz@lib@graph@name{\tikz@lib@graph@path
↪  \tikz@lib@graph@name@only}%
```

2. 执行 \tikz@lib@graph@handle@node@cont

\tikz@lib@graph@handle@node@cont

本命令的定义是：

```
\def\tikz@lib@graph@handle@node@cont{%
  \iftikzgraphsautonumbernodes%
    \tikz@lib@do@autonumber%
  \fi%
  \iftikz@lib@graph@fresh@node%
    \tikz@lg@if@local@node{\tikz@lib@graph@name}{\tikz@lg@find@fresh@name}{}%
  \fi%
  \let\tikzgraphnodeas\tikzgraphnodeas@default%
  \pgfutil@ifnextchar\pgf@stop{%
    \ifx\tikz@lib@graph@name@only\pgfutil@empty%
      \expandafter\tikz@lib@graph@node@empty@done%
    \else
      \expandafter\tikz@lib@graph@node@opt\expandafter[\expandafter]\expandafter[%
    \fi%
  }{\tikz@lib@graph@node@opt[]%
}%

\def\tikzgraphnodeas@default{%
  \tikz@lib@graph@typesetter%
}%

\def\tikz@lib@graph@node@opt[#1]#2[\pgf@stop{%
  \tikz@lib@graph@node@parsed{#1}%
}%
```

其中：

2 行 \iftikzgraphsautonumbernodes 与选项 /tikz/graphs/number nodes→P. 1052 相关。

5 行 \iftikz@lib@graph@fresh@node 与选项 /tikz/graphs/fresh nodes→P. 1053 相关。

8 行 \tikzgraphnodeas 与选项 /tikz/graphs/as→P. 1054 相关。

11 行 \tikz@lib@graph@node@empty@done 与 graphdrawing 库相关。

19 行 \tikz@lib@graph@typesetter 与选项 /tikz/graphs/typeset→P. 1054 相关。在默认下有

```
\tikzgraphnodeas=\tikzgraphnodeas@default=\tikz@lib@graph@typesetter=
↪  \tikzgraphnodetext=\tikz@lib@graph@node@text
```

宏 \tikz@lib@graph@node@text 保存顶点的内容，见 \tikz@lib@parse@normal@node→P. 1044.

23 行 在 \tikz@lib@graph@parse@node@text→P. 1044 那里，有

```
\let\tikz@lib@graph@node@parsed\tikz@lib@graph@node@opt@normal%
```

\tikz@lib@graph@node@opt@normal{⟨*options*⟩}

⟨*options*⟩ 是顶点的方括号里的选项。

本命令的定义是：

```
\def\tikz@lib@graph@node@opt@normal#1{%
  \expandafter\tikz@lib@graph@test@use@list\tikz@lib@graph@name@only\pgf@stop
  \iftikz@lib@graph@use@list%
    % Ok, make a list of the nodes stored in #1:
    \let\tikz@lg@temp\pgfutil@empty%
    \foreach \tikz@lg@node@name in \tikz@lib@graph@use@list {
    ↪  \expandafter\tikz@lib@graph@handle@use\expandafter{\tikz@lg@node@name}}
    % Ok, now add the nodes to the node list
    \expandafter\expandafter\expandafter\def%
    \expandafter\expandafter\expandafter\tikz@lib@graph@node@list%
    \expandafter\expandafter\expandafter{%
```

```
      \expandafter\tikz@lib@graph@node@list\tikz@lg@temp}%
    % Then color and initialize them:
    \let\tikz@lg@do\tikz@lib@graph@do@use%
    \tikz@lg@temp%
  \else%
    \edef\tikz@lib@graph@name{\tikz@lib@graph@path\tikz@lib@graph@name@only}%
    \expandafter\ifx\csname tikz@lib@graph@def@\tikz@lib@graph@name@only
    ↪  \endcsname\relax%
      \pgfkeysgetvalue{/tikz/graphs/placement/level}\tikz@temp%
      \c@pgf@counta=\tikz@temp\relax%
      \advance\c@pgf@counta by1\relax%
      \edef\tikz@temp{\the\c@pgf@counta}%
      \pgfkeyslet{/tikz/graphs/placement/level}\tikz@temp%
      \tikzgraphsset{
        level/.try=\pgfkeysvalueof{/tikz/graphs/placement/level},
        level \pgfkeysvalueof{/tikz/graphs/placement/level}/.try
      }
      {%
        \edef\tikz@lib@graph@node@list{\noexpand\tikz@lg@do{\tikz@lib@graph@name}}
        ↪  %
        \global\tikz@lib@graph@node@createdfalse%
        \pgfkeyslet{/tikz/graphs/@operators}\pgfutil@empty%
        \tikz@lib@activate@source@target@edge@syntax%
        \tikz@lg@if@local@node{\tikz@lib@graph@name}%
        {\tikz@lg@local@node@handle{#1}}%
        {%
          \tikz@lg@init@color{\tikz@lib@graph@name}{\tikz@lgc@all@true
          ↪  \tikz@lgc@source@true\tikz@lgc@target@true}%
          \iftikz@lib@graph@all%
            \tikzgraphsset{#1}\pgfkeysvalueof{/tikz/graphs/@operators}%
          \else%
            \pgfkeys{/tikz/graphs/placement/place}%
            \let\tikzgraphnodetext\tikz@lib@graph@node@text%
            \let\tikzgraphnodename\tikz@lib@graph@name@only%
            \let\tikzgraphnodepath\tikz@lib@graph@path%
            \let\tikzgraphnodefullname\tikz@lib@graph@name%
            \iftikz@lib@graph@fake@node%
              {% run options in protected mode...
                \pgfkeys{
                  /tikz/graphs/.cd,%
                  .unknown/.code=,
                  /tikz/graphs/@nodes styling,%
                  #1}%
                \pgfkeysgetvalue{/tikz/graphs/@operators}\tikz@lib@graph@op@save%
                \global\let\tikz@lib@graph@op@save\tikz@lib@graph@op@save%
              }%
            \else%
              \node [%
                name=\tikz@lib@graph@name,%
                execute at end node={%
                  \pgfkeysgetvalue{/tikz/graphs/@operators}\tikz@lib@graph@op@save
                  ↪  %
                  \global\let\tikz@lib@graph@op@save\tikz@lib@graph@op@save%
                },%
                graphs/redirect unknown to tikz,
                /tikz/graphs/.cd,%
```

```
                /tikz/graphs/@nodes styling,%
                #1]%
                {%
                  \tikzgraphnodeas%
                };%
            \fi%
            \tikz@lib@graph@op@save\global\let\tikz@lib@graph@op@save\relax%%
            \global\tikz@lib@graph@node@createdtrue%
          \fi
        }%
      }%
      \iftikz@lib@graph@node@created\tikz@lib@graph@placement@update\fi%
      \expandafter\expandafter\expandafter\def%
      \expandafter\expandafter\expandafter\tikz@lib@graph@node@list%
      \expandafter\expandafter\expandafter{%
        \expandafter\tikz@lib@graph@node@list\expandafter\tikz@lg@do\expandafter{
        ↪  \tikz@lib@graph@name}}%
      \iftikz@lib@graph@trie\tikzgraphsset{name=\tikz@lib@graph@name@only}\fi%
    \else
      % The name of the node is a graph name
      \tikz@lib@graph@handle@graph{#1}%
    \fi
  \fi%
}%
```

其中：

2 行 检查 `\tikz@lib@graph@name@only` 中保存的内容 (用户写出的顶点名称) 是否以开圆括号 (开头，

- 如果是，即用户写出的是 (⟨*listed name*⟩), 则导致

  ```
  \def\tikz@lib@graph@use@list{⟨listed name⟩}\tikz@lib@graph@use@listtrue
  ```

  注意，如果户写出的是 ({⟨*a list of name*⟩}), 那么宏 `\tikz@lib@graph@use@list` 保存带花括号的列表 {⟨*a list of name*⟩}.
- 如果不是，则导致

  ```
  \tikz@lib@graph@use@listfalse
  ```

17 行 检查控制序列 `\csname tikz@lib@graph@def@\tikz@lib@graph@name@only\endcsname` 是否等于 `\relax`, 这个控制序列由选项 `/tikz/graphs/declare` →P. 1119 定义。

31 行 命令 `\tikz@lib@activate@source@target@edge@syntax` 使得

- 选项 `>{`⟨*options*⟩`}` 等价于 `target edge style={`⟨*options*⟩`}`
- 选项 `>"`⟨*options*⟩`"` 等价于

  ```
  /tikz/node quotes mean={%
    target edge node={%
      node [every edge quotes,#2]{#1}
    }
  },
  /utils/exec=\tikz@enable@node@quotes,
  ⟨options⟩
  ```

- 选项 `<{`⟨*options*⟩`}` 等价于 `source edge style={`⟨*options*⟩`}`
- 选项 `<"`⟨*options*⟩`"` 等价于

```
/tikz/node quotes mean={%
  source edge node={%
    node [every edge quotes,#2]{#1}
  }
},
/utils/exec=\tikz@enable@node@quotes,
⟨options⟩
```

32 行 参考命令 \tikz@lg@if@local@node→P. 1062, 从 32 行到 72 行都属于这个命令。

如果已创建顶点 \tikz@lib@graph@name, 则执行 33 行，否则执行 34-72 行。

35 行 全局地定义控制序列 \csname tikz@lgc@\tikz@lib@graph@name\endcsname 使之保存 3 个真值

```
\tikz@lgc@all@true\tikz@lgc@source@true\tikz@lgc@target@true
```

36 行 \iftikz@lib@graph@all 的真值就是选项 /tikz/graphs/use existing nodes→P. 1054 的值。

39 行 计算顶点 node 的锚定点位置，并设置一个平移变换，针对之后的 \node 顶点。

44 行 \iftikz@lib@graph@fake@node 这个 TEX-if 与 graphdrawing 库相关。

56 行 用“全名”\tikz@lib@graph@name 给顶点 node 命名。

66 行 在默认下有

```
\tikzgraphnodeas=\tikzgraphnodeas@default=\tikz@lib@graph@typesetter=
↪  \tikzgraphnodetext=\tikz@lib@graph@node@text
```

宏 \tikz@lib@graph@node@text 保存顶点的内容，见 \tikz@lib@parse@normal@node→P. 1044.

69 行 执行 58, 59 行全局定义的 \tikz@lib@graph@op@save, 然后全局地清除它。

75-78 行 更新 (增加) 宏 \tikz@lib@graph@node@list 保存的内容，把

```
\tikz@lg@do{⟨ 彻底展开的 \tikz@lib@graph@name⟩}
```

添加到宏 \tikz@lib@graph@node@list 中 (右侧，这个宏指的是 27 行的花括号组之前的宏)。

79 行 \iftikz@lib@graph@trie 这个 TEX-if 与选项 /tikz/graphs/trie 相关。

\tikz@lib@graph@handle@use{⟨*a set name*⟩} or ⟨*a full node name*⟩

本命令的定义是：

```
\def\tikz@lib@graph@handle@use#1{%
  % Is #1 the name of a node set?
  \expandafter\let\expandafter\pgf@temp\csname tikz@lg@node@set #1\endcsname
  \ifx\pgf@temp\relax
    \pgfutil@g@addto@macro\tikz@lg@temp{\tikz@lg@do{#1}}
  \else%
    \expandafter\pgfutil@g@addto@macro\expandafter\tikz@lg@temp\expandafter{
    ↪  \pgf@temp}
  \fi
}%
```

本命令的处理是：

1. 如果本命令的参数是一个顶点集合名称 ⟨*a set name*⟩, 则将 \csname tikz@lg@node@set ⟨*a set name*⟩\endcsname 保存的内容全局地添加到 \tikz@lg@temp 中。
   关于这个控制序列保存的内容，参考 /tikz/set→P. 1058.
2. 否则，本命令的参数应当是某个顶点的全名 ⟨*a full node name*⟩, 此时将

   ```
   \tikz@lg@do{⟨a full node name⟩}
   ```

   全局地添加到 \tikz@lg@temp 中。

\tikz@lib@graph@do@use⟨*a full node name*⟩

本命令的定义是：

```
\def\tikz@lib@graph@do@use#1{%
  \tikz@lg@init@color{#1}{\tikz@lgc@all@true\tikz@lgc@source@true
  ↪  \tikz@lgc@target@true}%
}%
```

本命令使得顶点 ⟨*a full node name*⟩ 仅仅属于 all，source，target 这 3 个颜色类，即恢复顶点的初始颜色属性。

见 \tikz@lg@init@color→P. 1061.

58.5.2.5 用 \foreach 创建顶点

如果某个顶点以 \foreach 开头，那么 \tikz@lib@graph@grab@name→P. 1040 会识别它，并导致执行 \tikz@lib@graph@do@foreach.

当用户把 \foreach 语句用作顶点时，应当写出如下形式：

```
\foreach \var in {⟨list⟩}{⟨body⟩}
\foreach \var in \MacroOfList {⟨body⟩}
```

这 2 个形式都导致

```
\tikz@lib@graph@do@foreach@normal\var{⟨list⟩}{⟨body⟩}\pgf@stop
```

命令 \tikz@lib@graph@do@foreach@normal 会引入一个 \begingroup 与 \endgroup 的组合，在这个组合中使用 \foreach, \tikz@lib@graph@parse@group→P. 1032 创建顶点。

本命令的定义是：

```
\def\tikz@lib@graph@do@foreach@normal#1#2#3\pgf@stop{%
  % Ok, we do a parse on a foreach loop.
  \begingroup
    \let\tikz@lib@graph@node@list@saved\pgfutil@empty%
    \xdef\tikz@lib@graph@saved@placement{%
      {\pgfkeysvalueof{/tikz/graphs/placement/local depth}}%
      {\pgfkeysvalueof{/tikz/graphs/placement/local width}}%
      {\pgfkeysvalueof{/tikz/graphs/placement/chain count}}%
      {\pgfkeysvalueof{/tikz/graphs/placement/element count}}%
      {\pgfkeysvalueof{/tikz/graphs/placement/width}}%
      {\pgfkeysvalueof{/tikz/graphs/placement/depth}}%
    }%
    \foreach #1 in {#2}%
    {%
      \let\tikz@lib@graph@node@list\tikz@lib@graph@node@list@saved%
      \expandafter\tikz@lib@graph@setup@placement\tikz@lib@graph@saved@placement%
      \tikz@lib@graph@parse@group{#3}%
      \xdef\tikz@lib@graph@saved@placement{%
        {\pgfkeysvalueof{/tikz/graphs/placement/local depth}}%
        {\pgfkeysvalueof{/tikz/graphs/placement/local width}}%
        {\pgfkeysvalueof{/tikz/graphs/placement/chain count}}%
        {\pgfkeysvalueof{/tikz/graphs/placement/element count}}%
        {\pgfkeysvalueof{/tikz/graphs/placement/width}}%
        {\pgfkeysvalueof{/tikz/graphs/placement/depth}}%
      }%
      % TODO: Need to also save hints!
      \global\let\tikz@lib@graph@node@list@saved\tikz@lib@graph@node@list%
    }%
```

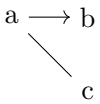
```
    \expandafter%
  \endgroup%
  \expandafter\expandafter\expandafter\def%
  \expandafter\expandafter\expandafter\tikz@lib@graph@node@list%
  \expandafter\expandafter\expandafter{\expandafter\tikz@lib@graph@node@list
  ↪  \tikz@lib@graph@node@list@saved}%
  \expandafter\tikz@lib@graph@setup@placement\tikz@lib@graph@saved@placement%
}%
```

58.5.3 顶点 node 的名称与引用

用户给出的顶点名称 ⟨*name*⟩ 是顶点的“本名”，保存在 `\tikz@lib@graph@name@only` 中。
graphs 库创建的顶点的名称是“全名”，保存在 `\tikz@lib@graph@name` 中。
“本名”与“全名”未必一样。引用已创建的顶点时，要注意使用“本名”还是“全名”。

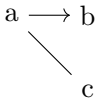
58.5.3.1 普通名称与引用

最简单的情况下，用户给出的“本名”就是“全名”。

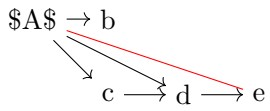



```
\tikz{
  \graph { a -> { b, c[not source] } };
  \draw (a) -- (c);
}
```

如果顶点使用了引号句法，并且引号之内含有特殊符号，那么 node 的名称会复杂一些。




```
\tikz{
  \graph { "a" -> { b, c[not source] } };
  \draw (a) -- (c);
}
```




```
\tikz{
  \graph {
    "$a$"/"\$A\$" -> { b, c -> d -> e},
    "$a$" -> d };
  \draw [red] (@DOLLAR SIGN@a@DOLLAR SIGN@) -- (e);
}
```

上面例子中，在 `\graph` 命令之内，用引号句法来引用已创建的 node 时，即使 node 名称中含有特殊符号，也可以直接使用引号句法，因为 graphs 库总能处理名称中 (不是内容中) 的特殊符号。但在 `\graph` 命令之外引用已创建的 node 时，名称中的特殊符号应当替换为相应的 Unicode 名称，参考 `\tikz@lib@graph@check@quotes` → P. 1033.

58.5.3.2 带前缀的名称与引用

顶点全名可以采用这样的结构：⟨*prefix*⟩⟨*separator*⟩⟨*name*⟩, 其中 ⟨*name*⟩ 是用户提供的本名。
这种全名保存在宏 `\tikz@lib@graph@name` 中。
本名 ⟨*name*⟩ 保存在宏 `\tikz@lib@graph@name@only` 中。

`/tikz/graphs/name=`⟨*prefix*⟩ (initially empty)

本选项设置顶点 node 的名称的前缀。

```
\let\tikz@lib@graph@path\pgfutil@empty
\tikzgraphsset{
  name separator/.store in=\tikz@lib@graph@name@separator,
  name separator=\space,
  name/.code={%
```

```
    \edef\tikz@lib@graph@path{\tikz@lib@graph@path#1\tikz@lib@graph@name@separator
    ↪ }%
  }%
}%
```

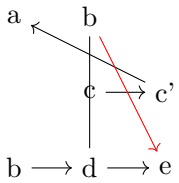
可见本选项对 \tikz@lib@graph@path 的重定义是“累积”的。

本选项的有效范围受到组的限制。

/tikz/graphs/name separator=⟨*separator*⟩ (initially \space)

本选项设置顶点 node 的“全名”中的分隔符号 (用于分隔前缀与本名)。其初始值是 \space.

注意，如果同时使用选项 name separator 和 graphs/name, 那么应当把选项 name separator 写在 graphs/name 之前，否则选项 name separator 不起作用。这是因为，按这 2 个选项的定义，选项 name separator 修改 \tikz@lib@graph@name@separator, 使之保存 ⟨*separator*⟩; 选项 graphs/name 修改 \tikz@lib@graph@path, 使之保存 ⟨*prefix*⟩\tikz@lib@graph@name@separator, 而在构造 node 的全名时，用的是宏 \tikz@lib@graph@path.




```
\tikz{
  \graph {
    a -> { [name separator=!!, name=prefix A, name=prefix B]
           b, c -> c' -> (a) },
    b -> d -> e,
    (prefix A!!prefix B!!b) ->[red] e};
  \draw (prefix A!!prefix B!!b) -- (d);
}
```

上面例子表明，选项 graphs/name 的有效范围受到组的限制，还会把 ⟨*prefix*⟩⟨*separator*⟩ 累积。

在选项 graphs/name 的有效范围内可以使用类似 (a) 这样的引用顶点 (顶点集) a 的圆括号形式。

58.5.3.3 带编号的名称与引用

顶点 node 的全名可以采用这样的结构：⟨*prefix*⟩⟨*separator*⟩⟨*name*⟩⟨*auto separator*⟩⟨*number*⟩, 其中的 ⟨*prefix*⟩, ⟨*separator*⟩, ⟨*name*⟩ 见前一种命名方式，⟨*number*⟩ 是编号。

这种方式要求 ⟨*prefix*⟩⟨*separator*⟩⟨*name*⟩ 非空，否则全名为空。

这种全名保存在宏 \tikz@lib@graph@name 中。

而 ⟨*name*⟩⟨*auto separator*⟩⟨*number*⟩ 保存在宏 \tikz@lib@graph@name@only 中。

\iftikzgraphsautonumbernodes

当这个 TEX-if 的真值为 true 时，采用这种命名方式。

/tikz/graphs/number nodes=⟨*start number*⟩ (default 1)

本选项设置 \iftikzgraphsautonumbernodes 的真值为 true, 并且将编号的起始值设为 ⟨*start number*⟩.

保存编号的是计数器 \tikz@lib@auto@number, 默认起始值是 1.

本选项的作用受到组的限制。

```
\newif\iftikzgraphsautonumbernodes
\newcount\tikz@lib@auto@number
\tikzgraphsset{number nodes/.code=%
  \pgfmathsetcount\tikz@lib@auto@number{#1}%
  \tikzgraphsautonumbernodestrue,%
  number nodes/.default=1,%
  number nodes sep/.code=\def\tikz@lib@auto@sep{#1}
}%
\def\tikz@lib@auto@sep{\space}%
```

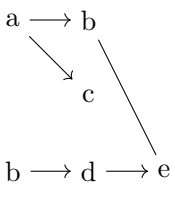
/tikz/graphs/number nodes sep=⟨*separator*⟩ (initially \space)

本选项设置顶点 node 的“全名”中的分隔符号 (用于分隔本名与编号)。其初始值是 \space.

\tikz@lib@do@autonumber

当 \iftikzgraphsautonumbernodes 的真值为 true 时，本命令被 \tikz@lib@graph@handle@node@cont →P. 1045 调用，用来构造顶点 node 的名称及编号，并全局地递增编号。

```
\def\tikz@lib@do@autonumber{%
  \ifx\tikz@lib@graph@name@only\pgfutil@empty%
  \else%
    \edef\tikz@lib@graph@name@only{\tikz@lib@graph@name@only\tikz@lib@auto@sep
    ↪  \the\tikz@lib@auto@number}%
    \global\advance\tikz@lib@auto@number by1\relax%
  \fi%
}%
```




```
\tikz{
  \graph { [number nodes=11, number nodes sep=-]
    a -> { [name separator=+, name=prefix] b, c },
    b -> d -> e,
  };
  \draw (prefix+b-12) -- (e-16);
}
```

注意，在选项 graphs/number nodes 的有效范围内不要使用类似 (a) 这样的引用顶点 (顶点集) a 的圆括号形式，否则 (a) 会被加上编号 (在 \tikz@lib@graph@handle@node@cont →P. 1045 那里)，导致错误 (在 \tikz@lib@graph@node@opt@normal →P. 1046 那里)。

58.5.3.4 fresh node

在默认下，当用户重复使用同一个顶点 node 名称时，后者将被看作是引用已创建的 node, 而不是新创建 node. 可以改变这一行为，也就是说，当用户重复使用同一个顶点 node 名称时，就会自动在重复的名称后面添加一个撇号“'”，或者加足够多的撇号，得到一个新名称，因而总是创建新 node, 是否采用这种 fresh node 行为由 \iftikz@lib@graph@fresh@node 的真值决定。

\iftikz@lib@graph@fresh@node

当这个 TEX-if 的真值为 true 时，graphs 库会采用 fresh node 行为。

/tikz/graphs/fresh nodes=true|false (initially false)

本选项决定是否采用 fresh node 行为。

本选项的作用受到组的限制。

```
\newif\iftikz@lib@graph@fresh@node
\tikzgraphsset{fresh nodes/.is if=tikz@lib@graph@fresh@node}%
```

\tikz@lg@if@local@node{⟨*node name*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

本命令检查控制序列 \tikz@lgc@⟨*node name*⟩\endcsname 是否有定义 (不等于 \relax)，如果有定义，则执行 ⟨*true code*⟩; 否则执行 ⟨*false code*⟩.

```
\def\tikz@lg@if@local@node#1#2#3{\expandafter\ifx\csname tikz@lgc@#1
↪  \endcsname\relax#3\else#2\fi}%
```

当 \iftikz@lib@graph@fresh@node 的真值为 true 时，本命令被 \tikz@lib@graph@handle@node@cont →P. 1045 调用：

```
\tikz@lg@if@local@node{\tikz@lib@graph@name}{\tikz@lg@find@fresh@name}{}%
```

这个命令的作用是检查名称为 \tikz@lib@graph@name 的 node 是否已经被创建，如果已创建，就执行 \tikz@lg@find@fresh@name.

\tikz@lg@find@fresh@name

本命令：

- 重定义 \tikz@lib@graph@name@only, \tikz@lib@graph@name, 在它们原来内容的后面添加一个撇号“'”.
- 然后调用 \tikz@lg@if@local@node 检查名称为 \tikz@lib@graph@name 的 node 是否已经被创建，如果已创建，就再次重定义 \tikz@lib@graph@name@only, \tikz@lib@graph@name, 在它们原来内容的后面添加一个撇号“'”.
- 重复以上步骤，直到加的撇号“'”足够多，得到一个新名称。

```
\def\tikz@lg@find@fresh@name{%
  \edef\tikz@lib@graph@name@only{\tikz@lib@graph@name@only'}%
  \edef\tikz@lib@graph@name{\tikz@lib@graph@path\tikz@lib@graph@name@only}%
  \tikz@lg@if@local@node{\tikz@lib@graph@name}{\tikz@lg@find@fresh@name}{}%
}%
```

58.5.3.5 非 fresh node

/tikz/graphs/use existing nodes=true|false (initially false)

本选项决定 \iftikz@lib@graph@all 的真值。

```
\tikzgraphsset{
  use existing nodes/.is if=tikz@lib@graph@all
}%
```

在 \tikz@lib@graph@alltrue 的范围内，所有的顶点，无论是否使用圆括号包裹，都被看作是已创建的，因而只是引用它们。

58.5.4 顶点 node 的内容

/tikz/graphs/as=⟨*node content*⟩

本选项的定义是：

```
\tikzgraphsset{
  as/.code=\def\tikzgraphnodeas{#1},%
}
```

/tikz/graphs/typeset=⟨*node content*⟩

本选项的定义是：

```
\tikzgraphsset{
  typeset/.store in=\tikz@lib@graph@typesetter,
  typeset=\tikzgraphnodetext
}
```

/tikz/graphs/math nodes (no value)

本选项的定义是：

```
\tikzgraphsset{
  math nodes/.style={/tikz/graphs/typeset=$\tikzgraphnodetext$},
}
```

将顶点 node 的内容放到数学模式中。

/tikz/graphs/empty nodes (no value)

本选项的定义是：

```
\tikzgraphsset{
  empty nodes/.style={/tikz/graphs/typeset=},
}
```

清空顶点 node 的内容。

- 解析顶点时，宏 \tikz@lib@graph@node@text 保存顶点的内容，见 \tikz@lib@parse@normal@node → P. 1044.
- 创建新顶点前，在一个花括号组中定义 \tikzgraphnodetext

```
{
  ...
  \let\tikzgraphnodetext\tikz@lib@graph@node@text%
  ..
  \node ...
  ...
}
```

见 \tikz@lib@graph@node@opt@normal → P. 1046.

- graphs 库自己执行

```
\tikzgraphsset{
  typeset/.store in=\tikz@lib@graph@typesetter,
  math nodes/.style={/tikz/graphs/typeset=$\tikzgraphnodetext$},
  empty nodes/.style={/tikz/graphs/typeset=},
  typeset=\tikzgraphnodetext
}%
```

导致 \def\tikz@lib@graph@typesetter{\tikzgraphnodetext}.

- graphs 库定义

```
\def\tikzgraphnodeas@default{%
  \tikz@lib@graph@typesetter%
}%
```

- 在解析顶点时

```
\let\tikzgraphnodeas\tikzgraphnodeas@default%
```

见 \tikz@lib@graph@handle@node@cont → P. 1045.

按以上定义，在默认下有

```
\tikzgraphnodeas=\tikzgraphnodeas@default=\tikz@lib@graph@typesetter=
↪  \tikzgraphnodetext=\tikz@lib@graph@node@text
```

创建新顶点的 \node 命令的内容就是 \tikzgraphnodeas. 如果希望修改顶点 node 的内容，可以参考上面的赋值顺序。

58.6 解析边

边的形式应当是，例如 ->[⟨*options*⟩].

参考 \tikz@lib@graph@node@naked → P. 1039.

1. 将边的类型保存到宏 \tikz@lib@graph@arrow@type, 例如

```
\def\tikz@lib@graph@arrow@type{<->}%
```

边的类型必须是 <->, <-, ->, --, -!- 这 5 种之一。

2. 执行 \tikz@lib@graph@after@arrow, 这导致

```
\tikz@lib@graph@after@arrow@opt[⟨options⟩]
```

又导致

(a) 修改颜色类

```
\tikzgraphinvokeoperator{recolor source by=source''}
\tikzgraphinvokeoperator{recolor target by=target'}
```

(b) 定义 \tikz@lib@graph@stored@actions

```
% Save action for next node
\expandafter\def\expandafter\tikz@lib@graph@stored@actions\expandafter{%
  \expandafter\tikz@lib@graph@joiner\expandafter{\tikz@lib@graph@arrow@type}{
  ↪ ⟨options⟩}}%
```

(c) 执行 \tikzgdeventgroupcallbackdescendants, 这与 graphdrawing 库相关。

(d) 执行 \tikz@lib@graph@parse@one, 解析下一个顶点 node.

\tikz@lib@graph@after@arrow@opt{⟨边的选项 ⟨options⟩⟩}

本命令的定义是：

```
\def\tikz@lib@graph@after@arrow@opt[#1]{%
  %
  % Ok, first recolor
  %
  \tikzgraphinvokeoperator{recolor source by=source''}
  \tikzgraphinvokeoperator{recolor target by=target'}
  % Save action for next node
  \expandafter\def\expandafter\tikz@lib@graph@stored@actions\expandafter{%
    \expandafter\tikz@lib@graph@joiner\expandafter{\tikz@lib@graph@arrow@type}{#1
    ↪ }}%
  %
  \tikzgdeventgroupcallback{descendants}%
  \tikz@lib@graph@parse@one%
}%
```

本命令：

- 参考 /tikz/graphs/color class → P. 1059 的定义，\tikz@lg@change@color → P. 1061.
  对于当前 \tikz@lib@graph@node@list 中保存的任意一个顶点，如果该顶点属于颜色类 source, 那么就让这个顶点不再属于 source, 而是属于颜色类 source''.
- 对于当前 \tikz@lib@graph@node@list 中保存的任意一个顶点，如果该顶点属于颜色类 target, 那么就让这个顶点不再属于 target, 而是属于颜色类 target'.
- 定义 \tikz@lib@graph@stored@actions, 它保存

```
\tikz@lib@graph@joiner{⟨ 当前边的类型⟩}{⟨ 边的选项 ⟨options⟩⟩}
```

这个宏的用处见 \tikz@lib@graph@@node → P. 1040, \tikz@lib@graph@scope → P. 1038.

- 执行 \tikzgdeventgroupcallbackdescendants, 这与 graphdrawing 库相关。

- 执行 \tikz@lib@graph@parse@one, 解析下一个顶点 node.

\tikz@lib@graph@joiner{⟨边的类型⟩}{⟨边的选项 ⟨options⟩⟩}

参数 ⟨边的类型⟩ 必须是 <->, <-, ->, --, -!- 这 5 种之一，或者保存其一的宏。

本命令的定义是：

```
1  \def\tikz@lib@graph@joiner#1#2{%
2    \tikzgraphinvokeoperator{recolor source by=source'}
3    \tikzgraphinvokeoperator{recolor source'' by=source}
4    {%
5      \tikz@enable@edge@quotes%
6      \pgfkeyssetvalue{/tikz/graphs/default edge kind}{#1}%
7      \pgfkeys{/tikz/graphs/.cd,@operators=,%
8        /tikz/graphs/.unknown/.code=\tikz@lib@graph@unknown@edge@option{##1},#2}%
9      \pgfkeysgetvalue{/tikz/graphs/@operators}\pgf@temp%
10     \ifx\pgf@temp\pgfutil@empty%
11       \edef\pgf@temp{\noexpand\pgfkeys{/tikz/graphs/.cd,\pgfkeysvalueof
         ↪  {/tikz/graphs/default edge operator}}}%
12       \pgf@temp%
13       \pgfkeysgetvalue{/tikz/graphs/@operators}\pgf@temp%
14     \fi%
15     \pgf@temp%
16   }%
17   \tikzgraphinvokeoperator{not source',not target'}
18 }%
19
20 \def\tikz@lib@graph@unknown@edge@option#1{%
21   \def\tikz@temp{/tikz/graphs/@edges styling/.append=}
22   \expandafter\expandafter\expandafter\pgfkeys%
23   \expandafter\expandafter\expandafter{\expandafter\tikz@temp\expandafter{
     ↪  \expandafter,\pgfkeyscurrentname={#1}}}
24 }%
```

如果执行本命令，那么其处理是：

2 行 参考 /tikz/graphs/color class →P. 1059 的定义，\tikz@lg@change@color →P. 1061.
对于当前 \tikz@lib@graph@node@list 中保存的任意一个顶点，如果该顶点属于颜色类 source, 那么就让这个顶点不再属于 source, 而是属于颜色类 source'.

3 行 对于当前 \tikz@lib@graph@node@list 中保存的任意一个顶点，如果该顶点属于颜色类 source'', 那么就让这个顶点不再属于 source'', 而是属于颜色类 source.

4 行 用 { 开启一个组

5 行 \tikz@enable@edge@quotes 是 quotes 库的命令，它激活使用选项的引号句法。

6 行 设置选项 graphs/default edge kind 的值为 ⟨边的类型⟩.

7 行 清空 @operators 保存的内容。

8 行 /tikz/graphs/.unknown/.code=\tikz@lib@graph@unknown@edge@option{#1} 的作用是：如果 ⟨边的选项 ⟨options⟩⟩ 中的某个键值对不能被识别 (通常这就是路径为 /tikz 的选项，将被用作 edge 操作的选项)，就把这个键值对放到 (添加到) 键 /tikz/graphs/@edges styling 中。

8 行 执行 ⟨边的选项 ⟨options⟩⟩, 其中的选项可能会改变 @operators 保存的内容。

9 行 将 @operators 保存的代码复制到 \pgf@temp 中。

10-14 行 如果 \pgf@temp 是空的，就执行选项 /tikz/graphs/default edge operator →P. 1077 保存的连接器 (connector, 一个 style)，从而为键 /tikz/graphs/@operators →P. 1066 赋值，然后把 @operators 保存的代码复制到 \pgf@temp 中。

15 行 执行 \pgf@temp. 也就是说，如果用户没有指定 @operators 保存的代码，就执行由

/tikz/graphs/default edge operator→P. 1077 指定的连接器 (一个 style) 所对应的、将要保存到键 @operators 中的代码。

参考 /tikz/graphs/matching and star→P. 1081.

如果是画多重图，见 /tikz/graphs/multi→P. 1076, 那么这一步就会在两个顶点之间创建边。

如果是画简单图，见 /tikz/graphs/simple→P. 1076, 那么这一步只会保存创建这个边的代码。

16 行 用 } 结束组。

17 行 使得当前 \tikz@lib@graph@node@list 中保存的任意一个顶点既不属于颜色类 source', 也不属于颜色类 target'.

简单地说，本命令的作用是：对于当前 \tikz@lib@graph@node@list 中保存的那些顶点，先修改它们的颜色类属性，再调用 @operators 中的代码来处理这些顶点之间的边，或者创建边，或者保存创建边的代码。然后再次修改那些顶点的颜色类属性。

58.7 顶点集合

一个顶点集合 ⟨*a set name*⟩ 就是一个控制序列 \csname tikz@lg@node@set ⟨*a set name*⟩\endcsname, 它保存的内容是形如

```
\tikz@lg@do{顶点 node 的全名}
```

的一系列记号。

/tikz/new set=⟨*a set name*⟩

注意这个键的路径是 /tikz. 本选项声明一个新的顶点集合名称，即 let 控制序列

\csname tikz@lg@node@set ⟨*a set name*⟩\endcsname

等于 \pgfutil@empty.

```
\tikzset{
  new set/.code={
    \expandafter\global\expandafter\let\csname tikz@lg@node@set #1
    ↪  \endcsname\pgfutil@empty%
  },
}
```

/tikz/set=⟨*a set name*⟩

注意这个键的路径是 /tikz.

这个键用作某个“新顶点”的选项，使得该顶点属于集合 ⟨*a set name*⟩.

本选项的定义是：

```
\tikzset{
  set/.code={
    \tikz@fig@mustbenamed%
    \ifcsname tikz@lg@node@set #1\endcsname\else
      \tikzerror{Undefined set `#1'}%
    \fi
    \expandafter\def\expandafter\tikz@alias\expandafter{\tikz@alias%
      \expandafter\def\expandafter\pgf@temp\expandafter{\csname tikz@lg@node@set
      ↪  #1\endcsname}%
      \expandafter\expandafter\expandafter\pgfutil@g@addto@macro
      ↪  \expandafter\pgf@temp\expandafter{\expandafter\tikz@lg@do\expandafter{
      ↪  \tikz@fig@name}}%
    }%
  },%
```

```
}
```

其中，命令 \tikz@fig@mustbenamed→P. 842 确保当前正在创将的新顶点有名称；本选项定义宏 \tikz@alias，这个宏会在 (创建 node 的过程中) 的命令 \tikz@node@finish→P. 845 之前被执行。

注意，如果同时使用 /tikz/alias→P. 804 以及本选项，那么本选项必须用在 /tikz/alias→P. 804 之后，否则无效。

如果给一个顶点 node 多次使用本选项，那么最后一个有正面效果。也就是说，一个顶点只能属于一个集合。

执行宏 \tikz@alias 的结果是：

1. 按 /tikz/alias→P. 804 的设置，给顶点 node 安排别名。
2. 将

   ```
   \tikz@lg@do{顶点 node 的全名}
   ```

   全局地添加到控制序列 \csname tikz@lg@node@set ⟨*a set name*⟩\endcsname 中。

58.8 颜色类

颜色类用来对顶点作分类，在画边时可能需要考虑在哪两个颜色类之间画边。

一个顶点可以属于 0 个或数个颜色类。名称为 ⟨顶点 *node* 的全名⟩ 的顶点属于哪个 (哪些) 颜色类，由控制序列

\csname tikz@lgc@⟨ 顶点 *node* 的全名⟩\endcsname

保存的内容决定，这个控制序列总是被全局地定义、全局地修改的。

/tikz/graphs/color class=⟨*a color class name*⟩

本选项被定义为一个样式：

```
\tikzgraphsset{
  color class/.style={%
    /utils/exec=\expandafter\tikz@lg@newif\csname iftikz@lgc@#1@\endcsname,
    #1/.style={operator={%
      \edef\tikz@lg@col{\expandafter\noexpand\csname tikz@lgc@#1@true\endcsname}%
      \let\tikz@lg@do\tikz@lg@colorize%
      \tikz@lib@graph@node@list%
    }},
    not #1/.style={operator={%
      \edef\tikz@lg@old@col{\expandafter\noexpand\csname tikz@lgc@#1@true
      ↪  \endcsname}%
      \let\tikz@lg@new@col\pgfutil@empty%
      \let\tikz@lg@do\tikz@lg@change@color%
      \tikz@lib@graph@node@list%
    }},
    recolor #1 by/.style={operator={%
      \edef\tikz@lg@old@col{\expandafter\noexpand\csname tikz@lgc@#1@true
      ↪  \endcsname}%
      \edef\tikz@lg@new@col{\expandafter\noexpand\csname tikz@lgc@##1@true
      ↪  \endcsname}%
      \let\tikz@lg@do\tikz@lg@change@color%
      \tikz@lib@graph@node@list%
    }},
    !#1/.style=not #1,
  },
}
```

执行 /tikz/graphs/color class=⟨*a color class name*⟩ 导致：

1. 将控制序列 \csname iftikz@lgc@⟨*a color class name*⟩@\endcsname 声明为一个 TEX-if, 它代表这个颜色类 ⟨*a color class name*⟩, 它的 2 个真值是

```
\csname tikz@lgc@⟨a color class name⟩@true\endcsname
\csname tikz@lgc@⟨a color class name⟩@false\endcsname
```

2. 定义样式 /tikz/graphs/⟨*a color class name*⟩,

   /tikz/graphs/⟨*a color class name*⟩

   执行这个样式会导致 operator 被执行，向 @operators 中添加代码。如果执行这些代码，那么就会使得 \tikz@lib@graph@node@list 中保存的那些顶点都属于颜色类 ⟨*a color class name*⟩.

3. 定义样式 /tikz/graphs/not ⟨*a color class name*⟩,

   /tikz/graphs/not ⟨*a color class name*⟩

   执行这个样式会导致 operator 被执行，向 @operators 中添加代码。如果执行这些代码，那么就会使得 \tikz@lib@graph@node@list 中保存的那些顶点都不属于颜色类 ⟨*a color class name*⟩.

4. 定义样式 /tikz/graphs/recolor ⟨*a color class name*⟩ by,

   /tikz/graphs/recolor ⟨*a color class name*⟩ by=⟨*a new color class name*⟩

   执行这个样式会导致 operator 被执行，向 @operators 中添加代码。如果执行这些代码，那么就会使得 \tikz@lib@graph@node@list 中保存的那些顶点，原来属于 ⟨*a color class name*⟩ 的都属于 ⟨*a new color class name*⟩, 而不再属于 ⟨*a color class name*⟩.

5. 定义样式 /tikz/graphs/!⟨*a color class name*⟩,

   /tikz/graphs/!⟨*a color class name*⟩

   等效于 not ⟨*a color class name*⟩.

文件《tikzlibrarygraphs.code.tex》中定义了 6 个颜色类：

```
\tikzgraphsset{
  color class=source,
  color class=source',
  color class=source'',
  color class=target,
  color class=target',
  color class=all
}%
```

控制序列 \csname tikz@lgc@⟨顶点 *node* 的全名⟩\endcsname 保存着 0 个或者数个颜色类的真值。例如命令 \tikz@lib@graph@node@opt@normal →P. 1046 在创建顶点时，会执行 \tikz@lg@init@color →P. 1061 来定义这个控制序列，使之保存 3 个颜色类真值：

```
\tikz@lgc@all@true\tikz@lgc@source@true\tikz@lgc@target@true
```

一个顶点可以属于多个颜色类。令某个顶点 ⟨顶点 *node* 的全名⟩ 属于颜色类 ⟨*a color class name*⟩, 就是把

\csname iftikz@lgc@⟨*a color class name*⟩@\endcsname

的 true 真值，即 \csname tikz@lgc@⟨*a color class name*⟩@true\endcsname, (全局地) 保存到 (添加到) 控制序列

\csname tikz@lgc@⟨ 顶点 *node* 的全名⟩\endcsname

中。

将某个顶点 ⟨顶点 *node* 的全名⟩ 从颜色类 ⟨*a color class name*⟩ 中排除，就是把控制序列

\csname tikz@lgc@⟨ 顶点 *node* 的全名⟩\endcsname

中的真值 \csname tikz@lgc@⟨*a color class name*⟩@true\endcsname (全局地) 替换为 \pgfutil@empty.

通常，控制序列 \csname tikz@lgc@⟨顶点 *node* 的全名⟩\endcsname 总是被全局定义的。

\tikz@lg@init@color{⟨顶点 *node* 的全名⟩}{⟨某个 *(某些)* 颜色类真值⟩}

这个命令“全局地”定义控制序列 \csname tikz@lgc@⟨顶点 *node* 的全名⟩\endcsname 的初始值，即使之保存 ⟨某个 *(某些)* 颜色类真值⟩.

命令 \tikz@lib@graph@node@opt@normal →P. 1046 在创建顶点时，会调用本命令来定义这个控制序列，使之保存 3 个颜色类真值：

```
\tikz@lgc@all@true\tikz@lgc@source@true\tikz@lgc@target@true
```

也就是说，在初始值下，新创建的顶点属于 all, source, target 这 3 个颜色类。

本命令的定义是：

```
\def\tikz@lg@init@color#1#2{%
  \expandafter\gdef\csname tikz@lgc@#1\endcsname{#2}%
}%
```

\tikz@lib@graph@cleanup{⟨顶点 *node* 的全名⟩}

本命令全局地 let 控制序列

\csname tikz@lgc@⟨ 顶点 *node* 的全名⟩\endcsname
\csname tikz@lgca@@⟨ 顶点 *node* 的全名⟩\endcsname
\csname tikz@lgcb@@⟨ 顶点 *node* 的全名⟩\endcsname

等于 \relax, 使得 ⟨顶点 *node* 的全名⟩ 不属于任何颜色类。

参考 \tikz@lib@annotate@ →P. 1072, /tikz/graphs/source edge style →P. 1072.

在 \tikz@lib@graph@main@done 之前会调用这个命令，对所有顶点作清理。

\tikz@lg@colorize{⟨顶点 *node* 的全名⟩}

使用本命令前需要先定义：

```
\edef\tikz@lg@col{\expandafter\noexpand\csname tikz@lgc@⟨a color class name⟩@true
↪  \endcsname}%
```

本命令全局地把 \csname tikz@lgc@⟨*a color class name*⟩@true\endcsname 添加到控制序列

\csname tikz@lgc@⟨ 顶点 *node* 的全名⟩\endcsname

中，使得 ⟨顶点 *node* 的全名⟩ 属于颜色类 ⟨*a color class name*⟩.

\tikz@lg@change@color{⟨顶点 *node* 的全名⟩}

使用本命令前需要先定义 \tikz@lg@old@col:

```
\edef\tikz@lg@old@col{\expandafter\noexpand\csname tikz@lgc@
↪  ⟨a old color class name⟩@true\endcsname}%
```

也需要提前定义 \tikz@lg@new@col.

本命令把控制序列 \csname tikz@lgc@⟨顶点 *node* 的全名⟩\endcsname 中的真值

\csname tikz@lgc@⟨*a old color class name*⟩@true\endcsname

(全局地) 替换为 \tikz@lg@new@col 的值：

- 如果定义 \tikz@lg@new@col 为

```
\edef\tikz@lg@new@col{\expandafter\noexpand\csname tikz@lgc@
↪ ⟨a new color class name⟩@true\endcsname}%
```

那么会使得 ⟨顶点 *node* 的全名⟩ 不属于颜色类 ⟨*a old color class name*⟩, 而属于颜色类 ⟨*a new color class name*⟩.

- 如果定义 \tikz@lg@new@col 为

```
\let\tikz@lg@new@col\pgfutil@empty%
```

那么会使得 ⟨顶点 *node* 的全名⟩ 不属于颜色类 ⟨*a old color class name*⟩.

\tikz@lg@if@has@color{⟨顶点 *node* 的全名⟩}{⟨*a color class name*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

本命令检查 ⟨顶点 *node* 的全名⟩ 是否属于颜色类 ⟨*a color class name*⟩, 如果属于，就执行 ⟨*true code*⟩; 否则执行 ⟨*false code*⟩.

\tikz@lg@if@local@node{⟨顶点 *node* 的全名⟩}{⟨*true code*⟩}{⟨*false code*⟩}

本命令检查控制序列 \csname tikz@lgc@⟨顶点 *node* 的全名⟩\endcsname 是否不等于 \relax, 也就是检查 ⟨顶点 *node* 的全名⟩ 是否属于某一个颜色类，如果属于某一个颜色类，就执行 ⟨*true code*⟩; 否则执行 ⟨*false code*⟩.

58.9 \tikz@lib@graph@node@list

宏 \tikz@lib@graph@node@list 的内容如下：

- 每当创建 (处理) 一个 (不以圆括号开头的) 顶点时，将该顶点放到一个花括号组中处理

```
% 此处的 \tikz@lib@graph@node@list
{
  \edef\tikz@lib@graph@node@list{\noexpand\tikz@lg@do{\tikz@lib@graph@name}}%
  处理一个顶点
}
```

在这个花括号组结束时，把组内的 \tikz@lib@graph@node@list 的内容，即

```
\tikz@lg@do{⟨ 彻底展开的 \tikz@lib@graph@name, 即顶点 node 的全名⟩}
```

添加到组之前的宏 \tikz@lib@graph@node@list 中 (右侧)。见 \tikz@lib@graph@node@opt@normal →P. 1046.

- 每当处理一个以圆括号开头的顶点 (⟨*listed name*⟩) (圆括号代表引用已创建的顶点) 后，被 ⟨*listed name*⟩ 引用的那些顶点 ⟨*full name of cited node*⟩, 即记号序列

```
\tikz@lg@do{⟨full name of cited node⟩}
```

会被添加到 \tikz@lib@graph@node@list 中。

见 \tikz@lib@graph@node@opt@normal →P. 1046, /tikz/set →P. 1058, \tikz@lib@graph@handle@use →P. 1049, \tikz@lib@graph@do@use →P. 1050.

- 每个链都会被放到一个组合中处理

```
% 此处的 \tikz@lib@graph@node@list
\begingroup
  处理一个链，得到 \tikz@lib@graph@node@list
\endgroup
```

对于这个组合：在 \begingroup 之后，会 \let\tikz@lib@graph@node@list\pgfutil@empty, 见 \tikz@lib@graph@main@parser →P. 1033.

在 \endgroup 结束这个组合前,把在这个组合中得到的 \tikz@lib@graph@node@list 添加到 \begingroup 之前的 \tikz@lib@graph@node@list 中 (右侧)。见 \tikz@lib@graph@graph@done → P. 1034.

- 每一个链组都会被放到一个组合中处理

```
% 此处的 \tikz@lib@graph@node@list
\begingroup
  处理一个链组，得到 \tikz@lib@graph@node@list
\endgroup
```

对于这个组合：在 \begingroup 之后，会 \let\tikz@lib@graph@node@list\pgfutil@empty，见 \tikz@lib@graphs@normal@main, \tikz@lib@graph@scope → P. 1038.

在 \endgroup 结束这个组合前,把在这个组合中得到的 \tikz@lib@graph@node@list 添加到 \begingroup 之前的 \tikz@lib@graph@node@list 中 (右侧)。见 \tikz@lib@graph@scope → P. 1038.

- 每一个子图都会被放到一个组合中处理

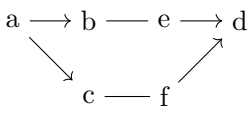
```
% 此处的 \tikz@lib@graph@node@list
\begingroup
  处理一个子图，得到 \tikz@lib@graph@node@list
\endgroup
```

此时对 \tikz@lib@graph@node@list 的处理也类似以上。

- 各处对 \tikz@lib@graph@node@list 的定义都不是全局地。
- 在执行 \tikz@lib@graph@main@done 之前，\tikz@lib@graph@node@list 中保存了各个顶点的某些信息。

  在执行 \tikz@lib@graph@main@done 之后，宏 \tikz@lib@graph@node@list 是没有定义的，因为它被限制在了组中。

按上述,在创建图的过程中,\tikz@lib@graph@node@list 的内容处于变化之中。如果某个选项或者命令是针对 (或者利用) \tikz@lib@graph@node@list 进行操作的,那么一定要注意当前的 \tikz@lib@graph@node@list 保存的是什么内容 (哪些顶点)。



macro:->\tikz@lg@do {a}\tikz@lg@do {PREFIX b}\tikz@lg@do {PREFIX c}\tikz@lg@do {e 1}\tikz@lg@do {f 2}\tikz@lg@do {d}\tikz@lg@do {x}\tikz@lg@do {y}

```
\makeatletter
\expandafter\def\expandafter\aaaa\expandafter{%
  \tikz@lib@graph@main@done}%
\def\bbbb{%
  \expandafter\gdef\expandafter\cccc\expandafter{%
    \tikz@lib@graph@node@list}}
\expandafter\expandafter\expandafter\def%
  \expandafter\expandafter\expandafter\tikz@lib@graph@main@done%
    \expandafter\expandafter\expandafter{%
  \expandafter\bbbb\aaaa}
\makeatother

\tikz{
```

```
  \graph {
    a -> {[name=PREFIX] b, c } -- {[number nodes] e, f } -> d,
    x -- y
  };
}

\ttfamily
\meaning\cccc
```

\tikz@lib@graph@node@list 的用处是对其中保存的各个 ⟨顶点 *node* 的全名⟩ 作某种操作:将 \tikz@lg@do let 为某个命令 \⟨*cmd*⟩, 这里的 \⟨*cmd*⟩ 是以 ⟨顶点 *node* 的全名⟩ 这个名称为参数的命令，然后再执行 \tikz@lib@graph@node@list, 从而实现操作。

\tikz@lib@graph@pack@node@list

本命令去掉 \tikz@lib@graph@node@list 中重复的

```
\tikz@lg@do{⟨ 顶点 node 的全名⟩}
```

具体操作是:

1. 用 { 开启一个组
2. 令

   ```
   \let\tikz@lg@packed\pgfutil@empty%
   \let\tikz@lg@do\tikz@lg@packer%
   ```

   命令 \tikz@lg@packer 能处理一个参数，即 ⟨顶点 *node* 的全名⟩。
3. 执行 \tikz@lib@graph@node@list, 由于 \tikz@lg@packer 的定义是

   ```
   \def\tikz@lg@packer#1{%
     \expandafter\ifx\csname tikz@lg@p@#1\endcsname\pgf@stop%
     \else%
       \expandafter\let\csname tikz@lg@p@#1\endcsname\pgf@stop%
       \expandafter\def\expandafter\tikz@lg@packed\expandafter{\tikz@lg@packed
       ↪  \tikz@lg@do{#1}}
     \fi
   }%
   ```

   所以结果就是将 \tikz@lib@graph@node@list 中的内容转存到 \tikz@lg@packed 中，在转存的同时去掉重复的

   ```
   \tikz@lg@do{⟨ 顶点 node 的全名⟩}
   ```

   在这个过程中利用 \csname tikz@lg@p@⟨顶点全名⟩\endcsname 检查重复。
4. 用 } 结束之前开启的组,结束组前,用 \tikz@lg@packed 的内容替换 \tikz@lib@graph@node@list 内容，并将 \tikz@lib@graph@node@list 的定义推到组外。

\tikzgraphforeachcolorednode{⟨*a color class name*⟩}\⟨*cmd*⟩

参数 ⟨*a color class name*⟩ 是一个有效的颜色类名称，或者返回颜色类名称的宏、命令。

参数 \⟨*cmd*⟩ 是能处理一个参数，即 ⟨顶点 *node* 的全名⟩ 的命令。

本命令检查 \tikz@lib@graph@node@list 中保存的那些 ⟨顶点 *node* 的全名⟩, 如果 ⟨顶点 *node* 的全名⟩ 属于颜色类 ⟨*a color class name*⟩, 则执行 \⟨*cmd*⟩{⟨顶点 *node* 的全名⟩}.

本命令的具体处理是:

1. 执行 \tikz@lib@graph@pack@node@list, 去掉 \tikz@lib@graph@node@list 中重复的

   ```
   \tikz@lg@do{⟨ 顶点 node 的全名⟩}
   ```
2. 定义

```
\expandafter\def\expandafter\iftikz@lib@graph@color@picker\expandafter{\csname
↪  iftikz@lgc@⟨a color class name⟩@\endcsname}%
\let\tikz@lib@graph@action\⟨cmd⟩%
\let\tikz@lg@do\tikz@lg@pick%
```

命令 \tikz@lg@pick 能处理一个参数，即 ⟨顶点 *node* 的全名⟩。

3. 执行 \tikz@lib@graph@node@list, 命令 \tikz@lg@pick 的定义是

```
\def\tikz@lg@pick#1{
  {%
    \csname tikz@lgc@#1\endcsname%
    \iftikz@lib@graph@color@picker
      \global\tikz@color@testtrue%
    \else%
      \global\tikz@color@testfalse%
    \fi%
  }%
  \iftikz@color@test\tikz@lib@graph@action{#1}\fi%
}%
```

其中 \csname tikz@lgc@⟨顶点 *node* 的全名⟩\endcsname 是在创建 ⟨顶点 *node* 的全名⟩ 这个顶点 node 时定义的控制序列，在最初定义它时，它保存 3 个真值

```
\tikz@lgc@all@true\tikz@lgc@source@true\tikz@lgc@target@true
```

但此时它可能还保存了其它的真值。

对 \tikz@lib@graph@node@list 中保存的那些 ⟨顶点 *node* 的全名⟩ 来说，这一步的处理是：

- 如果在 \csname tikz@lgc@⟨顶点 *node* 的全名⟩\endcsname 中保存了

  \csname iftikz@lgc@⟨*a color class name*⟩@\endcsname

  的 true 真值，也就是 ⟨顶点 *node* 的全名⟩ 属于颜色类 ⟨*a color class name*⟩, 则执行 \⟨*cmd*⟩{⟨顶点 *node* 的全名⟩}.
- 如果不是上一情况，则什么也不做。

\tikzgraphpreparecolor{⟨*a color class name*⟩}\⟨*counter*⟩{⟨*prefix*⟩}

参数 ⟨*a color class name*⟩ 是一个有效的颜色类名称。

参数 \⟨*counter*⟩ 是一个已声明的计数器名称。

参数 ⟨*prefix*⟩ 是一串符号。

本命令调用 \tikzgraphforeachcolorednode →P. 1064, 其处理是：

1. 定义

```
\let\tikz@lib@graph@count\⟨counter⟩%
\tikz@lib@graph@count0\relax% 注意归 0
\def\tikz@lib@graph@prefix{⟨prefix⟩}%
```

2. 调用 \tikzgraphforeachcolorednode →P. 1064, 效果是：
   (a) 执行 \tikz@lib@graph@pack@node@list →P. 1064, 去掉 \tikz@lib@graph@node@list 中重复的

   ```
   \tikz@lg@do{⟨ 顶点 node 的全名⟩}
   ```

   (b) 对 \tikz@lib@graph@node@list 中保存的那些 ⟨顶点 *node* 的全名⟩ 来说，如果 ⟨顶点 *node* 的全名⟩ 属于颜色类 ⟨*a color class name*⟩, 则有以下 2 个操作：

   ```
   \advance\tikz@lib@graph@count by1\relax%
   \expandafter\def\csname\tikz@lib@graph@prefix\the\tikz@lib@graph@count
   ↪  \endcsname{⟨ 顶点 node 的全名⟩}%
   ```

本命令没有用组限制以上 2 个操作。使用本命令后，计数器 \tikz@lib@graph@count 或 \⟨*counter*⟩ 的值就是那些属于颜色类 ⟨*a color class name*⟩ 的顶点的总数；可以用控制序列

\csname ⟨*prefix*⟩⟨ 编号⟩\endcsname

获取对应的顶点名称，注意 ⟨编号⟩ 从 1 开始。

58.10 operator

/tikz/graphs/@operators={⟨*code*⟩}

键 /tikz/graphs/@operators 保存某些代码，它是 graphs 库的内部处理过程使用的键。

/tikz/graphs/operator={⟨*code*⟩}

有定义：

```
\tikzgraphsset{
  @operators/.initial=,
  operator/.style={/tikz/graphs/@operators/.append={#1}},
}
```

键 /tikz/graphs/operator 向键 /tikz/graphs/@operators 中添加代码 (右侧)。

键 /tikz/graphs/operator 与 /tikz/graphs/@operators 中可以保存任何代码，不过它们的主要用处是：决定对 \tikz@lib@graph@node@list 中保存的各个顶点或者顶点之间的边采取什么操作。

1. 命令 \graph[⟨*graph options*⟩]{⟨*group specification*⟩}; 中，处理选项 ⟨*graph options*⟩ 时，先执行 @operators= 清空它保存的内容，然后再执行选项 ⟨*graph options*⟩. 用户可以在 ⟨*graph options*⟩ 中修改 @operators 保存的代码。

   然后将 @operators 保存的代码复制到宏 \tikz@lib@graph@outer@operators 中，之后才会解析 ⟨*group specification*⟩.

   在处理完毕 ⟨*group specification*⟩ 后，会执行宏 \tikz@lib@graph@outer@operators 中的代码。

2. 每当 \tikz@lib@graph@group@opt → P. 1033 处理一个组

   ```
   { [⟨options⟩] ... }
   ```

   时，会先执行 @operators= 清空它保存的内容，再执行组的选项 ⟨*options*⟩ (这可能会修改 @operators 保存的内容)。

   在处理完毕一个组后，\tikz@lib@graph@graph@group@done → P. 1035 会执行 @operators 保存的代码。

3. 每当创建一个新顶点前，会先

   ```
   \pgfkeyslet{/tikz/graphs/@operators}\pgfutil@empty%
   ```

   清空 @operators 保存的代码。

   然后执行顶点的选项，这可能重定义 @operators 保存的内容。

   然后将 @operators 保存的代码全局地复制到 \tikz@lib@graph@op@save 中。

   创建顶点后，再执行 \tikz@lib@graph@op@save (即 @operators 保存的代码)。

   然后令 \tikz@lib@graph@op@save 全局地等于 \relax.

   见 \tikz@lib@graph@node@opt@normal → P. 1046.

4. 与颜色类有关的键会利用 operator 来修改 @operators 保存的代码。

5. 各种连接算子 (connector) 会利用 operator.

6. 与边有关的选项可能会利用 operator.

7. 有的选项可能会利用 operator.

8. 各处对 operator 的赋值都应当是“局部地”。

在创建图的过程中，键 @operators 保存的代码是不断变化的，它的变化也不是全局地，因此组对 @operators 的内容有影响。如果在一个新顶点的选项中向 @operators 中添加某些代码，那么在创建这个顶点后，这些代码会被执行。如果在一个链组的选项中向 @operators 中添加某些代码，那么在这个链组结束时，这些代码会被执行。

\tikzgraphinvokeoperator{⟨*graph options*⟩}

参数 ⟨*graph options*⟩ 中的选项是路径为 /tikz/graphs 的键。
本命令在一个组内操作，先清空 @operators 保存的代码，再执行 ⟨*graph options*⟩。如果 ⟨*graph options*⟩ 向 /tikz/graphs/@operators 中添加了代码，还会在组后执行所添加的代码。
本命令的定义是：

```
\def\tikzgraphinvokeoperator#1{%
  {%
    \pgfkeyslet{/tikz/graphs/@operators}\pgfutil@empty%
    \pgfkeys{/tikz/graphs/.cd,#1}%
    \pgfkeysgetvalue{/tikz/graphs/@operators}\tikz@lib@graph@temp%
    \global\let\tikz@lib@graph@temp\tikz@lib@graph@temp%
  }%
  \tikz@lib@graph@temp%
  \global\let\tikz@lib@graph@temp\relax%
}%
```

58.11 与顶点有关的选项

/tikz/graphs/@nodes styling (style, initially empty)

这个样式是 graphs 库的内部处理过程使用的键。它的初始值是空的。

```
\tikzgraphsset{
  @nodes styling/.style=,
  nodes/.style={/tikz/graphs/@nodes styling/.append style={,#1}},
}
```

/tikz/graphs/nodes={⟨*options*⟩} (style)

这个键向样式 @nodes styling 中添加选项 (在右侧累积)，这些选项用于顶点。

```
\tikzgraphsset{
  @nodes styling/.style=,
  nodes/.style={/tikz/graphs/@nodes styling/.append style={,#1}},
}
```

58.12 与边有关的选项

58.12.1 边的类型

/tikz/graphs/new ->={⟨full name of edge left node⟩}{⟨full name of edge right node⟩}
{⟨edge options⟩}{⟨edge node spec⟩}

本选项的定义是：

```
\tikzgraphsset{
  new ->/.code n args={4}{%
    \path [->,every new ->/.try]
      (#1\tikzgraphleftanchor)
      edge[#3] #4
      (#2\tikzgraphrightanchor);},
}
```

本选项保存一个 \path 命令，用 edge 操作创建路径。
这个路径带有箭头选项 -> 并使用键 every new ->.

/tikz/every new ->=⟨something⟩

注意这个键的路径是 /tikz. 这个键是没有定义的，如果希望它起到某种作用，应当提前定义它，例如：

```
\tikzset{every new ->/.style=...} 或者
\tikzset{every new ->/.append style=...} 或者
\tikzset{every new ->/.code=...} 或者
\tikzset{every new ->/.append code=...} 或者其他
```

/tikz/graphs/new --={⟨full name of edge left node⟩}{⟨full name of edge right node⟩}
{⟨edge options⟩}{⟨edge node spec⟩}

本选项的定义是：

```
\tikzgraphsset{
  new --/.code n args={4}{
      \path [-,every new --/.try]
      (#1\tikzgraphleftanchor)
      edge[#3] #4
      (#2\tikzgraphrightanchor);},
}
```

本选项保存一个 \path 命令，用 edge 操作创建路径。
这个路径带有“空箭头”选项 - 并使用键 every new --.

/tikz/every new --=⟨something⟩

注意这个键的路径是 /tikz. 这个键是没有定义的，如果希望它起到某种作用，应当提前定义它，例如：

```
\tikzset{every new --/.style=...} 或者
\tikzset{every new --/.append style=...} 或者
\tikzset{every new --/.code=...} 或者
\tikzset{every new --/.append code=...} 或者其他
```

/tikz/graphs/new <->={⟨full name of edge left node⟩}{⟨full name of edge right node⟩}
{⟨edge options⟩}{⟨edge node spec⟩}

本选项的定义是：

```
\tikzgraphsset{
  new <->/.code n args={4}{
    \path [<->,every new <->/.try]
      (#1\tikzgraphleftanchor)
      edge[#3] #4
      (#2\tikzgraphrightanchor);},
}
```

本选项保存一个 \path 命令，用 edge 操作创建路径。

这个路径带有箭头选项 - 并使用键 every new <->.

/tikz/every new <->=⟨*something*⟩

注意这个键的路径是 /tikz. 这个键是没有定义的，如果希望它起到某种作用，应当提前定义它，例如：

```
\tikzset{every new <->/.style=...} 或者
\tikzset{every new <->/.append style=...} 或者
\tikzset{every new <->/.code=...} 或者
\tikzset{every new <->/.append code=...} 或者其他
```

/tikz/graphs/new -!-={⟨*any*⟩}{⟨*any*⟩}{⟨*any*⟩}{⟨*any*⟩}

本选项的定义是：

```
\tikzgraphsset{
  new -!-/.code n args={4}{},
}
```

本选项只是吃掉它的 4 个参数，什么也不做。

/tikz/graphs/new <-={⟨*full name of edge left node*⟩}{⟨*full name of edge right node*⟩}{⟨*edge options*⟩}{⟨*edge node spec*⟩}

本选项的定义是：

```
\tikzgraphsset{
  new <-/.code n args={4}{%
    \path [<-,every new <-/.try]
      (#1\tikzgraphleftanchor)
      edge[#3] #4
      (#2\tikzgraphrightanchor);
  }
}
```

本选项保存一个 \path 命令，用 edge 操作创建路径。

这个路径带有箭头选项 <- 并使用键 every new <-.

/tikz/every new <-=⟨*something*⟩

注意这个键的路径是 /tikz. 这个键是没有定义的，如果希望它起到某种作用，应当提前定义它，例如：

```
\tikzset{every new <-/.style=...} 或者
\tikzset{every new <-/.append style=...} 或者
\tikzset{every new <-/.code=...} 或者
\tikzset{every new <-/.append code=...} 或者其他
```

\tikzgraphleftanchor

在初始之下这个宏保存的内容是空的。

如果这个宏保存的内容非空，那么它保存的应当是 .⟨*anchor*⟩, 即边的起点的某个 anchor 位置 (注意其中有点号)。用户可以在适当的地方用选项 /tikz/graphs/left anchor → P. 1070 修改它。

\tikzgraphrightanchor

在初始之下这个宏保存的内容是空的。

如果这个宏保存的内容非空，那么它保存的应当是 .⟨*anchor*⟩, 即边的起点的某个 anchor 位置 (注意其中有点号)。用户可以在适当的地方用选项 /tikz/graphs/right anchor 修改它。

/tikz/graphs/default edge kind=⟨某一种类型的边⟩ (initially --)

这个键保存某种边的类型，必须是 --, ->, <-, <->, -!- 这 5 种之一。在初始之下它保存的是 --.

/tikz/graphs/-- (style, no value)

```
\tikzgraphsset{
  --/.style={default edge kind=--},
}
```

/tikz/graphs/-> (style, no value)

```
\tikzgraphsset{
  ->/.style={default edge kind=->},
}
```

/tikz/graphs/<- (style, no value)

```
\tikzgraphsset{
  <-/.style={default edge kind=<-},
}
```

/tikz/graphs/<-> (style, no value)

```
\tikzgraphsset{
  <->/.style={default edge kind=<->},
}
```

/tikz/graphs/-!- (style, no value)

```
\tikzgraphsset{
  -!-/.style={default edge kind=-!-},
}
```

/tikz/graphs/left anchor=⟨*anchor name*⟩ (initially empty)

这个键修改宏 \tikzgraphleftanchor 保存的 anchor 位置。

```
\tikzgraphsset{
  left anchor/.code=\tikz@lib@graph@store@anchor{#1}{\tikzgraphleftanchor},
  left anchor=,
}
```

/tikz/graphs/right anchor=⟨*anchor name*⟩ (initially empty)

这个键修改宏 \tikzgraphrightanchor 保存的 anchor 位置。

```
\tikzgraphsset{
  right anchor/.code=\tikz@lib@graph@store@anchor{#1}{\tikzgraphrightanchor},
  right anchor=
}
```

在命令 \tikz@lib@graph@default@new@edge@ 中展示了一种利用上述选项的方法：

```
\pgfkeys{/tikz/graphs/.cd,new \pgfkeysvalueof{/tikz/graphs/default edge kind}={#3}{#4}
↪  {#1}{#2}}%
```

58.12.2 边的样式，标签

/tikz/graphs/@edges styling={⟨*options*⟩} (initially empty)

这个选项是 graphs 库的内部处理过程使用的键。它的初始值是空的。

```
\tikzgraphsset{
  @edges styling/.initial=,
}
```

/tikz/graphs/edges={⟨*options*⟩} (style)

这个选项向键 `@edges styling` 中添加选项 (在右侧累积)，这些选项用于边，即 `edge` 操作的选项。

```
\tikzgraphsset{
  edges/.style={/tikz/graphs/@edges styling/.append={,#1}},
}
```

/tikz/graphs/edge={⟨*options*⟩} (style)

这个选项等效于 `edges`.

```
\tikzgraphsset{
  edge/.style={edges={#1}},
}
```

/tikz/graphs/@edges node={⟨*node spec*⟩} (initially empty)

这个选项是 graphs 库的内部处理过程使用的键。它的初始值是空的。

```
\tikzgraphsset{
  @edges node/.initial=,
}
```

/tikz/graphs/edge node={⟨*options*⟩} (style)

这个选项的参数是 node 语句。

这个选项向键 `@edges node` 中添加 node 语句 (在右侧累积)，这些 node 语句用作 `edge` 操作的标签。

```
\tikzgraphsset{
  edge node/.style={/tikz/graphs/@edges node/.append={#1}},
}
```

/tikz/graphs/edge label=⟨*label text*⟩ (style)

这个选项的参数是 node 标签的内容。

这个选项向键 `@edges node` 中添加 node 语句 (在右侧累积)，这些 node 语句用作 `edge` 操作的标签。所添加的 node 语句如下：

```
\tikzgraphsset{
  edge label/.style={/tikz/graphs/@edges node/.append={node[auto]{#1}}},
}
```

/tikz/graphs/edge label'=⟨*label text*⟩ (style)

这个选项的参数是 node 标签的内容。

这个选项向键 `@edges node` 中添加 node 语句 (在右侧累积)，这些 node 语句用作 `edge` 操作的标签。所添加的 node 语句如下：

```
\tikzgraphsset{
  edge label'/.style={/tikz/graphs/@edges node/.append={node[auto,swap]{#1}}},
}
```

58.12.3 与 source, target 有关的边的样式，标签，<, > 句法

\tikz@lib@annotate@{⟨tag⟩}{⟨顶点 node 的全名⟩}{⟨edge options⟩}{⟨edge node spec⟩}

参数 ⟨tag⟩ 是一个标签，它可以是字母 a (对应 source) 或者 b (对应 target).

参数 ⟨顶点 node 的全名⟩ 是某个顶点的全名。

参数 ⟨edge options⟩ 是针对边的外观样式的选项，通常就是 TikZ 的 edge 路径的选项。

参数 ⟨edge node spec⟩ 是 TikZ 的 edge 路径上的 node 语句，用作 edge 路径的标签。

本命令“全局地”定义 (或重定义) 控制序列

\csname tikz@lgc⟨tag⟩@@⟨ 顶点 node 的全名⟩\endcsname

使得这个控制序列保存 2 个花括号组：

```
{⟨edge options⟩}{⟨edge node spec⟩}
```

如果多次针对同一组 ⟨tag⟩ 和 ⟨顶点 node 的全名⟩ 使用本命令，那么参数 ⟨edge options⟩ 和 ⟨edge node spec⟩ 会被累积到那 2 个花括号组中：

```
{⟨edge style options 1⟩,⟨edge style options 2⟩,...}{⟨edge node spec 1⟩⟨edge node spec 2⟩...}
```

\tikz@lib@graph@add@edge@annotations{⟨tag⟩}{⟨顶点 node 的全名⟩}

参数 ⟨tag⟩ 是一个标签，它可以是字母 a (对应 source) 或者 b (对应 target). 当控制序列

\csname tikz@lgc⟨tag⟩@@⟨ 顶点 node 的全名⟩\endcsname

有定义时，本命令才有实际的操作。假设这个控制序列保存的是

```
{⟨edge options⟩}{⟨edge node spec⟩}
```

本命令提取这个控制序列保存的内容，人阿定义 3 个宏：

- \pgf@temp

  ```
  \expandafter\def\expandafter\pgf@temp\expandafter{\pgf@temp,⟨edge options⟩}%
  ```

  可见使用本命令前应当提前定义 \pgf@temp, 可以是空的，或者保存 edge 路径的选项。

- \pgf@temp@b

  ```
  \expandafter\def\expandafter\pgf@temp@b\expandafter{\pgf@temp@b⟨edge node spec⟩}%
  ```

  可见使用本命令前应当提前定义 \pgf@temp@b, 可以是空的，或者保存 edge 路径上的 node 语句。

- \tikz@lib@add@temp

  ```
  \let\tikz@lib@add@temp\tikz@lib@final@edge@style
  \def\tikz@lib@final@edge@style{,after source and target edge/.try}%
  ```

/tikz/graphs/source edge style={⟨edge options⟩}

这个选项“全局地”定义 (重定义) 控制序列

\csname tikz@lgca@@⟨ 当前顶点 node 的全名⟩\endcsname

向第 1 个花括号中添加 ,⟨edge options⟩.

本选项最好只用在顶点之后的方括号里，或者 \tikz@lib@graph@name 有定义的地方。

```
\tikzgraphsset{
  source edge style/.code=\tikz@lib@annotate@{a}{\tikz@lib@graph@name}{#1}{},
```

```
}
```

/tikz/graphs/source edge node={⟨edge node spec⟩}

这个选项“全局地”定义 (重定义) 控制序列

\csname tikz@lgca@@⟨ 当前顶点 node 的全名⟩\endcsname

向第 2 个花括号中添加 ⟨edge node spec⟩.

本选项最好只用在顶点之后的方括号里，或者 \tikz@lib@graph@name 有定义的地方。

```
\tikzgraphsset{
  source edge node/.code=\tikz@lib@annotate@{a}{\tikz@lib@graph@name}{}{#1},
}
```

/tikz/graphs/source edge clear

这个选项全局地 let 控制序列

\csname tikz@lgca@@⟨ 当前顶点 node 的全名⟩\endcsname

为 \relax.

本选项最好只用在顶点之后的方括号里，或者 \tikz@lib@graph@name 有定义的地方。

```
\tikzgraphsset{
  source edge clear/.code={
    \expandafter\global\expandafter\let\csname tikz@lgca@@\tikz@lib@graph@name
    ↪  \endcsname\relax%
  },
}
```

/tikz/graphs/target edge style={⟨edge options⟩}

这个选项“全局地”定义 (重定义) 控制序列

\csname tikz@lgcb@@⟨ 当前顶点 node 的全名⟩\endcsname

向第 1 个花括号中添加 ,⟨edge options⟩.

本选项最好只用在顶点之后的方括号里，或者 \tikz@lib@graph@name 有定义的地方。

```
\tikzgraphsset{
  target edge style/.code=\tikz@lib@annotate@{b}{\tikz@lib@graph@name}{#1}{},
}
```

/tikz/graphs/target edge node={⟨edge node spec⟩}

这个选项“全局地”定义 (重定义) 控制序列

\csname tikz@lgcb@@⟨ 当前顶点 node 的全名⟩\endcsname

向第 2 个花括号中添加 ⟨edge node spec⟩.

本选项最好只用在顶点之后的方括号里，或者 \tikz@lib@graph@name 有定义的地方。

```
target edge node/.code=\tikz@lib@annotate@{b}{\tikz@lib@graph@name}{}{#1},
```

/tikz/graphs/target edge clear

这个选项全局地 let 控制序列

\csname tikz@lgcb@@⟨ 当前顶点 node 的全名⟩\endcsname

为 \relax.

本选项最好只用在顶点之后的方括号里，或者 \tikz@lib@graph@name 有定义的地方。

```
\tikzgraphsset{
  target edge clear/.code={%
    \expandafter\global\expandafter\let\csname tikz@lgcb@@\tikz@lib@graph@name
    ↪  \endcsname\relax%
```

```
  }
}
```

/tikz/graphs/clear >

这个选项是个样式，它保存 target edge clear.

```
\tikzgraphsset{
  clear >/.style=target edge clear,
  clear </.style=source edge clear
}%
```

/tikz/graphs/clear <

这个选项是个样式，它保存 source edge clear.

\tikz@lib@activate@source@target@edge@syntax

本命令把符号 <，> 变成“首字符句法”，使得

- 选项 >{⟨*options*⟩} 等价于 target edge style={⟨*options*⟩}
- 选项 >"⟨*options*⟩" 等价于

```
/tikz/node quotes mean={%
  target edge node={%
    node [every edge quotes,#2]{#1}
  }
},
/utils/exec=\tikz@enable@node@quotes,
⟨options⟩
```

- 选项 <{⟨*options*⟩} 等价于 source edge style={⟨*options*⟩}
- 选项 <"⟨*options*⟩" 等价于

```
/tikz/node quotes mean={%
  source edge node={%
    node [every edge quotes,#2]{#1}
  }
},
/utils/exec=\tikz@enable@node@quotes,
⟨options⟩
```

其中 \tikz@enable@edge@quotes 是 quotes 库的命令，它激活使用选项的引号句法。

参考 \ifpgfkeys@syntax@handlers →P. 58.

graphs 库只是在 \tikz@lib@graph@node@opt@normal →P. 1046 那里调用本命令，所以本命令声明的首字符句法只在顶点之后的方括号里才有效。

\tikzlibignorecomparisonsINTERNAL

本命令把符号 <，> 变成“首字符句法”，使得

- 选项 >{⟨*options*⟩} 被吃掉，没有任何作用。
- 选项 <{⟨*options*⟩} 被吃掉，没有任何作用。

58.13 简单图，多重图

在简单图 (simple graph) 中，如果在两个顶点之间指定了两个或更多个边 (包括 -!- 类型的边，并且不考虑这两个顶点的次序)，则只画出最后一个边。在多重图中，如果在两个顶点之间指定了两个或更多个边，那么这些边都会被画出来。

注意，简单图、多重图是 graphs 库本身固有的处理边的流程。

\iftikz@lib@graph@simple

这个 TEX-if 用于决定是否画简单图。默认 \tikz@lib@graph@simplefalse, 即画多重图。

\tikz@lib@graph@set@simple@edge{⟨边的类型⟩}{⟨顶点 x 的全名⟩}{⟨顶点 y 的全名⟩}{⟨*edge options*⟩}{⟨*edge node spec*⟩}

在 \iftikz@lib@graph@simple 的真值为 true 的情况下，当需要在顶点 ⟨顶点 x 的全名⟩ 与 ⟨顶点 y 的全名⟩ 之间创建边 (包括 -!- 类型的边)——记为 ⟨当前边⟩ ——时，会调用本命令。

见 \tikz@lib@graph@default@new@edge@ → P. 1078.

本命令并不直接创建 ⟨当前边⟩，而是将创建 ⟨当前边⟩ 的代码“全局地”保存到控制序列

\csname tikz@lg@e@⟨顶点 x 的全名⟩@⟨顶点 y 的全名⟩\endcsname

中——这是全局定义这个控制序列；同时，还会清除之前保存的、创建顶点 ⟨顶点 x 的全名⟩ 与 ⟨顶点 y 的全名⟩ 之间的边的代码——如果有话，例如全局地 let 控制序列

\csname tikz@lg@e@⟨顶点 y 的全名⟩@⟨顶点 x 的全名⟩\endcsname

为 \relax.

```
\def\tikz@lib@graph@set@simple@edge#1#2#3#4#5{%
  % #1 = kind (<-, ->, ...)
  % #2 = from
  % #3 = to
  % #4 = options
  % #5 = edge nodes
  %
  % Ok, first, test, whether edge exists:
  \ifcsname tikz@lg@e@#3@#2\endcsname%
    \expandafter\global\expandafter\let\csname tikz@lg@e@#3@#2\endcsname\relax
    ↪  % reset
  \fi%
  \expandafter\gdef\csname tikz@lg@e@#2@#3\endcsname{
  ↪  \tikz@lib@graph@make@simple@edge{#1}{#2}{#3}{#4}{#5}}%
}%
```

\tikz@lib@graph@make@simple@edge{⟨边的类型⟩}{⟨顶点 x 的全名⟩}{⟨顶点 y 的全名⟩}{⟨*edge options*⟩}{⟨*edge node spec*⟩}

本命令的定义是：

```
\def\tikz@lib@graph@make@simple@edge#1#2#3#4#5{%
  \pgfqkeys{/tikz/graphs}{new #1={#2}{#3}{#4}{#5}}%
}%
```

例如 \tikz@lib@graph@make@simple@edge{->}{a}{b}{red}{node{TexT}} 导致

```
\path [->,every new ->/.try]
  (a\tikzgraphleftanchor)
  edge[red] node{TexT}
  (b\tikzgraphrightanchor);
```

\tikz@lib@graph@simple@done

本命令是一个循环操作：

1. 执行 \tikz@lib@graph@pack@node@list → P. 1064, 去掉 \tikz@lib@graph@node@list 中重复的

   ```
   \tikz@lg@do{⟨顶点 node 的全名⟩}
   ```

2. 之后的操作等价于下面的循环：对于 \tikz@lib@graph@node@list 中的那些顶点 x_1, x_2, ..., x_n, 执行如下操作：

```
{% 组
  for A = x_1,...,x_n do
    {% 组
      for B = x_1,...,x_n do
        如果控制序列 \csname tikz@lg@e@A@B\endcsname 不等于 \relax,
        则执行这个控制序列;
        然后全局地 let 这个控制序列为 \relax.
      end
    }
  end
}
```

这种循环操作是通过将 \tikz@lg@do 命令 let 为不同命令，再执行 \tikz@lib@graph@node@list 实现的。

/tikz/graphs/simple (no value)

本选项决定画简单图。

本选项的定义是：

```
\tikzgraphsset{
  simple/.code={
    \iftikz@lib@graph@simple%
      % is already simple, ignore
    \else
      \tikz@lib@graph@simpletrue%
      \pgfkeysalso{operator=\tikz@lib@graph@simple@done}%
    \fi%
  },
}
```

注意，本选项的有效范围决定了执行命令 \tikz@lib@graph@simple@done → P. 1075 的时机。通常将本选项用作链组的选项，在组结束时，会执行 \tikz@lib@graph@simple@done → P. 1075.

/tikz/graphs/multi (no value)

本选项决定画多重图。

本选项的定义是：

```
\tikzgraphsset{
  multi/.code={
    \tikz@lib@graph@simplefalse%
  }
}
```

58.14 connector

一种“连接器”(connector) 就是一种在顶点之间自动画边的规则或操作，使得顶点之间呈现某种连接状态。一个连接器通常就是一个样式，其中保存的是键 /tikz/graphs/operator → P. 1066, 例如

```
\tikzgraphsset{
  matching and star/.style 2 args={operator={ ⟨code⟩ }},
  matching and star/.default={target'}{source'}
}%
```

如果一个连接器被执行，那么其中保存的 operator 就会被执行，从而把 ⟨code⟩ 保存到键 /tikz/graphs/@operators → P. 1066 中。通常，预定义的连接器都会在 ⟨code⟩ 中利用 \tikz@lib@graph@node@list, 根据顶点的颜色类属性，对顶点之间的边作某种操作。

由于 \tikz@lib@graph@node@list 的内容是不断变化的，且是局部定义的，所以要注意应该把连接器用在什么地方：如果用作某个边的选项，那么当边右侧的顶点被创建后，连接操作会被执行，此时只对这个边两端的顶点有效；如果用作某个链组的选项，那么在组结束时，连接操作才会被执行，此时对整个组内的顶点有效。

默认的连接器是 matching and star.

注意，连接器的设计要考虑到 (或者说服从) “简单图、多重图” 的处理流程，以及顶点颜色类属性的变化方式。

/tikz/graphs/default edge operator=⟨*a connector*⟩ (initially matching and star)

本选项指定默认的连接器，在初始之下的默认连接器是 matching and star.

用户可以使用本选项指定一种默认的连接器。

如果用户没有使用选项 graphs/operator 向键 graphs/@operators 中添加代码，或者说用户提供的 graphs/operator 值是空的，或者当前状态超出 graphs/operator 的有效范围，那么就使用本选项指定的连接器。

```
\tikzgraphsset{
  default edge operator/.initial=matching and star,
}
```

命令 \tikz@lib@graph@joiner → P. 1057 展示了一种使用 default edge operator 的方法：

```
macro:->\tikzgraphpreparecolor {target'}\c@pgf@counta {tikz@lg}
\c@pgf@countb =0\relax \let \tikz@lg@prev \relax \tikzgraphforeachcolorednode
{source'}\tikz@lib@graph@flow@do \tikz@lib@graph@flow@rest
```

```
\makeatletter
\edef\pgf@temp{\noexpand\pgfkeys{/tikz/graphs/.cd,\pgfkeysvalueof{/tikz/graphs/default edge
↪  operator}}}%
\pgf@temp%
\pgfkeysgetvalue{/tikz/graphs/@operators}\pgf@temp%
\ttfamily\meaning\pgf@temp
\makeatother
```

如上，默认连接器 matching and star 保存的 “连接操作代码” 被复制到了 \pgf@temp 中。

\tikz@lib@graph@default@new@edge{⟨*full name of source node*⟩}{⟨*full name of target node*⟩}

本命令被多个预定义的连接器调用。

本命令：

1. 把选项 graphs/@edges styling, source edge style, target edge style 中保存的键值对 (路径为 /tikz 的，能用作 edge 操作的选项) 收集起来，保存到宏 \pgf@temp 中。
   另外也把 ,after source and target edge/.try 添加到宏 \pgf@temp 中 (右侧)。
2. 把选项 graphs/@edges node, source edge node, target edge node 中保存的 node 语句 (将用作 edge 操作的标签) 收集起来，保存到宏 \pgf@temp@b 中。
3. 调用 \tikz@lib@graph@default@new@edge@, 检查 \iftikz@lib@graph@simple 的真值，
   - 如果 \iftikz@lib@graph@simple 的真值是 true, 则

     ```
     \edef\tikz@temp{{\pgfkeysvalueof{/tikz/graphs/default edge kind}}
     ↪  {⟨full name of source node⟩}{⟨full name of target node⟩}}
     \expandafter\tikz@lib@graph@set@simple@edge\tikz@temp{\pgf@temp}{
     ↪  \pgf@temp@b}%
     ```

   - 如果 \iftikz@lib@graph@simple 的真值是 false, 则

```
\pgfkeys{/tikz/graphs/.cd,new \pgfkeysvalueof{/tikz/graphs/default edge
↪  kind}={⟨full name of source node⟩}{⟨full name of target node⟩}{\pgf@temp}{
↪  \pgf@temp@b}}%
```

如果 default edge kind 的值是 ->, 这就导致

```
\path [->,every new ->/.try]
  (⟨full name of source node⟩\tikzgraphleftanchor)
  edge[展开的 \pgf@temp] 展开的 \pgf@temp@b
  (⟨full name of target node⟩\tikzgraphrightanchor);
```

\tikz@lib@graph@default@new@edge@{⟨*full name of source node*⟩}{⟨*full name of target node*⟩}{⟨*edge options*⟩}{⟨*edge node spec* 俄⟩}

```
\def\tikz@lib@graph@default@new@edge@#1#2#3#4{%
  \iftikz@lib@graph@simple%
    \edef\tikz@temp{{\pgfkeysvalueof{/tikz/graphs/default edge kind}}{#3}{#4}}
    \expandafter\tikz@lib@graph@set@simple@edge\tikz@temp{#1}{#2}%
  \else%
    \pgfkeys{/tikz/graphs/.cd,new \pgfkeysvalueof{/tikz/graphs/default edge kind}=
    ↪  {#3}{#4}{#1}{#2}}%
  \fi%
}%
```

58.14.1 预定义的连接器：针对一个颜色类

graphs 库预定义了一些连接器，当这些连接器保存的操作被执行时，所针对的是当前 \tikz@lib@graph@node@list 内的、所有属于某个颜色类的那些顶点，按某种规则处置它们之间的边。

这些算子都是样式 (/.style)，保存着 operator 值。

/tikz/graphs/clique=⟨*color*⟩ (default all)

这里的 ⟨*color*⟩ 指的是“颜色类”。

本选项是个样式，其定义是：

```
\tikzgraphsset{
  clique/.style={operator={
      \tikzgraphpreparecolor{#1}\c@pgf@counta{tikz@lg}%
      \tikz@lg@clique@loop%
    }},
  clique/.default=all
}%
```

当这个样式保存的操作被执行时，其处理如下：

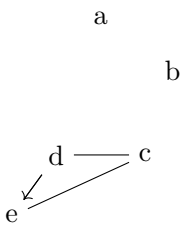
先执行 \tikzgraphpreparecolor→ P. 1065{⟨*color*⟩}\c@pgf@counta{tikz@lg}, 然后执行下面的循环：

假设当前 \tikz@lib@graph@node@list 内属于颜色类 ⟨*color*⟩ 的那些顶点是 $x_1, \ldots, x_m$,

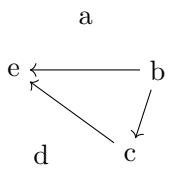
```
if m > 0 then
  for a = m, ... , 1 do
    b = a - 1
    if b > 0 then
      \tikz@lib@graph@default@new@edge{x_b}{x_a}%
    end
  end
end
```

这个操作会在属于颜色类 ⟨*color*⟩ 的任意两个顶点之间画边。

本选项的默认值是 all.




```
\tikz \graph [clockwise, n=5] {
  a,
  b,
  { [clique]
    c, d -> e }
};
```



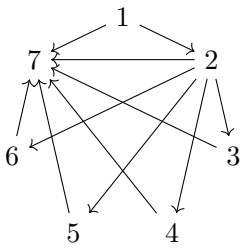

```
\tikz \graph [color class=red, clockwise, n=5]
{ [clique=red, ->]
  a, b[red], c[red], d, e[red]
};
```

/tikz/graphs/induced independent set=⟨*color*⟩ (default all)

当这个样式保存的操作被执行时，其处理如下：

假设把当前 \tikz@lib@graph@node@list 内的属于颜色类 ⟨*color*⟩ 的顶点看作是一个集合 T，$\forall a, b \in T$，本操作会在 a, b 之间使用一个画边命令 -!-. 在默认之下这个画边命令 -!- 不执行任何动作，但它仍然是个画边命令。所以在选项 simple 起作用的情况下，这个算子的作用恰好与 clique 相反，它会取消顶点 a, b 之间的边，因为选项 simple 只允许在顶点 a, b 之间使用一个画边命令（最后出现的画边命令）。

本选项的默认值是 all.




```
\tikz \graph [simple] {
  subgraph K_n [->, n=7, clockwise, radius=1.5cm];
  { [induced independent set] 1, 3, 4, 5, 6 }
};
```

/tikz/graphs/cycle=⟨*color*⟩ (default all)

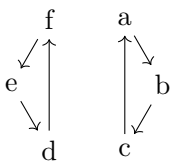
这个算子会把属于颜色类 ⟨*color*⟩ 的顶点联系为一个环。

本选项的默认值是 all.

当这个样式保存的操作被执行时，其处理如下：

假设当前 \tikz@lib@graph@node@list 内属于颜色类 ⟨*color*⟩ 的顶点是 $x_1, \ldots, x_m$，执行下面的分支、循环：

```
if m > 0 then
  for a = 1, ... , m do
    if a = 1 then
      \def\tikz@lg@prev{x_a}%
      \let\tikz@lg@first\tikz@lg@prev%
    else
      \tikz@lib@graph@default@new@edge{\tikz@lg@prev}{x_a}
      \def\tikz@lg@prev{x_a}%
    end
  end
  \tikz@lib@graph@default@new@edge{\tikz@lg@prev}{\tikz@lg@first}%
end
```

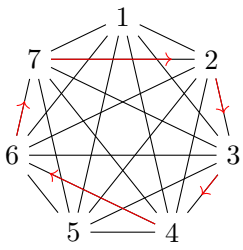



```
\tikz \graph [clockwise, n=6, phase=60] {
  { [cycle, ->] a, b, c },
  { [cycle, <-] d, e, f }
};
```

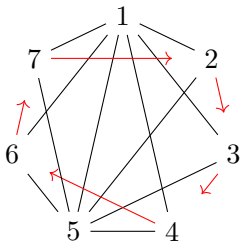
/tikz/graphs/induced cycle=⟨*color*⟩ (default all)

这个样式保存的内容是 induced independent set={⟨*color*⟩}, cycle={⟨*color*⟩}, 即先清空 ⟨*color*⟩ 顶点之间的边，然后将它们连成一个环。

本选项的默认值是 all.




```
\tikz \graph {
  subgraph K_n [n=7, clockwise, radius=1.5cm];
  { [induced cycle, ->, edge={red, shorten >=3mm}]
    2, 3, 4, 6, 7 },
};
```




```
\tikz \graph [simple] {
  subgraph K_n [n=7, clockwise, radius=1.5cm];
  { [induced cycle, ->, edge={red, shorten >=3mm}]
    2, 3, 4, 6, 7 },
};
```

/tikz/graphs/path=⟨*color*⟩ (default all)

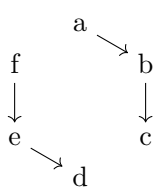
这个算子会把属于颜色类 ⟨*color*⟩ 的顶点联系为一个链。

本选项的默认值是 all.

当这个样式保存的操作被执行时，其处理如下：

假设当前 \tikz@lib@graph@node@list 内的属于颜色类 ⟨*color*⟩ 的顶点是 $x_1, \ldots, x_m$, 执行下面的分支、循环：

```
if m > 0 then
  for a = 1, ... , m do
    if a = 1 then
      \def\tikz@lg@prev{x_a}%
    else
      \tikz@lib@graph@default@new@edge{\tikz@lg@prev}{x_a}
      \def\tikz@lg@prev{x_a}%
    end
  end
end
```

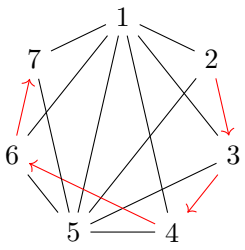



```
\tikz \graph [clockwise, n=6] {
  { [path, ->] a, b, c },
  { [path, <-] d, e, f }
};
```

/tikz/graphs/induced path=⟨*color*⟩ (default all)

这个样式保存的内容是 `induced independent set={⟨color⟩}, path={⟨color⟩}`, 即先清空 ⟨color⟩ 顶点之间的边，然后将它们连成一个链。

本选项的默认值是 `all`.




```
\tikz \graph [simple] {
  subgraph K_n [n=7, clockwise, radius=1.5cm];
  { [induced path, ->, edges=red] 2, 3, 4, 6, 7 },
};
```

`/tikz/graphs/grid=`⟨color⟩ (default `all`)

本选项的默认值是 `all`.

当这个样式保存的操作被执行时，其处理如下：

假设当前 `\tikz@lib@graph@node@list` 内的属于颜色类 ⟨color⟩ 的顶点是 $x_1, \ldots, x_m$,

1. 确定列数，记 `/tikz/graphs/wrap after` 的值是整数 p, 宏 `\tikzgraphVnum` 的值是整数 q,

```
if p = 0 then
  \edef\tikzgraphwrapafter{]√q[}% 开平方后取整
else
  \edef\tikzgraphwrapafter{p}
end
```

2. 记列数 `\tikzgraphVnum` 的值是 k,

```
a = m
if a > 0 then
  for b = 1, ... , m do
    if a > 0 then
      r = ( b - 1 ) mod k
      if r > 0 then
        \tikz@lib@graph@default@new@edge{x_b}{x_{b-1}}
      end
      if b > k then
        r = b - k
        \tikz@lib@graph@default@new@edge{x_{b-k}}{x_b}
      end
      a = a - 1
    end
  end
end
```

58.14.2 预定义的连接器：针对两个颜色类

graphs 库预定义了一些操作 (operator)，当这些操作被执行时，只是针对当前 `\tikz@lib@graph@node@list` 内的、并且属于某 2 个颜色类的顶点，按某种规则处置它们之间的边。

这些算子都是样式 (`/.style`)，保存着 `operator` 的值。

58.14.2.1 connector: matching and star

`/tikz/graphs/matching and star={`⟨from color⟩`}{`⟨to color⟩`}` (default `{target'}{source'}`)

这个样式的默认值是 `target'` 与 `source'`. 三

连接器 matching and star 的定义是：

```
\tikzgraphsset{
  matching and star/.style 2 args={operator=%
    {%
      \tikzgraphpreparecolor{#1}\c@pgf@counta{tikz@lg}
      \c@pgf@countb=0\relax%
      \let\tikz@lg@prev\relax
      \tikzgraphforeachcolorednode{#2}\tikz@lib@graph@flow@do%
      \tikz@lib@graph@flow@rest%
    }%
  },
  matching and star/.default={target'}{source'}
}%

\def\tikz@lib@graph@flow@do#1{%
  \advance\c@pgf@countb by1\relax%
  \ifnum\c@pgf@countb>\c@pgf@counta\relax%
    \c@pgf@countb=\c@pgf@counta\relax%
  \fi%
  \ifnum\c@pgf@countb>0\relax%
    \tikz@lib@graph@default@new@edge{\csname tikz@lg\the\c@pgf@countb\endcsname}{#1}%
  \fi%
  \def\tikz@lg@prev{#1}%
}%

\def\tikz@lib@graph@flow@rest{%
  \ifnum\c@pgf@countb<\c@pgf@counta\relax%
    \ifnum\c@pgf@countb>0\relax%
      \advance\c@pgf@countb by1\relax%
      \tikz@lib@graph@default@new@edge{\csname tikz@lg\the\c@pgf@countb\endcsname}{
      ↪  \tikz@lg@prev}%
      \expandafter\tikz@lib@graph@flow@rest%
    \fi%
  \fi%
}%
```

执行

```
1 \tikzgraphpreparecolor{target'}\c@pgf@counta{tikz@lg}
2 \c@pgf@countb=0\relax% 赋值 0
3 \let\tikz@lg@prev\relax
4 \tikzgraphforeachcolorednode{source'}\tikz@lib@graph@flow@do%
5 \tikz@lib@graph@flow@rest%
```

的意思是：

1 行 参考 \tikzgraphpreparecolor→P. 1065.

对于当前的 \tikz@lib@graph@node@list 中保存的顶点，利用计数器 \c@pgf@counta 为其中属于颜色类 target' 的顶点作编号，记被编号的顶点为

$$t'_1,\ t'_2,\ \ldots,\ t'_m,$$

有效的编号从 1 开始，计数器 \c@pgf@counta 保存着编号的终止值 m.

4 行 参考 \tikzgraphforeachcolorednode→P. 1064.

对于当前的 \tikz@lib@graph@node@list 中保存的顶点，其中属于颜色类 source' 的顶点是

$$s'_1,\ s'_2,\ \ldots,\ s'_n,$$

对这些属于 source' 的顶点逐一执行 \tikz@lib@graph@flow@do{s'_i}, 这个命令的意思是：

```
if m > 0 and n > 0 then
  for i = 1, ... , n do
    j = i
    if j > m then
      j=m
    end
    \tikz@lib@graph@default@new@edge{t'_j}{s'_i}%
  end
end
\def\tikz@lg@prev{s'_n}
```

注意，如果 $m < n$, 就会出现 t'_m 对应 $s'_m, s'_{m+1}, \ldots, s'_n$ 这种“一对多”的情况。

参考 \tikz@lib@graph@default@new@edge → P. 1077.

5 行 这一行的意思就是：

```
if n > 0 and m > n then
  for k = n+1, ... , m do
    \tikz@lib@graph@default@new@edge{t'_k}{s'_n}%
  end
end
```

注意，如果 $m > n$, 就会出现 $t'_{n+1}, t'_{n+2}, \ldots, t'_m$ 对应 s'_n 这种“多对一”的情况。

参考 \tikz@lib@graph@default@new@edge → P. 1077.

注意，边的左端顶点来自 target', 右端顶点来自 source'.

58.14.2.2 connector: matching

/tikz/graphs/matching={⟨*from color*⟩}{⟨*to color*⟩} (default {target'}{source'})

这个样式的默认值是 target' 与 source'.

当这个样式保存的操作被执行时，其处理如下：

记当前 \tikz@lib@graph@node@list 内属于颜色类 ⟨*from color*⟩ 的顶点是

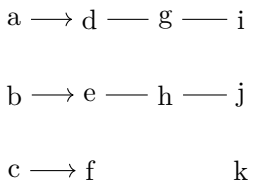
$$f_1, \ldots, f_m,$$

属于颜色类 ⟨*to color*⟩ 的顶点是

$$t_1, \ldots, t_n,$$

具体处理是：

```
{% 组
  if n > 0 then
    for i = 1, ... , n do
      if i > m then
      else
        \tikz@lib@graph@default@new@edge{f_i}{t_i}%
      end
    end
  end
}
```




```
\tikz \graph {
  {a, b, c} ->[matching]
  {d, e, f} --[matching]
  {g, h} --[matching]
  {i, j, k}
};
```

58.14.2.3 connector: butterfly, butterfly'

/tikz/graphs/butterfly={⟨*options*⟩} (no default)

这是个样式。

⟨*options*⟩ 是路径为 /tikz/graphs/butterfly 的键值对列表，默认 ⟨*options*⟩ 是空的。

在 ⟨*options*⟩ 中可以使用以下选项：

/tikz/graphs/butterfly/level=⟨*level*⟩ (initially 1)

这个键保存一个整数。

/tikz/graphs/butterfly/from=⟨*from color*⟩ (initially target')

这个键保存一个颜色类名称，其中的顶点作为边的 from 端。

/tikz/graphs/butterfly/to=⟨*to color*⟩ (initially source')

这个键保存一个颜色类名称，其中的顶点作为边的 to 端。

样式 butterfly 构造“蝶形网络”（butterfly network），当这个样式保存的操作被执行时，其处理如下：

1. 执行 ⟨*options*⟩.
2. 检查 /tikz/graphs/butterfly/level 的当前值 ⟨*level*⟩，
   - 如果 ⟨*level*⟩= 0, 那么

     ```
     \tikzgraphinvokeoperator{matching and star={\pgfkeysvalueof
     ↪  {/tikz/graphs/butterfly/from}}{\pgfkeysvalueof
     ↪  {/tikz/graphs/butterfly/to}}}%
     ```

     也就是执行 matching and star 保存的操作，参考 \tikzgraphinvokeoperator → P. 1067.
   - 如果 ⟨*level*⟩≠ 0, 那么，记当前 \tikz@lib@graph@node@list 内属于颜色类 ⟨*from color*⟩ 的顶点是

     $$f_1, \ldots, f_m,$$

     属于颜色类 ⟨*to color*⟩ 的顶点是

     $$t_1, \ldots, t_n,$$

     然后

     (a) 执行

     ```
     {% 组
       if n > 0 then
         for i = 1, ... , n do
           b = i - 1
           c = ⟨level⟩
           c2 = b mod ( 2 * c )
           if c2 < c then
             c = b + c + 1
           else
             c = b - c + 1
           end
     ```

```
      if c > i then
        c = i
      end
      \tikz@lib@graph@default@new@edge{f_c}{t_i}%
    end
  end
}
```

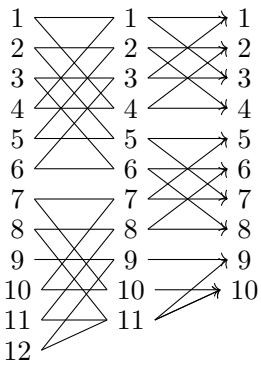
(b) 检查 \iftikz@butterfly@prime 的真值，

– 如果它的真值是 true, 则什么也不做；

– 如果它的真值是 false, 则

```
\tikzgraphinvokeoperator{matching and star={\pgfkeysvalueof
↪  {/tikz/graphs/butterfly/from}}{\pgfkeysvalueof
↪  {/tikz/graphs/butterfly/to}}}
```

也就是执行 matching and star 保存的操作,参考 \tikzgraphinvokeoperator → P. 1067.



```
\tikz \graph [left anchor=east, right anchor=west,
branch down=4mm, grow right=15mm] {
  subgraph I_n [n=12, name=A] --[butterfly={level=3}]
  subgraph I_n [n=11, name=B] ->[butterfly={level=2}]
  subgraph I_n [n=10, name=C]
};
```

\iftikz@butterfly@prime

这个 TEX-if 的真值影响 butterfly 的行为。

/tikz/graphs/butterfly'={⟨*options*⟩} (no default)

这是个样式。

```
\tikzgraphsset{
  butterfly'/.style={operator={}{\tikz@butterfly@primetrue\pgfkeysalso{butterfly=
  ↪  {#1}}}},
  butterfly'/.default=,
}
```

58.14.2.4 connector: complete bipartite

/tikz/graphs/complete bipartite={⟨*from color*⟩}{⟨*to color*⟩} (default {target'}{source'})

这里的 ⟨*from color*⟩，⟨*to color*⟩ 是颜色类名称。本选项的默认值是 {target'}{source'}.

这个样式的定义是：

```
\tikzgraphsset{
  complete bipartite/.style 2 args={operator={
    \def\tikz@lg@shoreb{#2}%
    \tikzgraphforeachcolorednode{#1}\tikz@lib@graph@bipartite@outer
  }},
  complete bipartite/.default={target'}{source'},
}%
```

当这个样式保存的操作被执行时，其处理如下：

记当前 `\tikz@lib@graph@node@list` 内属于颜色类 ⟨*from color*⟩ 的顶点是

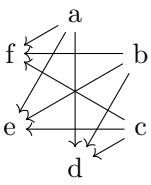
$$f_1,\ \ldots,\ f_m,$$

属于颜色类 ⟨*to color*⟩ 的顶点是

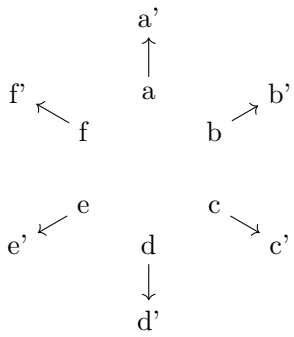
$$t_1,\ \ldots,\ t_n,$$

执行下面的分支、循环：

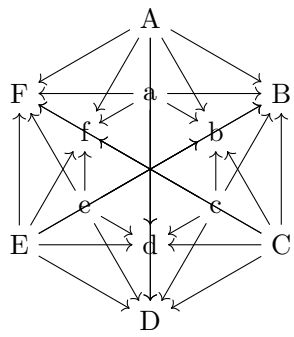
```
if m > 0 then
  for i = 1, ... , m do
    {% 组
      if n > 0 then
        for j = 1, ... , n do
          if t_j = f_i then
          else
            \tikz@lib@graph@default@new@edge{f_i}{t_j}%
          end
        end
      end
    }
  end
end
```



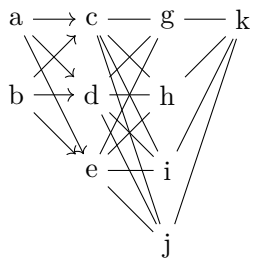

```
\tikz \graph [color class=red, color class=green, clockwise, n=6] {
  [complete bipartite={red}{green}, ->]
  a [red], b[red], c[red], d[green], e[green], f[green]
};
```



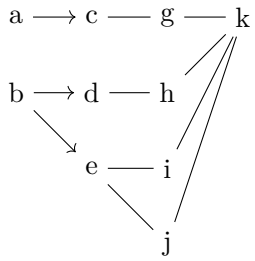

```
\tikz \graph [color class=red, color class=green,
    clockwise, n=6] {
  {a [red], b[green], c[red], d[green], e[red], f[green]}
    ->
  {a' [red], b'[green], c'[red], d'[green], e'[red], f'[green]}
};
```




```
\tikz \graph [color class=red, color class=green,
    clockwise, n=6] {
  {a [red], b[green], c[red], d[green], e[red], f[green]}
    ->[complete bipartite={red}{green}]
  {A [red], B[green], C[red], D[green], E[red], F[green]}
};
```




```
\tikz \graph {
  {a, b} ->[complete bipartite]
  {c, d, e} --[complete bipartite]
  {g, h, i, j} --[complete bipartite]
  k };
```




```
\tikz \graph [complete bipartite]{
  {a, b} ->
  {c, d, e} --
  {g, h, i, j} --
  k };
```

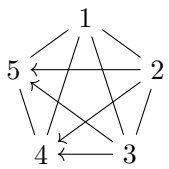
上面图形中的 `complete bipartite` 没有任何作用，因为这个选项的默认值是 `{target'}{source'}`，但在把各个顶点、边解析完毕后，就没有被标记为 `source'` 或 `target'` 的顶点了。

`/tikz/graphs/induced complete bipartite`={⟨*from color*⟩}{⟨*to color*⟩}(default `{target'}{source'}`)

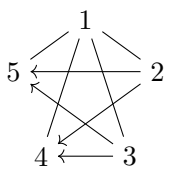
这个样式的定义是：

```
\tikzgraphsset{
  induced complete bipartite/.style 2 args={
    induced independent set={#1},
    induced independent set={#2},
    complete bipartite={#1}{#2}
  },
  induced complete bipartite/.default={target'}{source'},
}%
```

这个样式的操作是：先取消 ⟨*from color*⟩ 内部顶点之间的边，再取消 ⟨*to color*⟩ 内部顶点之间的边，然后使用 `complete bipartite` 的操作。




```
\tikz \graph [simple] {
  subgraph K_n [n=5, clockwise];
  {2, 3} ->[complete bipartite] {4, 5}
};
```




```
\tikz \graph [simple] {
  subgraph K_n [n=5, clockwise];
  {2, 3} ->[induced complete bipartite] {4, 5}
};
```

58.14.2.5 connector: no edges

`/tikz/graphs/no edges` (style)

这个样式取消创建边的代码。

```
\tikzgraphsset{no edges/.style={operator=\relax}}%
```

58.15 \tikz@lib@graph@stored@actions

宏 \tikz@lib@graph@stored@actions 与边的创建有关。

在 \tikz@lib@graph@main@parser→P. 1033 那里，在 \begingroup 之后，读取一个链之前，清空这个宏：

```
\let\tikz@lib@graph@stored@actions\pgfutil@empty%
```

在解析边的过程中，在 \tikz@lib@graph@after@arrow@opt→P. 1056 那里，会重定义这个宏，它保存

```
\tikz@lib@graph@joiner{⟨ 当前边的类型⟩}{⟨ 边的选项 ⟨options⟩⟩}
```

见 \tikz@lib@graph@joiner→P. 1057.

这个宏用在 2 个地方：

- 当处理完毕一个顶点 (新顶点,或者以圆括号开头的引用顶点) 后,执行这个宏,见 \tikz@lib@graph@@node→P. 1040.
- 当处理完毕一个链组后，执行这个宏，见 \tikz@lib@graph@scope→P. 1038.

也就是说，每当需要在某处创建一个边 (包括 -!- 类型的边) 时，都会调用这个宏，这个宏保存的命令会对创建这个边的代码 (命令) 作出处理：

◇ 如果当下是创建多重图，就立即执行创建这个边的代码。

◇ 如果当下是创建简单图，就保存创建这个边的代码，待到适当的时机再调用保存的代码创建这个边。这个“时机”要看 \iftikz@lib@graph@simple→P. 1075 的 true 值的有效范围,或者选项 /tikz/graphs/simple→P. 1076 的有效范围。通常这个选项用作链组的选项,当这个组结束时,执行 \tikz@lib@graph@graph@group@done→P. 1035,导致 \tikz@lib@graph@simple@done→P. 1075 被执行，在这个组中保存的创建各个边的代码会被执行，创建出各个边。

58.16 改变顶点颜色类属性的默认方式

graphs 库默认了一套改变顶点颜色类属性的方式。

顶点颜色类属性的变化总是全局的。

顶点颜色类属性发生变化的地方有：

- 当新建一个顶点时，\tikz@lib@graph@node@opt@normal→P. 1046 调用 \tikz@lg@init@color→P. 1061 设置新顶点的颜色类属性的初始值为

  ```
  \tikz@lgc@all@true\tikz@lgc@source@true\tikz@lgc@target@true
  ```

- 当用圆括号引用顶点或者顶点集合所指定的顶点时，\tikz@lib@graph@node@opt@normal→P. 1046 调用 \tikz@lib@graph@do@use→P. 1050 恢复那些被引用顶点的初始值为

  ```
  \tikz@lgc@all@true\tikz@lgc@source@true\tikz@lgc@target@true
  ```

- 当解析一个边时,\tikz@lib@graph@after@arrow@opt→P. 1056 调用 \tikzgraphinvokeoperator→P. 1067 修改当前 \tikz@lib@graph@node@list 中保存的顶点的颜色类属性，即

  recolor source by=source'' 以及 recolor target by=target'

- 当宏 \tikz@lib@graph@stored@actions (也就是命令 \tikz@lib@graph@joiner→P. 1057) 被执行时，调用 \tikzgraphinvokeoperator→P. 1067 修改当前 \tikz@lib@graph@node@list 中保存的顶点的颜色类属性，首先

  recolor source by=source' 以及 recolor source'' by=source

  然后执行创建边的命令或者保存创建边的命令，然后

  not source' 以及 not target'

- 在 \graph 命令的结尾处,在 \tikz@lib@graph@main@done 之前会调用 \tikz@lib@graph@cleanup→P. 1061,清理所有顶点的颜色类属性。

显然，如果不存在边与某个顶点相关，那么这个顶点的颜色类属性就不会改变。

假设采用默认的连接器 matching and star 并创建多重图，以 {a, b, c} -> {d, e -- f} 为例，其中顶点的颜色类变化是：

1. 处理第一个链组
   ```
   a  all  source  target
   b  all  source  target
   c  all  source  target
   ```
2. 解析 ->
   ```
   a  all  source''  target'
   b  all  source''  target'
   c  all  source''  target'
   ```
3. 读取第 2 个链组，创建 d
   ```
   a  all  source''  target'
   b  all  source''  target'
   c  all  source''  target'
   d  all  source    target
   ```
4. 创建 e
   ```
   a  all  source''  target'
   b  all  source''  target'
   c  all  source''  target'
   d  all  source    target
   e  all  source    target
   ```
5. 解析 --
   ```
   a  all  source''  target'
   b  all  source''  target'
   c  all  source''  target'
   d  all  source    target
   e  all  source''  target'
   ```
6. 创建 f
   ```
   a  all  source''  target'
   b  all  source''  target'
   c  all  source''  target'
   d  all  source    target
   e  all  source''  target'
   f  all  source    target
   ```
7. 创建边 -- 前
   ```
   a  all  source''  target'
   b  all  source''  target'
   c  all  source''  target'
   d  all  source    target
   e  all  source    target'
   f  all  source'   target
   ```
8. 创建边 -- 后

```
a  all  source''  target'
b  all  source''  target'
c  all  source''  target'
d  all  source    target
e  all  source
f  all            target
```

9. 创建边 -> 前

```
a  all  source   target'
b  all  source   target'
c  all  source   target'
d  all  source'  target
e  all  source'
f  all           target
```

10. 创建边 -> 后

```
a  all  source
b  all  source
c  all  source
d  all          target
e  all
f  all          target
```

58.17 顶点的位置

graphs 库提供了一个计算顶点位置的框架，这个框架维护着一些变量：

```
\tikzgraphsset{
  placement/.cd,
  element count/.initial=0,
  chain count/.initial=0,
  depth/.initial=0,
  width/.initial=0,
  level/.initial=0,
  logical node depth/.code=\def\pgfmathresult{1},
  logical node width/.code=\def\pgfmathresult{1},
}%
% 内部处理过程也会使用：
%\pgfkeyssetvalue{/tikz/graphs/placement/local depth}{0}%
%\pgfkeyssetvalue{/tikz/graphs/placement/local width}{0}%
```

可以利用这些变量编写计算顶点位置的代码并保存到 /tikz/graphs/placement/compute position → P. 1091 中；\tikz@lib@graph@node@opt@normal → P. 1046 在创建一个新顶点前，会执行

```
\pgfkeys{/tikz/graphs/placement/place}%
```

调用保存的计算代码获得一个点坐标 (这就是新顶点的位置) 并执行变换选项 /tikz/shift，然后再创建顶点。

这些变量是一些键，姑且称它们为“逻辑性变量”；“逻辑”一词强调的是“并非某种实际意义，只是一种标签”；能用逻辑性变量做什么事情，主要取决于代码的设计。

命令 \graph[⟨*graph options*⟩]{⟨*group specification*⟩}；中的 {⟨*group specification*⟩} 是个用花括号构造的层级结构，花括号的套嵌方式决定了各个链、组的处理次序。不过，其中所有的花括号组都会被命令

\tikz@lib@graph@parse@group→P. 1032 读取,而花括号也只是命令 \tikz@lib@graph@parse@group→P. 1032 的参数定界标志，在读取时会被忽略。

在处理过程中,命令 \tikz@lib@graph@parse@group→P. 1032 与 \tikz@lib@graph@graph@group@done→P. 1035 并不会引入额外的组；真正引入组的是

- 命令 \tikz@lib@graph@node@opt@normal→P. 1046 会把创建新顶点的过程限制在一个花括号组中。
- 命令 \tikz@lib@graph@scope→P. 1038 引入 \begingroup 与 \endgroup 的组合，对“用作顶点的链组”的处理限制在这个组中。
- \tikz@lib@graph@main@parser→P. 1033 与 \tikz@lib@graph@graph@done→P. 1034 引入 \begingroup 与 \endgroup, 对一个链的处理限制在这个组中。

注意以上 3 种命令引入的组会形成套嵌。

上述作为变量的键都是局部的，它们的值受到组的影响。在计算键值时可能会用到某些宏，有的宏可能是全局的 (为了超出组的限制)。一开始，前述变量有各自的初始值，之后，它们的值就随着组的开始、结束而变化。

\tikz@lib@graph@setup@placement{⟨*ld*⟩}{⟨*lw*⟩}{⟨*c*⟩}{⟨*e*⟩}{⟨*w*⟩}{⟨*ds*⟩}

本命令的定义是：

```
\def\tikz@lib@graph@setup@placement#1#2#3#4#5#6{%
  \pgfkeyssetvalue{/tikz/graphs/placement/local depth}{#1}%
  \pgfkeyssetvalue{/tikz/graphs/placement/local width}{#2}%
  \pgfkeyssetvalue{/tikz/graphs/placement/chain count}{#3}%
  \pgfkeyssetvalue{/tikz/graphs/placement/element count}{#4}%
  \pgfkeyssetvalue{/tikz/graphs/placement/width}{#5}%
  \pgfkeyssetvalue{/tikz/graphs/placement/depth}{#6}%
}%
```

/tikz/graphs/placement/place (no value)

本选项的定义是：

```
\tikzgraphsset{
  placement/.cd,
  place/.code={%
    \pgfkeys{/tikz/graphs/placement/compute position}%
    \aftergroup\tikz@lib@graph@reset@locals%
    \pgfkeyssetvalue{/tikz/graphs/placement/element count}{0}%
    \pgfkeyssetvalue{/tikz/graphs/placement/chain count}{0}%
    \pgfkeyssetvalue{/tikz/graphs/placement/depth}{0}%
    \pgfkeyssetvalue{/tikz/graphs/placement/width}{0}%
    \pgfkeyssetvalue{/tikz/graphs/placement/local depth}{0}%
    \pgfkeyssetvalue{/tikz/graphs/placement/local width}{0}%
  },
}
\def\tikz@lib@graph@reset@locals{%
  \gdef\tikz@lib@graph@group@depth{0}%
  \gdef\tikz@lib@graph@group@width{0}%
}%
```

/tikz/graphs/placement/compute position=⟨*code*⟩

这个键被 placement/place 调用，应当提前定义它，例如

```
\tikzgraphsset{
  placement/compute position/.code={⟨code⟩}
}
```

在 ⟨*code*⟩ 中可以使用任何代码，不过，⟨*code*⟩ 至少应当能计算一个坐标位置，并使用平移变换平移到这个位置。

58.17.1 各个变量的变化情况

逻辑性变量有以下变化情况：

1. 在 `\graph[⟨graph options⟩]{⟨group specification⟩};` 的处理中，先执行 ⟨*graph options*⟩, 然后

```
\pgfkeyssetvalue{/tikz/graphs/placement/depth}{0}%
\pgfkeyssetvalue{/tikz/graphs/placement/width}{0}%
\pgfkeyssetvalue{/tikz/graphs/placement/level}{0}%
\tikz@lib@graph@start@hint@group%
  \tikz@lib@graph@parse@group{⟨group specification⟩}%
\tikz@lib@graph@end@hint@group
```

注意，链组的形式是 `{[⟨options⟩]⟨group content⟩}`, 用户可以在 ⟨*options*⟩ 中指定逻辑性变量的值。

`\tikz@lib@graph@start@hint@group`

本命令的定义是：

```
\def\tikz@lib@graph@start@hint@group{%
  \pgfkeyssetvalue{/tikz/graphs/placement/local depth}{0}%
  \pgfkeyssetvalue{/tikz/graphs/placement/local width}{0}%
  \pgfkeyssetvalue{/tikz/graphs/placement/chain count}{0}%
  \pgfkeyssetvalue{/tikz/graphs/placement/element count}{0}%
}%
```

注意本命令定义了变量 (键) `placement/local depth`, `placement/local width`.

`\tikz@lib@graph@end@hint@group`

本命令的定义是：

```
\def\tikz@lib@graph@end@hint@group{%
  % Get local depth and width outside
  \xdef\tikz@lib@graph@group@depth{\pgfkeysvalueof
  ↪  {/tikz/graphs/placement/local depth}}
  \xdef\tikz@lib@graph@group@width{\pgfkeysvalueof
  ↪  {/tikz/graphs/placement/local width}}
}%
```

2. 当用 `\node` 命令创建新顶点时：

```
\pgfkeysgetvalue{/tikz/graphs/placement/level}\tikz@temp%
\c@pgf@counta=\tikz@temp\relax%
\advance\c@pgf@counta by1\relax%
\edef\tikz@temp{\the\c@pgf@counta}%
\pgfkeyslet{/tikz/graphs/placement/level}\tikz@temp%
...
% 组之前
{
  \pgfkeys{/tikz/graphs/placement/place}%
  用 \node 命令创建顶点
}
% 按 placement/place 的定义，这里有 \tikz@lib@graph@reset@locals，即
%\gdef\tikz@lib@graph@group@depth{0}%
%\gdef\tikz@lib@graph@group@width{0}%
```

```
\tikz@lib@graph@placement@update
% 组之后
```

键	组开始之前的值	组开始之后，创建顶点之前的值	创建顶点之后，组结束之前的值	组结束之后的值
element count	e	0	0	$e+1$
chain count	c	0	0	c
width	w	0	0	$lnw+w$
depth	d	0	0	d
level	$l=l+1$	l	l	l
logical node width	lnw	lnw	lnw	lnw
logical node depth	lnd	lnd	lnd	lnd
local width	lw	0	0	$lnw+lw$
local depth	ld	0	0	$\max(lnd,ld)$

\tikz@lib@graph@placement@update

本命令的定义是：

```
\def\tikz@lib@graph@placement@update{%
  \pgfkeys{/tikz/graphs/placement/logical node depth/.expand once=
  ↪  \tikz@lib@graph@name}
  \let\tikz@lib@graph@node@depth\pgfmathresult
  \pgfkeys{/tikz/graphs/placement/logical node width/.expand once=
  ↪  \tikz@lib@graph@name}
  \let\tikz@lib@graph@node@width\pgfmathresult
  % This is a single node, so update the lengths accordingly
  \pgfkeysgetvalue{/tikz/graphs/placement/width}\tikz@temp@h%
  \pgfkeysgetvalue{/tikz/graphs/placement/local width}\tikz@temp@lh%
  \pgfkeysgetvalue{/tikz/graphs/placement/local depth}\tikz@temp@lw%
  \pgfmathsetmacro\tikz@temp@h{\tikz@lib@graph@node@width+\tikz@temp@h}
  \pgfmathsetmacro\tikz@temp@lh{\tikz@lib@graph@node@width+\tikz@temp@lh}
  \pgfmathsetmacro\tikz@temp@lw{max(\tikz@lib@graph@node@depth,\tikz@temp@lw)}
  \pgfkeyslet{/tikz/graphs/placement/width}\tikz@temp@h%
  \pgfkeyslet{/tikz/graphs/placement/local width}\tikz@temp@lh%
  \pgfkeyslet{/tikz/graphs/placement/local depth}\tikz@temp@lw%
  %
  \pgfkeysgetvalue{/tikz/graphs/placement/element count}\tikz@temp%
  \c@pgf@counta=\tikz@temp\relax%
  \advance\c@pgf@counta by1\relax%
  \edef\tikz@temp{\the\c@pgf@counta}%
  \pgfkeyslet{/tikz/graphs/placement/element count}\tikz@temp%
}%
```

3. 对于一个链 ⟨*a chain*⟩ 的处理：

```
% 组之前
\begingroup%
  \pgfkeyssetvalue{/tikz/graphs/placement/local depth}{0}%
  \pgfkeyssetvalue{/tikz/graphs/placement/local width}{0}%
  %
  处理一个链 ⟨a chain⟩
  %
```

```
  \xdef\tikz@lib@graph@chain@depth{\pgfkeysvalueof{/tikz/graphs/placement/local
  ↪  depth}}
  \xdef\tikz@lib@graph@chain@width{\pgfkeysvalueof{/tikz/graphs/placement/local
  ↪  width}}
\endgroup%
\tikz@lib@graph@placement@after@chain@update
% 组之后
```

键	组开始之前的值	组开始之后，处理链之前的值	处理链之后，组结束之前的值	组结束之后的值
element count	e	e	e'	e
chain count	c	c	c	$c+1$
width	w	w	w'	w
depth	d	d	d'	$d+ld'$
level	l	l	l'	l
logical node width	lnw	lnw	lnw	lnw
logical node depth	lnd	lnd	lnd	lnd
local width	lw	0	lw'	$\max(lw, lw')$
local depth	ld	0	ld'	$ld+ld'$

\tikz@lib@graph@placement@after@chain@update

本命令的定义是：

```
\def\tikz@lib@graph@placement@after@chain@update{%
  \pgfkeysgetvalue{/tikz/graphs/placement/depth}\tikz@temp@w%
  \pgfkeysgetvalue{/tikz/graphs/placement/local width}\tikz@temp@lh%
  \pgfkeysgetvalue{/tikz/graphs/placement/local depth}\tikz@temp@lw%
  \pgfmathsetmacro\tikz@temp@w{\tikz@lib@graph@chain@depth+\tikz@temp@w}%
  \pgfmathsetmacro\tikz@temp@lw{\tikz@lib@graph@chain@depth+\tikz@temp@lw}%
  \pgfmathsetmacro\tikz@temp@lh{max(\tikz@lib@graph@chain@width,\tikz@temp@lh)
  ↪  }%
  \pgfkeyslet{/tikz/graphs/placement/depth}\tikz@temp@w%
  \pgfkeyslet{/tikz/graphs/placement/local width}\tikz@temp@lh%
  \pgfkeyslet{/tikz/graphs/placement/local depth}\tikz@temp@lw%
  %
  \pgfkeysgetvalue{/tikz/graphs/placement/chain count}\tikz@temp%
  \c@pgf@counta=\tikz@temp\relax%
  \advance\c@pgf@counta by1\relax%
  \edef\tikz@temp{\the\c@pgf@counta}%
  \pgfkeyslet{/tikz/graphs/placement/chain count}\tikz@temp%
}%
```

4. 对于“用作顶点的链组”`{[`⟨*options*⟩`]` ⟨*group content*⟩`}` 的处理：

```
% 组之前
\begingroup%
  \tikz@lib@graph@start@hint@group%
      \tikz@lib@graph@parse@group{[⟨options⟩]⟨group content⟩ }%
  \tikz@lib@graph@end@hint@group%
\endgroup%
\tikz@lib@graph@hint@aftergroup%
% 组之后
```

注意，链组的形式是 {[⟨*options*⟩]⟨*group content*⟩}, 用户可以在 ⟨*options*⟩ 中指定逻辑性变量的值。

键	组开始之前的值	组开始之后，处理链组之前的值	处理链组之后，组结束之前的值	组结束之后的值
element count	e	0 或用户值	0 或用户值	$e+1$
chain count	c	0 或用户值	c'	c
width	w	w 或用户值	w'	$w+lw'$
depth	d	d 或用户值	d'	d
level	l	l 或用户值	l'	l
logical node width	lnw	lnw 或用户值	lnw 或用户值	lnw
logical node depth	lnd	lnd 或用户值	lnd 或用户值	lnd
local width	lw	0 或用户值	lw'	$lw+lw'$
local depth	ld	0 或用户值	ld'	$\max(ld, ld')$

\tikz@lib@graph@hint@aftergroup

本命令的定义是：

```
\def\tikz@lib@graph@hint@aftergroup{%
  \pgfkeysgetvalue{/tikz/graphs/placement/width}\tikz@temp@h%
  \pgfkeysgetvalue{/tikz/graphs/placement/local width}\tikz@temp@lh%
  \pgfkeysgetvalue{/tikz/graphs/placement/local depth}\tikz@temp@lw%
  \pgfmathsetmacro\tikz@temp@h{\tikz@lib@graph@group@width+\tikz@temp@h}
  \pgfmathsetmacro\tikz@temp@lh{\tikz@lib@graph@group@width+\tikz@temp@lh}
  \pgfmathsetmacro\tikz@temp@lw{max(\tikz@lib@graph@group@depth,\tikz@temp@lw)
  ↪  }
  \pgfkeyslet{/tikz/graphs/placement/width}\tikz@temp@h%
  \pgfkeyslet{/tikz/graphs/placement/local width}\tikz@temp@lh%
  \pgfkeyslet{/tikz/graphs/placement/local depth}\tikz@temp@lw%
  %
  \pgfkeysgetvalue{/tikz/graphs/placement/element count}\tikz@temp%
  \c@pgf@counta=\tikz@temp\relax%
  \advance\c@pgf@counta by1\relax%
  \edef\tikz@temp{\the\c@pgf@counta}%
  \pgfkeyslet{/tikz/graphs/placement/element count}\tikz@temp%
}%
```

58.17.2 各个变量的意义

/tikz/graphs/placement/element count

这个键的通常意义仅限于“处理一个链的 \begingroup 与 \endgroup 组合之内”。

假设下面的组

```
{[⟨options⟩] ... ⟨a chain⟩ ...}
```

是包含链 ⟨*a chain*⟩ 的最小的组。

在开始处理 ⟨*a chain*⟩ 的第 1 个顶点前，这个键的值是 i_0, 这个 i_0 就是用户在 ⟨*options*⟩ 中指定的 element count 值，或者是 graphs 库提供的初始值 0.

每当用 \node 命令创建链 ⟨*a chain*⟩ 上的一个顶点后，或者处理完毕链 ⟨*a chain*⟩ 上的一个“用作顶点的链组”后，这个键的值加 1.

注意链 ⟨*a chain*⟩ 上的以圆括号开头的引用形式的顶点不会引起这个键值的变化。

当链 ⟨*a chain*⟩ 上的各个非引用的顶点被处理完毕时，这个键的值就是这些顶点的个数。
在内部计算中，这个键借助计数器 `\c@pgf@counta` 来变化。

`/tikz/graphs/placement/chain count`

这个键的正常意义仅限于一个链组之内 (因为每个链组的处理都被限制在了组中)。
考虑下面的链组

```
{[⟨options⟩] ...}
```

中所包含的那些 (按层次结构是) “最大的” 链。
在开始处理链组中的第 1 个链之前 (包裹链的 `\begingroup` 之前)，这个键的值是 i_0, 这个 i_0 就是用户在 ⟨*options*⟩ 中指定的 `chain count` 值，或者是 graphs 库提供的初始值 0.
每当处理完毕一个链后 (包裹链的 `\endgroup` 之后)，这个键的值加 1.
当链组内的各个链被处理完毕时，这个键的值就是链组内的链的个数。
在内部计算中，这个键借助计数器 `\c@pgf@counta` 来变化。

`/tikz/graphs/placement/logical node width`=⟨顶点 *node* 的全名⟩

在 graphs 库的默认处理流程中，这个键执行一个计算，计算结果被保存到宏 `\pgfmathresult` 中，代表当前顶点的所占据的 “水平宽度”。
在 graphs 库中是这样使用这个键的：

```
\pgfkeys{/tikz/graphs/placement/logical node depth/.expand once=
↪  \tikz@lib@graph@name}
\let\tikz@lib@graph@node@depth\pgfmathresult
\pgfkeys{/tikz/graphs/placement/logical node width/.expand once=
↪  \tikz@lib@graph@name}
\let\tikz@lib@graph@node@width\pgfmathresult
\pgfmathsetmacro\tikz@temp@h{\tikz@lib@graph@node@width+\tikz@temp@h}
\pgfmathsetmacro\tikz@temp@lh{\tikz@lib@graph@node@width+\tikz@temp@lh}
```

所以这个键保存的代码应当能够定义宏 `\pgfmathresult` 的值。如果这个键保存一个函数，那么这个函数应当能够把 ⟨顶点 *node* 的全名⟩，即 `\tikz@lib@graph@name` 的展开值，作为参数。
见 `\tikz@lib@graph@placement@update` → P. 1093.
这个键的初始值是：

```
\tikzgraphsset{
  placement/logical node depth/.code=\def\pgfmathresult{1}
}
```

如果把这个键定义为一个函数，那么在使用这个键时，它的参数通常是 ⟨顶点 *node* 的全名⟩，即当前 `\tikz@lib@graph@name` 的展开值，例如

```
\tikzgraphsset{
  placement/logical node depth/.code=\tikz@lib@graph@width@sep{#1}{1em}
}
```

在 graphs 库中，函数 `\tikz@lib@graph@width@sep{`⟨*node name*⟩`}{`⟨*dimension*⟩`}` 先计算 ⟨*node name*⟩ 的 `west` 点与 `east` 点之间的水平距离 ⟨*h*⟩，再执行 `\pgfmathparse{`⟨*dimension*⟩`+`⟨*h*⟩`}`. 也就是说，此时的键 `logical node depth` 计算一个 “水平间隔”。
也可以重定义键 `logical node depth`, 让它计算别的东西，无论计算什么，都要把计算结果保存到宏 `\pgfmathresult` 中。

`/tikz/graphs/placement/logical node depth`=⟨顶点 *node* 的全名⟩

类似 `/tikz/graphs/placement/logical node width`.

当在一个直角坐标系中描述一个点时，可以使用横坐标与纵坐标。类似地，在一个图中描述一个顶点时，可以使用“逻辑宽度”(logical width) 与“逻辑深度”(logical depth)。如果把图中的顶点排布到直角网格的格点上，那么逻辑宽度、逻辑深度就相当于横坐标、纵坐标。如果对直角网格做一个拓扑变换，那么再用横坐标、纵坐标来描述一个顶点就不太合适了，所以这里使用逻辑宽度、逻辑深度这种说法。

/tikz/graphs/placement/width

在 graphs 库的默认处理中，这个键保存的数值代表当前点 (在整个图中) 所处的逻辑宽度，是某些顶点“占据的水平宽度的累加”。

/tikz/graphs/placement/depth

在 graphs 库的默认处理中，这个键保存的数值代表当前点 (在整个图中) 所处的逻辑深度，是某些顶点“占据的纵向深度的累加”。

/tikz/graphs/placement/level

在 graphs 库的默认处理中，这个键保存的整数代表当前点 (在整个图中) 所处的层。
每当创建一个 (不以圆括号开头的) 新顶点前，会令这个键的值加 1.
这个键值受到组的限制，可以当作“层编号”，在默认下，层编号从 1 开始。
在内部计算中，这个键借助计数器 \c@pgf@counta 来变化。

/tikz/graphs/placement/local width

当用 \node 创建一个顶点、退出 } 组、执行 \tikz@lib@graph@placement@update →P. 1093 后，这个键保存刚才创建的顶点“所延伸到”的逻辑宽度。
当处理完毕一个链、退出 \endgroup 组、执行 \tikz@lib@graph@placement@after@chain@update →P. 1094 后，这个键保存刚才处理的 (整个) 链“所延伸到”的逻辑宽度。
当处理完毕一个链组、退出 \endgroup 组、执行 \tikz@lib@graph@hint@aftergroup →P. 1095 后，这个键保存刚才处理的 (整个) 链组“所延伸到”的逻辑宽度。
注意在进入组后、处理对象之前，这个键的值会被重置。

/tikz/graphs/placement/local depth

当用 \node 创建一个顶点、退出 } 组、执行 \tikz@lib@graph@placement@update →P. 1093 后，这个键保存刚才创建的顶点“所延伸到”的逻辑深度。
当处理完毕一个链、退出 \endgroup 组、执行 \tikz@lib@graph@placement@after@chain@update →P. 1094 后，这个键保存刚才处理的 (整个) 链“所延伸到”的逻辑深度。
当处理完毕一个链组、退出 \endgroup 组、执行 \tikz@lib@graph@hint@aftergroup →P. 1095 后，这个键保存刚才处理的 (整个) 链组“所延伸到”的逻辑深度。
注意在进入组后、处理对象之前，这个键的值会被重置。

\tikz@lib@graph@group@width

当用 \node 创建一个顶点、退出 } 组后、执行 \tikz@lib@graph@placement@update →P. 1093 前，这个键会被全局地定义为 0, 见 \tikz@lib@graph@reset@locals.
当处理完毕链组后，退出 \endgroup 组前，全局定义

```
\xdef\tikz@lib@graph@group@width{\pgfkeysvalueof{/tikz/graphs/placement/local
↪  width}}
```

见 \tikz@lib@graph@end@hint@group →P. 1092.

\tikz@lib@graph@group@depth

当用 \node 创建一个顶点、退出 } 组后、执行 \tikz@lib@graph@placement@update →P. 1093 前，这个键会被全局地定义为 0, 见 \tikz@lib@graph@reset@locals.

当处理完毕链组后，退出 \endgroup 组前，全局定义

```
\xdef\tikz@lib@graph@group@depth{\pgfkeysvalueof{/tikz/graphs/placement/local
↪  depth}}
```

见 \tikz@lib@graph@end@hint@group → P. 1092.

\tikz@lib@graph@chain@width

当处理完毕一个链后，退出 \endgroup 组前，全局定义

```
\xdef\tikz@lib@graph@chain@width{\pgfkeysvalueof{/tikz/graphs/placement/local
↪  width}}
```

见 \tikz@lib@graph@graph@done → P. 1034.

\tikz@lib@graph@chain@depth

当处理完毕一个链后，退出 \endgroup 组前，全局定义

```
\xdef\tikz@lib@graph@chain@depth{\pgfkeysvalueof{/tikz/graphs/placement/local
↪  depth}}
```

见 \tikz@lib@graph@graph@done → P. 1034.

58.17.3 手工指定顶点位置

/tikz/graphs/no placement

这个选项的定义是：

```
\tikzgraphsset{
  no placement/.style={
    placement/place,
    placement/compute position/.code=%
  }
}
```

这个选项的作用是恢复逻辑性变量的初始值，并清空 placement/compute position 保存的代码，这样就不会计算顶点位置，所有顶点都把原点作为锚定点。

/tikz/graphs/x=⟨x⟩ (initially 0)

这个选项导致

```
\pgfqkeys{/tikz}{at/.expand once=(⟨x⟩,⟨y⟩)}%
```

/tikz/graphs/y=⟨y⟩ (initially 0)

这个选项导致

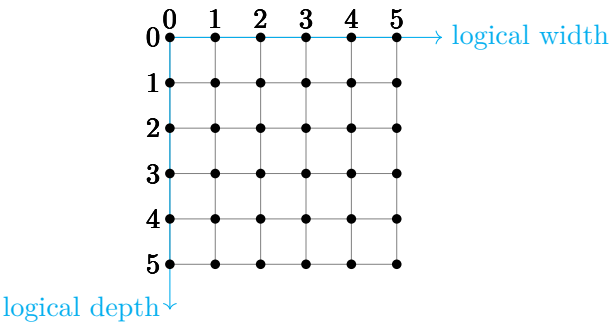
```
\pgfqkeys{/tikz}{at/.expand once=(⟨x⟩,⟨y⟩)}%
```

也可以使用 /tikz/at 来指定顶点 node 的锚定点。

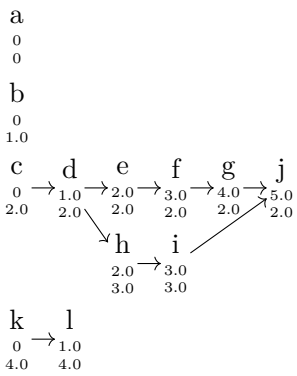
58.17.4 排布策略：Cartesian placement

/tikz/graphs//tikz/graphs/Cartesian placement (no value)

这个选项引入一种自动排布顶点的策略，并且是 graphs 库默认的排布策略。这个策略的排布结果类似直角网格的格点，其中顶点的逻辑宽度、逻辑深度如下图：



例如：




```
\tikz
  \graph [nodes={align=center, inner sep=1pt}, grow right=7mm,
      typeset={\tikzgraphnodetext\\[-8pt]
        \tiny\mywidth\\[-10pt]\tiny\mydepth},
      placement/compute position/.append code=
        \pgfkeysgetvalue{/tikz/graphs/placement/width}{\mywidth}
        \pgfkeysgetvalue{/tikz/graphs/placement/depth}{\mydepth}]
  { a,
    b,
    c -> d -> { e -> f -> g,
                h -> i }      -> j,
    k -> l };
```

策略 Cartesian placement 的定义是：

```
\tikzgraphsset{
  Cartesian placement/.style={
    placement/place,
    placement/compute position/.code=\tikz@lib@graph@linear@pos%
  },
  chain shift/.initial={(1,0)},
  group shift/.initial={(0,-1)},
%... 略
}
```

58.17.4.1 有关的选项

\tikz@lib@graph@linear@pos

本命令的定义是：

```
\def\tikz@lib@graph@linear@pos{%
  \pgfkeysgetvalue{/tikz/graphs/chain shift}\tikz@temp
  \expandafter\tikz@scan@one@point\expandafter\pgf@process\tikz@temp
  \pgf@process{\pgfpointscale{\pgfkeysvalueof{/tikz/graphs/placement/width}}{}}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgfkeysgetvalue{/tikz/graphs/group shift}\tikz@temp
  \expandafter\tikz@scan@one@point\expandafter\pgf@process\tikz@temp
  \pgf@process{\pgfpointscale{\pgfkeysvalueof{/tikz/graphs/placement/depth}}{}}%
  \advance\pgf@xa by\pgf@x%
  \advance\pgf@ya by\pgf@y%
  \edef\tikz@lib@graph@shift{(\the\pgf@xa,\the\pgf@ya)}
  \pgfkeys{/tikz/graphs/nodes/.expanded={shift={\tikz@lib@graph@shift}}}
}%
```

假设选项值

/tikz/graphs/chain shift=⟨*coordinate1*⟩ 和 /tikz/graphs/group shift=⟨*coordinate2*⟩

其中的⟨*coordinate1*⟩ 和 ⟨*coordinate2*⟩ 都是 Ti*k*Z 坐标，或者是展开为 Ti*k*Z 坐标的宏，将被 \tikz@scan@one@point→ 处理。

假设某个顶点的逻辑宽度是 w(不带长度单位), 逻辑深度是 d(不带长度单位), 本命令计算平移向量

$$w \cdot \langle coordinate1 \rangle + d \cdot \langle coordinate2 \rangle,$$

然后执行

```
\pgfkeys{/tikz/graphs/nodes/.expanded={shift={(⟨ 计算的坐标尺寸, 带单位⟩)}}}
```

将平移选项保存到样式 tikz/graphs/@nodes styling 中。

在 chain shift, group shift 的初始值下，计算出来的平移向量是

$$w \cdot (1,0) + d \cdot (0,-1) = (w,-d).$$

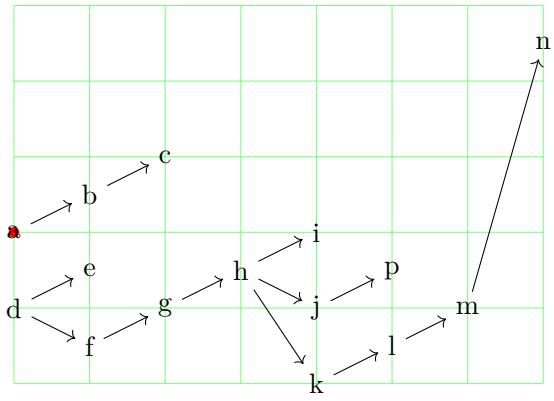
/tikz/graphs/chain shift=⟨*coordinate*⟩ (no default, initially (1,0))

见 \tikz@lib@graph@linear@pos→P. 1099.

⟨*coordinate*⟩ 是 Ti*k*Z 的坐标，将被 \tikz@scan@one@point→P. 710 解析。

本选项的初始值是 (1,0).

如果 ⟨*coordinate*⟩ 是极坐标形式 (⟨*angle*⟩:⟨*t*⟩)，那么通过简单的计算可知，若原来的某个边是角度为 0 度的水平边，那么这个边会变成角度为 ⟨*angle*⟩ 度的边，并且这个边的始点中心与终点中心的间距就是 ⟨*t*⟩；如果 ⟨*t*⟩ 不带长度单位就默认其单位是 cm.




```
\tikz {
  \draw [green!50] (0,-2) grid (7,3);
  \fill [red] (0,0) circle (2pt);
  \graph [chain shift={(1,0.5)}] {
    a -> b -> c;
    d -> { e,
           f -> g -> h -> { i,
                            j -> p,
                            k -> l-> m }}
      -> n[not source]
  };
  \draw [->] (m)--(n);
}
```

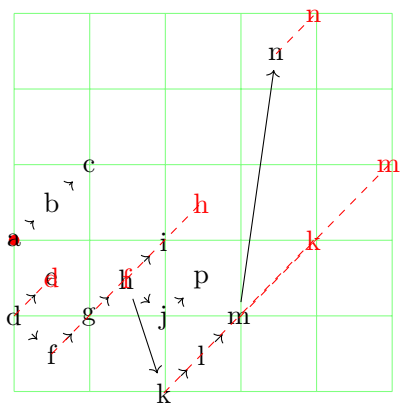
/tikz/graphs/group shift=⟨*coordinate*⟩ (no default, initially (0,-1))

见 \tikz@lib@graph@linear@pos→P. 1099.

⟨*coordinate*⟩ 是 Ti*k*Z 的坐标，将被 \tikz@scan@one@point→P. 710 解析。

本选项的初始值是 (0,-1).

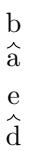
如果 ⟨*coordinate*⟩ 是极坐标形式 (⟨*angle*⟩:⟨*t*⟩)，那么通过简单的计算可知，若原来的某个边是角度为 ±90° 的竖直边，那么这个边会被旋转角度 ⟨*angle*⟩，并且这个边的始点中心与终点中心的间距就是 ⟨*t*⟩；如果 ⟨*t*⟩ 不带长度单位就默认其单位是 cm.



```
\tikz {
  \draw [green!50] (0,-2) grid (5,3);
  \fill [red] (0,0) circle (2pt);
  \graph [chain shift=(45:7mm),] {
    a -> b -> c;
    d -> { e,
           f -> g -> h -> { i,
                            j -> p,
                            k -> l-> m }}
      -> n[not source]
  };
  \draw [->] (m)--(n);
  \foreach \dingdian/\shendu in
    {d/1,f/2,h/2,k/4,m/4,n/1}
    \draw [dashed,red] (\dingdian.center)
      --++($\shendu*(-45:7mm)+(0,\shendu)$)
      node{\dingdian};
}
```

/tikz/graphs/grow up=⟨*distance*⟩ (default 1)

本选项等效于 chain shift={(0,⟨*distance*⟩)}, 效果是使得链向上展开，链的上下相邻的两个顶点的间距（顶点中心之间的间距）是 ⟨*distance*⟩.



```
\tikz \graph [grow up=5mm]
{ a -> b, d -> e};
```

本选项的定义是：

```
\tikzgraphsset{
%...
  grow up/.style={
    placement/place,
    chain shift={(0,#1)},
    @auto anchor horizontal=center,
    placement/logical node width/.code=\def\pgfmathresult{1}
  },
  grow up/.default=1,
%...
}
```

/tikz/graphs/grow down=⟨*distance*⟩ (default 1)

本选项使得链向下展开，链的上下相邻的两个顶点的间距（顶点中心之间的间距）是 ⟨*distance*⟩.

本选项的定义是：

```
\tikzgraphsset{
%...
  grow down/.style={
    placement/place,
    chain shift={(0,-#1)},
    @auto anchor vertical=center,
    placement/logical node width/.code=\def\pgfmathresult{1}
  },
  grow down/.default=1,
%...
}
```

/tikz/graphs/grow left=⟨*distance*⟩ (default 1)

本选项使得链向左展开，链的左右相邻的两个顶点的间距（顶点中心之间的间距）是 ⟨*distance*⟩.

c ← b ← a

```
\tikz \graph [grow left=7mm] { a -> b -> c};
```

本选项的定义是：

```
\tikzgraphsset{
%...
  grow left/.style={
    placement/place,
    chain shift={(-#1,0)},
    @auto anchor horizontal=center,
    placement/logical node width/.code=\def\pgfmathresult{1}
  },
  grow left/.default=1,
%...
}
```

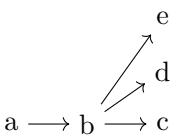
/tikz/graphs/grow right=⟨*distance*⟩ (default 1)

本选项使得链向右展开，链的左右相邻的两个顶点的间距（顶点中心之间的间距）是 ⟨*distance*⟩.
本选项的定义是：

```
\tikzgraphsset{
%...
  grow right/.style={
    placement/place,
    chain shift={(#1,0)},
    @auto anchor horizontal=center,
    placement/logical node width/.code=\def\pgfmathresult{1}
  },
  grow right/.default=1,
%...
}
```

/tikz/graphs/branch up=⟨*distance*⟩ (default 1)

本选项使得链的分支向上展开，相邻两个分支的上下间距（顶点中心之间的间距）是 ⟨*distance*⟩.




```
\tikz \graph [branch up=7mm] { a -> b -> {c, d, e} };
```

本选项的定义是：

```
\tikzgraphsset{
%...
  branch up/.style={
    placement/place,
    group shift={(0,#1)},
    @auto anchor vertical=center,
    placement/logical node depth/.code=\def\pgfmathresult{1}
  },
  branch up/.default=1,
%...
}
```

/tikz/graphs/branch down=⟨*distance*⟩ (default 1)

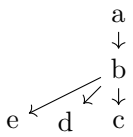
本选项使得链的分支向下展开，相邻两个分支的上下间距（顶点中心之间的间距）是 ⟨*distance*⟩.
本选项的定义是：

```
\tikzgraphsset{
%...
  branch down/.style={
    placement/place,
    group shift={(0,-#1)},
    @auto anchor vertical=center,
    placement/logical node depth/.code=\def\pgfmathresult{1}
  },
  branch down/.default=1,
%...
}
```

/tikz/graphs/branch left=⟨*distance*⟩ (default 1)

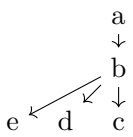
在使用选项 grow down 或 grow up 的情况下，本选项使得链的分支向左展开，相邻两个分支的左右间距（顶点中心之间的间距）是 ⟨*distance*⟩.

```
\tikz \graph [branch left=7mm, grow down=7mm] { a -> b -> {c, d, e}};
```



上面图形中最后一行的三个字母有点凹凸，如下修改对齐：

```
\tikz \graph [branch left=7mm, grow down=7mm] {[nodes/.expanded=
↪  {anchor=base}] a -> b -> {c, d, e}};
```



本选项的定义是：

```
\tikzgraphsset{
%...
  branch left/.style={
    placement/place,
    group shift={(-#1,0)},
    @auto anchor horizontal=center,
    placement/logical node depth/.code=\def\pgfmathresult{1}
  },
  branch left/.default=1,
%...
}
```

/tikz/graphs/branch right=⟨*distance*⟩ (default 1)

在使用选项 grow down 或 grow up 的情况下，本选项使得链的分支向右展开，相邻两个分支的左右间距（顶点中心之间的间距）是 ⟨*distance*⟩.
本选项的定义是：

```
\tikzgraphsset{
%...
  branch right/.style={
    placement/place,
    group shift={(#1,0)},
    @auto anchor horizontal=center,
    placement/logical node depth/.code=\def\pgfmathresult{1}
  },
```

```
  branch right/.default=1,
%...
}
```

/tikz/graphs/@auto anchor horizontal=⟨anchor ⟨h⟩⟩ (initially center)

这是个样式，参数 ⟨anchor ⟨h⟩⟩ 可以是 center, north, west, south, east. 初始值是 center. 本选项的定义是：

```
\tikzgraphsset{
  @auto anchor horizontal/.style={
    nodes={anchor=\csname tikz@lib@graph@auto@\tikz@lib@graph@auto@h @
    ↪  \tikz@lib@graph@auto@v\endcsname},
    /utils/exec=\def\tikz@lib@graph@auto@h{#1}
  },
}

\def\tikz@lib@graph@auto@h{center}%
\def\tikz@lib@graph@auto@v{center}%

\def\tikz@lib@graph@auto@center@center{center}%
\def\tikz@lib@graph@auto@west@center{west}%
\def\tikz@lib@graph@auto@east@center{east}%
\def\tikz@lib@graph@auto@center@north{north}%
\def\tikz@lib@graph@auto@west@north{north west}%
\def\tikz@lib@graph@auto@east@north{north east}%
\def\tikz@lib@graph@auto@center@south{south}%
\def\tikz@lib@graph@auto@west@south{south west}%
\def\tikz@lib@graph@auto@east@south{south east}%
```

执行这个样式会把

```
anchor=\csname tikz@lib@graph@auto@\tikz@lib@graph@auto@h @\tikz@lib@graph@auto@v
↪  \endcsname},
/utils/exec=\def\tikz@lib@graph@auto@h{⟨anchor ⟨h⟩⟩}
```

添加到样式 /tikz/graphs/@nodes styling 中，用于指定新创建顶点的 anchor 位置。

/tikz/graphs/@auto anchor vertical=⟨anchor ⟨h⟩⟩ (initially center)

这是个样式，参数 ⟨anchor ⟨h⟩⟩ 可以是 center, north, west, south, east. 初始值是 center. 本选项的定义是：

```
\tikzgraphsset{
  @auto anchor vertical/.style={
    nodes={anchor=\csname tikz@lib@graph@auto@\tikz@lib@graph@auto@h @
    ↪  \tikz@lib@graph@auto@v\endcsname},
    /utils/exec=\def\tikz@lib@graph@auto@v{#1}
  },
}
```

58.17.4.2 在排布时考虑顶点 node 的尺寸

grow up，branch right 等选项在排布顶点时不考虑顶点 node 的尺寸，只是把顶点的中心对准某个网格位置，所以当顶点 node 的尺寸过大时就会显得太拥挤甚至发生重叠。

xxxx yyyyy

```
\tikz \graph  { xxxx ->[red] yyyyy };
```

在很多情况下，使用下面的选项可以避免“拥挤、重叠”。

\tikz@lib@graph@width@sep{⟨*full node name*⟩}{⟨*expression*⟩}

参数 ⟨*full node name*⟩ 是某个 node 的全名。

参数 ⟨*expression*⟩ 是能被 \pgfmathparse 处理的表达式。

本命令的定义是：

```
\def\tikz@lib@graph@width@sep#1#2{%
  \pgf@process{\pgfpointdiff{\pgfpointanchor{#1}{west}}{\pgfpointanchor{#1}{east
  ↪  }}}%
  \pgfmathparse{#2+\the\pgf@x}%
}%
```

本命令先计算 ⟨*full node name*⟩ 的 west 位置点与 east 位置点之间的间距，然后用 \pgfmathparse 解析这个间距的水平分量 (\the\pgf@x) 与 ⟨*expression*⟩ 的和。

本命令的计算结果保存在 \pgfmathresult 中。

\tikz@lib@graph@depth@sep{⟨*full node name*⟩}{⟨*expression*⟩}

类似 \tikz@lib@graph@width@sep.

本命令的定义是：

```
\def\tikz@lib@graph@depth@sep#1#2{%
  \pgf@process{\pgfpointdiff{\pgfpointanchor{#1}{south}}{\pgfpointanchor{#1}{north
  ↪  }}}%
  \pgfmathparse{#2+\the\pgf@y}%
}%
```

/tikz/graphs/grow right sep=⟨*distance*⟩ (default 1em)

本选项的定义是：

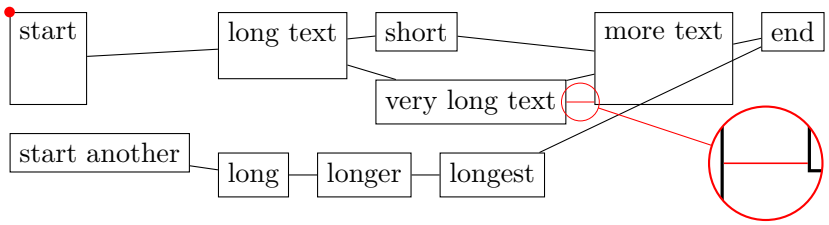
```
\tikzgraphsset{
  grow right sep/.style={
    Cartesian placement,
    chain shift={(1pt,0)},
    @auto anchor horizontal=west,
    placement/logical node width/.code=\tikz@lib@graph@width@sep{##1}{#1}
  },
  grow right sep/.default=1em,
}
```

可见选项 grow right sep 会选定 Cartesian placement 策略。

参数 ⟨*distance*⟩ 将被用作 \tikz@lib@graph@width@sep 的参数，会被命令 \pgfmathparse 解析。注意本样式使用的是 chain shift={(1pt,0)}, 这是 PGF 的 canvas 坐标系的单位向量，所以，如果参数 ⟨*distance*⟩ 不带长度单位，就默认其单位是 pt.

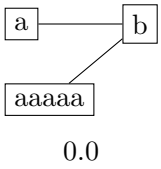
使用本选项后，各个顶点 node 的 west 位置处于其锚定点上 (见 /tikz/graphs/@auto anchor horizontal → P. 1104)。由选项 logical node width 计算出来的、顶点占据的水平宽度，就是顶点的 west 位置点与 east 位置点之间的水平间距加上 ⟨*distance*⟩.

本选项的效果如下：



```
\tikz [spy using outlines={circle, magnification=3, size=1.5cm, connect spies}]{
\graph [grow right sep,branch down sep, nodes=draw] {
  {
    start[text depth=2em],
    start another
  } --
    {
      long text[text depth=1em] --
        {
          short,
          very long text
        } --
          more text[text depth=2em],
      long --
        longer --
          longest
    } --
      end
};
  \fill [red] (0,0) circle (2pt);
  \draw [red, very thin] (very long text.east) -- ++(1em,0);
  \spy [red] on ($(very long text.east)!0.5!(more text.west |- very long text.east)$) in node at
  ↪  (10,-2);
}
```

从本选项的效果上看，一个链上的相邻 2 个顶点之间的间距是 ⟨*distance*⟩; 当链组用作顶点时，⟨*distance*⟩ 是链组之间的间距。例如，对于 `{a,aaaaa}--b`, 其中 `{a,aaaaa}` 与 `b` 各成一组，所以二者的间距是 `aaaaa` 与 `b` 的间距，



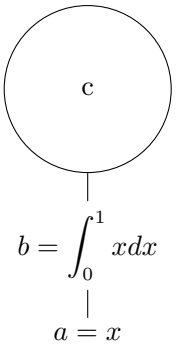

```
\tikz{
  \graph [grow right sep,nodes/.expanded={draw}] { {a,aaaaa} -- b };
  \path let \p1=(aaaaa.east), \p2=(b.west) in
    node [below=2mm]at(current bounding box.south){\pgfmathparse{\x2-
    ↪  \x1-1em}\pgfmathresult};
}
```

`/tikz/graphs/grow left sep=`⟨*distance*⟩ (default `1em`)

longlonglong — longlong — long

```
\tikz \graph [grow left sep] { long -- longlong -- longlonglong };
```

`/tikz/graphs/grow up sep=`⟨*distance*⟩ (default `1em`)

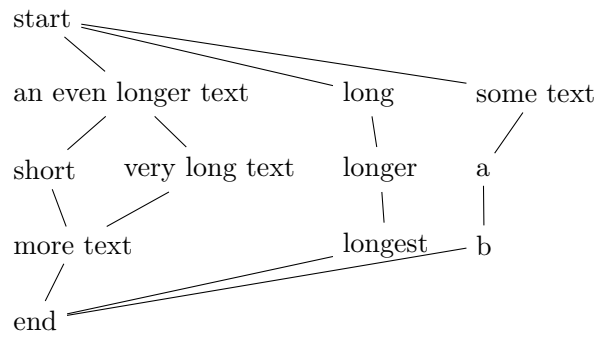



```
\tikz \graph [grow up sep] {
  a / $a=x$ --
  b / {$b=\displaystyle \int_0^1 x dx$} --
  c [draw, circle, inner sep=7mm]
};
```

`/tikz/graphs/grow down sep=`⟨*distance*⟩ (default `1em`)

`/tikz/graphs/branch right sep=`⟨*distance*⟩ (default `1em`)

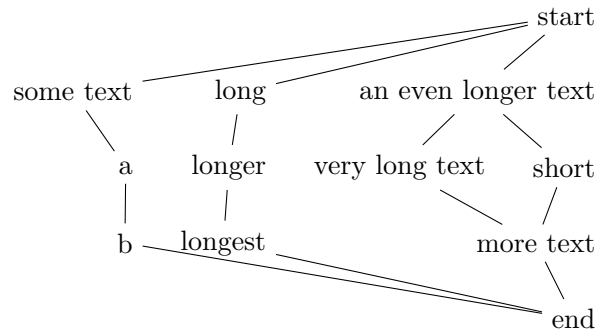
本选项通常需要配合 `grow down` 或 `grow up` 使用。




```
\tikz \graph [grow down, branch right sep] {
  start -- {
    an even longer text -- {short, very long text} -- more text,
    long -- longer -- longest,
    some text -- a -- b
  } -- end
};
```

/tikz/graphs/branch left sep=⟨*distance*⟩ (default 1em)

本选项通常需要配合 grow down 或 grow up 使用。




```
\tikz \graph [grow down, branch left sep] {
  start -- {
    an even longer text -- {short, very long text} -- more text,
    long -- longer -- longest,
    some text -- a -- b
  } -- end
};
```

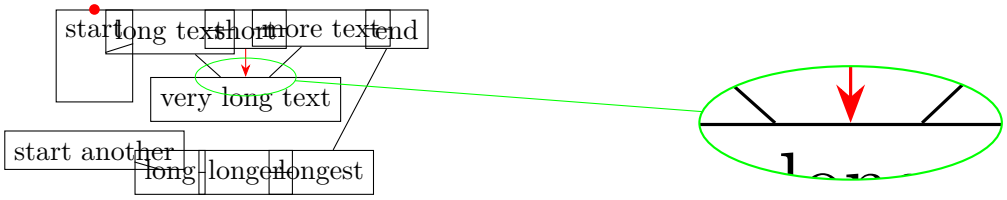
/tikz/graphs/branch up sep=⟨*distance*⟩ (default 1em)

/tikz/graphs/branch down sep=⟨*distance*⟩ (default 1em)

本选项的定义是：

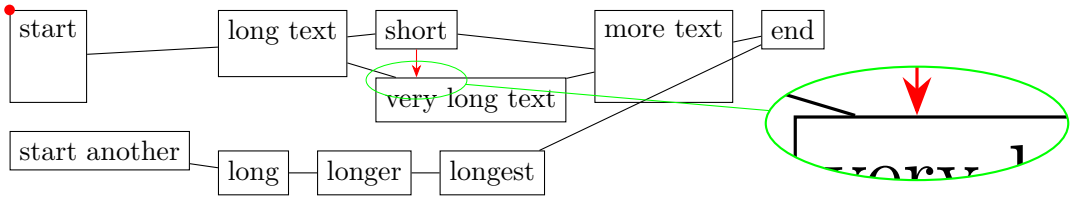
```
\tikzgraphsset{
  branch down sep/.style={
    Cartesian placement,
    group shift={(0,-1pt)},
    @auto anchor vertical=north,
    placement/logical node depth/.code=\tikz@lib@graph@depth@sep{##1}{#1}
  },
  branch down sep/.default=1em,
}
```

可见选项 branch down sep 会选定 Cartesian placement 策略。使用本选项后，各个顶点 node 的 north 位置处于其锚定点上。




```
\tikz [spy using outlines={ellipse, magnification=3, width=4cm, height=1.5cm, connect spies}]{
  \graph [branch down sep, nodes=draw] {
    {start[text depth=2em],start another} -- {
      long text[] -- {short, very long text} -- more text,
      long -- longer -- longest
    } -- end
  };
  \fill [red] (0,0) circle (2pt);
  \draw [-Stealth, red] (short.south) -- ++(0,-1em);
  \spy [green] on ($(short.south)+(0,-1em)$) in node at(10,-1.5);
}
```

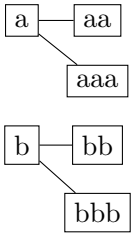
上面例子中，选项 `branch down sep` 会把各个顶点 node 的锚位置设为 `north`，各个顶点 node 所占据的“深度”就是：顶点 node 本身的高度，即从顶点 node 的锚位置 `north` 到锚位置 `south` 的距离，再加上默认的 `1em`.




```
\tikz [spy using outlines={ellipse, magnification=3, width=4cm, height=1.5cm, connect spies}]
{
\graph [grow right sep,branch down sep, nodes=draw] {
  {
    start[text depth=2em],
    start another
  } --
    {
      long text[text depth=1em] --
        {
          short,
          very long text
        } --
          more text[text depth=2em],
      long --
        longer --
          longest
    } --
      end
};
\fill [red] (0,0) circle (2pt);
\draw [-Stealth, red] (short.south) -- ++(0,-1em);
\spy [green] on ($(short.south)+(0,-1em-1pt)$) in node at(12,-1.5);
}
```

上面例子中，选项 `grow right sep, branch down sep` 会通过两个平移把各个顶点 node 的锚位置 `north west` 放到锚定点上，各个顶点 node 所占据的“深度”就是：顶点 node 本身的高度，即从顶点 node 的锚位置 `north` 到锚位置 `south` 的距离，再加上默认的 `1em`.

从本选项的效果上看，选项值 ⟨*distance*⟩ 是“链间距”，例如，对于 `a--{aa,aaa}, b--{bb,bbb}`, 其中 `a--{aa,aaa}` 与 `b--{bb,bbb}` 各成一链，所以二者的间距 ⟨*distance*⟩ 是 `aaa` 与 `bb` 的间距，例如



0.0

```
\tikz{
  \graph [branch down sep,nodes/.expanded={draw}] { a--{aa,aaa}, b--
  ↪  {bb,bbb} };
  \path let \p1=(aaa.south), \p2=(bb.north) in
    node [below=2mm] at(current bounding box.south){\pgfmathparse{\y1-
    ↪  \y2-1em}\pgfmathresult};
}
```

58.17.5 排布策略：grid placement

/tikz/graphs/grid placement (no value)

这个选项确定一种排布策略。当需要把顶点排成矩阵形式时可以使用本选项。这个策略的思路是：逐行排布顶点，每行的顶点个数相同，最后一行可能排不满。先计算列数（一行中顶点的个数），再依据列数逐行地排布顶点。计算出的列数保存在宏 `\tikzgraphwrapafter` 中。有两个办法确定列数：

1. 使用选项 /tikz/graphs/wrap after→P. 1125=⟨*number*⟩ 将列数指定为 ⟨*number*⟩, 这个选项的初始值是 0. 如果 ⟨*number*⟩ ≠ 0, 则将列数 ⟨*number*⟩ 保存在宏 `\tikzgraphwrapafter` 中。
2. 如果选项 `wrap after` 的值是 0，则利用宏 `\tikzgraphVnum` 的值 (应该是一个非负整数，代表顶点总个数) 来计算列数。
   - 选项 /tikz/graphs/V→P. 1122=⟨*list of vertices*⟩ 给出一个顶点列表，顶点总个数会被自动保存在宏 `\tikzgraphVnum` 中。
   - 选项 /tikz/graphs/n→P. 1122=⟨*number*⟩ 直接把顶点总个数指定为 ⟨*number*⟩, 并把顶点总个数自动保存在宏 `\tikzgraphVnum` 中。这个选项实际是利用 `/tikz/graphs/V` 来起作用的。

   在确定宏 `\tikzgraphVnum` 的值后，列数就是 $\lceil\sqrt{\texttt{\textbackslash tikzgraphVnum}}\rceil$(开平方后取整数部分)，然后将列数保存在宏 `\tikzgraphwrapafter` 中。

这个策略是通过逐个地对顶点做平移来实现“逐行排布顶点”的。初始之下，所有顶点 node 的锚定点都是原点。假设：

- 列数 `\tikzgraphwrapafter` 的值是 n;
- 当前键值 `chain shift`=⟨*vec1*⟩, `group shift`=⟨*vec2*⟩;
- 当前顶点 V 的逻辑宽度值是 w_V, 即键 `/tikz/graphs/placement/width` 的当前值;
- 当前顶点 V 的逻辑深度值是 d_V, 即键 `/tikz/graphs/placement/depth` 的当前值;
- 取二者的较大者 $\texttt{max}(w_V, d_V) = M_V$;
- 取模 $\texttt{mod}(M_V, n) = c$(不带长度单位);
- 做整数除法 $\texttt{div}(M_V, n) = r$(不带长度单位),

那么针对当前顶点 V 的平移向量是

$$r \cdot \langle vec2 \rangle + c \cdot \langle vec1 \rangle.$$

按这个排布方法，grid placement 策略很适合对一个链做排布，但对多个链做排布时可能会导致顶点重叠，也就是说，某些顶点的平移向量可能相等。例如最简单的例子是：

a — b

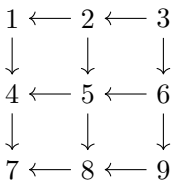
```
\tikz \graph [grid placement,n=4] { a--b,c };
```

上面例子中的顶点 `b`，`c` 重合，因为 `b` 的宽度、深度是 `<1,0>`, `c` 的宽度、深度是 `<0,1>`, 针对它们的平移向量相等。

1 2 3

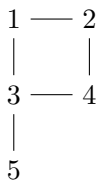
4 5 6

```
\tikz \graph [grid placement] { subgraph I_n[n=6, wrap after=3] };
```

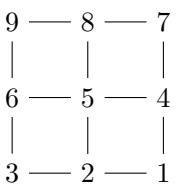


```
\tikz \graph [grid placement] { subgraph Grid_n [n=9, ->] };
```

上面两个例子表明，默认选项 grid placement 先排第一行，再排第二行，再排第三行……

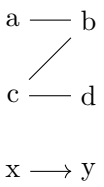


```
\tikz \graph [grid placement,] { subgraph Grid_n [n=5] };
```



```
\tikz \graph [grid placement, branch up, grow left]
{ subgraph Grid_n [n=9] };
```

使用选项 grid placement 后，可以把 n=⟨*number*⟩ 或 wrap after=⟨*column number*⟩ 作为某个链组的选项来确定该链组顶点矩阵的列数。



```
\tikz \graph  {
  {[grid placement, n=4] a -- b -- c -- d},
  {[nodes={shift=(-90:2)}] x -> y }
};
```

58.17.6 排布策略：circular placement

选项 circular placement 确定一种排布策略，在默认下它会把具有相同逻辑宽度的顶点放到同一个圆上。另外选项 clockwise 或 counterclockwise 会自动使用 circular placement 排布策略。

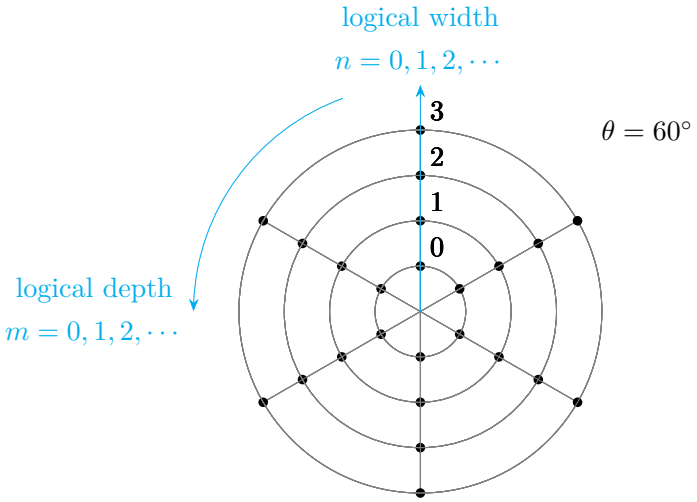
/tikz/graphs/circular placement (no value)

这个选项确定一种排布策略。

下面是 circular placement 策略的默认排布方式：

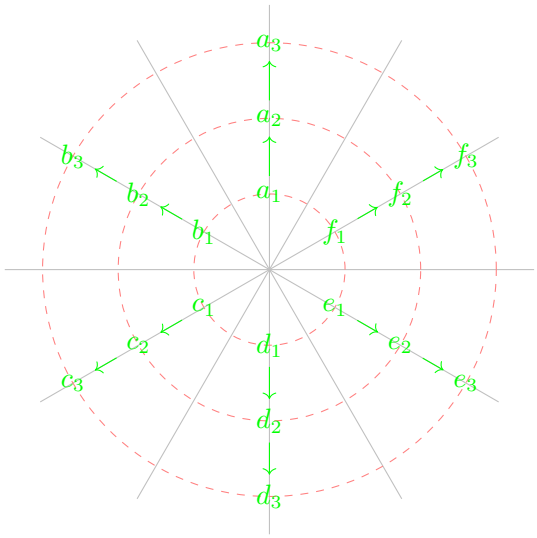
- 以方向是 (⟨*p*⟩ : 1) 的射线为初始边构造一个极坐标系，其中 ⟨*p*⟩ 是选项 phase=⟨*p*⟩ 指定的角度（此角度的初始值是 90°）；
- 以原点为圆心，以 ⟨*d*⟩ + *n* cm 为半径作圆 C_n，其中 ⟨*d*⟩ 是选项 radius=⟨*d*⟩ 规定的尺寸（这个尺寸的初始值是 1cm），*n* 是非负整数；
- 设 θ 是个角度，*m* 是非负整数，以原点为始点、以角度 ⟨*p*⟩° + $m\theta$ 为方向作射线，此射线与圆 C_n 相交于点 C_{nm}，点 C_{nm} 就是需要考虑的格点；
- 格点 C_{nm} 的 logical width 是 *n*，logical depth 是 *m*；格点的 logical width 对应其半径，格点的 logical depth 对应其角度；
- 如果没有其他相关选项（如 group polar shift=(⟨*angle*⟩:⟨*radius*⟩) 或 clockwise=⟨*number*⟩），则默认 $\theta = 60°$.

在 Cartesian placement 排布策略与 circular placement 排布策略之间有一种对应关系:记 Cartesian placement 排布策略下 logical width 和 logical depth 是 <n, m> 的格点为 P,记 circular placement 排布策略下 logical width 和 logical depth 也是 <n, m> 的格点为 Q,可以令 P 对应 Q.



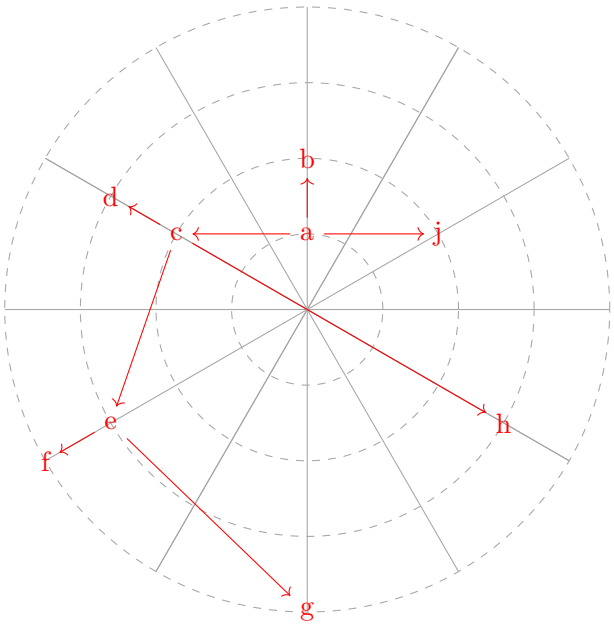
circular placement 排布策略的定义是:

```
\tikzgraphsset{,
  circular placement/.style={
    placement/place,
    placement/compute position/.code=\tikz@lib@graph@circular@pos,%
    placement/logical node depth/.code=\def\pgfmathresult{1},
    placement/logical node width/.code=\def\pgfmathresult{1},
  },
%...
}
```




```
\begin{tikzpicture}
  \foreach \ang[count=\i] in {0,60,-60}
  {  \draw [gray!50, rotate=\ang](-3.5,0) -- (3.5,0) (0,-3.5)--(0,3.5);
     \draw [red!50,dashed](0,0) circle [radius=\i];
  }
  \graph [math nodes, circular placement, nodes=green, edge=green] {
    a_1 -> a_2 -> a_3;
    b_1 -> b_2 -> b_3;
    c_1 -> c_2 -> c_3;
    d_1 -> d_2 -> d_3;
    e_1 -> e_2 -> e_3;
```

```
    f_1 -> f_2 -> f_3;
  };
\end{tikzpicture}
```




```
\tikz {
  \foreach \r in {1,2,3,4}
  { \draw [gray!70,dashed] (0,0) circle (\r);
    \draw [gray!70,rotate=60*\r] (-4,0)--(4,0) (0,-4)--(0,4);}
  \graph [circular placement, nodes=red, edge=red]
  { a -> { b,
           c -> { d,
                  e -> {f, g},
                  h},
           j
         }
  };
}
```

\tikz@lib@graph@circular@pos

命令 \tikz@lib@graph@circular@pos 对顶点做平移。初始之下，所有顶点 node 的锚定点都是原点。平移向量如下确定，假设：

- 当前键值 /tikz/graphs/chain polar shift=(a_c:s_c), 解析 \pgfmathparse{s_c} 的结果是 S_c(不带长度单位);
- 当前键值 /tikz/graphs/group polar shift=(a_g:s_g), 解析 \pgfmathparse{s_g} 的结果是 S_g(不带长度单位);
- 解析 \pgfmathparse{\pgfkeysvalueof{/tikz/graphs/radius}} 的结果是 r(不带长度单位);
- 解析 \pgfmathparse{\pgfkeysvalueof{/tikz/graphs/phase}} 的结果是 p(不带长度单位);
- 当前顶点 V 的逻辑宽度是 w_V, 即键 /tikz/graphs/placement/width 的当前值 (不带长度单位);
- 当前顶点 V 的逻辑深度是 d_V, 即键 /tikz/graphs/placement/depth 的当前值 (不带长度单位);

那么针对当前顶点 V 的平移向量是

$$(w_V \cdot a_c + d_V \cdot a_g + p \ : \ w_V \cdot S_c\mathtt{pt} + d_V \cdot S_g\mathtt{pt} + r\mathtt{pt})$$

其中的 $w_V \cdot a_c + d_V \cdot a_g + p$ 没有长度单位。

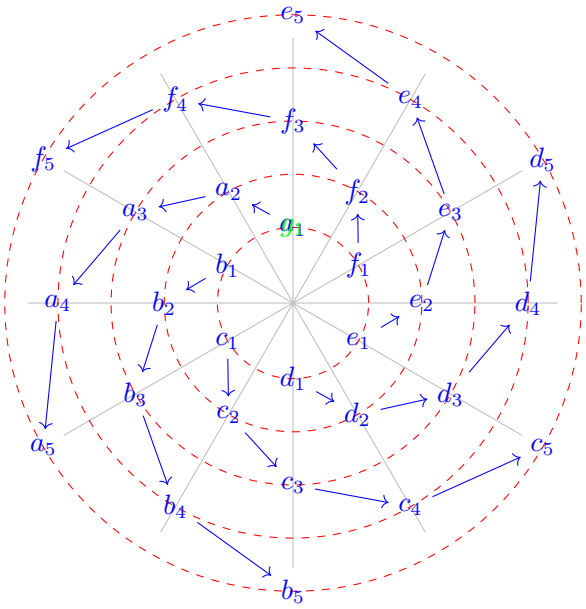
```
\def\tikz@lib@graph@circular@pos{%
  \pgfkeysgetvalue{/tikz/graphs/chain polar shift}\tikz@temp
  \expandafter\tikz@lib@graph@decompose@polar\tikz@temp%
  \pgf@process{\pgfpointscale{\pgfkeysvalueof{/tikz/graphs/placement/width}}{}}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgfkeysgetvalue{/tikz/graphs/group polar shift}\tikz@temp
  \expandafter\tikz@lib@graph@decompose@polar\tikz@temp%
  \pgf@process{\pgfpointscale{\pgfkeysvalueof{/tikz/graphs/placement/depth}}{}}%
  \advance\pgf@xa by\pgf@x%
  \advance\pgf@ya by\pgf@y%
  \pgfmathparse{\pgfkeysvalueof{/tikz/graphs/radius}}%
  \advance\pgf@ya by\pgfmathresult pt%
  \pgfmathsetmacro\tikz@temp{\the\pgf@xa+\pgfkeysvalueof{/tikz/graphs/phase}}%
  \edef\tikz@lib@graph@shift{(\tikz@temp:\the\pgf@ya)}
  \pgfkeys{/tikz/graphs/nodes/.expanded={shift={\tikz@lib@graph@shift}}}
}%


\def\tikz@lib@graph@decompose@polar(#1:#2){%
  \pgfmathsetlength\pgf@x{#1}%
  \pgfmathsetlength\pgf@y{#2}%
}%
```

下面是可以与选项 `circular placement` 配合使用的选项。

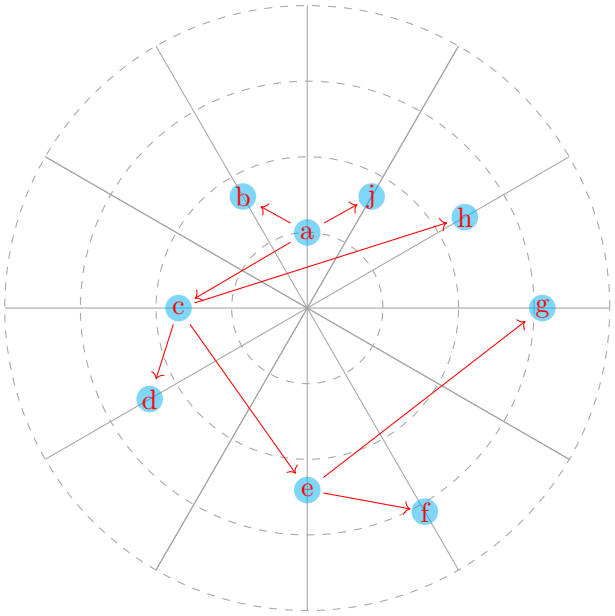
`/tikz/graphs/chain polar shift=(`⟨*angle*⟩`:`⟨*radius*⟩`)` (no default, initially `(0:1)`)

在 `circular placement` 排布策略下，本选项会使得每个顶点被平移（参见前文）。注意，如果其中的 ⟨*radius*⟩ 不带长度单位，则默认其长度单位是 `pt`.




```
\begin{tikzpicture}
  \foreach \ang[count=\i] in {0,60,-60}
    \draw [gray!50, rotate=\ang](-3.5,0) -- (3.5,0) (0,-3.5)--(0,3.5);
  \foreach \ri in {0,1,2,3,4}
    \draw [dashed,red](0,0) circle [radius=1cm+\ri*20 pt];
  \graph [math nodes,circular placement,chain polar shift=(30:20),nodes=blue,edge=blue] {
    a_1 -> a_2 -> a_3 -> a_4 -> a_5;
    b_1 -> b_2 -> b_3 -> b_4 -> b_5;
    c_1 -> c_2 -> c_3 -> c_4 -> c_5;
```

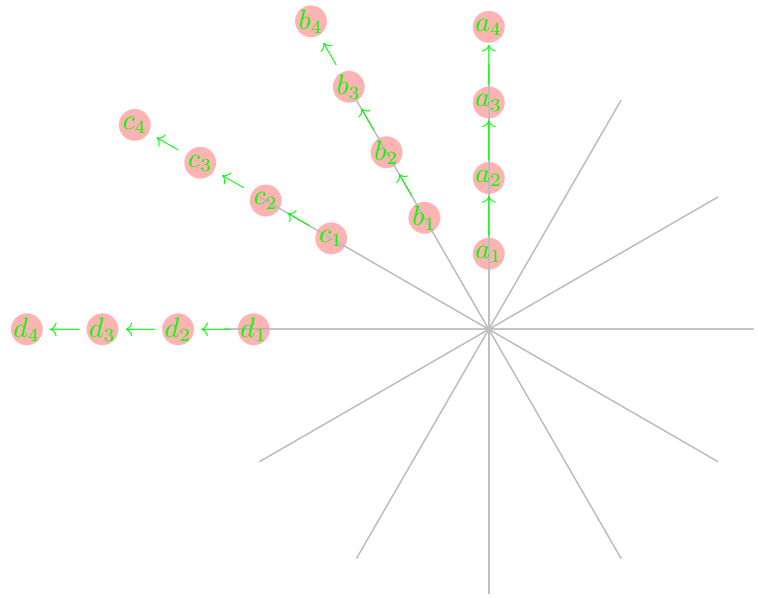
```
    d_1 -> d_2 -> d_3 -> d_4 -> d_5;
    e_1 -> e_2 -> e_3 -> e_4 -> e_5;
    f_1 -> f_2 -> f_3 -> f_4 -> f_5;
    g_1[green];
  };
\end{tikzpicture}
```




```
\tikz {
  \tikzmath{ \jiao=30; \jing=20; }
  \foreach \r in {1,2,3,4}
  { \draw [gray!70,dashed] (0,0) circle (\r);
    \draw [gray!70,rotate=60*\r] (-4,0)--(4,0) (0,-4)--(0,4);
  }
  \foreach \logicwidth/\logicdepth in {0/0,1/0,1/1,2/1,2/2,3/2,3/3,2/4,1/5}
    \fill [cyan, opacity=0.5] (90+\logicdepth*60+\logicwidth*\jiao:1cm+\logicwidth*\jing pt)
    ↪  circle (5pt);
  \graph [circular placement, nodes=red, edge=red, chain polar shift=(\jiao:\jing)]
  { a -> { b,
           c -> { d,
                  e -> {f,g},
                  h},
           j
         }
  };
}
```

/tikz/graphs/group polar shift=(⟨*angle*⟩:⟨*radius*⟩) (no default, initially (45:0))

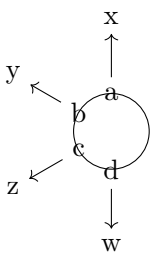
在 circular placement 排布策略下，本选项的作用参见前文。注意，如果其中的 ⟨*radius*⟩ 不带长度单位，则默认其长度单位是 pt.




```
\begin{tikzpicture}
  \foreach \ang / \m in {0/0,30/1,60/2,90/3}
    \foreach \n in {1,2,3,4}
    {\draw [gray!50, rotate=\ang](-3.5,0) -- (3.5,0) (0,-3.5)--(0,3.5);
     \fill [red, opacity=0.3](90 + \ang : \n cm + \m*20 pt) circle (6pt);}
  \graph [math nodes,circular placement,nodes=green,edge=green, group polar shift=(30:20)] {
    a_1 -> a_2 -> a_3 -> a_4;
    b_1 -> b_2 -> b_3 -> b_4;
    c_1 -> c_2 -> c_3 -> c_4;
    d_1 -> d_2 -> d_3 -> d_4;
  };
\end{tikzpicture}
```

/tikz/graphs/radius=⟨*dimension*⟩ (no default, initially 1cm)

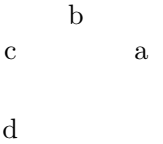
在 circular placement 排布策略下，本选项设置最内层的顶点所在的圆的半径，即 logical width 等于 0 的顶点所在的圆的半径。




```
\tikz{
  \graph [circular placement, radius=5mm]
    { a->x, b->y, c->z, d->w };
  \draw circle (5mm);}
```

/tikz/graphs/phase=⟨*angle*⟩ (no default, initially 90)

在 circular placement 排布策略下，本选项决定排布策略所用的极坐标系初始边的方向，初始之下，极坐标系初始边的方向是 90°.




```
\tikz \graph [circular placement, phase=30]
  { a, b, c, d };
```

/tikz/graphs/clockwise=⟨*number*⟩ (default \tikzgraphVnum)

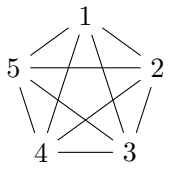
使用本选项后，当 logical width 等于 1 的顶点个数为 ⟨*number*⟩ 个时，这 ⟨*number*⟩ 个顶点恰好按顺时针方向均匀排布在圆上，相邻两个顶点的角度之差是 $\theta = \frac{360^\circ}{\langle number \rangle}$，这个 θ 就是 circular placement 排布策略所使用的 θ.

a
```
\tikz \graph [clockwise=4] { a, b, c, d };
```
d b
c

a
```
\tikz \graph [clockwise=4] { a, b, c };
```
b
c

a
```
\tikz \graph [clockwise=4] { a, b, c, d, e[red] };
```
d b
c

本选项的默认值是宏 \tikzgraphVnum 的值，如果使用选项 n=⟨*number*⟩，则宏 \tikzgraphVnum 的值就是 ⟨*number*⟩.




```
\tikz \graph [clockwise] { subgraph K_n [n=5] };
```

/tikz/graphs/counterclockwise=⟨*number*⟩ (default \tikzgraphVnum)

类似 clockwise，只是按逆时针方向排布同一 logical width 的顶点。

58.17.7 层与层样式

命令 \tikz@lib@graph@node@opt@normal → P. 1046 在新建顶点前会执行：

```
\pgfkeysgetvalue{/tikz/graphs/placement/level}\tikz@temp%
\c@pgf@counta=\tikz@temp\relax%
\advance\c@pgf@counta by1\relax%
\edef\tikz@temp{\the\c@pgf@counta}%
\pgfkeyslet{/tikz/graphs/placement/level}\tikz@temp%
\tikzgraphsset{
  level/.try=\pgfkeysvalueof{/tikz/graphs/placement/level},
  level \pgfkeysvalueof{/tikz/graphs/placement/level}/.try
}
...
{
  ...
  用 \node 创建顶点
  ...
}
\tikz@lib@graph@placement@update
```

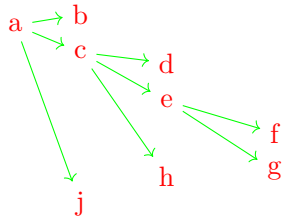
可见在新建顶点前会将 /tikz/graphs/placement/level 的值加 1, 所以，如果用户不指定其他值，那么层编号从 1 开始。

/tikz/graphs/level=⟨*level*⟩ (style, no default)

这个键是没有定义的，如果希望它发挥某种作用，需要提前定义它，例如

```
\tikzgraphsset{level/.style=...} 或者
\tikzgraphsset{level/.append style=...} 或者
\tikzgraphsset{level/.code=...} 或者
\tikzgraphsset{level/.append code=...} 或者其他
```

可以令 `level` 保存的代码能处理一个参数，即处理 `/tikz/graphs/placement/level` 的当前值。
定义这个键后，它对当前点之后的任何一个层都有效。



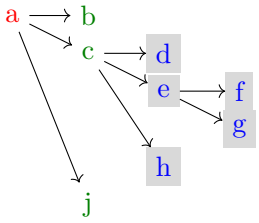

```
\tikz \graph [level/.style={nodes={red,shift=(135:8mm - #1*2 mm)},
    edges=green,branch down=5mm}]
{ a -> { b,
         c -> { d,
                e -> {f,g},
                h },
         j
       }
};
```

/tikz/graphs/level ⟨*level number*⟩ (style, no default)

这里 ⟨*level number*⟩ 是某个层的序号。
这个键是没有定义的，如果希望它发挥某种作用，需要提前定义它，例如

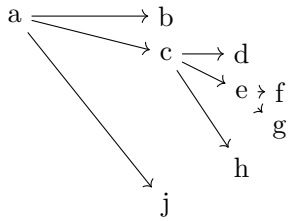
```
\tikzgraphsset{level ⟨level number⟩/.style=...} 或者
\tikzgraphsset{level ⟨level number⟩/.append style=...} 或者
\tikzgraphsset{level ⟨level number⟩/.code=...} 或者
\tikzgraphsset{level ⟨level number⟩/.append code=...} 或者其他
```

定义这个键后，它对第 ⟨*level number*⟩ 层，以及之后的各个层都有效。




```
\tikz \graph [
    branch down=5mm,
    level 1/.style={nodes=red},
    level 2/.style={nodes=green!50!black},
    level 3/.style={nodes={blue, fill=gray!30}}]
{ a -> { b,
         c -> { d,
                e -> {f,g},
                h },
         j
       }
};
```

注意上面例子中，`level 3` 的样式不仅作用于第 3 层，也作用于第 4 层。

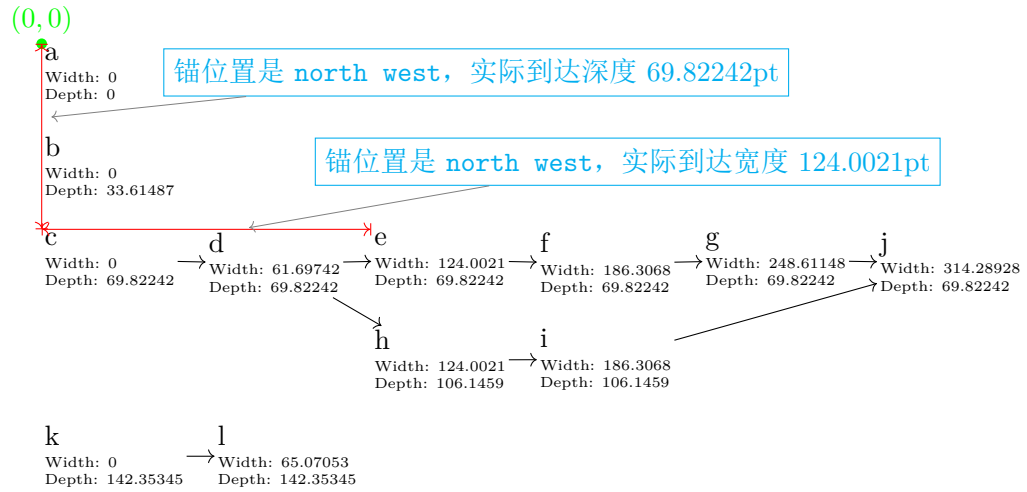



```
\tikz \graph [
    branch down=5mm,
    level 1/.style={grow right=2cm},
    level 2/.style={grow right=1cm},
    level 3/.style={grow right=5mm}]
{ a -> { b,
         c -> { d,
                e -> {f,g},
                h },
         j
       }
};
```

58.17.8 定义新的排布策略

利用逻辑性变量，参考前文。

注意，在创建一个新的顶点前，会调用 /tikz/graphs/placement/place→P. 1091，另外，各个预定义的排布策略，以及与策略相关的选项也会调用这个选项。当切换排布策略，或者更改策略选项时注意这一点。

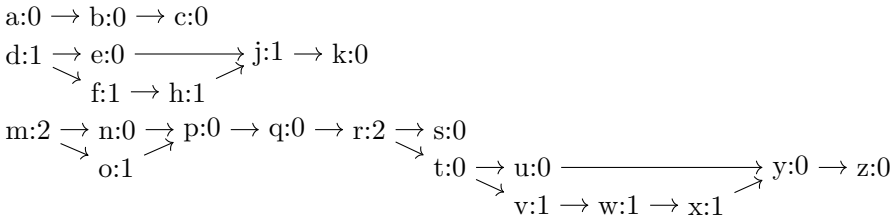



```
\tikz {
  \graph [grow right sep, branch down sep, nodes={align=left, inner sep=1pt},
          typeset={\tikzgraphnodetext\\[-8pt] \tiny Width: \mywidth\\[-10pt] \tiny Depth: \mydepth},
          placement/compute position/.append code=
            \pgfkeysgetvalue{/tikz/graphs/placement/width}{\mywidth}
            \pgfkeysgetvalue{/tikz/graphs/placement/depth}{\mydepth}]
  { a,
    b,
    c -> d -> { e -> f -> g,
                h -> i } -> j,
    k -> l
    };

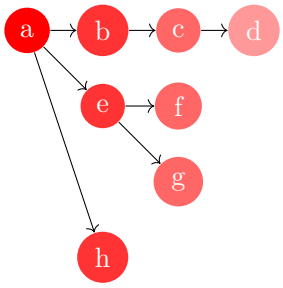
  \fill [green] circle (2pt) node [above] {$(0,0)$};

  \draw [{Bar[]<}-{>[]Bar}, red]
    let \p1=(a.north west),
        \p2=(c.north west),
        \p3=($(\p1)-(\p2)$)
    in (0,0)-- node [pos=0.4, pin={[draw,cyan, pin distance=1.5cm,pin edge={<-}]10:
      {锚位置是 \texttt{north west}，实际到达深度 \y3}}] {} ++(0,-\y3);

  \draw [{Bar[]<}-{>[]Bar}, red]
    let \p1=(e.north west),
        \p2=(c.north west),
        \p3=($(\p1)-(\p2)$)
    in (c.north west)-- node [pos=0.6, pin={[draw,cyan,pin distance=1cm,pin edge={<-}]30:
      {锚位置是 \texttt{north west}，实际到达宽度 \x3}}] {} ++(\x3,0);
}
```



```
\tikz \graph [
    grow right sep, branch down=5mm, typeset=\tikzgraphnodetext:\mynum,
    placement/compute position/.append code=
      \pgfkeysgetvalue{/tikz/graphs/placement/chain count}{\mynum}]
  { a -> b -> c,
    d -> { e, f->h} -> j -> { k },
    m -> { n, o } -> { p -> q} -> r -> { s, { t -> {u, v -> w -> x} -> y -> z }}
  };
```




```
\newcount\mycount
\def\lightendeepernodes{ \pgfmathsetcount{\mycount}{
    100-20*\pgfkeysvalueof{/tikz/graphs/placement/width}}
  \edef\mydepth{\the\mycount}
  \tikzset{nodes={fill=red!\mydepth,circle,text=white}}
}
\tikz
  \graph [placement/compute position/.append code=
            \lightendeepernodes]
  { a -> { b -> c -> d,
           e -> { f,
                  g },
           h }
  };
```

58.18 图宏，子图

图宏（graph macro）的作用是在某个位置插入一个子图。在绘制图的代码中不能直接使用 TEX 宏，因为 graphs 库不会直接展开不能处理的 TEX 宏，在图的各个顶点、边都被确定后才会展开 TEX 宏。

用下面的选项声明一个图宏：

/tikz/graphs/declare={⟨*graph name*⟩}{[⟨*options B*⟩] ⟨*content*⟩} (no default)

这里 ⟨*graph name*⟩ 是需要插入的子图的名称，而 {[⟨*options*⟩] ⟨*content*⟩} 相当于一个链组，是子图代码。

```
\tikzgraphsset{
  declare/.code 2 args={\expandafter\def\csname tikz@lib@graph@def@#1\endcsname{
  ↪  \tikz@lib@graph@do@graph{#2}}}%
}%
\def\tikz@lib@graph@do@graph#1{%
  \tikz@lib@graph@parse@group{#1}%
}%
```

参考 \tikz@lib@graph@parse@group → P. 1032.

执行 declare={⟨*graph name*⟩}{⟨*specification*⟩} 导致定义控制序列

\csname tikz@lib@graph@def@⟨*graph name*⟩\endcsname

使之保存 \tikz@lib@graph@do@graph{[⟨*options*⟩] ⟨*content*⟩}.

当把 ⟨*graph name*⟩[⟨*options A*⟩] 作为一个顶点使用时，命令 \tikz@lib@graph@node@opt@normal → P. 1046 会识别它，并执行 \tikz@lib@graph@handle@graph{⟨*options A*⟩}.

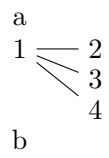
\tikz@lib@graph@handle@graph{⟨*options*⟩}

本命令的定义是：

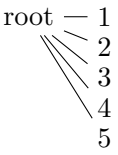
```
\def\tikz@lib@graph@handle@graph#1{%
  \begingroup%
    \let\tikz@lib@graph@node@list\pgfutil@empty%
    \tikzgraphsset{@extra group options/.style={#1}}%
    \tikz@lib@graph@start@hint@group%
```

```
      \csname tikz@lib@graph@def@\tikz@lib@graph@name@only\endcsname%
    \tikz@lib@graph@end@hint@group%
    \expandafter%
  \endgroup%
  \expandafter\expandafter\expandafter\def%
  \expandafter\expandafter\expandafter\tikz@lib@graph@node@list%
  \expandafter\expandafter\expandafter{\expandafter\tikz@lib@graph@node@list
  ↪  \tikz@lib@graph@node@list}%
  \tikz@lib@graph@hint@aftergroup%
}%
```

可见本命令会把子图 ⟨*graph name*⟩ 放在一个 \begingroup 与 \endgroup 的组合中来处理。选项 ⟨*options A*⟩ 将在 \tikz@lib@graph@group@opt →P. 1033 那里被执行，并且先于 ⟨*options B*⟩ 被执行。

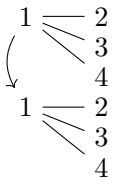



```
\tikz \graph [branch down=4mm, declare={claw}{1 -- {2,3,4}}] {
  a;
  claw;
  b;
};
```

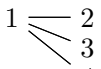



```
\tikz \graph [ n/.code=\def\n{#1}, branch down=4mm,
    declare={star}{root -- { \foreach \i in {1,...,\n} {\i} }}]
{ star [n=5]; };
```

如果多次使用同一个图宏，最好给图宏带上选项 name=⟨*name*⟩ 来为其命名，⟨*name*⟩ 会成为其中所有顶点 node 的名称前缀（见前文的 /tikz/graphs/name →P. 1051 选项）。




```
\tikz {
  \graph [branch down=4mm, declare={claw}{1 -- {2,3,4}}] {
    claw [name=left],
    claw [name=right] };
  \draw [->](left 1) to [ bend right] (right 1);}
```




```
\tikz {
  \graph [branch down=4mm, declare={claw}{1 -- {2,3,4}}] {
    claw,
    claw };
}
```

58.19 Quick graphs

graphs 库提供 quick 方法来创建一个图。当创建的图较大时，可以使用 quick 方法。

\iftikz@graph@quick

这个 TEX-if 决定是否启用 quick 方法。

/tikz/graphs/quick=true|false

本选项决定 \iftikz@graph@quick 的真值，即决定是否启用 quick 方法。

当启用 quick 方法后，\graph[⟨*options*⟩]{⟨*group spec*⟩} 在执行 ⟨*options*⟩ 后，就会调用 \tikz@lib@graphs@parse@quick@graph, 而不再调用 \tikz@lib@graphs@normal@main. 此时对顶点、链、链组的格式有以下要求：

- 指定顶点的句法必须是

  "⟨*node name*⟩"
  "⟨*node name*⟩"/"⟨*node text*⟩"
  "⟨*node name*⟩"/"⟨*node text*⟩"[⟨*options*⟩]

  其中名称、内容外围的双引号是必须的，⟨*node name*⟩ 中不要使用特殊符号。

- 不能使用类似 `a -- { b, c}` 这种把一个链组用作顶点的句法。
- 不能使用类型为 `-!-` 的边。
- 不能用选项 `simple`.
- 不能用图宏、子图。
- 不能用顶点集合。
- 不能用颜色类。
- 不能用与 fresh node 相关的选项。
- 不能用与 sublayouts 相关的选项 (与 graphdrawing 库相关)。
- 不能用 edge annotations, 即 `graphs/source edge style` 等选项。
- 指定顶点组或链组的句法是

  { [⟨*options*⟩] ⟨*chains and groups*⟩ };

  其中的 [⟨*options*⟩] 是必须写出的的。

- 选项 `number nodes` 仍然有效。
- graphs 库的自动排布顶点的策略无效，可以使用选项 `at={`⟨坐标⟩`}`，`x=`⟨x⟩，`y=`⟨y⟩ 指定顶点 node 的位置，或者用 graphdrawing 库的自动化 algorithm, 如 `layered layout` 来排布顶点。

58.20 预定义项目

58.20.1 图宏库 graphs.standard

```
\usetikzlibrary{graphs.standard} % LaTeX and plain TeX
\usetikzlibrary[graphs.standard] % ConTeXt
```

库 graphs.standard 定义了几个图宏：`subgraph I_n`, `subgraph I_nm`, `subgraph K_n`, `subgraph K_nm`, `subgraph C_n`, `subgraph P_n`, `subgraph Grid_n`, `subgraph G_np`.

关于这些图宏的设计思路可以参考网页 https://mathworld.wolfram.com/topics/SimpleGraphs.html

下面几个选项在《tikzlibrarygraphs.code.tex》中定义：

```
\tikzgraphsset{
  % Grids:
  wrap after/.initial=0,
  % Node sets:
  V/.code={%
    \def\tikzgraphV{#1}%
```

```
    \c@pgf@counta=0\foreach \tikz@dummy in {#1} {\global\advance\c@pgf@counta by1
    ↪  \relax}%
    \edef\tikzgraphVnum{\the\c@pgf@counta}
  },
  V={1},
  n/.style={V={1,...,#1},name shore V/.style={name=V}},
  W/.code={%
    \def\tikzgraphW{#1}%
    \c@pgf@counta=0\foreach \tikz@dummy in {#1} {\global\advance\c@pgf@counta by1
    ↪  \relax}%
    \edef\tikzgraphWnum{\the\c@pgf@counta}
  },
  W={1},
  m/.style={W={1,...,#1},name shore W/.style={name=W}},
  % Shores:
  name shore V/.style=,
  name shore W/.style=,
}%
```

/tikz/graphs/V={⟨*list of vertices*⟩} (initial 1)

这个选项中的大写字母“V”代表顶点（Vertices），⟨*list of vertices*⟩ 是顶点列表。当 graph 命令或者某个图宏带上这个选项后，graph 命令或者图宏就创建 ⟨*list of vertices*⟩ 中列举的顶点。顶点列表 ⟨*list of vertices*⟩ 保存在宏 \tikzgraphV 中，列表 ⟨*list of vertices*⟩ 中顶点的个数保存在宏 \tikzgraphVnum 中 (利用了 \foreach 循环来计数)。

/tikz/graphs/n=⟨*number*⟩ (no default)

这个选项是 V={1,...,⟨*number*⟩}, name shore V={name=V} 的简写，这里 V={1,...,⟨*number*⟩} 是顶点组 (会被 \foreach 处理)，name shore V={name=V} 指定这个顶点组的名称为 V，这个名称 V 是组中各个顶点的名称前缀，参考 /tikz/graphs/name → P. 1051, 例如 V 1 就是顶点 1 的名称。

/tikz/graphs/name shore V (style, initially empty)

这是个样式，初始之下它保存空内容。

/tikz/graphs/W={⟨*list of vertices*⟩} (initial 1)

类似选项 V，本选项用于设置一组顶点。顶点列表 ⟨*list of vertices*⟩ 保存在宏 \tikzgraphW 中，列表 ⟨*list of vertices*⟩ 中顶点的个数保存在宏 \tikzgraphWnum 中。

/tikz/graphs/name shore W (style, initially empty)

这是个样式，初始之下它保存空内容。

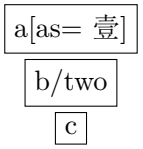
/tikz/graphs/m=⟨*number*⟩ (no default)

这个选项是 W={1,...,⟨*number*⟩}, name shore W={name=W} 的简写，这里 W={1,...,⟨*number*⟩} 是顶点组，name shore W={name=W} 指定这个顶点组的名称为 W，这个名称 W 是组中各个顶点的名称前缀，例如 W 1 就是顶点 1 的名称。

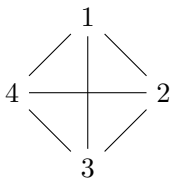
图宏会把一个“子图”插入到当前位置。图宏实际上是个“构图命令”，它后面可以加方括号，方括号内可以写入各种选项。

58.20.1.1 图宏 subgraph I_n

这个图宏可以创建 n 个顶点，任意两个顶点之间没有边相联系。




```
\tikz \graph [branch down sep=2pt, nodes=draw]
  { subgraph I_n [V={a[as= 壹], b/two, c}] };
```




```
\tikz \graph [clockwise, clique] { subgraph I_n [n=4] };
```

subgraph I_n 的定义是：

```
\tikzgraphsset{
  declare={subgraph I_n}%
  {
    \foreach \tikz@lib@graph@node@num in \tikzgraphV
    { \tikz@lib@graph@node@num }
  },
}
```

58.20.1.2 图宏 subgraph I_nm

观察下面的例子：

1 a

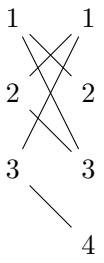
2 b

3 c

```
\tikz \graph { subgraph I_nm [V={1,2,3}, W={a,b,c}] };
```

在这个图宏的选项中为该图宏设置两组顶点，第一组用选项 V 设置，第二组用选项 W 设置，这两组顶点都是没有边的。

如果一个图由两组顶点构成，那么可以考虑使用图宏 subgraph I_nm.




```
\tikz \graph {subgraph I_nm [n=3, m=4];
  V 1 -- { W 2, W 3 };
  V 2 -- { W 1, W 3 };
  V 3 -- { W 1, W 4 };
};
```

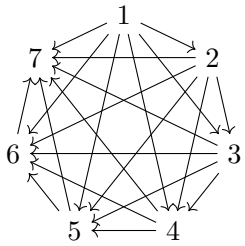
这个例子中的 V 1，W 1 等都是顶点的全名（即 ⟨前缀⟩⟨ 空格⟩⟨ 名称⟩）。

subgraph I_nm 的定义是：

```
\tikzgraphsset{
  declare={subgraph I_nm}%
  {
    subgraph I_n [name shore V] -- [no edges]
    subgraph I_n [name shore W, V/.expand once=\tikzgraphW]
  },
}
```

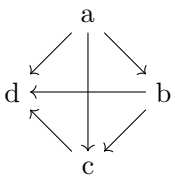
58.20.1.3 图宏 subgraph K_n

使用这个图宏时需要在它的选项中用 V 或 n 为它指定一个顶点组，这个图宏会在其所辖的顶点组内，在任意两个顶点之间连线画边，即在顶点组内使用 clique 算子（见后文）。




```
\tikz \graph [clockwise, radius=1.5cm]
  { subgraph K_n [n=7, ->] };
```

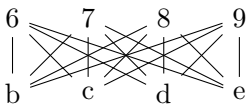
```
\tikzgraphsset{
  declare={subgraph K_n}%
  {
    [clique]
    subgraph I_n
  },
}
```



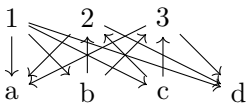

```
\tikz \graph [clockwise]
  { subgraph K_n [->, V={a, b, c, d}] };
```

58.20.1.4 图宏 subgraph K_nm

使用这个图宏时需要在它的选项中用 V 或 n，W 或 m 为它指定两个顶点组。假设 x 是第一个顶点组的任意一个顶点，y 是第二个顶点组的任意一个顶点，则这个图宏会在 x 与 y 之间连线画边。




```
\tikz \graph [branch right, grow down]
  { subgraph K_nm [V={6,...,9}, W={b,...,e}] };
```




```
\tikz \graph [simple, branch right, grow down]
{
  subgraph K_nm [V={1,2,3}, W={a,b,c,d}, ->];
  subgraph K_nm [V={2,3}, W={b,c}, <-];
};
```

58.20.1.5 图宏 subgraph P_n

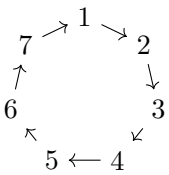
使用这个图宏时需要在它的选项中用 V 或 n 为它指定一个顶点组，这个图宏会把这些顶点连接为一个链。

a — b — c — d

```
\tikz \graph [branch right]
  { subgraph P_n [V={a, b, c, d}] };
```

58.20.1.6 图宏 subgraph C_n

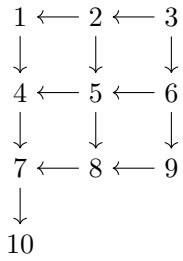
使用这个图宏时需要在它的选项中用 V 或 n 为它指定一个顶点组，这个图宏会把这些顶点连接为一个环。




```
\tikz \graph [clockwise]
  { subgraph C_n [n=7, ->] };
```

58.20.1.7 图宏 subgraph Grid_n

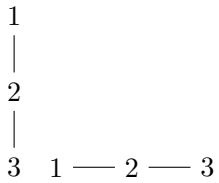
使用这个图宏时需要在它的选项中用 V 或 n 为它指定一个顶点组，这个图宏主要配合排布策略选项 grid placement 来使用，把顶点排布成点阵状并画边。



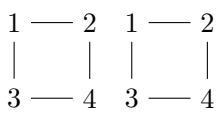

```
\tikz \graph [grid placement]
  { subgraph Grid_n [n=10, ->] };
```

/tikz/graphs/wrap after=⟨*number*⟩ (initial 0)

当使用排布策略选项 grid placement 和图宏 subgraph Grid_n 时，顶点排布成点阵状，本选项指顶点阵的列数为⟨*number*⟩. 词组“wrap after...”的意思是“在... 之后卷折（换行）”。




```
\tikz \graph [grid placement]
  { subgraph Grid_n [n=3,wrap after=1] };
\tikz \graph [grid placement]
  { subgraph Grid_n [n=3,wrap after=3] };
```




```
\tikz \graph [grid placement]
  { subgraph Grid_n [n=4,wrap after=2] };
\tikz \graph [grid placement]
  { subgraph Grid_n [n=4] };
```

58.21 其他

58.21.1 处理未知键

/tikz/graphs/redirect unknown to tikz (no value)

执行这个样式会导致键 /tikz/graphs/.unknown 被定义，这个“未定义”键的处理是：当遇到未定义的键值对时，就把默认路径改为 /tikz, 然后再执行这个键值对；然后把默认路径恢复为 /tikz/graphs. 当混合使用路径为 /tikz/graphs 的键与路径为 /tikz 的键时，可以考虑执行这个样式。

```
\tikzgraphsset{redirect unknown to tikz/.style={
    /tikz/graphs/.unknown/.code={%
      \let\tikz@key\pgfkeyscurrentname%
```

```
      \pgfkeys{tikz/.cd,\tikz@key={##1},/tikz/graphs/.cd}%
    }}
}%
```

58.21.2 修改创建边的 edge 路径

选项 /tikz/graphs/new --→P. 1068 的定义是：

```
\tikzgraphsset{
  new --/.code n args={4}{
      \path [-,every new --/.try]
      (#1\tikzgraphleftanchor)
      edge[#3] #4
      (#2\tikzgraphrightanchor);},
}
```

这是 -- 类型的边的定义。edge 路径类似 node, 是主路径的“附加物”。在默认下 edge 路径利用 to 操作的方式来创建路径 (初始之下是一个线段)，如果想把 edge 路径画成一条曲线，可以参考 /tikz/edge macro→P. 746, /tikz/to path→P. 740.

第五十九章　Lindenmayer System 分形图

PGF 有《pgflibrarylindenmayersystems.code.tex》。

Ti*k*Z 有《tikzlibrarylindenmayersystems.code.tex》。

```
\usepgflibrary{lindenmayersystems} % LaTeX and plain TeX and pure pgf
\usepgflibrary[lindenmayersystems] % ConTeXt and pure pgf
\usetikzlibrary{lindenmayersystems} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[lindenmayersystems] % ConTeXt when using TikZ
```

本程序库提供构造平面 L-S 的方法。受到 TEX 内存的限制，能画出的 L-S 分形图形都不是非常复杂。

59.1　pgf 中的 L-S

见《pgf-notes》。

59.2　Ti*k*Z 中的 L-S

59.2.1　命令

`\path ... lindenmayer system[⟨keys⟩] ...;`

在 Ti*k*Z 处理这个路径命令的过程中，当命令 `\tikz@scan@next@command` 遇到 `lindenmayer system` 时，就执行 `\tikz@lsystem`.

⟨*keys*⟩ 中的选项应当有前缀 `/tikz`, 并且至少应当使用选项 `/tikz/lindenmayer system`→P. 1129`={`⟨*set keys*⟩`}` 或者选项 `/tikz/l-system`→P. 1129`={`⟨*set keys*⟩`}` 来指定或定义一种 L-S.

下面几个句法等效：

```
\draw lindenmayer system [lindenmayer system={Hilbert curve, axiom=4, order=3}];
```

```
\draw [lindenmayer system={Hilbert curve, axiom=4, order=3}] lindenmayer system;
```

```
\tikzset{lindenmayer system={Hilbert curve, axiom=4, order=3}}
  \draw lindenmayer system;
```

`\path ... l-system[⟨keys⟩] ...;`

这个句法与上一个句法等效。

`\tikz@@lsystem-system`

本命令的定义是：

```
\def\tikz@@lsystem-system{%
    \pgfutil@ifnextchar[{\tikz@lsystem@options}{\tikz@lsystem@options[]}}%
```

`\tikz@@lsystem indenmayer system`

本命令的定义是：

```
\def\tikz@lsystem indenmayer system{%
    \pgfutil@ifnextchar[{\tikz@lsystem@options}{\tikz@lsystem@options[]}}%
```

\tikz@lsystem@options[⟨*options*⟩]

本命令：

1. 执行选项 ⟨*options*⟩

```
\tikzset{⟨options⟩}%
```

2. 检查 \ifx\tikz@lsystem@rules\pgfutil@empty,
   - 如果检查结果为 true, 即在 ⟨*options*⟩ 没有使用选项 /pgf/lindenmayer system/rule set 指定替换规则，那么什么也不做。
   - 如果检查结果为 false, 即在 ⟨*options*⟩ 使用选项 /pgf/lindenmayer system/rule set 指定了替换规则，那么
     (a) \tikz@lsystem@declare, 这个命令先 let 控制序列

     \csname pgf@lsystem@tikz@temp\endcsname

     为 \relax, 然后声明名称为 tikz@temp 的 L-S:

```
\pgfdeclarelindenmayersystem{tikz@temp}{%
        \expandafter\tikz@lsystem@parse@rules\tikz@lsystem@rules,
        ↪  \tikz@stop,%
    }%
```

     相当于

```
\pgfdeclarelindenmayersystem{tikz@temp}{%
    \rule{...}%
    \rule{...}%
    ....
}%
```

     (b) \def\tikz@lsystem@name{tikz@temp}
3. 检查 \ifx\tikz@lsystem@anchor\pgfutil@empty,
   - 如果检查结果为 true, 即在 ⟨*options*⟩ 没有使用选项 /pgf/lindenmayer system/anchor 指定一个 anchor 位置，那么
     (a) move-to 到当前点

```
\pgfpathmoveto{\pgfqpoint{\tikz@lastxsaved}{\tikz@lastysaved}}%
```

     (b) 创建名称为 \tikz@lsystem@name 的 L-S:

```
\pgflindenmayersystem{\tikz@lsystem@name}{\tikz@lsystem@axiom}{
↪  \tikz@lsystem@order}%
```

     其中的 \tikz@lsystem@axiom, \tikz@lsystem@order 由选项 /pgf/lindenmayer system/axiom, /pgf/lindenmayer system/order 规定。
   - 如果检查结果为 false, 即在 ⟨*options*⟩ 使用选项 /pgf/lindenmayer system/anchor 指定了一个 anchor 位置，那么
     (a) 将当前点保存到 \tikz@lsystem@pos,

```
\pgfextract@process\tikz@lsystem@pos{\pgfqpoint{\tikz@lastxsaved}{
↪  \tikz@lastysaved}}%
```

     (b) 设置盒子

```
\setbox\pgfnodeparttextbox=\hbox{%
    \pgfinterruptpicture%
    \pgfpicture%
        \pgfpathmoveto{\pgfpointorigin}%
        \pgflindenmayersystem{\tikz@lsystem@name}{\tikz@lsystem@axiom}{
        ↪  \tikz@lsystem@order}%
        \begingroup%
            \tikz@finish%
    \endpgfpicture%
    \endpgfinterruptpicture%
}%
```

在盒子内，以原点为当前点，创建分形图形。

(c) 在一个组内创建包含分形图的 node,

```
{%
    \pgftransformshift{\tikz@lsystem@pos}%
    \tikzset{inner sep=0pt, outer sep=0pt, minimum size=0pt}%
    \pgfmultipartnode{rectangle}{\tikz@lsystem@anchor}{lindenmayer
    ↪  system}{\pgfusepath{discard}}%
}%
```

参考 `\pgfmultipartnode`. 关于这个 node:

- 其锚定点是当前点，
- 其 anchor 位置是 `\tikz@lsystem@anchor`, 在之前由选项 `/pgf/lindenmayer system/anchor` 指定，
- 设置了选项 `inner sep=0pt, outer sep=0pt, minimum size=0pt`,
- 形状为 `rectangle`,
- 名称为 `lindenmayer system`,

4. 执行 `\tikz@scan@next@command`, 解析之后的命令。

59.2.2 选项

`/pgf/lindenmayer system=`{⟨*keys*⟩} (style, no default)

`/tikz/lindenmayer system=`{⟨*keys*⟩} (style, no default)

这个选项用于临时声明一个 L-S, 声明内容由 ⟨*keys*⟩ 指定，⟨*keys*⟩ 中使用的选项应该具有路径前缀 `/pgf/lindenmayer system`.

```
\tikzset{%
    lindenmayer system/.style={/pgf/lindenmayer system/.cd, #1,/tikz/.cd},
    l-system/.style={lindenmayer system={#1}},
}%
```

`/pgf/l-system=`{⟨*keys*⟩} (style, no default)

`/tikz/l-system=`{⟨*keys*⟩} (style, no default)

等效于上一选项。

`/pgf/lindenmayer system/name=`{⟨*name*⟩} (no default)

⟨*name*⟩ 是已经被声明的 L-S 的名称，本选项调用这个 L-S, 画出其图形。

- 如果使用本选项,那么仍然需要使用选项 `/pgf/lindenmayer system/axiom, order` 指定 ⟨*axiom*⟩, ⟨*order*⟩, 但不要再使用选项 `/pgf/lindenmayer system/rule set` 指定替换规则。

- 如果不使用本选项，那么应当使用选项 /pgf/lindenmayer system/rule set, axiom, order 来定义 \pgflindenmayersystem 的参数。

```
\pgfkeys{/pgf/lindenmayer system/.cd,
    name/.code=\edef\tikz@lsystem@name{#1}\let\tikz@lsystem@rules=\pgfutil@empty,%
}
```

/pgf/lindenmayer system/axiom={⟨*string*⟩} (no default)

设置 L-S 的初始符号串 ⟨*string*⟩.

```
\pgfkeys{/pgf/lindenmayer system/.cd,
    axiom/.store in=\tikz@lsystem@axiom,%
}
```

/pgf/lindenmayer system/order={⟨*integer*⟩} (no default)

指定符号替换的次数，获得第 ⟨*integer*⟩ 次符号替换后的结果。

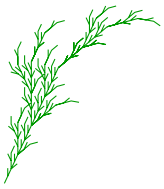
```
\pgfkeys{/pgf/lindenmayer system/.cd,
    order/.store in=\tikz@lsystem@order,
}
```

/pgf/lindenmayer system/rule set={⟨*list*⟩} (no default)

指定符号替换规则，各规则之间用逗号分隔，每个规则都使用“⟨前任⟩->⟨继任⟩”的格式。

```
\pgfkeys{/pgf/lindenmayer system/.cd,
    rule set/.store in=\tikz@lsystem@rules
}
\let\tikz@lsystem@rules=\pgfutil@empty%
```

下面是一个用选项定义 L-S 并画出其图形的例子。



```
\tikz[rotate=65]\draw [green!60!black] l-system
  [l-system={rule set={F -> F[+F]F[-F]}, axiom=F, order=4,
    angle=25,step=3pt}];
```

/pgf/lindenmayer system/anchor={⟨*anchor*⟩} (no default)

如果不使用这个选项，当启用 L-S 画笔开始画分形图时，画笔会从当前点开始画起，“当前点”就是启用画笔时程序处理过程恰好达到的点。

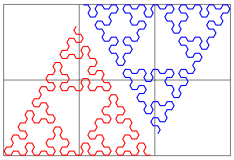
使用这个选项后，所画的分形图会被放入一个矩形 node 中，关于这个 node:

- 其锚定点是当前点，
- 其 anchor 位置是 \tikz@lsystem@anchor, 在之前由选项 /pgf/lindenmayer system/anchor 指定，
- 设置了选项 inner sep=0pt, outer sep=0pt, minimum size=0pt,
- 形状为 rectangle,
- 名称为 lindenmayer system,

本选项的定义是：

```
\pgfkeys{/pgf/lindenmayer system/.cd,
    anchor/.store in=\tikz@lsystem@anchor,%
}
\let\tikz@lsystem@anchor=\pgfutil@empty%
```

下面的例子中，先定义一个 L-S，然后在路径命令中调用这个定义，画出其图形。



```
\begin{tikzpicture}[l-system={step=1.75pt, order=5, angle=60}]
  \pgfdeclarelindenmayersystem{Sierpinski triangle}{
    \symbol{X}{\pgflsystemdrawforward}
    \symbol{Y}{\pgflsystemdrawforward}
    \rule{X -> Y-X-Y}
    \rule{Y -> X+Y+X}
  }
  \draw [help lines] grid (3,2);
  \draw [red] (0,0) l-system [l-system={Sierpinski triangle,
  ↪  axiom=+++X, anchor=south west}]; % 锚位置 south west 在 (0,0) 点上
  \draw [blue] (3,2) l-system [l-system={Sierpinski triangle, axiom=X,
  ↪  anchor=north east}]; % 锚位置 north east 在 (3,2) 点上
\end{tikzpicture}
```

/pgf/lindenmayer system/.unknown

```
\pgfkeys{/pgf/lindenmayer system/.cd,
    .unknown/.code={%
        \pgfutil@ifundefined{pgf@lsystem@\pgfkeyscurrentname}{%
            \pgfkeys{/errors/unknown key={/pgf/lindenmayer system/
            ↪  \pgfkeyscurrentname}{#1}}%
            \let\tikz@lsystem@name=\pgfutil@empty%
        }%
        {\edef\tikz@lsystem@name{\pgfkeyscurrentname}}
    },%
}
```

第六十章　math 库

TikZ Library math

```
\usetikzlibrary{math} % LaTeX and plain TeX
\usetikzlibrary[math] % ConTeXt
```

这个程序库定义了一种简单的数学语言，能执行一些基本的数学运算。PGF 的数学引擎用起来有点繁琐，尤其是在为多个变量赋值时。Ti*k*Z 的 calc 程序库也能执行计算，但一般情况下 calc 只用于 Ti*k*Z 的绘图命令。当调用程序库 math 以及 calc 后，可以在程序库 math 的语句中使用 calc 的句法。程序库 math 提供赋值语句、循环语句、条件语句、自定义函数语句、输出语句等句法来实现相应的运算。

程序库 math 使用 PGF 的数学引擎来完成各种赋值、解析、计算工作，受限于 TEX 的计算能力，程序库 math 的计算精度有限。可以用程序库 fp 或 fpu 提高计算精度。

本程序库提供命令 `\tikzmath` 和选项 `/tikz/evaluate` 来使用本程序库的功能。

60.1　基本命令

`\tikzmath{`⟨*statements*⟩`}`

这个命令等于 `\tikz@math`

```
\let\tikzmath=\tikz@math
```

参数 ⟨*statements*⟩ 是一些语句，注意：

- 每个语句都要以分号“;”结束，否则报错；分号只是作为句子的定界标志，不属于句子本身。
- ⟨*statements*⟩ 中不能有空行，否则报错。
- 语句的开头最好不是开花括号“`{`”。
- 在 ⟨*statements*⟩ 中的任何位置，最好不要添加多余的花括号。
- 在处理 ⟨*statements*⟩ 时，本命令不会引入额外的组来限制处理过程，所以处理结果可以用在后续的计算中。例外的是关键词 `print`, 见后文。

⟨*statements*⟩ 中的语句有 2 种：

1. 一种是以关键词 (⟨*keyword*⟩) 开头的语句，例如

   0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765,

```
\tikzmath{
  function fibonacci(\n) {  % 定义函数 fibonacci 并以 \n 为参数
    if \n == 0 then {  % 使用条件语句
      return 0;
      }
    else {
      return fibonacci2(\n, 0, 1);
      };  % 条件语句结束
```

```
  };  % 函数定义结束
  function fibonacci2(\n, \p, \q) {  % 定义函数 fibonacci2 并以 \n, \p, \q 为参数
    if \n == 1 then {  % 使用条件语句
      return \q;
      }
    else {
      return fibonacci2(\n-1, \q, \p+\q);
      };  % 条件语句结束
  };  % 函数定义结束
  int \f, \i;  % 声明整数变量
  for \i in {0,1,...,20}{  % 使用循环语句
    \f = fibonacci(\i);
    print {\f, };  % 输出计算结果
  };  % 循环语句结束
}
```

2. 一种是以命令 (一个命令通常以反斜线开头) \⟨*cmd*⟩ 开头的语句，例如

```
\newcount\mycount
\newdimen\mydimen
\tikzmath{
\a = 4*5+6;
\b = sin(30)*4;
\mycount = log10(2048) / log10(2);
\mydimen = 15^2;
}
\a, \b, \the\mycount, \the\mydimen
```

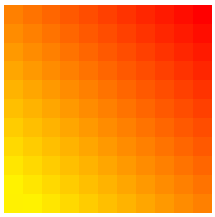
26.0, 2.0, 11, 225.0pt

本命令导致：

```
\tikz@math@parse⟨statements⟩@;
```

/tikz/evaluate=⟨*statements*⟩ (no default)

相当于执行 \tikzmath{⟨*statements*⟩}.



```
\tikz[x=0.25cm,y=0.25cm,
  evaluate={
    int \i, \j;
    for \i in {0,...,10}{
      for \j in {0,...,10}{
        \a{\i,\j} = (\i+\j)*5;
      };
    };
  }
]
\foreach \i in {0,...,10}
  \foreach \j in {0,...,10}
    \fill [red!\a{\i,\j}!yellow] (\i,\j) rectangle ++(1, 1);
```

\tikz@math@parse⟨*statement*⟩;

本命令解析一个句子。本命令：

- 检查句子 ⟨*statement*⟩ 是否以开花括号“{”开头——最好不要这么做。
- 检查 ⟨*statement*⟩ 是否以某个关键词 ⟨*keyword*⟩ 开头，如果是，就

  ```
  \tikz@math@parse@keyword⟨statement⟩;
  ```

- 检查 ⟨*statement*⟩ 是否以某个命令 \⟨*cmd*⟩ 开头，如果是，就

  ```
  \tikz@math@parse@nokeyword⟨statement⟩;
  ```

处理完一个句子后，本命令会继续处理后面的句子，直到处理完所有句子。

\tikz@math@parse@hook@before

\tikzmath 处理参数 ⟨*statements*⟩ 前，执行本命令。本命令的初值值是 \pgfutil@empty.

\tikz@math@parse@hook@after

\tikzmath 处理参数 ⟨*statements*⟩ 后，执行本命令。本命令的初值值是 \pgfutil@empty.

60.2 以关键词开头的语句

math 库认为的关键词 ⟨*keyword*⟩ 有 2 种：

1. PGF 的函数名称，即 \pgfmath⟨*keyword*⟩ 是 PGF 的函数 (包括预定义函数、自定义的函数)。
2. math 库定义的关键词，如果命令 \tikz@math@process@keyword@⟨*keyword*⟩ 在 math 库中有定义，那么 ⟨*keyword*⟩ 就是关键词。关键词有：
   - count, 对应命令 \tikz@math@process@keyword@count
   - length, 对应命令 \tikz@math@process@keyword@length
   - coordinate
   - point, 等于 coordinate
   - integer
   - int, 等于 integer
   - real
   - let
   - print
   - if
   - function
   - return
   - for

当一个句子以关键词 ⟨*keyword*⟩ 开头时，在 ⟨*keyword*⟩ 的后面应当使用 空格，(，{ 这 3 个符号之一来界定关键词；其他符号不能界定关键词，因为 math 库的解析命令 (\tikz@math@parse@keyword) 会以逐个字符的方式读取 ⟨*keyword*⟩ 中的符号，直到遇到这 3 个符号之一，才认定关键词 ⟨*keyword*⟩, 然后执行这个关键词对应的命令。

注意，在关键词开头的句子 ⟨*keyword*⟩ ⟨*following*⟩; 中：

- 如果 ⟨*following*⟩ 中含有分号，那么可以尝试用花括号将整个 ⟨*following*⟩ 包裹起来，也可以尝试把 ⟨*following*⟩ 中的分号单独用花括号包裹起来，不过这种添加花括号的做法未必被总是可接受的。
- 如果在 ⟨*following*⟩ 中有多余的花括号，那么可能导致错误。

60.2.1 以 pgf 的函数名为关键词

如果一个句子以 PGF 的某个函数名 ⟨*fun name*⟩ 开头 (例如正弦函数名称 sin)，那么这个句子应当是函数 ⟨*fun name*⟩ 的表达式，即 ⟨*fun name*⟩(...);, 这会导致：

```
\pgfmathparse{⟨fun name⟩(...)}%
\tikz@math@parse%
```

例如

0.5
```
\tikzmath{
  sin(30);
  print \pgfmathresult;
}
```

60.2.2 关键词 count，length

关键词 count，length 用于声明 count，dimen 类型的记号变量。

声明语句 count \⟨*var 1*⟩，\⟨*var 2*⟩，...；导致

\tikz@math@process@keyword@count\⟨*var 1*⟩，\⟨*var 2*⟩，...；

这又导致

```
\newcount\⟨var 1⟩%
\newcount\⟨var 2⟩%
...
```

声明语句 length \⟨*var 1*⟩，\⟨*var 2*⟩，...；导致

```
\newdimen\⟨var 1⟩%
\newdimen\⟨var 2⟩%
...
```

\count454
\count455

```
\tikzmath{
  count \aaaa, \bbbb;
}
\meaning\aaaa\par
\meaning\bbbb
```

60.2.3 关键词 integer，int，real，coordinate

这几个关键词规定 math 库的 3 种“变量类型”，即整数 integer, 实数 real, 坐标 coordinate, 这 3 种类型由下面的宏代表：

\tikz@math@keyword@coordinate

这个宏代表变量类型 coordinate.

```
\def\tikz@math@keyword@coordinate{coordinate}
```

\tikz@math@keyword@integer

这个宏代表变量类型 integer.

```
\def\tikz@math@keyword@integer{integer}
```

\tikz@math@keyword@real

这个宏代表变量类型 real.

```
\def\tikz@math@keyword@real{real}
```

声明变量类型的句子，例如：

integer \⟨*var*⟩;

integer \⟨*var 1*⟩，\⟨*var 2*⟩，...;

这个句子导致

\tikz@math@process@keyword@integer\⟨*var 1*⟩，\⟨*var 2*⟩，...

进一步地，对于每一个变量 \⟨*var i*⟩ 执行

```
\tikz@math@setvartype\⟨var i⟩\tikz@math@keyword@integer
```

也就是使得控制序列

\csname tikz@math@var@vartype@\string\⟨*var i*⟩\endcsname

等于 \tikz@math@keyword@integer, 即保存单词 integer.

关键词 int，real，coordinate 的声明也有类似的处理。

\tikz@math@setvartype\⟨*cmd*⟩\⟨*var type*⟩

本命令使得 \⟨*cmd*⟩ 的变量类型是 \⟨*var type*⟩ 代表的类型。

```
\def\tikz@math@setvartype#1#2{%
    \expandafter\let\csname tikz@math@var@vartype@\string#1\endcsname=#2%
}%
```

本命令使得控制序列

\csname tikz@math@var@vartype@\string\⟨*cmd*⟩\endcsname

等于 \⟨*var type*⟩.

\tikz@math@getvartype\⟨*save type*⟩\⟨*cmd*⟩

本命令将 \⟨*cmd*⟩ 的变量类型保存到 \⟨*save type*⟩.

```
\def\tikz@math@getvartype#1#2{%
    \def\tikz@math@marshal{\let#1=}%
    \expandafter\tikz@math@marshal\csname
    tikz@math@var@vartype@\string#2\endcsname%
}%
```

本命令将 \⟨*save type*⟩ let 为控制序列

\csname tikz@math@var@vartype@\string\⟨*cmd*⟩\endcsname

即保存 \⟨*cmd*⟩ 的变量类型。

macro:->real

```
\tikzmath{
  real \aaaa;
}
\makeatletter
\tikz@math@getvartype\savetemp\aaaa
\makeatother
\meaning\savetemp
```

60.2.4 关键词 let

关键词 let 引起对 \⟨*cmd*⟩ 的赋值，即 \edef\⟨*cmd*⟩....

macro:->XXX

macro:#1->\csname tikz@math@var@indexed@dddd@#1\endcsname

```
\def\aaaa{XXX}
\def\bbbb{\aaaa}
\tikzmath{
  let \cccc=\bbbb;
  let \dddd{index}=\linewidth;
}
\meaning\cccc\par
\meaning\dddd
```

上面例子中，\cccc 返回 \bbbb 的彻底展开值 (利用了 \edef)。

let \⟨*cmd*⟩...;

这个情况是：“关键词 let 引导 \⟨*cmd*⟩”，此时的 \⟨*cmd*⟩ 最好只是一个 macro, 可以是未定义的 macro, 但最好不要是寄存器 (dimen, count)，也不要是 integer, real, coordinate 类型的变量。这个句子：

1. 设置 \iftikz@math@let 的真值为 true.
2. 把 \⟨*cmd*⟩...; 看作是以命令开头的句子，解析它，即

```
\tikz@math@parse@nokeyword\⟨cmd⟩...;
```

这个句子通常导致：

- 如果是 `let \⟨cmd⟩=⟨something⟩;`, 则定义

```
\edef\⟨cmd⟩{⟨something⟩}
```

- 如果是 `let \⟨cmd⟩⟨index⟩=⟨something⟩;`, 则

```
\tikz@math@assign@index{⟨cmd⟩}{⟨something⟩}
```

参考 \tikz@math@assign@index→ P. 1144.

- 然后设置真值 `\tikz@math@letfalse`.

60.2.5 关键词 print

句子 `print ⟨code⟩;` 导致

```
\begingroup⟨code⟩\endgroup\tikz@math@parse
```

即在一个组内执行 ⟨code⟩, 然后继续解析后面的句子。

可以用花括号包裹 ⟨code⟩, 但这并非总是可行的。

下面的代码正常 (参考 \pgfmathsmuggle→ P. 40)：

3.0
```
\tikzmath{
  print
    \pgfmathparse{1+2}
    \pgfmathsmuggle\pgfmathresult;
  print \pgfmathresult;
}
```

但用花括号包裹被执行的命令后会导致错误，如下：

```
\tikzmath{
  print {
    \pgfmathparse{1+2}
    \pgfmathsmuggle\pgfmathresult
  };
}
```

错误信息是 `! Argument of \pgfmathsmuggle has an extra }.`

60.2.6 关键词 if

关键词 `if` 引起条件分支。

`if` ⟨condition⟩ `then {`⟨if-non-zero-statements⟩`};`

`if {`⟨condition⟩`} then {`⟨if-non-zero-statements⟩`};`

或者

`if` ⟨condition⟩ `then {`⟨if-non-zero-statements⟩`} else {`⟨if-zero-statements⟩`};`

`if {`⟨condition⟩`} then {`⟨if-non-zero-statements⟩`} else {`⟨if-zero-statements⟩`};`

这个句子中的参数 ⟨condition⟩ 应当是能被 \pgfmathparse 处理的表达式。如果 ⟨condition⟩ 被花括号包裹，那么几乎不会导致错误。

⟨if-non-zero-statements⟩ 和 ⟨if-zero-statements⟩ 是能被 \tikzmath 处理的一个或者一些语句 (注意用花括号包裹)。

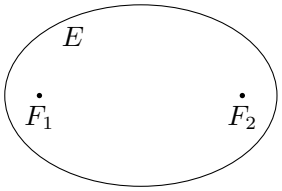
这个句子：

1. 处理条件 `\pgfmathparse{`⟨condition⟩`}`, 如果这个处理利用了 fpu 函数 (此时 \ifpgfmathfloatparseactive→ P. 594 的真值是 true)，那么就把结果转为定点数：

```
\ifpgfmathfloatparseactive%
    \pgfmathfloattofixed{\pgfmathresult}%
\fi%
```

2. 检查处理结果 \pgfmathresult 是否为 0,
   - 如果不是 0, 则用 \tikz@math 处理 ⟨*if-non-zero-statements*⟩,
   - 如果是 0, 则用 \tikz@math 处理 ⟨*if-zero-statements*⟩.

```
\begin{tikzpicture}
\tikzmath{
  int \x;
  for \k in {0,10,...,350}{
    if \k>260 then { let \c = orange; } else {
      if \k>170 then { let \c = blue; } else {
        if \k>80 then { let \c = red; } else {
          let \c = green; }; }; };  % 结束条件语句
% 下面是 for 语句中的非 0 语句，因为它是个绘图命令，不是本程序库的语句，
% 所以要用花括号把它括起来，且在闭花括号后加分号
    {
     \path [fill=\c!50, draw=\c] (\k:0.5cm) -- (\k:1cm) -- (\k+5:1cm) -- (\k+5:0.5cm) -- cycle;
    };  % 注意这里加分号
  };  % 结束 for 语句
}
\end{tikzpicture}
```




```
\tikzmath{
  \xrdsize=1.8; \yrdsize=1.2;
  \focusrdsize=sqrt(abs(\xrdsize*\xrdsize-\yrdsize*\yrdsize));
  coordinate \focuspointA,\focuspointB;
  if \xrdsize>\yrdsize then {%
    \focuspointA=(\focusrdsize,0); \focuspointB=(-\focusrdsize,0);
  } else {%
    \focuspointA=(0,\focusrdsize); \focuspointB=(0,-\focusrdsize);
  };
}
\begin{tikzpicture}
  \draw (0,0) ellipse[x radius=\xrdsize cm, y radius=\yrdsize cm];
  \node [below] at(120:\xrdsize cm and \yrdsize cm) {$E$};
  \fill (\focuspointA) circle [radius=1pt] node [below] {$F_2$};
  \fill (\focuspointB) circle [radius=1pt] node [below] {$F_1$};
\end{tikzpicture}
```

60.2.7 关键词 function，return

句子 return ⟨*expression*⟩; 或者 return {⟨*expression*⟩}; 导致

```
\pgfmathparse{⟨expression⟩}\tikz@math@parse
```

关键词 function 用于声明函数，句式是：

function ⟨*fun name*⟩ (⟨*arguments list*⟩) {⟨*definition body*⟩};

function ⟨*fun name*⟩ () {⟨*definition body*⟩};

function ⟨*fun name*⟩ {⟨*definition body*⟩};

上面句式中参数 ⟨*fun name*⟩ 是函数名称 (一个字符串)；⟨*arguments list*⟩ 是用逗号分隔的变量列表；⟨*definition body*⟩ 是定义内容；包裹 ⟨*definition body*⟩ 的花括号不能省略。

假设列表 ⟨*arguments list*⟩ 是 \⟨*var 1*⟩，\⟨*var 2*⟩，...，那么以上句式最终导致：

```
\pgfmathdeclarefunction{⟨fun name⟩}{⟨ 变量总个数⟩}%
  {%
    \def\return{0}%
    \tikz@math{%
      \⟨var 1⟩=#1;
      \⟨var 2⟩=#2;
      ...
      ⟨definition body⟩
    }%
  }%
\tikz@math@parse
```

所以 ⟨*definition body*⟩ 是一些能被 \tikzmath 处理的语句。

在 ⟨*definition body*⟩ 中使用的变量最好不超出 ⟨*arguments list*⟩ 列出的变量。

在 ⟨*definition body*⟩ 的末尾可以使用 return {⟨*expression*⟩}; (在 ⟨*expression*⟩ 中使用的变量最好不超出 ⟨*arguments list*⟩ 列出的变量)，这样，当函数 ⟨*fun name*⟩ 完成计算后，计算结果就保存在宏 \pgfmathresult 中 (解析 ⟨*expression*⟩ 的结果)。

60.2.8 关键词 for

关键词 for 引起循环，它是 \foreach 命令的删减版。它的句式是：

for \⟨*var*⟩ in {⟨*list*⟩} {⟨*action*⟩};

for \⟨*var*⟩⟨*index*⟩ in {⟨*list*⟩} {⟨*action*⟩};

for \⟨*var*⟩{⟨*index*⟩} in {⟨*list*⟩} {⟨*action*⟩};

for 只能识别一个变量记号 \⟨*var*⟩, 多余的变量记号会被忽略。

变量 \⟨*var*⟩ 可以带有引用标识 ⟨*index*⟩, 只要 ⟨*index*⟩ 中不含单词 in, 就可以不用花括号包裹 ⟨*index*⟩.

变量 \⟨*var*⟩ 可以是未定义的，也可以是事先由关键词 integer, int, real, coordinate, count, length 声明的变量 (此时注意变量值应当与变量类型匹配)。

类似 \foreach 句子，在 ⟨*list*⟩ 中可以使用省略号。

⟨*action*⟩ 是一些能被 \tikzmath 处理的语句。如果 ⟨*action*⟩ 中不含分号，也可以省略包裹 ⟨*action*⟩ 的花括号。

关键词 for 导致命令 \tikz@math@for, 这个命令首先将计数器 \tikz@math@for@depth 的值加 1；在套嵌使用 for 语句时，这个计数器的值指示了当前的套嵌层次。

当一个 for 语句结束时，执行命令 \tikz@math@for@scan@end, 这个命令先将计数器 \tikz@math@for@depth 的值减 1，然后执行 \tikz@math@parse 解析后面的句子。所以，当各个层次的 for 语句结束时，计数器 \tikz@math@for@depth 的值是 0.

注意 for 语句不会统计 ⟨*list*⟩ 中列表项 (变量值) 的个数。

这个语句的处理大致如下。

在套嵌 for 语句的情况下，用 \⟨*var i*⟩ 代表第 i 层循环的变量；用 ⟨*action i*⟩ 代表第 i 层循环的循环体；用 ⟨*list i*⟩ 代表第 i 层循环的变量值列表；用 ⟨*value* i, j⟩ 代表由 ⟨*list i*⟩ 决定的第 j 个变量值，其中 $i, j > 0$, 那么对 ⟨*value* i, j⟩ 的处理是：

1. 将当前的变量值 ⟨*value* i, j⟩ 保存到 `\tikz@math@for@value`

```
\def\tikz@math@for@value{⟨value i, j⟩}
```

2. 将当前变量 `\`⟨*var* i⟩ 的引用标识 ⟨*index*⟩ 保存到 `\tikz@math@current@index`
3. 将当前循环体 ⟨*action* i⟩ 保存到 `\tikz@math@action`

```
\tikz@math@for@namegetvalue{action}{\tikz@math@action}%
```

4. 为 `\`⟨*var* i⟩ 赋值

```
\tikz@math@doassignment{\⟨var i⟩}{⟨value i, j⟩}
```

这个命令将 ⟨*value* i, j⟩ 处理为 ⟨*VALUE* i, j⟩, 保存在宏 `\tikz@math@last@assigned@value` 中。

5. 保存 ⟨*VALUE* i, j⟩

```
\tikz@math@for@namelet{value}=\tikz@math@last@assigned@value%
```

这实际是把控制序列

```
\csname tikz@math@for@def@value@i\endcsname
```

的值 let 为 ⟨*VALUE* i, j⟩.

6. 执行循环体 ⟨*action* i⟩

```
\tikz@math@for@namegetvalue{execute}{\tikz@math@execute}%
\expandafter\tikz@math@execute\expandafter{\tikz@math@action}%
```

在默认下这就是

```
\tikz@math{⟨action i⟩}
```

7. 保存变量值 ⟨*VALUE* $i, j-1$⟩,

```
\tikz@math@for@namegetvalue{prevvalue}{\tikz@math@prevvalue}%
\tikz@math@for@namelet{prevprevvalue}=\tikz@math@prevvalue%
```

这实际是把控制序列

```
\csname tikz@math@for@def@prevprevvalue@i\endcsname
```

的值 let 为 ⟨*VALUE* $i, j-1$⟩.

8. 保存变量值 ⟨*VALUE* i, j⟩,

```
\tikz@math@for@namegetvalue{value}{\tikz@math@value}%
\tikz@math@for@namelet{prevvalue}=\tikz@math@value%
```

这实际是把控制序列

```
\csname tikz@math@for@def@prevvalue@i\endcsname
```

的值 let 为 ⟨*VALUE* i, j⟩.

9. 处理下一个变量值 ⟨*value* $i, j+1$⟩

可见在上面步骤执行循环体 ⟨*action* i⟩ 的过程中：

- 控制序列

```
\csname tikz@math@for@def@value@i\endcsname
```

的值是 ⟨*VALUE* i, j⟩, 使用

```
\tikz@math@for@namegetvalue{value}{\value@ij}
```

可以将 ⟨*VALUE* i, j⟩ 保存到宏 `\value@ij`

- 控制序列

```
\csname tikz@math@for@def@prevvalue@i\endcsname
```

的值是 ⟨*VALUE* $i, j-1$⟩, 使用

```
\tikz@math@for@namegetvalue{prevvalue}{\prevvalue@ij}
```

可以将 ⟨*VALUE* $i, j-1$⟩ 保存到宏 \prevvalue@ij

- 控制序列

  \csname tikz@math@for@def@prevprevvalue@⟨*i*⟩\endcsname

  的值是 ⟨*VALUE* $i, j-2$⟩, 使用

```
\tikz@math@for@namegetvalue{prevprevvalue}{\prevprevvalue@ij}
```

可以将 ⟨*VALUE* $i, j-2$⟩ 保存到宏 \prevprevvalue@ij

注意，变量值 ⟨*value* $i, -1$⟩, ⟨*value* $i, 0$⟩ 是空的。

$x_1 = 5, x_2 = 50, y = 2250$

```
\tikzmath{
  int \x, \y;
  \y = 0;
  for \x1 in {1,...,5}{
    for \x2 in {10,20,...,50}{
      \y = \y+\x1*\x2;
    };
  };
}
$x_1=\x1, x_2=\x2, y=\y$
```

60.3 以命令或 let 开头的语句

以命令 \⟨*cmd*⟩ 开头的语句是 \⟨*cmd*⟩...;, 以 let 开头的语句是 let \⟨*cmd*⟩...;, 这两种句子都导致

```
\tikz@math@parse@nokeyword\⟨cmd⟩...;
```

以 let 开头的语句见前文，下面主要看以命令 \⟨*cmd*⟩ 开头的语句。

注意，在句子 \⟨*cmd*⟩⟨*following*⟩; 中，如果 ⟨*following*⟩ 中含有分号，那么：

- 可以尝试用花括号将整个 ⟨*following*⟩ 包裹起来，这个做法未必被总是可接受的。
- 可以尝试把 ⟨*following*⟩ 中的分号单独用花括号包裹起来，这个做法也未必被总是可接受的。

60.3.1 命令的属性

命令 \⟨*cmd*⟩ 可以有以下属性：

- 把 \⟨*cmd*⟩ 看作一个记号，那么它有记号类别属性，这些类别体现在下面命令中：

  \tikz@math@getmeaning⟨*a token*⟩

  本命令用 \meaning 检查参数 ⟨*a token*⟩ 的类型，其类型可能是

  – macro, 包括未定义的情况
  – dimen, 包括弹性尺寸和刚性尺寸
  – count, 计数器
  – null, 除了前 3 种类型，其他类型都属于这一类型

  检查类型后，将类型保存在宏 \tikz@math@meaning 中。

  macro

```
\makeatletter
\tikz@math@getmeaning\aaaa@bbbb
\tikz@math@meaning
\makeatother
```

- 把 \⟨*cmd*⟩ 看作一个变量，那么它有变量类别属性，可以是 `coordinate`, `integer`, `real`.
- 是否被关键词 `let` 引导，即是否写出 `let` \⟨*cmd*⟩`...;` 这样的句子。
- \⟨*cmd*⟩ 是否有引用标识，见下文。

60.3.2 引用标识

以命令 \⟨*cmd*⟩ 开头的语句有 2 种情况：

第一 不带引用标识的句式：\⟨*cmd*⟩=⟨*something*⟩;, 例如

```
\aaaa=...;
```

第二 带引用标识的句式：\⟨*cmd*⟩⟨*index*⟩=⟨*something*⟩;, 例如

```
\aaaa3=...;
\bbbb{12}=...;
\cccc{index}=...;
```

引用标识的作用如下面的例子：

XXX
```
\makeatletter
\def\test@example#1{%
  \csname test@example@#1\endcsname%
}
\expandafter\def\csname test@example@index\endcsname{XXX}
\test@example{index}
\makeatother
```

上面例子中，`index` 用作命令 `\testexample` 的引用标识，作用是调用一个控制序列。

60.3.3 处理不带引用标识的句式

对 \⟨*cmd*⟩=⟨*something*⟩; 这种句式的处理是：

1. 清除 \⟨*cmd*⟩ 的引用命令

```
\tikz@math@clearvarindexed\⟨cmd⟩
```

\tikz@math@clearvarindexed\⟨*cmd*⟩

本命令的定义是：

```
\def\tikz@math@clearvarindexed#1{%
    \expandafter\let\csname tikz@math@var@subtype@\string#1\endcsname=\relax%
}%
```

也就是说，如果之前有 \⟨*cmd*⟩⟨*index*⟩=...;, 那么此后 ⟨*index*⟩ 不能再作为 \⟨*cmd*⟩ 引用标识。

2. 清空宏 `\tikz@math@current@index`

```
\let\tikz@math@current@index=\pgfutil@empty
```

3. 执行

```
\tikz@math@doassignment{\⟨cmd⟩}{⟨something⟩}
\tikz@math@parse
```

60.3.4 处理带引用标识的句式

对 \⟨*cmd*⟩⟨*index*⟩=⟨*something*⟩; 这种句式的处理是：

1. 执行

```
\tikz@math@setvarindexed\⟨cmd⟩
```

\tikz@math@setvarindexed\⟨*cmd*⟩

本命令的定义是：

```
\def\tikz@math@setvarindexed#1{%
    \expandafter\let\csname
    tikz@math@var@subtype@\string#1\endcsname=\tikz@math@subtype@indexed%
}%
```

2. 定义宏 \tikz@math@current@index

```
\edef\tikz@math@current@index{⟨index⟩}
```

3. 执行

```
\tikz@math@doassignment{\⟨cmd⟩}{⟨something⟩}
\tikz@math@parse
```

可见一个命令 \⟨*cmd*⟩ 是否有引用标识决定于控制序列

\csname tikz@math@var@subtype@\string\⟨*cmd*⟩\endcsname

是否有定义。

60.3.5 赋值命令

\tikz@math@doassignment\⟨*cmd*⟩{⟨*something*⟩}

本命令利用 ⟨*something*⟩ 为 \⟨*cmd*⟩ “赋值”。

本命令：

- 如果 \⟨*cmd*⟩ 是 coordinate 类型的变量，那么 ⟨*something*⟩ 应当是 TikZ 的坐标或关于坐标的算式，
  - 如果 ⟨*something*⟩ 中含有符号 $, 或者没有载入 TikZ 的 calc 库，则

    ```
    \tikz@scan@one@point\tikz@math@assign@coordinate⟨something⟩
    ```

  - 如果 ⟨*something*⟩ 中不含有符号 $, 但载入了 TikZ 的 calc 库，则

    ```
    \tikz@scan@one@point\tikz@math@assign@coordinate($⟨something⟩$)
    ```

  参考 \tikz@scan@one@point → P. 710.
- 如果 \⟨*cmd*⟩ 不是 coordinate 类型的变量，而 \iftikz@math@let 的真值为 true, 也就是说，实际写出的句子是以关键词 let 开头的句子 let \⟨*cmd*⟩...;, 则
  - 如果 \⟨*cmd*⟩ 没有引用标识 ⟨*index*⟩, 则定义

    ```
    \edef\⟨cmd⟩{⟨something⟩}
    ```

  - 如果 \⟨*cmd*⟩ 有引用标识 ⟨*index*⟩, 则

    ```
    \tikz@math@assign@index{⟨cmd⟩}{⟨something⟩}
    ```

  然后设置真值 \tikz@math@letfalse.
- 如果 \⟨*cmd*⟩ 不是 coordinate 类型的变量，而 \iftikz@math@let 的真值为 false, 则
  - 如果 \⟨*cmd*⟩ 没有引用标识 ⟨*index*⟩, 则
    - 如果 \⟨*cmd*⟩ 是 dimen 类型的记号，那么 ⟨*something*⟩ 应当能被 \pgfmathsetlength 处理，则

      ```
      \pgfmathsetlength{\⟨cmd⟩}{⟨something⟩}%
      \let\tikz@math@last@assigned@value=\pgfmathresult%
      ```

* 如果 \⟨*cmd*⟩ 是 count 类型的记号，那么 ⟨*something*⟩ 应当能被 \pgfmathsetcount 处理，则

```
\pgfmathsetcount{\⟨cmd⟩}{⟨something⟩}%
\let\tikz@math@last@assigned@value=\pgfmathresult%
```

* 如果 \⟨*cmd*⟩ 是 int 类型的变量，那么 ⟨*something*⟩ 应当能被 \pgfmathparse 处理，则

```
\pgfmathparse{int(⟨something⟩)}%
\let\tikz@math@last@assigned@value=\pgfmathresult%
\let\⟨cmd⟩=\pgfmathresult%
```

* 如果 \⟨*cmd*⟩ 不是以上情况，那么 ⟨*something*⟩ 应当能被 \pgfmathparse 处理，则

```
\pgfmathparse{⟨something⟩}%
\let\tikz@math@last@assigned@value=\pgfmathresult%
\let\⟨cmd⟩=\pgfmathresult%
```

– 如果 \⟨*cmd*⟩ 有引用标识 ⟨*index*⟩, 则

* 如果 \⟨*cmd*⟩ 是 int 类型的变量，那么 ⟨*something*⟩ 应当能被 \pgfmathparse 处理，则

```
\pgfmathparse{int(⟨something⟩)}%
\tikz@math@assign@index{⟨cmd⟩}{⟨something⟩}
\let\tikz@math@last@assigned@value=\pgfmathresult%
```

* 如果 \⟨*cmd*⟩ 不是 int 类型的变量，那么 ⟨*something*⟩ 应当能被 \pgfmathparse 处理，则

```
\pgfmathparse{⟨something⟩}%
\tikz@math@assign@index{⟨cmd⟩}{⟨something⟩}
\let\tikz@math@last@assigned@value=\pgfmathresult%
```

\tikz@math@assign@coordinate{⟨*PGF point*⟩}

本命令：

1. 处理 PGF 点

```
\pgf@process{⟨PGF point⟩}
```

2. 检查 \⟨*cmd*⟩ 是否带有引用标识 ⟨*index*⟩,

- 如果 \⟨*cmd*⟩ 不带有引用标识 ⟨*index*⟩, 则

```
\edef\⟨cmd⟩{\the\pgf@x,\the\pgf@y}%
\edef\⟨cmd⟩x{\the\pgf@x}%
\edef\⟨cmd⟩y{\the\pgf@y}%
```

- 如果 \⟨*cmd*⟩ 带有引用标识 ⟨*index*⟩, 则

```
\tikz@math@assign@index{⟨cmd⟩}{\the\pgf@x,\the\pgf@y}%
\tikz@math@assign@index{⟨cmd⟩x}{\the\pgf@x}%
\tikz@math@assign@index{⟨cmd⟩y}{\the\pgf@y}%
```

\tikz@math@assign@index{⟨⟨*cmd*⟩ *name*⟩}{⟨*something*⟩}

当 \⟨⟨*cmd*⟩ *name*⟩ 带有引用标识 ⟨*index*⟩ 时执行本命令。本命令导致 2 个定义：

```
\def\csname⟨⟨cmd⟩ name⟩\endcsname#1{\csname
    tikz@math@var@indexed@⟨⟨cmd⟩ name⟩@#1\endcsname}%
\edef\csname
    tikz@math@var@indexed@⟨⟨cmd⟩ name⟩@⟨index⟩\endcsname{⟨something⟩}\relax%
```

也就是说，\⟨⟨*cmd*⟩ *name*⟩⟨*index*⟩ 导致

\csname tikz@math@var@indexed@⟨⟨*cmd*⟩ *name*⟩@⟨*index*⟩\endcsname

导致 ⟨*something*⟩.

综合以上，对 \⟨*cmd*⟩ 的赋值有以下情况：

1. 如果 \⟨*cmd*⟩ 被 `let` 引导，
   - 如果 \⟨*cmd*⟩ 没有引用标识 ⟨*index*⟩, 则

     ```
     \edef\⟨cmd⟩{⟨something⟩}
     ```

   - 如果 \⟨*cmd*⟩ 有引用标识 ⟨*index*⟩, 则定义宏 \⟨*cmd*⟩, 引用标识 ⟨*index*⟩ 可以用作它的参数：\⟨*cmd*⟩{⟨*index*⟩}, 这返回 ⟨*something*⟩ 的彻底展开值。
2. 如果 \⟨*cmd*⟩ 是 `coordinate` 类型的命令，
   - 如果 \⟨*cmd*⟩ 没有引用标识 ⟨*index*⟩, 则定义 3 个宏 \⟨*cmd*⟩, \⟨*cmd*⟩x, \⟨*cmd*⟩y

     ```
     \edef\⟨cmd⟩{\the\pgf@x,\the\pgf@y}%
     \edef\⟨cmd⟩x{\the\pgf@x}%
     \edef\⟨cmd⟩y{\the\pgf@y}%
     ```

     ```
     \tikzmath{
       coordinate \APoint;
       \APoint=(1,2);
     }
     \APoint\par
     \APointx\par
     \APointy
     ```

     28.45274pt,56.90549pt
     28.45274pt
     56.90549pt

   - 如果 \⟨*cmd*⟩ 有引用标识 ⟨*index*⟩, 则定义 3 个宏 \⟨*cmd*⟩, \⟨*cmd*⟩x, \⟨*cmd*⟩y, 引用标识 ⟨*index*⟩ 可以用作它们的参数：\⟨*cmd*⟩{⟨*index*⟩}, \⟨*cmd*⟩x{⟨*index*⟩}, \⟨*cmd*⟩y{⟨*index*⟩}
3. 如果 \⟨*cmd*⟩ 是 `dimen` 类型的记号，
   - 如果 \⟨*cmd*⟩ 没有引用标识 ⟨*index*⟩, 则

     ```
     \pgfmathsetlength{\⟨cmd⟩}{⟨something⟩}%
     \let\tikz@math@last@assigned@value=\pgfmathresult%
     ```

   - 如果 \⟨*cmd*⟩ 有引用标识 ⟨*index*⟩,
     - 如果 \⟨*cmd*⟩ 是 `int` 类型的变量，那么 ⟨*something*⟩ 应当能被 `\pgfmathparse` 处理，则
       (a) `\pgfmathparse{int(`⟨*something*⟩`)}`
       (b) 则定义宏 \⟨*cmd*⟩, 引用标识 ⟨*index*⟩ 可以用作它的参数:\⟨*cmd*⟩{⟨*index*⟩}, 这返回 ⟨*something*⟩ 的彻底展开值。
       (c) `\let\tikz@math@last@assigned@value=\pgfmathresult`
     - 如果 \⟨*cmd*⟩ 不是 `int` 类型的变量，那么 ⟨*something*⟩ 应当能被 `\pgfmathparse` 处理，则
       (a) `\pgfmathparse{`⟨*something*⟩`}`
       (b) 则定义宏 \⟨*cmd*⟩, 引用标识 ⟨*index*⟩ 可以用作它的参数:\⟨*cmd*⟩{⟨*index*⟩}, 这返回 ⟨*something*⟩ 的彻底展开值。
       (c) `\let\tikz@math@last@assigned@value=\pgfmathresult`
4. 如果 \⟨*cmd*⟩ 是 `count` 类型的记号，
   - 如果 \⟨*cmd*⟩ 没有引用标识 ⟨*index*⟩, 则

     ```
     \pgfmathsetcount{\⟨cmd⟩}{⟨something⟩}%
     \let\tikz@math@last@assigned@value=\pgfmathresult%
     ```

   - 如果 \⟨*cmd*⟩ 有引用标识 ⟨*index*⟩,
     - 如果 \⟨*cmd*⟩ 是 `int` 类型的变量，那么 ⟨*something*⟩ 应当能被 `\pgfmathparse` 处理，则
       (a) `\pgfmathparse{int(`⟨*something*⟩`)}`
       (b) 则定义宏 \⟨*cmd*⟩, 引用标识 ⟨*index*⟩ 可以用作它的参数:\⟨*cmd*⟩{⟨*index*⟩}, 这返回 ⟨*something*⟩

的彻底展开值。
(c) \let\tikz@math@last@assigned@value=\pgfmathresult
– 如果 \⟨cmd⟩ 不是 int 类型的变量，那么 ⟨something⟩ 应当能被 \pgfmathparse 处理，则
(a) \pgfmathparse{⟨something⟩}
(b) 则定义宏 \⟨cmd⟩, 引用标识 ⟨index⟩ 可以用作它的参数:\⟨cmd⟩{⟨index⟩}, 这返回 ⟨something⟩ 的彻底展开值。
(c) \let\tikz@math@last@assigned@value=\pgfmathresult

5. 如果 \⟨cmd⟩ 不是以上情况，
- 如果 \⟨cmd⟩ 没有引用标识 ⟨index⟩,
– 如果 \⟨cmd⟩ 是 int 类型的变量，那么 ⟨something⟩ 应当能被 \pgfmathparse 处理，则
(a) \pgfmathparse{int(⟨something⟩)}
(b) \let\tikz@math@last@assigned@value=\pgfmathresult
(c) \let\⟨cmd⟩=\pgfmathresult
– 如果 \⟨cmd⟩ 不是 int 类型的变量，那么 ⟨something⟩ 应当能被 \pgfmathparse 处理，则
(a) \pgfmathparse{⟨something⟩}
(b) \let\tikz@math@last@assigned@value=\pgfmathresult
(c) \let\⟨cmd⟩=\pgfmathresult
- 如果 \⟨cmd⟩ 有引用标识 ⟨index⟩,
– 如果 \⟨cmd⟩ 是 int 类型的变量，那么 ⟨something⟩ 应当能被 \pgfmathparse 处理，则
(a) \pgfmathparse{int(⟨something⟩)}
(b) 则定义宏 \⟨cmd⟩, 引用标识 ⟨index⟩ 可以用作它的参数:\⟨cmd⟩{⟨index⟩}, 这返回 ⟨something⟩ 的彻底展开值。
(c) \let\tikz@math@last@assigned@value=\pgfmathresult
– 如果 \⟨cmd⟩ 不是 int 类型的变量，那么 ⟨something⟩ 应当能被 \pgfmathparse 处理，则
(a) \pgfmathparse{⟨something⟩}
(b) 则定义宏 \⟨cmd⟩, 引用标识 ⟨index⟩ 可以用作它的参数:\⟨cmd⟩{⟨index⟩}, 这返回 ⟨something⟩ 的彻底展开值。
(c) \let\tikz@math@last@assigned@value=\pgfmathresult

可见 \⟨cmd⟩ 的某些属性：

1. 是否有引用标识 ⟨index⟩
2. 是否被关键词 let 引导
3. 是否 dimen, count 类型的记号
4. 是否 coordinate, int, real 类型的变量

其中第 1 个属性可以与其他属性兼容；而后 3 个属性也有可能相互兼容，但比较复杂，最好让它们相互独立；也就是说，例如，如果 \⟨cmd⟩ 已经被声明为 dimen 类型的记号，那么就不要再用关键词 let 引导 \⟨cmd⟩.

还可以看出，除了关键词 let 引导 \⟨cmd⟩ 的情况外，其他为 \⟨cmd⟩ 赋值的情况都要用到命令 \pgfmathparse, 参考 \tikz@scan@one@point →P. 710.

60.4 与 fpu 库的协作

math 库一开始就会调用 fpu 库：

```
\usetikzlibrary{fpu}
```

在 \pgfkeys{/pgf/fpu} 的情况下，命令 \pgfmathparse 等于 \pgfmathfloatparse→P. 596, 这对所有用到命令 \pgfmathparse 的句子都有影响，其中，为 coordinate 变量赋值的句子一般会产生错误。由于 print 句子会被放在一个组中执行，所以可以在中 print 句子启用 {/pgf/fpu=true}.

第六十一章　matrix 库

TikZ Library matrix

```
\usetikzlibrary{matrix} % LaTeX and plain TeX
\usetikzlibrary[matrix] % ConTeXt
```

这个库定义了一些 style, option 用于创建矩阵。

61.1　矩阵中的 node

如果一个 TikZ 矩阵的元素都是 node，那么用下一个矩阵选项会比较便利：

`/tikz/matrix of nodes`　(no value)

这个选项会在每个元素代码的开头加“`\node{`”，在每个元素代码的结尾加“`};`”，所以，如果元素代码是文字、数字、数学模式、LaTeX 表格等内容，这个选项会把元素代码完善为一个完整的 node 语句。并且，这个选项还会把元素图形的锚位置（anchor）设为 `base`，还会将每个元素图形的名称（name）设为 ⟨*matrix name*⟩-⟨*row number*⟩-⟨*column number*⟩ 这种形式，以便于引用。

8 1 6
3 5 7
4 9 2

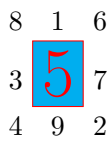
```
\begin{tikzpicture}
  \matrix (magic) [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & 7 \\
    4 & 9 & 2 \\
  };
  \draw[thick,red,->] (magic-1-1) |- (magic-2-3);
\end{tikzpicture}
```

如果要单独设置某个元素图形的样式，可以使用 `row` ⟨*row number*⟩ `column` ⟨*column number*⟩ 这个样式 key 来设置：

8 1 6
3 5 7
4 9 2

```
\begin{tikzpicture}
  [row 2 column 2/.style={font=\Huge,red,shift={(0,-1mm)}}]
  \matrix [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & 7 \\
    4 & 9 & 2 \\
  };
\end{tikzpicture}
```

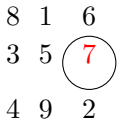
选项 `matrix of nodes` 只是把“`\node{`”和“`};`”加在元素代码上，针对单个元素图形的特殊选项还需要个别设置。在使用选项 `matrix of nodes` 的情况下，针对单个元素图形的选项放在方括号里，方括号还必须放在该元素代码的前面，还要在方括号前后加竖线（作为定界）。



```
\begin{tikzpicture}
  [row 2 column 2/.style={font=\Huge,red,shift={(0,-1mm)}}]
  \matrix [matrix of nodes]
  {
    8 &                    1 & 6 \\
    3 & |[draw,fill=cyan]|5 & 7 \\
    4 &                    9 & 2 \\
  };
\end{tikzpicture}
```

其中需要在方括号前后加竖线定界符，是因为分列符 & 本身可以带有方括号选项，如果不加竖线定界符，Ti*k*Z 会把方括号看作是属于分列符 & 的。

在使用选项 matrix of nodes 的情况下，如果某个元素图形的代码以 \path，\draw，\node，\fill 等命令开头，或者以能够展开为这些命令的符号串开头，那么对于该元素而言，选项 matrix of nodes 的添加符号“\node{”和“};”的作用会被抑制，这样就可以绘制该元素图形。



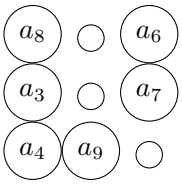
```
\begin{tikzpicture}
  \matrix [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & \node[red]{7}; \draw(0,0) circle(10pt);\\
    4 & 9 & 2 \\
  };
\end{tikzpicture}
```

/tikz/matrix of math nodes (no value)

这个选项的作用与 matrix of nodes 类似，它会在每个元素代码的开头加“\node{$”，在每个元素代码的结尾加“$};”。

/tikz/nodes in empty cells=⟨*true or false*⟩ (default true)

这个选项会在空元素（没有代码的元素）的位置处设置一个内容为空的 node.

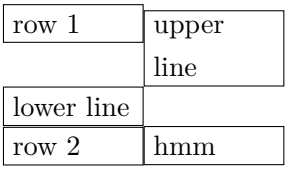


```
\begin{tikzpicture}
  \matrix [matrix of math nodes,nodes={circle,draw},nodes in empty
  ↪  cells]
  {
    a_8 &     & a_6 \\
    a_3 &     & a_7 \\
    a_4 & a_9 &     \\
  };
\end{tikzpicture}
```

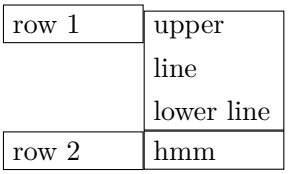
61.2 换行符号与矩阵行的结束符号

符号 \\ 是 TEX 的换行符号，也是 Ti*k*Z 矩阵的分行符号。如果矩阵的某个元素是以文字为内容的 node，并且文字内使用 \\ 换行，就可能会造成歧义。此时，应用以下规则：

1. 在矩阵内部，\\ 是分行符号。
2. 如果在 \\ 与它前面的分列符 & 之间只有一层花括号，并且开括号“{”紧跟在 & 之后，那么 \\ 是属于这层花括号之内的文本换行符号。




```
\begin{tikzpicture}
  \matrix [matrix of nodes,nodes={text width=16mm,draw}]
  {
    row 1 & upper line \\ lower line \\
    row 2 & hmm \\
  };
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [matrix of nodes,nodes={text width=16mm,draw}]
  {
    row 1 & {upper line \\ lower line} \\
    row 2 & hmm \\
  };
\end{tikzpicture}
```

注意 `a&b{c\\d}\\` 这种形式是错的，因为 `&` 与 `{` 这两个符号不是紧邻的，它们之间有 `b`.

61.3 定界符

定界符通常具有括号的形式，或具有类似括号用处的其它符号，例如矩阵两侧的括号是一种定界符。任何 node 都可以带有定界符，只需要为它添加以下选项。

/tikz/left delimiter=⟨*delimiter*⟩ (no default)

当一个 node 带有这个选项后，⟨*delimiter*⟩ 会成为这个 node 的左侧定界符。这里要求 node 有各种标准的 anchor 位置（`north`, `south` 等）。⟨*delimiter*⟩ 是那些可用作 TEX 数学模式命令 `\left` 的参数的符号。

/tikz/right delimiter=⟨*delimiter*⟩ (no default)

类似 `left delimiter`.

/tikz/above delimiter=⟨*delimiter*⟩ (no default)

类似 `left delimiter`.

/tikz/below delimiter=⟨*delimiter*⟩ (no default)

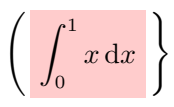
类似 `left delimiter`.

/tikz/every left delimiter (style, initially empty)

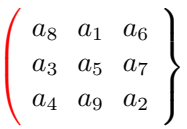
/tikz/every right delimiter (style, initially empty)

/tikz/every above delimiter (style, initially empty)

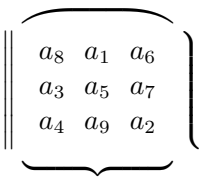
/tikz/every below delimiter (style, initially empty)




```
\begin{tikzpicture}
  \node [fill=red!20,left delimiter=(,right delimiter=\}]
    {$\displaystyle\int_0^1 x\,\mathrm{d}x$};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  [every left delimiter/.style={red,xshift=1ex},
   every right delimiter/.style={xshift=-1ex}]
  \matrix [matrix of math nodes,
    left delimiter=(,  right delimiter=\}]
  {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [matrix of math nodes,%
    left delimiter=\|, right delimiter=\rmoustache,%
    above delimiter=(, below delimiter=\}]
  {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```

第六十二章　三点透视图程序库

TikZ Library perspective

```
\usetikzlibrary{perspective} % LaTeX and plain TeX
\usetikzlibrary[perspective] % ConTeXt
```

这个程序库提供的工具能绘制有 1 个、2 个或 3 个没影点的透视图。

62.1　Coordinate system three point perspective

坐标系统 three point perspective 与 xyz 坐标系统非常类似，它对坐标施加“透视投影”操作。这个坐标系统有一个别名 tpp.

坐标系统 tpp 以下面的矩阵为基础做计算：

```
\pgfmathsetmacro\pgf@H@tpp@aa{+1}
  \pgfmathsetmacro\pgf@H@tpp@ab{+0}
    \pgfmathsetmacro\pgf@H@tpp@ac{+0}
\pgfmathsetmacro\pgf@H@tpp@ba{+0}
  \pgfmathsetmacro\pgf@H@tpp@bb{+1}
    \pgfmathsetmacro\pgf@H@tpp@bc{+0}
\pgfmathsetmacro\pgf@H@tpp@ca{+0}
  \pgfmathsetmacro\pgf@H@tpp@cb{+0}
    \pgfmathsetmacro\pgf@H@tpp@cc{+1}
\pgfmathsetmacro\pgf@H@tpp@da{+0}
  \pgfmathsetmacro\pgf@H@tpp@db{+0}
    \pgfmathsetmacro\pgf@H@tpp@dc{+0}
```

以上代码定义的宏 \pgf@H@tpp@aa 等构成矩阵

$$\mathbf{M} = \begin{bmatrix} aa & ab & ac & 0 \\ ba & bb & bc & 0 \\ ca & cb & cc & 0 \\ da & db & dc & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

单位矩阵作为矩阵 $\left[\begin{smallmatrix} aa & ab & ac & 0 \\ ba & bb & bc & 0 \\ ca & cb & cc & 0 \\ da & db & dc & 1 \end{smallmatrix}\right]$ 的初始值。

创建透视图形的基本思路是：首先将矩阵 $\mathbf{M}$ 改为透视矩阵，然后用这个矩阵对某些线段的端点做变换，从而使得线段被变换，变换后的线段表现出透视效果。

坐标系统 tpp 的定义是：

```
\tikzdeclarecoordinatesystem{three point perspective}
{%
  \tikzset{cs/.cd,x=0,y=0,z=0,#1}%
  \pgfpointperspectivexyz{\tikz@cs@x}{\tikz@cs@y}{\tikz@cs@z}
}
```

```
\tikzaliascoordinatesystem{tpp}{three point perspective}
```

/tikz/cs/x=⟨number⟩ (no default, initially 0)

本选项提供坐标点的 x 分量，⟨number⟩ 不能带有单位。

/tikz/cs/y=⟨number⟩ (no default, initially 0)

本选项提供坐标点的 y 分量，⟨number⟩ 不能带有单位。

/tikz/cs/z=⟨number⟩ (no default, initially 0)

本选项提供坐标点的 z 分量，⟨number⟩ 不能带有单位。

\pgfpointperspectivexyz{⟨x⟩}{⟨y⟩}{⟨z⟩}

将本命令的参数看作齐次坐标 (⟨x⟩,⟨y⟩,⟨z⟩,1)，本命令：

1. 计算乘积和齐次化操作：

$$\begin{bmatrix} aa & ab & ac & 0 \\ ba & bb & bc & 0 \\ ca & cb & cc & 0 \\ da & db & dc & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w \end{pmatrix} \xrightarrow{\text{齐次化}} \begin{pmatrix} pp@x \\ pp@y \\ pp@z \\ 1 \end{pmatrix}$$

2. 规定尺寸寄存器 \pgf@x 和 \pgf@y 的值，

$$\begin{pmatrix} \backslash\text{pgf@x} \\ \backslash\text{pgf@y} \end{pmatrix} = pp@x \cdot \begin{pmatrix} xx \\ xy \end{pmatrix} + pp@y \cdot \begin{pmatrix} yx \\ yy \end{pmatrix} + pp@z \cdot \begin{pmatrix} zx \\ zy \end{pmatrix}$$

其中的 $\binom{xx}{xy}$, $\binom{yx}{yy}$, $\binom{zx}{zy}$ 分别是当前的 xyz 坐标系统的 x, y, z 轴的单位向量，参考 \pgfsetxvec → P. 253 等命令。寄存器 \pgf@x 和 \pgf@y 分别作为横坐标和纵半轴确定 canvas 坐标系中的一个点。

将矩阵 **M** 改为透视矩阵后，这个矩阵就决定了一个透视坐标系，这个透视坐标系以当前的 xyz 坐标系统为基础 (参照)。坐标 (tpp cs:x=⟨x⟩,y=⟨y⟩,z==⟨z⟩) 就代表这个透视坐标系中的一个点。

62.2 设置视角

/tikz/3d view={⟨azimuth⟩}{⟨elevation⟩} (default {-30}{15})

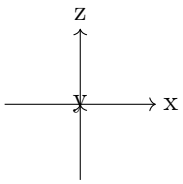
⟨azimuth⟩ 代表经度 θ, 经度是围绕 z 轴旋转的角度，绕 z 轴右旋为正；约定 y 轴负方向的经度是 0. ⟨elevation⟩ 代表纬度 φ, 北纬为正，南纬为负。默认值是 {-30}{15}.

本选项重新设置 xyz 坐标系，使得原点，单位球上的点 (θ, φ), “眼睛”这三点共线，从而确定视角。

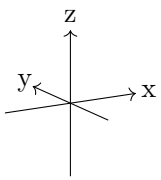
本选项将以下 canvas 坐标系中的点：

- \pgfpoint{$\cos(\theta)$ cm}{$-\sin(\theta)\sin(\varphi)$ cm}
- \pgfpoint{$\sin(\theta)$ cm}{$\cos(\theta)\sin(\varphi)$ cm}
- \pgfpoint{0 cm}{$\cos(\varphi)$ cm}

分别做成 xyz 坐标系的 x, y, z 轴的单位向量。



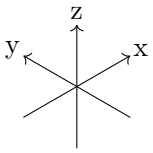

```
\begin{tikzpicture}[3d view={0}{0}]
  \draw[->] (-1,0,0) -- (1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-1,0) -- (0,1,0) node[pos=1.1]{y};
  \draw[->] (0,0,-1) -- (0,0,1) node[pos=1.1]{z};
\end{tikzpicture}
```




```
\begin{tikzpicture}[3d view]
  \draw[->] (-1,0,0) -- (1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-1,0) -- (0,1,0) node[pos=1.1]{y};
  \draw[->] (0,0,-1) -- (0,0,1) node[pos=1.1]{z};
\end{tikzpicture}
```

/tikz/isometric view (style, no value)

本选项选定一个特殊的视角，即 isometric view, 正等轴测图，等价于 `3d view={-45}{35.26}`, 其中 $35.26 \approx \arctan(1/\sqrt{2})$.




```
\begin{tikzpicture}[isometric view]
  \draw[->] (-1,0,0) -- (1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-1,0) -- (0,1,0) node[pos=1.1]{y};
  \draw[->] (0,0,-1) -- (0,0,1) node[pos=1.1]{z};
\end{tikzpicture}
```

上面的 isometric view 视角中，如果把画出的轴线看作是平面上的 3 条直线，那么这 3 条直线等分圆周角。

62.3 自定义透视

自定义透视就是自定义前面说的透视矩阵 $\mathbf{M}$, 即定义它的元素。

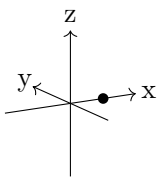
/tikz/perspective/p={(⟨x⟩,⟨y⟩,⟨z⟩)} (no default, initially (0,0,0))

本选项的定义是：

```
\pgfkeys{
  /perspective/.cd,
    p/.code args={(#1,#2,#3)}{
      \pgfmathparse{ifthenelse(#1,int(1),int(0))}
      \ifnum\pgfmathresult=0\else
        \pgfmathsetmacro\pgf@H@tpp@ba{#2/#1}
        \pgfmathsetmacro\pgf@H@tpp@ca{#3/#1}
        \pgfmathsetmacro\pgf@H@tpp@da{ 1/#1}
      \fi
    },
}
```

可见本选项修改透视矩阵 $\mathbf{M}$ 的元素 $ba = \frac{\langle y\rangle}{\langle x\rangle}, ca = \frac{\langle z\rangle}{\langle x\rangle}, da = \frac{1}{\langle x\rangle}$.

本选项规定的没影点 p 如下例子所示：




```
\begin{tikzpicture}[3d view]
  \draw[->] (-1,0,0) -- (1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-1,0) -- (0,1,0) node[pos=1.1]{y};
  \draw[->] (0,0,-1) -- (0,0,1) node[pos=1.1]{z};
  \fill[/perspective/p={(1,0,0)}] (tpp cs:x=1,y=0,z=0) circle
  ↪  [radius=2pt];
\end{tikzpicture}
```

参数 (⟨x⟩,⟨y⟩,⟨z⟩) 可以看作是当前 xyz 坐标系中的点。

/tikz/perspective/q={(⟨x⟩,⟨y⟩,⟨z⟩)} (no default, initially (0,0,0))

本选项修改透视矩阵 $\mathbf{M}$ 的元素 $ab = \frac{\langle x\rangle}{\langle y\rangle}, cb = \frac{\langle z\rangle}{\langle y\rangle}, db = \frac{1}{\langle y\rangle}$.

本选项设置没影点 q, 参数 (⟨x⟩,⟨y⟩,⟨z⟩) 可以看作是当前 xyz 坐标系中的点。

/tikz/perspective/r={(⟨x⟩,⟨y⟩,⟨z⟩)} (no default, initially (0,0,0))

本选项修改透视矩阵 $\mathbf{M}$ 的元素 $ac = \frac{\langle x\rangle}{\langle z\rangle}, bc = \frac{\langle y\rangle}{\langle z\rangle}, dc = \frac{1}{\langle z\rangle}$.

本选项设置没影点 r, 参数 (⟨x⟩,⟨y⟩,⟨z⟩) 可以看作是当前 xyz 坐标系中的点。

/tikz/perspective={⟨*vanishing points*⟩} (default p={(10,0,0)},q={(0,10,0)},r={(0,0,20)})

本选项的定义是：

```
\tikzset{
  perspective/.append code={\pgfkeys{/perspective/.cd,#1}},
  perspective/.default={
    p={(10,0,0)},
    q={(0,10,0)},
    r={(0,0,20)}},
}
```

可见本选项的默认值是 p={(10,0,0)},q={(0,10,0)},r={(0,0,20)}.

参数 ⟨*vanishing points*⟩ 应当是前缀路径为 /perspective 的选项，如前面定义的 /perspective/p, /perspective/q, /perspective/r.

本选项指定 3 个没影点 p, q, r.

以上选项会把透视矩阵 $\mathbf{M}$ 修改为

$$\begin{bmatrix} 1 & q_x/q_y & r_x/r_z & 0 \\ p_y/p_x & 1 & r_y/r_z & 0 \\ p_z/p_x & q_z/q_y & 1 & 0 \\ 1/p_x & 1/q_y & 1/r_z & 1 \end{bmatrix}$$

对这个矩阵的元素约定：如果元素的分母为 0, 则该元素为 0.

62.4 缺点

PGF 实际上使用 2D 坐标系统来模拟 3 维点，这对实现透视效果来说是个限制。使用本程序库时注意以下几点：

- 选项 shift, xshift, yshift, rotate around x, rotate around y, rotate around z 等没有正常的作用。
- 本程序库只接受 xyz 坐标系统的坐标，也就是说给出的坐标不能带有长度单位。
- 本程序库不兼容 3d 程序库中的多数选项。

62.5 例子

第六十三章　Plot Mark 库

TikZ Library plotmarks

```
\usepgflibrary{plotmarks} % LaTeX and plain TeX and pure pgf
\usepgflibrary[plotmarks] % ConTeXt and pure pgf
\usetikzlibrary{plotmarks} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[plotmarks] % ConTeXt when using TikZ
```

如果不调用这个程序库，在默认下只有 `*`，`+`，`x` 这 3 种类型的点标记可用。plotmarks 库提供多种类型的点标记。

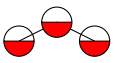
这个库定义的各点标记中，带有星号 `*` 的点标记可以填充颜色，例如 `oplus*`，`otimes*`，`square*`，`triangle*`，`diamond*`，`halfdiamond*`，`halfsquare*`，`halfsquare left*`，`pentagon*`，`halfcircle*`.

点标记可以被旋转：

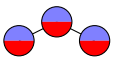
```
mark options={rotate=90}
every mark/.append style={rotate=90}.
```

`/pgf/mark color=`{⟨*color*⟩} (no default, initially empty)

这个选项可以为 `halfcircle`，`halfcircle*`，`halfdiamond*`，`halfsquare*` 这几种点标记设置填充色，其中带星号的点标记可以有两种填充色：一种填充色使用选项 `mark options=fill=`⟨*color*⟩ 设置，另一种填充色使用本选项设置。如果本选项值留空，则填充色是白色 (white, 这是初始值)。如果选项值是 `none`，则取消填充。



```
\tikz \draw plot[mark=halfcircle,mark size=2mm,mark color=red,mark
↪  options={fill=blue!50}]
 coordinates{(0,0)(0.5,0.25)(1,0)};
```



```
\tikz \draw plot[mark=halfcircle*,mark size=2mm,mark color=red,mark
↪  options={fill=blue!50}]
 coordinates{(0,0)(0.5,0.25)(1,0)};
```

上面例子看出，点标记 `halfcircle` 的一半是没有填充色的。

`/pgf/text mark=`{⟨*text*⟩} (no default, initially `p`)

把 ⟨*text*⟩ 用作点标记，⟨*text*⟩ 可以是任何 TEX 内容，比如文字，图形，数学公式，表格等。

`/pgf/text mark as node=`{⟨*boolean*⟩} (no default, initially `false`)

使用选项 `text mark=`{⟨*text*⟩} 后，如果本选项的值是 `true`, 则把 ⟨*text*⟩ 作为命令 `\node` 的内容——将 node 用作点标记；如果本选项的值是 `false`, 则把 ⟨*text*⟩ 作为命令 `\pgftext` 的内容——用作点标记。

`/pgf/text mark style=`{⟨*options for* `mark=text`⟩} (no default)

使用选项 `text mark=`{⟨*text*⟩} 后，这个选项设置点标记 ⟨*text*⟩ 的样式。

如果 `/pgf/text mark as node=false`,那么本选项的值只能是 `left`，`right`，`top`，`bottom`，`base`，`rotate` 等基本的选项。

如果 /pgf/text mark as node=true，那么在本选项的值中可以使用 node 操作能接受的诸多选项，包括 anchor，scale，fill，draw，rounded corners 等等。

第六十四章　shadings 程序库

TikZ Library shadings

```
\usepgflibrary{shadings} % LaTeX and plain TeX and pure pgf
\usepgflibrary[shadings] % ConTeXt and pure pgf
\usetikzlibrary{shadings} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shadings] % ConTeXt when using TikZ
```

这个程序库定义了数种颜色渐变模式。

颜色渐变用于填充路径，参考 \shade^{→P. 768}, /tikz/shade^{→P. 776}, /tikz/shading^{→P. 776}.

下文中的 axis，ball，radial 这三种模式在不调用本程序库时也可以直接使用，因为这三种模式是在基本层中定义的。关于颜色渐变的详细介绍参考基本层的内容。

Shading axis

这个颜色渐变模式是沿着 x 轴方向（横向）或 y 轴方向（纵向）的渐变。例如，在 x 轴方向指定左、中、右三种颜色，这个模式就能在这三种颜色之间实现渐变。

/tikz/top color=⟨*color*⟩ (no default)

这个选项设置纵向渐变的上部颜色。当使用这个选项时，系统会有以下反应：

1. 选定 shade 选项。
2. 选定 shading=axis 选项。
3. 中间颜色首先会被设定为上、下颜色的中间颜色。
4. 渐变的旋转角度设为 0 度。

```
\tikz \draw[top color=red] (0,0) rectangle (2,1);
```

/tikz/bottom color=⟨*color*⟩ (no default)

这个选项设置纵向渐变的下部颜色，作用与选项 top color 类似。

/tikz/middle color=⟨*color*⟩ (no default)

这个选项设置纵向渐变或者横向渐变的中部颜色，作用与选项 top color 类似，但不设置颜色渐变的方向角度。

注意，由于选项 top color，bottom color 会重设中间颜色，所以选项 middle color 应该用在这两个选项之后。

```
\tikz \draw[top color=white,bottom color=black,middle color=red]
  (0,0) rectangle (2,1);
```

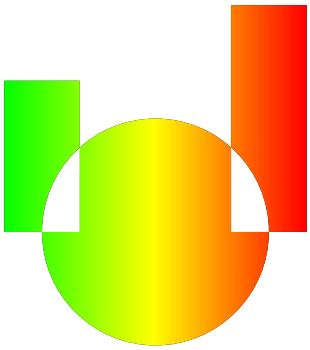
/tikz/left color=⟨*color*⟩ (no default)

这个选项设置横向渐变的左侧颜色，作用与选项 top color 类似。

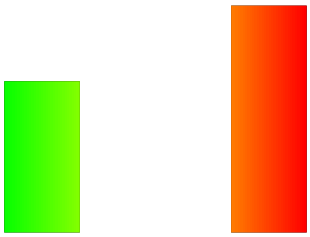
`/tikz/right color=`⟨*color*⟩ (no default)

这个选项设置横向渐变的右侧颜色，作用与选项 `top color` 类似。

当用颜色渐变填充路径时，在整个路径边界盒子内都充满渐变效果。下面图形中的空白是由于判断区域内外部的规则所导致的。



```
\tikz \fill [left color=green,right color=red,
    middle color=yellow]
  (0,0)rectangle(1,2)
  (2,0)circle(1.5cm)
  (3,0)rectangle(4,3);
```



```
\tikz \fill [left color=green,right color=red,
    middle color=yellow]
  (0,0)rectangle(1,2)
  (3,0)rectangle(4,3);
```

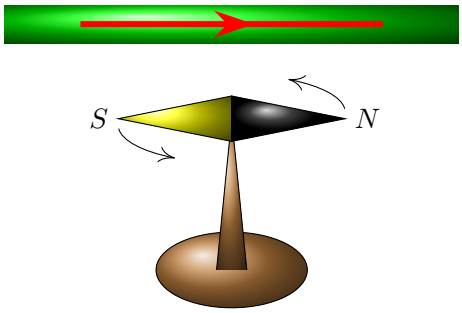
Shading `ball`

这个颜色渐变模式实现立体光影效果，渐变颜色默认用蓝色。

`/tikz/ball color=`⟨*color*⟩ (no default)

这个选项设置 ball 渐变的颜色，也会自动设置 `shade` 和 `shading=ball` 选项。

```
\begin{tikzpicture}
  \shade[ball color=white] (0,0) circle (2ex);
  \shade[ball color=red] (1,0) circle (2ex);
  \shade[ball color=black] (2,0) circle (2ex);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \fill [ball color=green] (-3,0) -- ++ (6,0)-- ++(0,0.5)-- ++(-6,0)--cycle;
  \filldraw [ball color=brown] (0,-3) ellipse (1 and 0.5);
  \filldraw [ball color=brown] (-0.2,-3)--(0,-1)--(0.2,-3) (-0.2,-3);
  \filldraw [ball color=yellow ]
    (-1.5,-1) node [left] {$S$} -- (0,-0.7) -- (0,-1.3) -- cycle;
  \filldraw [ball color=black]
    (1.5,-1) node [right] {$N$} -- (0,-0.7) -- (0,-1.3) -- cycle;
  \draw [line width=0.08cm,red,  -{Stealth[width=10pt,length=15pt,sep=1.7cm]
      Stealth[width=0.001pt 0 0 ,length=0.001pt 0 0]}]
    (-2,0.25)--(2,0.25);
  \draw [-{>[bend, width=6pt,length=6pt]}](0,-1) ++(5:1.5) arc (5:60:1.5 and 0.5);
  \draw [-{>[bend, width=6pt,length=6pt]}](0,-1) ++(185:1.5) arc (185:240:1.5 and 0.5);
\end{tikzpicture}
```

Shading bilinear interpolation

本模式通过指定矩形四个角的颜色，在该矩形内产生渐变效果。四个角的名称是 lower left, lower right, upper left, upper right，这四个名称也是四个选项，修改这四个选项的颜色可以改变渐变的颜色。

/tikz/lower left=⟨*color*⟩ (no default)

这个选项设置左下角的颜色，也会自动设置 shade 和 shading=bilinear interpolation 选项。

/tikz/upper left=⟨*color*⟩ (no default)

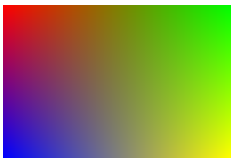
类似上一选项。

/tikz/lower right=⟨*color*⟩ (no default)

类似上一选项。

/tikz/upper right=⟨*color*⟩ (no default)

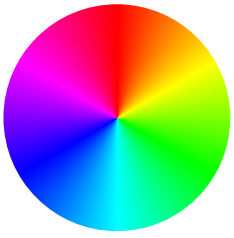
类似上一选项。



```
\tikz\shade[upper left=red,upper right=green,
    lower left=blue,lower right=yellow]
  (0,0) rectangle (3,2);
```

Shading color wheel

本模式生成一个色轮。



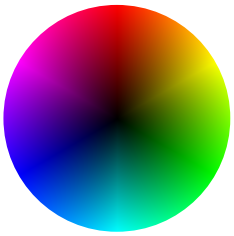
```
\tikz \shade[shading=color wheel] (0,0) circle (1.5);
```

下面的例子中使用了 even odd rule 规则来填充颜色：

```
\tikz \shade[shading=color wheel] [even odd rule]
  (0,0) circle (1.5)
  (0,0) circle (1);
```

Shading color wheel black center

本模式生成一个色轮，色轮中心的亮度是 0.



```
\tikz \shade[shading=color wheel black center]
  (0,0) circle (1.5);
```

Shading color wheel white center

本模式生成一个色轮，色轮中心的饱和度是 0.

```
\tikz \shade[shading=color wheel white center]
  (0,0) circle (1.5);
```

Shading Mandelbrot set

此模式会产生一个 Mandelbrot 分形集，是由 PDF 渲染器计算出来的，可以任意放缩，不是 bit 图。

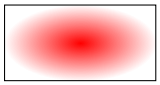
```
\tikz \shade[shading=Mandelbrot set] (0,0) rectangle (2,2);
```

Shading radial

本选项确定辐射渐变模式，辐射中心在被填充路径的边界盒子的中心，在默认下，内部中心颜色是灰色，外部边界颜色是白色。可以用下面的选项修改渐变颜色。

`/tikz/inner color=`⟨*color*⟩ (no default)

这个选项设置辐射渐变的内部中心的颜色，也会自动设置 `shade` 和 `shading=radial` 选项。



```
\tikz \draw[inner color=red] (0,0) rectangle (2,1);
```

`/tikz/outer color=`⟨*color*⟩ (no default)

这个选项设置辐射渐变的外部边界的颜色，也会自动设置 `shade` 和 `shading=radial` 选项。

```
\tikz \draw[outer color=red,inner color=white]
  (0,0) rectangle (2,1);
```

第六十五章　shadows 程序库

TikZ Library shadows

```
\usepgflibrary{shadows} % LaTeX and plain TeX and pure pgf
\usepgflibrary[shadows] % ConTeXt and pure pgf
\usetikzlibrary{shadows} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shadows] % ConTeXt when using TikZ
```

本程序库可以为路径或者 node 添加透明阴影。

65.1　Overview

阴影（shadow）通常是黑色或者灰色的，并且相对于路径有一定的位置偏移或者尺寸放缩。本程序库提供一些选项来实现阴影效果，实际上这是使用选项 /tikz/preaction →P. 781 两次利用路径的结果，第一用路径画阴影，第二次画出路径，阴影相对于路径有一定偏移。

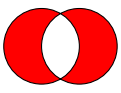
注意只能针对“路径”加阴影，不能针对整个 scope 图形画阴影。

65.2　一般的阴影选项

/tikz/general shadow=⟨*shadow options*⟩　(default empty)

这个选项只能作为路径或者 node 的选项。本选项的作用是，先执行 ⟨*shadow options*⟩, 利用路径完成阴影，然后对阴影图形做画布变换（放缩和平移），然后再画出路径。这个选项实际上使用选项 preaction 来工作。

在 ⟨*shadow options*⟩ 中可以使用路径前缀为 /tikz/ 的选项，例如选项 fill=⟨*color*⟩ 把填充色 ⟨*color*⟩ 作为阴影的颜色，如果不指定填充色，那么就默认不填充颜色，也就没有阴影效果。



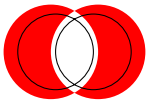
```
\tikz [even odd rule]
  \draw [general shadow={fill=red}]
    (0,0) circle (.5) (0.5,0) circle (.5);
```

还有下面的选项可以使用。

/tikz/shadow scale=⟨*factor*⟩　(no default, initially 1)

本选项设置阴影相对于原路径的尺寸比例，即使用放缩变换，放缩相对于阴影路径的边界盒子的中心。

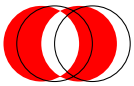
注意 PGF 用画布变换实现这个选项的作用。



```
\tikz [even odd rule]
  \draw [general shadow={fill=red,shadow scale=1.25}]
    (0,0) circle (.5) (0.5,0) circle (.5);
```

/tikz/shadow xshift=⟨*dimension*⟩　(no default, initially 0pt)

本选项设置阴影相对于原路径在水平方向的偏移量。注意 PGF 用画布变换实现此选项的作用。



```
\tikz [even odd rule]
  \draw [general shadow={fill=red,shadow xshift=-5pt}]
    (0,0) circle (.5) (0.5,0) circle (.5);
```

/tikz/shadow yshift=⟨*dimension*⟩ (no default, initially 0pt)

本选项设置阴影相对于原路径在竖直方向的偏移量。注意 PGF 用画布变换实现此选项的作用。

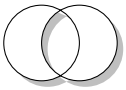
65.3 预定义的阴影

65.3.1 Drop Shadows

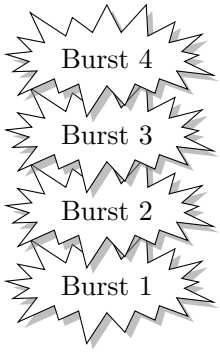
/tikz/drop shadow=⟨*shadow options*⟩ (default empty)

本选项给路径或 node 添加 drop shadow. 本选项实际使用 general shadow 来工作, ⟨*shadow options*⟩ 是关于图形外观的选项设置。在执行 ⟨*shadow options*⟩ 之前会先执行以下选项：

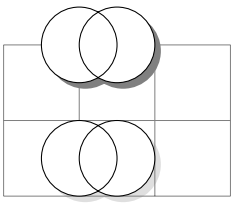
```
shadow scale=1, shadow xshift=.5ex, shadow yshift=-.5ex,
opacity=.5, fill=black!50, every shadow
```



```
\tikz [even odd rule]
  \filldraw [drop shadow,fill=white]
    (0,0) circle (.5) (0.5,0) circle (.5);
```




```
\begin{tikzpicture}
  \foreach \i in {1,...,4}
    \node[starburst,drop shadow,fill=white,draw]
      at (0,\i) {Burst \i};
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \filldraw [drop shadow={opacity=1},fill=white]
    (1,2) circle (.5) (1.5,2) circle (.5);
  \filldraw [drop shadow={opacity=0.25},fill=white]
    (1,.5) circle (.5) (1.5,.5) circle (.5);
\end{tikzpicture}
```

/tikz/every shadow (style, initially empty)

为每个阴影设置样式。

65.3.2 Copy Shadows

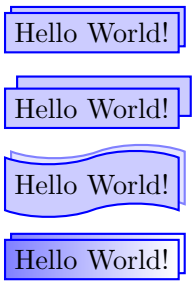
一个 copy shadow 实际上不是阴影，而是原路径的复制品，只是被原路径遮挡了，并且相对于原路径有一定偏移。

/tikz/copy shadow=⟨*shadow options*⟩ (default empty)

⟨*shadow options*⟩ 是关于图形外观的选项设置。在执行 ⟨*shadow options*⟩ 之前会先执行以下选项：

```
shadow scale=1, shadow xshift=.5ex, shadow yshift=-.5ex, every shadow
```

另外，主路径的 fill=⟨*color*⟩, draw=⟨*color*⟩ 也会应用于 copy shadow.




```
\begin{tikzpicture}
  \node [copy shadow,fill=blue!20,draw=blue,thick]
    {Hello World!};
  \node at (0,-1) [copy shadow={shadow xshift=1ex,
      shadow yshift=1ex},fill=blue!20,draw=blue,thick]
    {Hello World!};
  \node at (0,-2) [copy shadow={opacity=.5},tape,
      fill=blue!20,draw=blue,thick]
    {Hello World!};
  \node at (0,-3) [copy shadow={left color=blue!50},
      left color=blue!50,draw=blue,thick]
    {Hello World!};
\end{tikzpicture}
```

/tikz/double copy shadow=⟨*shadow options*⟩ (default empty)

将原路径复制两次来制作阴影，第二次复制时的平移量是第一次的 2 倍。


```
\begin{tikzpicture}
  \node [double copy shadow,fill=blue!20,draw=blue,thick]
    {Hello World!};
  \node at (0,-1) [double copy shadow={shadow xshift=1ex,
      shadow yshift=1ex},fill=blue!20,draw=blue,thick]
    {Hello World!};
  \node at (0,-2) [double copy shadow={opacity=.5},tape,
      fill=blue!20,draw=blue,thick]
    {Hello World!};
  \node at (0,-3) [double copy shadow={left color=blue!50},
      left color=blue!50,draw=blue,thick]
    {Hello World!};
\end{tikzpicture}
```

65.4 针对圆形的阴影

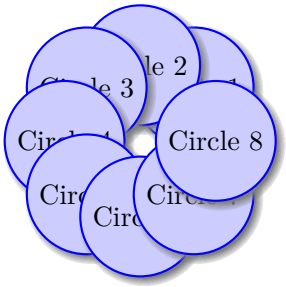
下面的阴影对圆形路径或圆形 node 的效果较好，如果用于其它形状的路径则可能会显得很奇怪。

/tikz/circular drop shadow=⟨*shadow options*⟩ (no default)

⟨*shadow options*⟩ 是关于图形外观的选项设置。在执行 ⟨*shadow options*⟩ 之前会先执行以下选项：

```
shadow scale=1.1, shadow xshift=.3ex, shadow yshift=-.3ex,
fill=black, path fading={circle with fuzzy edge 15 percent},
every shadow,
```

其中有选项 /tikz/path fading →P. 909 设置了填充色的透明度渐变。

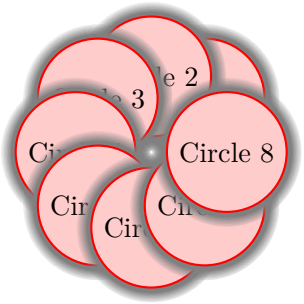



```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
    \node[circle,circular drop shadow,draw=blue,
        fill=blue!20,thick]
      at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

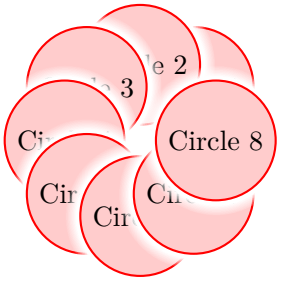
/tikz/circular glow=⟨*shadow options*⟩ (no default)

⟨*shadow options*⟩ 是关于图形外观的选项设置。在执行 ⟨*shadow options*⟩ 之前会先执行以下选项：

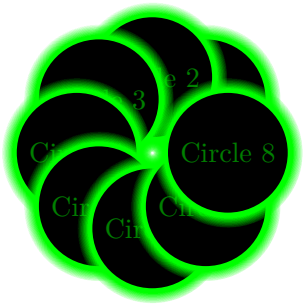
```
shadow scale=1.25, shadow xshift=0pt, shadow yshift=0pt,
fill=black, path fading={circle with fuzzy edge 15 percent},
every shadow,
```



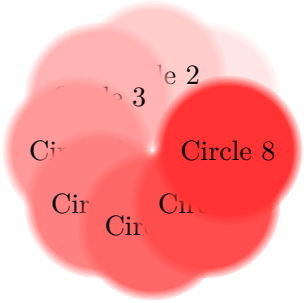

```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
    \node[circle,circular glow,
        fill=red!20,draw=red,thick]
      at (\i*45:1) {Circle \i};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
    \node[circle,circular glow={fill=white},
        fill=red!20,draw=red,thick]
      at (\i*45:1) {Circle \i};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
    \node[circle,circular glow={fill=green},
        fill=black,text=green!50!black]
      at (\i*45:1) {Circle \i};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
    \node[circle,circular glow={fill=red!\i0}]
      at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

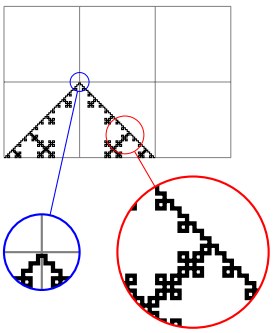
第六十六章　Spy 程序库：将图形的局部放大

TikZ Library spy

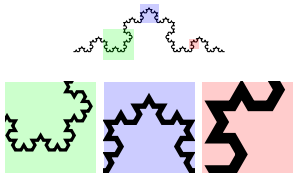
```
\usetikzlibrary{spy} % LaTeX and plain TeX
\usetikzlibrary[spy] % ConTeXt
```

有时候图形的某个局部有一些重要细节需要呈现，但这个局部的细节太细小，由于图形尺寸的限制，不便于用眼睛观察，此时可能需要将这个局部放大，就像用一个放大镜来观察这个局部位置一样。本宏包提供这样的局部放大功能。

66.1　将图形的某个局部放大



```
\begin{tikzpicture}[spy using outlines={
      circle, magnification=4, size=2cm, connect spies}]
  \draw [help lines] (0,0) grid (3,2);
  \draw [decoration=Koch curve type 1]
    decorate {decorate{decorate{decorate{(0,0) -- (2,0)}}}};
  \spy [red] on (1.6,0.3) in node [left] at (3.5,-1.25);
  \spy [blue, size=1cm] on (1,1) in node [right] at (0,-1.25);
\end{tikzpicture}
```



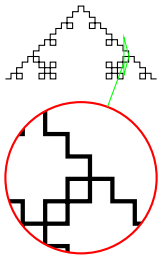
```
\begin{tikzpicture}[spy using overlays={size=12mm}]
  \draw [decoration=Koch snowflake]
    decorate {decorate{decorate{decorate{(0,0) -- (2,0)}}}};
  \spy [green,magnification=3] on (0.6,0.1) in node at (-0.3,-1);
  \spy [blue,magnification=5] on (1,0.5) in node at (1,-1);
  \spy [red,magnification=10] on (1.6,0.1) in node at (2.3,-1);
\end{tikzpicture}
```

如上面的例子所示，首先给 {tikzpicture} 环境或者 {scope} 环境添加选项 spy scope 或者其它隐含 spy scope 的选项（如上面例子中的选项 spy using overlays），把该环境做成一个 spy 域，在这个域中可以使用命令 \spy 来做局部放大图。命令 \spy 只能用在 spy 域中。注意，创建被放大图形的命令要放在 spy 域中，并且可以放在 \spy 的后面。

在 spy 域中，首先把通常的路径（由 \draw，\node 等创建的路径）创建出来并保存，在 spy 域结束时才会执行命令 \spy. 命令 \spy 创建两个 node：一个用于标示被放大的区域，此 node 称为 spy-on node；一个用于展示放大效果，此 node 称为 spy-in node. 假设原来图形上的点 p 是 spy-on node 的中心点，记 spy-in node 的中心是点 q. 在 spy-in node 的文字盒子里画出原图并对图形做“画布变换”，且使得变换后的点 p 与原来是点 q 重合，同时用 spy-in node 的边界路径剪切变换后的图形，这样就得到局部放大图。

spy-in node 的默认名称是 tikzspyinnode，spy-on node 的默认名称是 tikzspyonnode. 如果有多个 spy-in node 和 spy-on node，那么这两个名称所指的 node 是之前最近出现者。

通常，你需要指定 spy-in node 的尺寸以及放大的倍数，spy 程序库会根据这两个参数自动计算 spy-on node 的尺寸。例如，如果指定 spy-in node 的尺寸是 size=2cm，放大倍数是 magnification=2，那么 spy-on node 的尺寸就是 1cm.



```
\begin{tikzpicture}
  \begin{scope}[spy using outlines={ isosceles triangle,
        isosceles triangle apex angle=150,
        magnification=4, width=2cm, connect spies}]
    \draw [decoration=Koch curve type 1]
      decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
    \spy [green] on (1.6,0.3) in node (a) [red,circle,below=3mm]
      at (current bounding box.south);
    \end{scope}
\end{tikzpicture}
```

上面图形中，spy-on node 的外形是顶角在右侧的等腰三角形，它的顶角角度是 150°，中心点 p 在原来图形的 $(1.6, 0.3)$ 处，它的底边长度是 5mm（是选项值 width=2cm 的四分之一）；spy-in node 的外形是圆，圆心 q 在分形图的下边界之下 1.3cm 处（根据选项 width=2cm, below=3mm）。对原图做画布变换，并让变换后的点 p 与原图中的圆心 q 重合，同时用 spy-in node 的边界路径做剪切，得到局部放大图。

66.2 spy scopes

/tikz/spy scope=⟨*options*⟩ (default empty)

这是个样式，它用作 {tikzpicture} 环境或者 {scope} 环境的选项。这个样式为当前的 (只是当前的) {scope} 环境装备一些选项、命令，把该环境做成一个 spy 域，在这个域中可以使用命令 \spy 来做局部放大图。命令 \spy 只能用在 spy 域中。

本选项的定义是：

```
\newbox\tikz@lib@spybox

\let\tikz@lib@spy@collection=\pgfutil@empty%

\tikzset{spy scope/.style={
    size/.style={minimum size=##1},
    height/.style={minimum height=##1},
    width/.style={minimum width=##1},
    execute at begin scope={%
      \let\tikz@lib@spy@save=\tikz@lib@spy@collection%
      \setbox\tikz@lib@spybox=\hbox\bgroup\bgroup%
      \let\spy=\tikz@lib@spy@parse},
    execute at end scope={%
      \egroup\egroup%
      {%
        \copy\tikz@lib@spybox%
        \tikz@lib@spy@collection%
      }%
      \global\let\tikz@lib@spy@collection=\tikz@lib@spy@save%
    },%
    tikz@lib@spy@style/.style={#1},
    tikz@lib@reset@gs
  },
}%

\tikzset{
  tikz@lib@reset@gs/.style={black,thin,solid,opaque,line cap=butt,line join=miter}
```

```
}%
```

可见本样式的作用是定义一些选项，执行一些选项。
本样式的参数 ⟨*options*⟩ 将成为样式 `tikz@lib@spy@style` 的参数。

在执行 `\tikzset{spy scope={⟨options⟩}}` 的过程中：

- 以下样式被定义：

`/tikz/size`=⟨*dimension*⟩ (no default)

这是 `/pgf/minimum size` →P. 811 的简写，针对 spy-in node.
这个样式的定义是：

```
\tikzset{
  size/.style={minimum size=#1},
}
```

参考 `/tikz/minimum size` →P. 811.

`/tikz/height`=⟨*dimension*⟩ (no default)

这是 `/pgf/minimum height` →P. 811 的简写，针对 spy-in node.
这个样式的定义是：

```
\tikzset{
  height/.style={minimum height=#1},
}
```

参考 `/tikz/minimum height` →P. 811.

`/tikz/width`=⟨*dimension*⟩ (no default)

这是 `/pgf/minimum width` →P. 811 的简写，针对 spy-in node.
这个样式的定义是：

```
\tikzset{
  width/.style={minimum width=#1},
}
```

参考 `/tikz/minimum width` →P. 811.

`/tikz/tikz@lib@spy@style`={⟨*options*⟩} (no default)

这个样式的定义是：

```
\tikzset{
  tikz@lib@spy@style/.style={⟨options⟩},
}
```

- 以下 (保存命令的) 键被执行：
  - 参考 `/tikz/execute at begin scope` →P. 672

```
\tikzset{
  execute at begin scope={%
    \let\tikz@lib@spy@save=\tikz@lib@spy@collection%
    \setbox\tikz@lib@spybox=\hbox\bgroup\bgroup%
    \let\spy=\tikz@lib@spy@parse},
}
```

这个键在 `scope` 环境的开头被执行，参考 `\begin{scope}` 或者 `\scope`.
这个键定义盒子 `\tikz@lib@spybox`, 在盒子内定义命令 `\spy`.

`\tikz@lib@spybox`

这个盒子保存 spy 域的内容，即当前 scope 环境的内容。环境内的图形命令都保存在这个

盒子里。

– 参考 /tikz/execute at end scope→P. 672

```
\tikzset{
  execute at end scope={%
    \egroup\egroup%
    {%
      \copy\tikz@lib@spybox%
      \tikz@lib@spy@collection%
    }%
    \global\let\tikz@lib@spy@collection=\tikz@lib@spy@save%
  },%
}
```

这个键在 scope 环境的结尾被执行，参考 \end{scope} 或者 \endscope.

\copy 的作用是插入盒子的内容，但不清空盒子。

- 以下样式被执行：

/tikz/tikz@lib@reset@gs (no value)

这个样式的定义是：

```
\tikzset{
  tikz@lib@reset@gs/.style={black,thin,solid,opaque,line cap=butt,line
  ↪  join=miter}
}%
```

66.3 其他选项

/tikz/lens=⟨*transform options*⟩ (no default)

⟨*transform options*⟩ 中应该是变换选项，例如 /tikz/scale，rotate，这些变换选项会被作为画布变换，其效果显现在 spy-on node 内的图形中。

本选项的定义是：

```
\tikzset{
  lens/.store in=\tikz@lib@spy@lens,
  lens=,
}
```

本选项将其参数保存在宏 \tikz@lib@spy@lens 中，这个宏的初始值是空的。

/tikz/magnification=⟨*number*⟩ (no default)

这个选项用来设置放大倍数，等效于 lens={scale=⟨*number*⟩}.

本选项的定义是：

```
\tikzset{
  magnification/.style={lens={scale=#1}},
}
```

/tikz/spy connection path=⟨*code*⟩ (no default, initially empty)

这里 ⟨*code*⟩ 是绘图命令或者其它代码。在 spy-in node 和 spy-on node 被创建后再执行 ⟨*code*⟩, 所以在 ⟨*code*⟩ 中可以使用 spy-in node 和 spy-on node 的名称。

本选项的定义是：

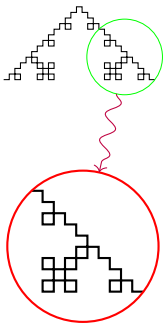
```
\tikzset{
  spy connection path/.store in=\tikz@lib@spy@path,
```

```
  spy connection path=
}
```

本选项将其参数保存在宏 \tikz@lib@spy@path 中，这个宏的初始值是空的。

例如

```
spy connection path={\draw[thin] (tikzspyonnode) -- (tikzspyinnode);}
```



```
\begin{tikzpicture}
    [spy using outlines={circle,magnification=2, width=2cm,},
      spy connection path={
        \draw[->,thin,purple,decorate,decoration=snake]
          (tikzspyonnode) -- (tikzspyinnode);
    }]
  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [green] on (1.6,0.3) in node (a) [red,below=1cm]
      at (current bounding box.south);
\end{tikzpicture}
```

/tikz/every spy on node (style, no value)

这个键用作样式，先用手柄 /.style 定义它。这个样式用于每个 spy-on node.

spy-on node 的默认名称是 tikzspyonnode，如果需要给 spy-on node 另行命名，可以在 every spy on node 中使用 name 选项。

/tikz/every spy in node (style, no value)

这个键用作样式，先用手柄 /.style 定义它。这个样式用于每个 spy-in node.

spy-in node 的默认名称是 tikzspyinnode，如果需要给 spy-in node 另行命名，可以在 every spy in node 中使用 name 选项，也可以在 \spy 命令的 ⟨*node options*⟩ 中使用圆括号命名 (node name).

66.4 spy 命令

在 spy 域中，命令 \spy 等于命令 \tikz@lib@spy@parse.

\spy[⟨*options*⟩] on ⟨*coordinate*⟩ in node ⟨*node options*⟩;

\spy[⟨*options*⟩] on ⟨*coordinate*⟩ in node ⟨*node options*⟩ (⟨*node name*⟩) at ⟨*at coordinate*⟩;

命令 \spy 只能用在 spy 域中。注意 \spy 不是 Ti*k*Z 的路径命令 \path 的处理过程之内的子命令。

在 \spy 命令的句法中，on 后面的 ⟨*coordinate*⟩ 就是 spy-on node 的中心。in node 后面的 ⟨*node options*⟩ 是针对 spy-in node 的选项设置，spy-in node 就是个通常的 node，它的内容是放大图，所有能用于设置 node 的选项都可以用在 ⟨*node options*⟩ 中。注意 ⟨*node options*⟩ 后面是分号，没有花括号。

⟨*options*⟩, ⟨*coordinate*⟩, ⟨*node options*⟩ 会被保存起来，直到 spy 域结束时才被调出，因此在 spy 域结束之前的各种 node，坐标都可以用在这三个参数中，即使在命令 \spy 之后才被定义的 node 也可以被引用。在 spy 域结束时，创建 spy-in node 和 the spy-on node.

\tikz@lib@spy@parse[⟨*options*⟩] on ⟨*TikZ point*⟩ in node ⟨*node options*⟩;

本命令导致下面的命令：

\tikz@lib@spy@parse@opta[⟨*options*⟩] on ⟨*TikZ point*⟩ in node ⟨*node options*⟩;

本命令的定义是：

```
\def\tikz@lib@spy@parse@opta[#1]on#2in node#3;{%
```

```
  \pgfutil@g@addto@macro\tikz@lib@spy@collection{\tikz@lib@spy@do{#1}{#2}{#3}}
  ↪ %
}%
```

可见本命令将 \spy 命令的实际内容，即 \tikz@lib@spy@do{⟨*options*⟩}{⟨*TikZ point*⟩}{⟨*node options*⟩}}，“全局地”添加到宏 \tikz@lib@spy@collection 中保存起来。

\tikz@lib@spy@do{⟨*options*⟩}{⟨*TikZ point*⟩}{⟨*node options*⟩}

本命令的处理是：

1. 开启一个 scope 环境

```
\scope[tikz@lib@spy@style,⟨options⟩]
```

其中的样式 tikz@lib@spy@style 的内容就是样式 /tikz/spy scope → P. 1168 的参数。这个样式以及 ⟨*options*⟩ 中的选项对 spy-in node 和 spy-on node 都有效。

2. 创建 spy-on node:

```
\node [alias=tikzspyonnode,inner sep=0pt,outer sep=0pt,every spy on node/.try,
  /utils/exec={
    {%
      \let\tikz@transform=\relax
      \pgftransformreset%
      \expandafter\tikzset\expandafter{\tikz@lib@spy@lens}
      \pgftransforminvert%
      \pgfgettransformentries\a\b\c\d\e\f%
      \global\let\pgf@lib@svg@a=\a%
      \global\let\pgf@lib@svg@b=\b%
      \global\let\pgf@lib@svg@c=\c%
      \global\let\pgf@lib@svg@d=\d%
    }%
    \tikz@addtransform{%
      \tikz@scan@one@point\pgftransformshift⟨TikZ point⟩%
      \pgftransformcm{\pgf@lib@svg@a}{\pgf@lib@svg@b}{\pgf@lib@svg@c}{
      ↪  \pgf@lib@svg@d}{\pgfpointorigin}%
    }
  }]{};
```

可见名称 tikzspyonnode 是 spy-on node 的别名。注意 spy-on node 采用的变换矩阵，其平移部分由 ⟨*TikZ point*⟩ 决定，但其旋转、放缩部分，是选项 lens，magnification 所指定的旋转、放缩变换的逆变换。

3. 决定 spy-in node 的 anchor 位置：

```
\expandafter\pgfutil@switch\expandafter\pgfutil@ifstrequal\expandafter{
↪  \tikz@anchor}{%
  {north}      {\def\tikz@spy@anchor{south}}%
  {north east}{\def\tikz@spy@anchor{south west}}%
  {east}       {\def\tikz@spy@anchor{west}}%
  {south east}{\def\tikz@spy@anchor{north west}}%
  {south}      {\def\tikz@spy@anchor{north}}%
  {south west}{\def\tikz@spy@anchor{north east}}%
  {west}       {\def\tikz@spy@anchor{east}}%
  {north west}{\def\tikz@spy@anchor{south east}}%
}{}{\def\tikz@spy@anchor{center}}%
```

参考 \pgfutil@switch → P. 33, \pgfutil@ifstrequal → P. 27.

4. 创建 spy-in node:

```
\node [alias=tikzspyinnode,inner sep=0pt,outer sep=0pt,at={⟨TikZ point⟩},every
↪  spy in node/.try,
  path picture={\node[anchor=\tikz@spy@anchor,tikz@lib@reset@gs]{\nullfont%
    \pgfpicture\relax\pgfsetbaseline{default}\pgfsettrimleft{default}
    ↪  \pgfsettrimright{default}%
    \pgftransformreset%
    \let\tikz@transform=\relax%
    \expandafter\tikzset\expandafter{\tikz@lib@spy@lens}%
    \pgflowlevelsynccm%
    \tikz@scan@one@point\tikz@lib@spy@shift⟨TikZ point⟩%
    \pgflowlevelsynccm%
    \copy\tikz@lib@spybox%
    \endpgfpicture};}]⟨node options⟩{};
```

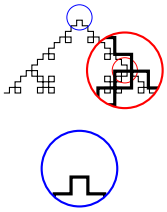
可见名称 `tikzspyinnode` 是 spy-in node 的别名。spy-in node 所包含的放大图形是由选项 `/tikz/path picture` 引入的一个 node.

注意放大图采用的变换是画布变换，其平移由 ⟨*TikZ point*⟩ 决定，旋转、放缩部分由选项 `lens`, `magnification` 指定。

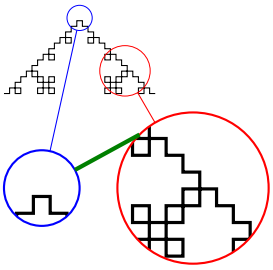
创建被放大图形的命令要放在 spy 域中，这样就会被包含到盒子 `\tikz@lib@spybox` 中，从而被画布变换，创建放大图形。

也可以看出，在默认下，spy-in node 和 spy-on node 的锚定点都是 ⟨*TikZ point*⟩. 如果希望 spy-in node 的锚定点是其他点，可以在 ⟨*node options*⟩ 中用选项 `at=`⟨*another TikZ point*⟩ 指定。

5. 执行 `\tikz@lib@spy@path`, 选项 `spy connection path` 指定的命令保存在这个宏里面。当 spy-in node 和 spy-on node 被创建后，再执行这个宏来连接它们。
6. 执行 `\endscope` 结束 scope 环境。

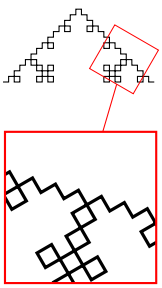


```
\begin{tikzpicture}[spy using outlines={circle,
      magnification=3, size=1cm}]
  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [red] on (1.6,0.3) in node;
  \spy [blue] on (1,1) in node at (1,-1);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \begin{scope}[spy using outlines={circle,
       magnification=3, size=2cm, connect spies}]
    \draw [decoration=Koch curve type 1]
      decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
    \spy [red] on (1.6,0.3) in node (a) [left] at (3.5,-1.25);
    \spy [blue, size=1cm]
      on (1,1) in node (b) [right] at (0,-1.25);
  \end{scope}
  \draw [ultra thick, green!50!black] (b) -- (a.north west);
\end{tikzpicture}
```

注意 spy-on node 采用的变换矩阵。例如 `lens={rotate=30}`，观察下图，spy-in node 和 spy-on node 之间有 30° 的旋转差别：



```
\begin{tikzpicture}[spy using outlines={lens={scale=3,rotate=30},
      size=2cm, connect spies}]
  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [red] on (1.6,0.3) in node [below=5mm]
    at (current bounding box.south);
\end{tikzpicture}
```

66.5 总结

spy 域就是一个 scope 环境，与普通的 scope 环境不同，环境内容被保存到盒子 `\tikz@lib@spybox` 中。在 `\endscope` 那里，TikZ 在一个花括号组中释放盒子 `\tikz@lib@spybox`, 使得其中的绘图命令被执行，而其中的 `\spy` 命令则将 `\tikz@lib@spy@do{{`⟨*options*⟩`}{`⟨*TikZ point*⟩`}{`⟨*node options*⟩`}}` “全局地”添加到宏 `\tikz@lib@spy@collection` 中保存起来。然后展开宏 `\tikz@lib@spy@collection`, 执行它保存的命令。

如果只是使用选项 `spy scope`, 并且不再向键 `execute at begin scope`, `execute at end scope` 中添加其他命令 (例如使用手柄 `/.add code`)，那么 spy 域就是如下的样子：

```
\pgfscope%
\begingroup%
  \let\tikz@atbegin@scope=\pgfutil@empty% 清空
  \let\tikz@atend@scope=\pgfutil@empty% 清空
  \let\tikz@options=\pgfutil@empty%
  \tikz@clear@rdf@options%
  \let\tikz@mode=\pgfutil@empty%
  \let\tikz@id@name=\pgfutil@empty%
  \tikz@transparency@groupfalse%
  \tikzset{every scope/.try}%
  \tikzset{#1}%
  \tikz@options%
  \tikz@do@rdf@pre@options%
  \iftikz@transparency@group%
    \expandafter\pgftransparencygroup\expandafter[\tikz@transparency@group@options]
    ↪  \tikz@blend@group%
  \fi%
  \tikz@is@nodefalse%
  \tikz@call@id@hook%
  \pgfidscope%
  \tikz@do@rdf@post@options%
  \begingroup%
    \let\tikz@id@name\pgfutil@empty%
%     \tikz@atbegin@scope%
    \let\tikz@lib@spy@save=\tikz@lib@spy@collection%
    \setbox\tikz@lib@spybox=\hbox\bgroup% 设置盒子 \tikz@lib@spybox
      \bgroup%
        \let\spy=\tikz@lib@spy@parse
        \pgfclearid%
        ⟨ 被 \expandafter 展开的 \tikz@lib@scope@check⟩%
        ⟨scope 环境的内容⟩
%     \tikz@atend@scope%
      \egroup
    \egroup%
    {%
      \copy\tikz@lib@spybox%
      \tikz@lib@spy@collection%
    }%
    \global\let\tikz@lib@spy@collection=\tikz@lib@spy@save%
  \endgroup%
  \endpgfidscope%
```

```
  \iftikz@transparency@group\endpgftransparencygroup\fi%
\endgroup%
\endpgfscope%
\tikz@lib@scope@check%
```

从以上代码可见，在套嵌使用 spy 域时：

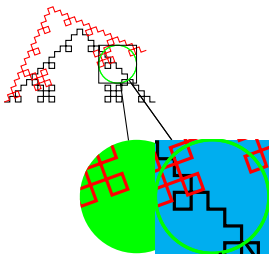
1. 应当给各个套嵌着的绘图环境分别加上选项 spy scope (或包含此选项的键)；
2. 各层次的 spy 域利用同一个盒子 \tikz@lib@spybox, 这个盒子的定义、盒子的内容都被限制在组中，盒子的内容也被放在组中执行；
3. 套嵌的 spy 域导致套嵌的盒子定义；
4. 内层 spy 域开始时，会把“外层”spy 域中的各个 \spy 命令保存的内容会被转存到 \tikz@lib@spy@save 中：

   ```
   \let\tikz@lib@spy@save=\tikz@lib@spy@collection
   ```

   当内层的 spy 域结束时，就会再次全局地保存外层的 \spy 命令保存的内容：

   ```
   \global\let\tikz@lib@spy@collection=\tikz@lib@spy@save
   ```

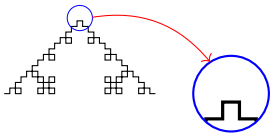
5. 命令 \copy 释放盒子 \tikz@lib@spybox 的内容，但不删除这个盒子保存的内容；
6. 各层次的 spy 域中的普通绘图命令，如 \draw, 会按它们在代码中出现的次序被执行；当前 spy 域中的普通绘图命令都被执行后，才会执行当前 spy 域中的 \spy 命令保存的代码；所以创建被放大图形的普通绘图命令应当与针对被放大图形的 \spy 命令处于同一层次的 spy 域内，可以放在 \spy 命令的后面；而 \spy 命令可以引用当前 spy 域内的任何坐标位置；
7. \spy 命令的实际作用是把 \tikz@lib@spy@do 全局地添加到宏 \tikz@lib@spy@collection 中，并且在 \endscope 那里展开这个宏，所以，如果外层 spy 域的某个 \spy 命令位于内层 spy 域的前面，那么内层 spy 域结束时会先执行这个外层的 \spy 命令保存的内容，然后再执行这个内层 spy 域中的各个 \spy 命令保存的内容；因此，如果这个外层的 \spy 命令引用了属于这个内层 spy 域的 spy-in node 或 spy-on node, 就可能导致错误；为了避免这种错误，外层 spy 域的 \spy 命令应当放在内层 spy 域之后。
8. 内层的 spy 域先完成“定义盒子、释放盒子、展开宏 \tikz@lib@spy@collection”的操作，所以，Ti*k*Z 先创建内层 spy 域中的 spy-in node 和 spy-on node, 再创建外层 spy 域中的 spy-in node 和 spy-on node, 这可能导致外层 spy node 遮挡内层 spy node 的结果；
9. 由于受到组的限制，内层 spy 域的非全局的、局部的内容不会影响到外层 spy 域；
10. 当前 spy 域的 \spy 命令放大当前 spy 域内的图形，所以，内层 spy 域的 \spy 命令放大内层的图形；外层 spy 域的 \spy 命令所放大的图形包含了内层 spy 域的所有图形；例如



```
\begin{tikzpicture}
  \begin{scope}[
    spy using outlines={
      rectangle,
      magnification=3, size=1.5cm, connect spies,
      every spy in node/.style={fill=cyan}
    }]
    \draw [decoration=Koch curve type 1]
```

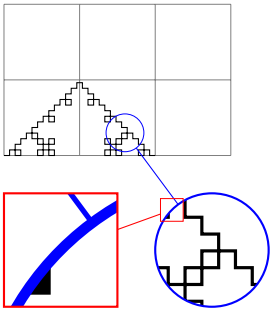
```
      decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
    \spy on (1.5,0.5) in node (zoom) [left] at (3.5,-1.25);% 外层的放大命令
    \begin{scope}[
      spy using outlines={
        circle,
        magnification=3, size=1.5cm, connect spies,
        every spy on node/.style={draw=green},
        every spy in node/.style={fill=green}
      }]
      \draw [draw=red, rotate=20, decoration=Koch curve type 1]
        decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
      \spy on (1.5,0.5) in node (zoom) [left] at (2.5,-1.25);% 内层的放大命令
    \end{scope}
  \end{scope}
\end{tikzpicture}
```

11. 当需要创建多个 spy-in node 和 spy-on node, 并且需要分别为它们命名时，可以在各个 \spy 命令之后，将命令 \pgfnodealias →P. 464 或 \pgfnoderename →P. 464 添加到宏 \tikz@lib@spy@collection 中，例如，



```
\begin{tikzpicture}
  \begin{scope}[
    spy using outlines={circle,
      magnification=3, size=1cm}]
    \draw [decoration=Koch curve type 1]
      decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
    \spy [blue] on (1,1) in node at (3,0);
    \makeatletter
    \pgfutil@g@addto@macro\tikz@lib@spy@collection{%
      \pgfnodealias{new name on 1}{tikzspyonnode}%
      \pgfnodealias{new name in 1}{tikzspyinnode}%
    }%
    \makeatother
  \end{scope}
  \draw [->, draw=red](new name on 1) to[bend left] (new name in 1);
\end{tikzpicture}
```

利用套嵌的 spy 域可以制作“放大图”的“放大图”，即利用外层的 \spy 命令来放大内层的 \spy 命令创建的放大图，此时外层的 \spy 命令应当放在内层的 spy 域之后。



```
\begin{tikzpicture}[spy using outlines={rectangle, red,
      magnification=5, size=1.5cm, connect spies}]
  \begin{scope}[spy using outlines={circle, blue,
        magnification=3, size=1.5cm, connect spies}]
    \draw [help lines] (0,0) grid (3,2);
    \draw [decoration=Koch curve type 1]
      decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
    \spy on (1.6,0.3) in node (zoom) [left] at (3.5,-1.25);% 内层的，圆形
  \end{scope}
  \spy on (zoom.north west) in node [right] at (0,-1.25);% 外层的，矩形
\end{tikzpicture}
```

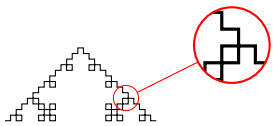
66.6 预定义的 spy 样式

下面预定义的 spy 样式隐含选项 spy scope，预定义的 spy 样式会把其选项设置传递给 spy scope.

```
\tikzset{
  spy using outlines/.style={
    spy scope={
      every spy on node/.style={very thin,draw},
      every spy in node/.style={thick,draw},
      #1
    }
  },
  spy using overlays/.style={
    spy scope={
      every spy on node/.style={fill,fill opacity=0.2,text opacity=1},
      every spy in node/.style={fill,fill opacity=0.2,text opacity=1},
      #1
    }
  },
  connect spies/.style={
    spy connection path={\draw[thin] (tikzspyonnode) -- (tikzspyinnode);}
  }
}%
```

/tikz/spy using outlines=⟨*options*⟩ (default empty)

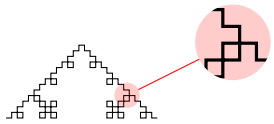
这个选项使得 spy-in node 用 thick 线宽画出，但不填充；使得 spy-on node 用 very thin 线宽画出，但不填充。



```
\begin{tikzpicture}[spy using outlines={circle, magnification=3,
       size=1cm, connect spies}]
  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [red] on (1.6,0.3) in node at (3,1);
\end{tikzpicture}
```

/tikz/spy using overlays=⟨*options*⟩ (default empty)

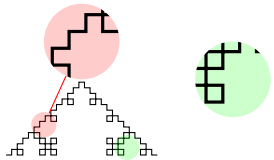
这个选项使得 spy-in node 和 spy-on node 被填充，填充色的不透明度是 20%，但不画出边界线条。



```
\begin{tikzpicture}[spy using overlays={circle, magnification=3,
       size=1cm, connect spies}]
  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [red] on (1.6,0.3) in node at (3,1);
\end{tikzpicture}
```

/tikz/connect spies (no value)

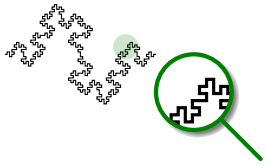
这个选项会在 spy-in node 和 spy-on node 之间画线。



```
\begin{tikzpicture}
     [spy using overlays={circle, magnification=3, size=1cm}]
  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [green] on (1.6,0.1) in node at (3,1);
  \spy [red,connect spies] on (0.5,0.4) in node at (1,1.5);
\end{tikzpicture}
```

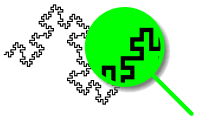
66.7 例子

下面的例子中把 spy-in node 的形状设为 `magnifying glass`，这个 node 形状需要调用 shapes.symbols 程序库。



```
\tikzset{spy using mag glass/.style={
    spy scope={
        every spy on node/.style={
              circle,
              fill, fill opacity=0.2, text opacity=1},
        every spy in node/.style={
              magnifying glass, circular drop shadow,
              fill=white, draw, ultra thick, cap=round},
        #1
    }}}
\begin{tikzpicture}[spy using mag glass={magnification=3, size=1cm}]
  \draw [decoration=Koch curve type 2] decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy [green!50!black] on (1.6,0.1) in node at (2.5,-0.5);
\end{tikzpicture}
```

上面例子中的选项 `circular drop shadow` 需要调用 shadows 程序库。也可以把放大镜直接放到被放大区域上。



```
\begin{tikzpicture}[spy scope={magnification=4, size=1cm},
    every spy in node/.style={
        magnifying glass, circular drop shadow,
        fill=white, draw, ultra thick, cap=round}]
  \draw [decoration=Koch curve type 2]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
  \spy on (1.6,0.1) in node[green];
\end{tikzpicture}
```

第六十七章　through 程序库

TikZ Library through

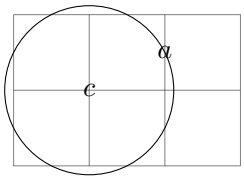
```
\usetikzlibrary{through} % LaTeX and plain TeX
\usetikzlibrary[through] % ConTeXt
```

这个程序库提供选项 /tikz/circle through=⟨*coordinate*⟩，这个选项用作 node 的选项。

/tikz/circle through=⟨*coordinate*⟩ (no default)

如果某个 node 带有这个选项，则：

- 此 node 的 inner sep 与 outer sep 都被设为 0pt.
- 此 node 的 shape 被设为 circle.
- 此 node 的边界路径（圆）的中心是它的锚定点（可以用 at 算子或 at={...} 选项指定），并且边界路径（圆）经过指定的坐标点 ⟨*coordinate*⟩.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \node (a) at (2,1.5) {$a$};
  \node [draw] at (1,1) [circle through={(a)}] {$c$};
\end{tikzpicture}
```

第六十八章　topaths 程序库

TikZ Library topaths

TikZ 会自动调用这个程序库。本程序库定义了数个选项，专门用于 to 操作或 edge 操作构建的路径。本程序库的定义文件是《tikzlibrarytopaths.code.tex》.

68.1　直线

/tikz/line to　(no value)

这个选项使得 to 操作或 edge 操作创建直线段，其定义是

```
\tikzset{line to/.style={to path={-- (\tikztotarget) \tikztonodes}}}
```

参考 /tikz/to path → P. 740.

```
\tikz {\draw (0,0) to[line to] (1,1);}
```

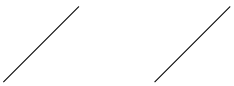
68.2　Move-To

/tikz/move to　(no value)

这个选项使得 to 或 edge 执行 move to 操作，其定义是

```
\tikzset{move to/.style={to path={(\tikztotarget) \tikztonodes}}}
```

```
\tikz \draw (0,0) to[line to] (1,1)
  to[move to] (2,0) to[line to] (3,1);
```



68.3　曲线

/tikz/curve to　(no value)

这个选项使得 to 或 edge 在两点之间构建一段控制曲线，其定义是

```
\tikzset{curve to/.style={to path=\tikz@to@curve@path}}
```

其中的 \tikz@to@curve@path 是个内部宏，它保存了 to 路径的代码。

假设 to 路径是：

```
(x0pt, y0pt) .. controls (x1pt, y1pt) and (x2pt, y2pt) .. (x3pt, y3pt)
```

记起点 $P_0(x_0\texttt{pt}, y_0\texttt{pt})$, 终点 $P_3(x_3\texttt{pt}, y_3\texttt{pt})$, 起点和终点是给定的。按如下的方式确定控制点 $P_1 = (x_1\texttt{pt}, y_1\texttt{pt})$ 和 $P_2 = (x_2\texttt{pt}, y_2\texttt{pt})$.
记 $Q = (|x_3 - x_0|\texttt{pt}, |y_3 - y_0|\texttt{pt}) = (x_Q\texttt{pt}, y_Q\texttt{pt})$, $n_Q = (x_{n_Q}\texttt{pt}, y_{n_Q}\texttt{pt})$, n_Q 与 Q 同向平行并且 $\|n_Q\| = 1\texttt{pt}$.
4 个参数 m_{in}, M_{in}, m_{out}, M_{out} 的值参考选项 /tikz/out min distance →P. 1183 等。

- 如果 $[65536 \cdot \max\{x_{n_Q}, y_{n_Q}\}] \neq 0$

$$r_{out} = \min\{\max\{m_{out}, r'_{out}\}, M_{out}\}, \quad r'_{out} = l_{out} \cdot 0.3915 \cdot \frac{16^2 \cdot \max\{x_Q\texttt{pt}, y_Q\texttt{pt}\}}{\left[\frac{[65536 \cdot \max\{x_{n_Q}, y_{n_Q}\}]}{255}\right]},$$

$$r_{in} = \min\{\max\{m_{in}, r'_{in}\}, M_{in}\}, \quad r'_{in} = l_{in} \cdot 0.3915 \cdot \frac{16^2 \cdot \max\{x_Q\texttt{pt}, y_Q\texttt{pt}\}}{\left[\frac{[65536 \cdot \max\{x_{n_Q}, y_{n_Q}\}]}{255}\right]}.$$

其中的方括号表示数值的整数部分，注意 $\max\{x_{n_Q}, y_{n_Q}\}$ 是不带长度单位的；使用因子 65536 是因为做了单位转换 $1\texttt{pt} = 65536\texttt{sp}$. 这个单位转换是通过寄存器实现的，例如当设置

```
\newdimen\dddd
\newcount\cccc
\dddd=1pt
\cccc=\dddd
```

后，\the\cccc 就是 65536. 实际上，$\left[\frac{[65536 \cdot \max\{x_{n_Q}, y_{n_Q}\}]}{255}\right]$ 是由

```
\c@pgf@counta=max{x_Qpt, y_Qpt}%
\divide\c@pgf@counta by 255\relax%
```

实现的，其中 \c@pgf@counta 的定义是 \newcount\c@pgf@counta, 它只能保存整数。

- 如果 $[65536 \cdot \max\{x_{n_Q}, y_{n_Q}\}] = 0$

$$r_{out} = \min\{\max\{m_{out}, r'_{out}\}, M_{out}\}, \quad r'_{out} = l_{out} \cdot 0.3915 \cdot x_Q\texttt{pt},$$

$$r_{in} = \min\{\max\{m_{in}, r'_{in}\}, M_{in}\}, \quad r'_{in} = l_{in} \cdot 0.3915 \cdot x_Q\texttt{pt}.$$

然后平移 P_0 得到 P_1, 平移 P_3 得到 P_2:

$$P_1 = (\texttt{[shift=(}\theta_{out}\texttt{:}r_{out}\texttt{)]}P_0), \quad P_2 = (\texttt{[shift=(}\theta_{in}\texttt{:}r_{in}\texttt{)]}P_3),$$

在起点 $P_0(x_0\texttt{pt}, y_0\texttt{pt})$ 和终点 $P_3(x_3\texttt{pt}, y_3\texttt{pt})$ 给定的情况下，上面表达式中可变的参数是 θ_{out}, l_{out}, m_{out}, M_{out} 以及 θ_{in}, l_{in}, m_{in}, M_{in}, 其中 θ_{out}, l_{out}, θ_{in}, l_{in} 对控制曲线的形态有很直接的影响。
下面的选项可以调节前述可变参数，从而调整控制曲线的形态。

/tikz/out=⟨*angle*⟩ (no default)

这个选项对应参数 θ_{out}, 它确定控制曲线在始点的方向角度，如果“出发地”是 node（假设 node 的名称是 a），那么始点就是点 (a.⟨*angle*⟩).

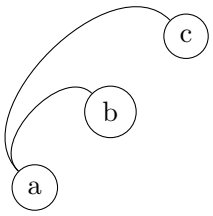
```
\begin{tikzpicture}[out=45,in=135]
  \draw (0,0) to (1,0)
        (0,0) to (2,0)
        (0,0) to (3,0);
\end{tikzpicture}
```

/tikz/in=⟨*angle*⟩ (no default)

这个选项对应参数 θ_{in}, 它确定控制曲线在终点的方向角度。如果“目标地”是 node（假设 node 的名称是 a），那么终点就是点 (a.⟨*angle*⟩).

/tikz/relative=⟨*true or false*⟩ (default true)

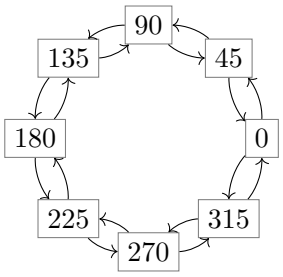
这个选项决定由选项 in，out 确定的角度是“绝对的”（absolute）还是“相对的”（relative）。假如一个坐标系的 x 轴方向在页面上是水平向右的，y 轴方向在页面上是竖直向上的，并且这个坐标系不接受变换，那么这个坐标系就是绝对的，在这个坐标系内确定的角度就是“绝对的”。以控制曲线的起点为原点，以起点到终点的有向直线为 x 轴，而 y 轴与 x 轴成右手系，这个坐标系就是相对的，在这个坐标系内确定的角度就是“相对的”。如果不使用本选项，那么就默认“绝对坐标系”。




```
\begin{tikzpicture}[out=90,in=90,relative]
  \node [circle,draw] (a) at (0,0) {a};
  \node [circle,draw] (b) at (1,1) {b};
  \node [circle,draw] (c) at (2,2) {c};
  \path (a) edge (b)
            edge (c);
\end{tikzpicture}
```

/tikz/bend left=⟨*angle*⟩ (default 30)

这个选项等效于 out=⟨*angle*⟩, in=180-⟨*angle*⟩, relative. 如果不明确给出 ⟨*angle*⟩，那么程序就向前查找最近使用过的选项 bend left 或 bend right 所确定的角度值并使用这个角度值，如果找不到，就默认 ⟨*angle*⟩ 的值是 30.



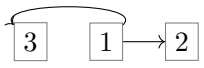

```
\begin{tikzpicture}
  \foreach \angle in {0,45,...,315}
  \node[rectangle,draw=black!50] (\angle) at (\angle:1.5) {\angle};
  \foreach \from/\to in {0/45,45/90,90/135,135/180,180/225,225/270,270/315,315/0}
  \path (\from) edge [->,bend right=22,looseness=0.8] (\to)
                edge [<-,bend left=22,looseness=0.8] (\to);
\end{tikzpicture}
```

/tikz/bend right=⟨*angle*⟩ (default last value)

类似 bend left, 本选项规定“向右偏”的角度，即一个左手系中的角度。本选项的 ⟨*angle*⟩ 默认为 30, 但这是“向右偏”的 30 度。

/tikz/bend angle=⟨*angle*⟩ (default last value)

这个选项声明一个角度 ⟨*angle*⟩，如果选项中还有 bend left 或 bend right，就把角度 ⟨*angle*⟩ 作为这两个选项的角度值。注意本选项只是声明一个角度，不会引入选项 curve to 或 relative.




```
\begin{tikzpicture}
  \node[rectangle,draw=black!50] (1) {1};
  \node[rectangle,draw=black!50] (2) at(1,0) {2};
  \node[rectangle,draw=black!50] (3) at(-1,0) {3};
  \path (1) edge [->,bend angle=30] (2)
            edge [->,bend angle=30,looseness=0.8] (3);
\end{tikzpicture}
```

上面例子中，选项 looseness=0.8 引入 curve to 操作。

/tikz/looseness=⟨*number*⟩ (no default, initially 1)

这个选项把参数 l_{out}, l_{in} 都设置为 ⟨*number*⟩, 它确定控制曲线的“松弛程度”。⟨*number*⟩ 是个实数，它的值越大，控制曲线就越“圆满”。当 looseness=1 且 in, out 的角度值相差 90° 时，得到的控制曲线是 $\frac{1}{4}$ 圆弧。

```
\tikz \draw (0,0) to [out=0,in=-90] (1,1);
\tikz \draw (0,0) to [out=0,in=-90,looseness=0.5] (1,1);
\tikz \draw (0,0) to [out=-45,in=-135,relative] (0,1);
```

上面例子中第一个命令绘制一段 $\frac{1}{4}$ 圆弧，这段圆弧的 4 个控制点应该是 $(0,0)$, $(\frac{4}{3}(\sqrt{2}-1),0)$, $(1,\frac{4}{3}(\sqrt{2}-1))$, $(1,1)$. 按转换关系 $1\text{cm}=28.45274\text{pt}$, 计算 r'_{out}:

$$0.3915\times\frac{16^2\times 28.45274\text{pt}}{\left[\frac{[65536\times\frac{\sqrt{2}}{2}]}{255}\right]}\approx 0.5537237569061\times 28.45274\text{pt}\approx 15.75495808707\text{pt}.$$

/tikz/out looseness=⟨*number*⟩ (default 1)

这个选项设置参数 l_{out} 为 ⟨*number*⟩.

/tikz/in looseness=⟨*number*⟩ (default 1)

这个选项设置参数 l_{in} 为 ⟨*number*⟩.

/tikz/min distance⟨*distance*⟩ (no default)

这个选项把参数 m_{in}, m_{out} 都设置为 ⟨*distance*⟩.

/tikz/max distance⟨*distance*⟩ (no default)

这个选项把参数 M_{in}, M_{out} 都设置为 ⟨*distance*⟩.

/tikz/out min distance⟨*distance*⟩ (initial 0pt)

这个选项把参数 m_{out} 设置为 ⟨*distance*⟩.

/tikz/out max distance⟨*distance*⟩ (initial 10000pt)

这个选项把参数 M_{out} 设置为 ⟨*distance*⟩.

/tikz/in min distance⟨*distance*⟩ (initial 0pt)

这个选项把参数 m_{in} 设置为 ⟨*distance*⟩.

/tikz/in max distance⟨*distance*⟩ (initial 10000pt)

这个选项把参数 M_{in} 设置为 ⟨*distance*⟩.

/tikz/distance⟨*distance*⟩ (no default)

本选项同时设置 min distance=⟨*distance*⟩ 和 max distance=⟨*distance*⟩.

/tikz/out distance⟨*distance*⟩ (no default)

本选项同时设置 out min distance=⟨*distance*⟩ 和 out max distance=⟨*distance*⟩.

/tikz/in distance⟨*distance*⟩ (no default)

本选项同时设置 in min distance=⟨*distance*⟩ 和 in max distance=⟨*distance*⟩.

/tikz/out control⟨*coordinate*⟩ (no default)

本选项直接指定第一个支撑点 P_1，这里 ⟨*coordinate*⟩ 可以使用相对坐标形式，例如 +(1,1)，它相对于起点计算。

/tikz/in control⟨*coordinate*⟩ (no default)

本选项直接指定第一个支撑点 P_2，这里 ⟨*coordinate*⟩ 可以使用相对坐标形式，例如 `+(1,1)`，它相对于终点计算。

/tikz/controls⟨*coordinate1*⟩ and ⟨*coordinate2*⟩ (no default)

本选项直接指定支撑点 P_1, P_2. 如果 ⟨*coordinate1*⟩ 是相对坐标形式，则它相对于起点计算。如果 ⟨*coordinate2*⟩ 是相对坐标形式，则它相对于终点计算。

```
\tikz \draw (0,0) to [controls=+(90:1) and +(90:1)] (2,0);
```

68.4 Loops

/tikz/loop (no value)

本选项类似 `curve to`，也构建一段控制曲线，不同的是：(1) 本选项确定的控制曲线的起点与终点重合，因此本选项构建的控制曲线像是一个“环”(loop)，而且在使用本选项时，代码中的“终点”只需要用一对圆括号代表；(2) 每当使用本选项时，都会使用固定选项值 `looseness=8`，`min distance=5mm`，也就是说，当使用本选项时这两个选项值不能改变（对于这两个固定选项值来说，当 `out`, `in` 的角度相差 30° 时，得到的控制曲线较美观。）


```
\begin{tikzpicture}
  \node [circle,draw] {a} edge [in=30,out=60,loop] ();
\end{tikzpicture}
```

/tikz/loop above (style, no value)

这是个样式（style），这个样式创建的环（控制曲线）总是位于起点（终点）的上方，并且，如果给这个环添加 node 标签，那么该标签会位于环的上方。


```
\begin{tikzpicture}
  \node [circle,draw] {a} edge [loop above] node {x} ();
\end{tikzpicture}
```

/style/loop below (style, no value)

这是个样式（style），这个样式创建的环（控制曲线）总是位于起点（终点）的下方，并且，如果给这个环添加 node 标签，那么该标签会位于环的下方。

/style/loop left (style, no value)

这是个样式（style），类似样式 `loop above`.

/style/loop right (style, no value)

这是个样式（style），类似样式 `loop above`.

/tikz/every loop (style, initially `->,shorten >=1pt`)

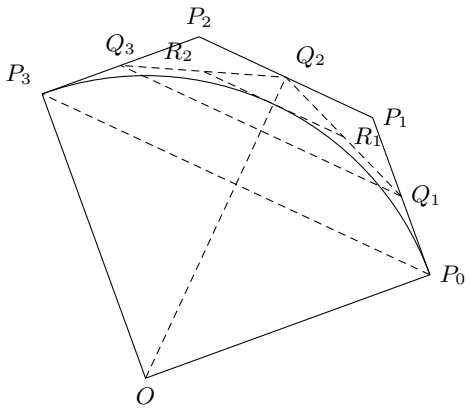
这是个样式（style），针对的是每个 `loop`，所设置的选项会用在每个 `loop` 的开头。

```
\begin{tikzpicture}[every loop/.style={}]
  \draw (0,0) to [loop above] () to [loop right] ()
              to [loop below] () to [loop left] ();
\end{tikzpicture}
```

68.5 关于 curve to 选项的系数

下面看看用 3 次 Bézier 曲线来近似椭圆弧的情况。

设 $0 < \varphi \leqslant \frac{\pi}{2}$, 考虑单位圆上从点 $P_0 = (\cos\alpha, \sin\alpha)^T$ 到点 $P_3 = (\cos(\alpha+\varphi), \sin(\alpha+\varphi))^T$ 的一段圆弧 S. 现在用一段 3 次 Bézier 曲线来近似圆弧 S. 控制曲线的控制点是 P_0, P_1, P_2, P_3, 在最好的近似情况下，这 4 个点的位置应当如下图所示：



其中 Q_1 是 P_0P_1 的中点，R_1 是 Q_1Q_2 的中点……可以计算出 P_1, P_2 的坐标是：

$$P_1 = \frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot\begin{pmatrix}-\sin\alpha\\ \cos\alpha\end{pmatrix} + \begin{pmatrix}\cos\alpha\\ \sin\alpha\end{pmatrix},$$
$$P_2 = \frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot\begin{pmatrix}\sin(\alpha+\varphi)\\ -\cos(\alpha+\varphi)\end{pmatrix} + \begin{pmatrix}\cos(\alpha+\varphi)\\ \sin(\alpha+\varphi)\end{pmatrix}.$$

当 $\varphi = \frac{\pi}{2}$ 时，记系数 $t = \frac{4}{3}\cdot\tan(\frac{\varphi}{4}) = \frac{4}{3}(\sqrt{2}-1) \approx 0.5522847498308$, 上述公式是：

$$P_0 = (\cos\alpha, \sin\alpha)^T,\ P_3 = (\cos(\alpha+\frac{\pi}{2}), \sin(\alpha+\frac{\pi}{2}))^T,$$
$$P_1 = t\cdot P_3 + P_0, \qquad P_2 = t\cdot P_0 + P_3.$$

当 $\varphi = -\frac{\pi}{2}$ 时，记系数 $-t = -\frac{4}{3}\cdot\tan(\frac{\varphi}{4}) = -\frac{4}{3}(\sqrt{2}-1) \approx -0.5522847498308$, 上述公式形式不变。

上面公式是针对单位圆的，容易变化到下面两个情况：

(i) 对于圆心在原点，半径为 r 的圆，上述公式就是：

$$P_1 = \frac{4}{3}\tan(\frac{\varphi}{4})\cdot r\cdot\begin{pmatrix}-\sin\alpha\\ \cos\alpha\end{pmatrix} + r\cdot\begin{pmatrix}\cos\alpha\\ \sin\alpha\end{pmatrix},$$
$$P_2 = \frac{4}{3}\tan(\frac{\varphi}{4})\cdot r\cdot\begin{pmatrix}\sin(\alpha+\varphi)\\ -\cos(\alpha+\varphi)\end{pmatrix} + r\cdot\begin{pmatrix}\cos(\alpha+\varphi)\\ \sin(\alpha+\varphi)\end{pmatrix}.$$

记 $(r\cos\theta, r\sin\theta)^T = (\theta : r)$, 上述公式就是：

$$P_0 = (\alpha : r),$$
$$P_1 = \frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot(\alpha+\frac{\pi}{2} : r) + (\alpha : r),$$
$$P_2 = \frac{4}{3}\cdot\tan(\frac{\varphi}{4})\cdot(\alpha+\varphi-\frac{\pi}{2} : r) + (\alpha+\varphi : r),$$
$$P_3 = (\alpha+\varphi : r).$$

(ii) 对于椭圆 $\begin{pmatrix}a\cos\theta\\ b\sin\theta\end{pmatrix}$, 只需要对上面公式中的点做个仿射变换：

$$(r\cos\theta, r\sin\theta)^T \to (a\cos\theta, b\sin\theta)^T = (\theta : a\ \&\ b),$$

上述公式变成：

$$
\begin{aligned}
P_0 &= (\alpha : a \,\&\, b),\\
P_1 &= \frac{4}{3} \cdot \tan(\frac{\varphi}{4}) \cdot (\alpha + \frac{\pi}{2} : a \,\&\, b) + (\alpha : a \,\&\, b),\\
P_2 &= \frac{4}{3} \cdot \tan(\frac{\varphi}{4}) \cdot (\alpha + \varphi - \frac{\pi}{2} : a \,\&\, b) + (\alpha + \varphi : a \,\&\, b),\\
P_3 &= (\alpha + \varphi : a \,\&\, b).
\end{aligned}
$$

注意这个变换保持圆 (椭圆) 的横轴与纵轴相正交。

第六十九章　trees 程序库

```
\usetikzlibrary{trees} % LaTeX and plain TeX
\usetikzlibrary[trees] % ConTeXt
```

这个程序库为 TikZ 的绘制树的 child 操作提供一些样式选项。

69.1　生长函数

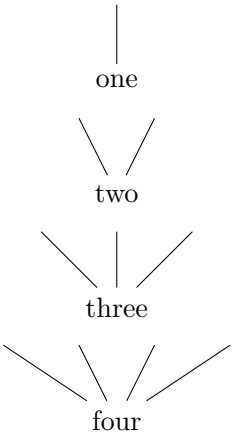
trees 程序库提供两个新的生长函数，使用下面的选项来调用：

/tikz/grow via three points=one child at (⟨x⟩) and two children at (⟨y⟩) and (⟨z⟩) (no default)

这个选项提供的生长函数是这样的：假设父节点是 p，把坐标系的原点平移到 p 的某个锚位置上，然后在这个坐标系中排布 p 的子节点；如果 p 只有一个子节点 c，那么 c 的某个锚位置放在坐标点 $\langle x\rangle$ 上；如果 p 有两个子节点 c1 和 c2，那么这两个子节点的某个锚位置分别放在坐标点 $\langle y\rangle$ 和 $\langle z\rangle$ 上；如果 p 的子节点的总个数 $n>2$，那么第 $i(1\leqslant i)$ 个子节点的某个锚位置位于坐标点

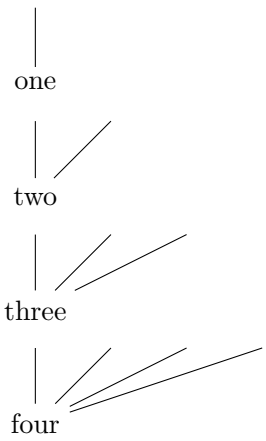
$$c_i = x + \frac{n-1}{2}(y-x) + (i-1)(z-y) = \frac{3-n}{2}x + \frac{n-2i+1}{2}y + (i-1)z,$$

这个点是 $\langle x\rangle$, $\langle y\rangle$, $\langle z\rangle$ 的线性组合。



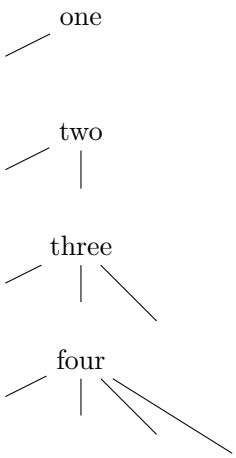

```
\begin{tikzpicture}[grow via three points={one child at
    (0,1)and two children at (-.5,1) and (.5,1)}]
  \node at (0,0) {one} child;
  \node at (0,-1.5) {two} child child;
  \node at (0,-3) {three} child child child;
  \node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

上面例子中 $x=(0,1)$，$y=(-0.5,1)$，$z=(0.5,1)$，$c_i=\left(\frac{4i-n-3}{4},1\right)$.




```
\begin{tikzpicture}[grow via three points={one child at
    (0,1) and two children at (0,1) and (1,1)}]
  \node at (0,0) {one} child;
  \node at (0,-1.5) {two} child child;
  \node at (0,-3) {three} child child child;
  \node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

上面例子中 $x=(0,1)$，$y=(0,1)$，$z=(1,1)$，$c_i=(i-1,1)$.




```
\begin{tikzpicture}[grow via three points={one child at
    (-1,-.5) and two children at (-1,-.5) and (0,-.75)}]
  \node at (0,0) {one} child;
  \node at (0,-1.5) {two} child child;
  \node at (0,-3) {three} child child child;
  \node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

上面例子中 $x=(-1,-0.5)$，$y=(-1,-0.5)$，$z=(0,-0.75)$，$c_i=\left(i-2,\frac{-i-1}{4}\right)$.

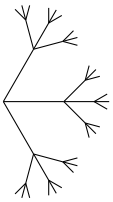
/tikz/grow cyclic (no value)

这个选项提供的生长函数是这样的：子节点的某个锚位置被放置在以父节点（的某个锚位置）为圆心的圆上；此时圆的半径是宏 \tikzleveldistance 的值，也就是说同一层的子节点位于同一个圆上；一层之内的相邻两个子节点的角度之差是宏 \tikzsiblingangle 的值，这个值由下面的选项指定：

/tikz/sibling angle=⟨*angle*⟩ (no default)

这个选项确定一层之内的相邻两个子节点的角度之差。

注意这个生长函数不仅会平移坐标系，还会旋转坐标系以使得树的各个分支具有适当的“延伸方向”。

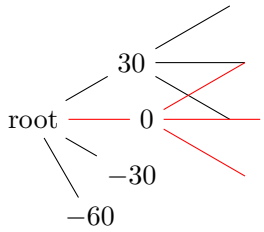


```
\begin{tikzpicture} [grow cyclic,
    level 1/.style={level distance=8mm,sibling angle=60},
    level 2/.style={level distance=4mm,sibling angle=45},
    level 3/.style={level distance=2mm,sibling angle=30}]
  \coordinate [rotate=-90]
    child foreach \x in {1,2,3}
      {child foreach \x in {1,2,3}
        {child foreach \x in {1,2,3}}};
\end{tikzpicture}
```

/tikz/clockwise from=⟨*angle*⟩ (no default)

这个选项提供的生长函数是这样的：子节点的某个锚位置按照顺时针的方向，放置在以父节点（的某个锚位置）为圆心的圆上；此时圆的半径是宏 \tikzleveldistance 的值，也就是说同一层的子节点位于同一个圆上；从父节点到第一个子节点的方向是角度 ⟨*angle*⟩；一层之内的相邻两个子节点的角度之差也是由选项 /tikz/sibling angle=⟨*angle*⟩ 指定。

注意这个生长函数仅平移坐标系，而不会旋转坐标系。




```
\begin{tikzpicture}
  \node {root} [clockwise from=30,sibling angle=30]
    child {node {$30$} child child child}
    child {node {$0$} [red] child child child}
    child {node {$-30$}}
    child {node {$-60$}};
\end{tikzpicture}
```

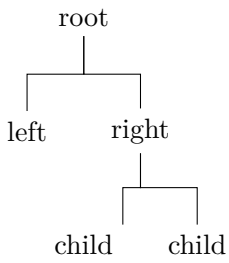
/tikz/counterclockwise from=⟨*angle*⟩ (no default)

类似 `clockwise from` 只是方向相反。

69.2 从父节点到子节点的边

/tikz/edge from parent fork down (style, no value)

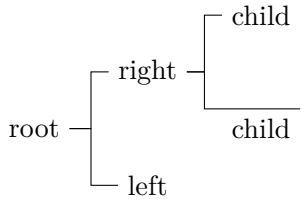
这个选项画出的边如下所示：




```
\begin{tikzpicture}
  \node {root} [edge from parent fork down]
    child {node {left}}
    child {node {right}
      child[child anchor=north east] {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

/tikz/edge from parent fork right (style, no value)

这个选项画出的边如下所示：




```
\begin{tikzpicture}
  \node {root} [edge from parent fork right,grow=right]
    child {node {left}}
    child {node {right}
      child[child anchor=north east] {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

/tikz/edge from parent fork left (style, no value)

/tikz/edge from parent fork up (style, no value)

第七十章　Views Library

TikZ Library views

```
\usetikzlibrary{views} % LaTeX and plain TeX
\usetikzlibrary[views] % ConTeXt
```

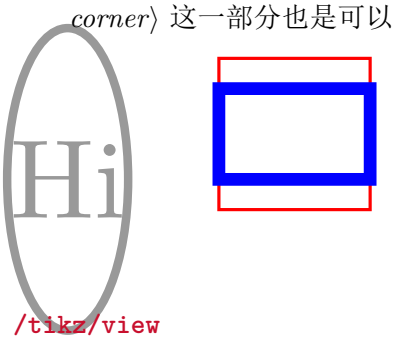
这个库用于创建 view, 常用于动画中。

一个 view 就是一个 window, 透过这个窗口可以看到图形（graphic）。为了创建一个 view, 需要创建两个矩形，一个矩形作为“固定窗口”，另一个矩形作为“可变矩形”（称之为 to-be-viewed rectangle）。固定窗口是不变的，tikz 会对可变矩形和其它图形做 画布变换，对其进行平移、放缩，使得可变矩形的中心与固定窗口的中心重合，并且：（1）固定窗口恰好把可变矩形容纳在内（这个“容纳”不考虑线宽），这是选项 meet，view 的情况；或者（2）可变矩形恰好把固定窗口容纳在内（这个“容纳”不考虑线宽），这是选项 slice 的情况。这实际上就是利用两个矩形来调整画布。

参考环境 pgfviewboxscope → P. 273.

/tikz/meet=⟨*to-be-viewed corner*⟩ rectangle ⟨*to-be-viewed corner*⟩
at ⟨*window corner*⟩ rectangle ⟨*window corner*⟩ (no default)

这个选项用作环境选项，它设置“固定窗口”和“可变矩形”。“可变矩形”由 ⟨*to-be-viewed corner*⟩ rectangle ⟨*to-be-viewed corner*⟩ 指定，“固定窗口”由 ⟨*window corner*⟩ rectangle ⟨*window corner*⟩ 指定；其中的两个单词 rectangle 都是可以省略的；而且 at ⟨*window corner*⟩ rectangle ⟨*window corner*⟩ 这一部分也是可以省略的，若省略这一部分，就默认“固定窗口”和“可变矩形”相同。



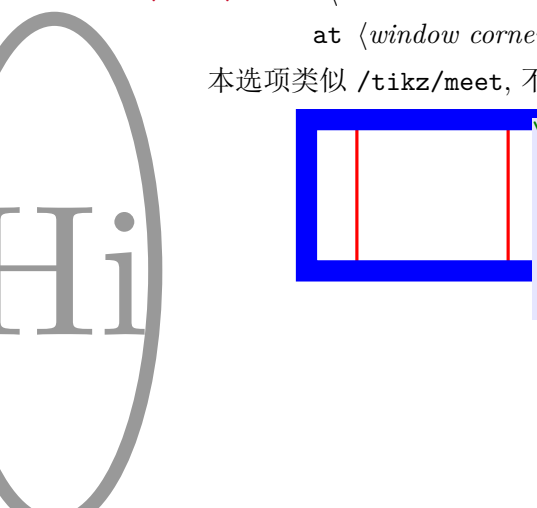

```
\tikz {
  \draw [red, very thick] (0,0) rectangle (20mm,20mm);
  \begin{scope}[meet = {(1.5,1.5) (2,1.8) at (0,0) (2,2)}]
    \draw [blue, very thick] (1.5,1.5) rectangle (2,1.8);
    \draw [thick,opacity=0.4] (1,1.5) circle [x radius=2mm, y
    ↪ radius=5mm] node {Hi};
  \end{scope} }
```

/tikz/view

这是 /tikz/meet 的别名。

/tikz/slice=⟨*to-be-viewed corner*⟩ rectangle ⟨*to-be-viewed corner*⟩
at ⟨*window corner*⟩ rectangle ⟨*window corner*⟩ (no default)

本选项类似 /tikz/meet, 不过是让可变矩形恰好把固定窗口容纳在内。




```
\tikz {
  \draw [red, very thick] (0,0) rectangle (20mm,20mm);
  \begin{scope}[slice = {(1.5,1.5) (2,1.8) at (0,0) (2,2)}]
    \draw [blue, very thick] (1.5,1.5) rectangle (2,1.8);
    \draw [thick,opacity=0.4] (1,1.5) circle [x radius=2mm, y
    ↪ radius=5mm] node {Hi};
  \end{scope} }
```

当把 view 用于动画时，属性 `:view` 的值只是可变矩形的对角点坐标 ⟨*to-be-viewed corner*⟩ `rectangle` ⟨*to-be-viewed corner*⟩（其中的 `rectangle` 可以省略），例如：

```
\tikz [animate = {
  my scope:view = {
    begin on = { click, of next = here },
      0s = "{(0.5,0.5) (2.5,1.5)}",
      2s = "{(0.5,0) (1.5,2)}", forever
}}] {
  \draw [red, fill=red!20, very thick, name=here]
    (0,0) rectangle (20mm,20mm);
  \begin{scope}[name = my scope,
                meet = {(0.5,0.5) (2.5,1.5) at (0,0) (2,2)}]
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [x radius=5mm, y radius=10mm] node {Hi};
  \end{scope} }
```

```
\tikz {
  \draw [red, fill=red!20, very thick, name=here]
    (0,0) rectangle (20mm,20mm);
  \begin{scope}[animate = { myself: = { :view = {
      begin on = { click, of = here },
      0s = "{(0.5,0.5) (2.5,1.5)}",
      2s = "{(0.5,0) (1.5,2)}", forever }}},
    slice = {(0.5,0.5) (2.5,1.5) at (0,0) (2,2)}]
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [x radius=5mm, y radius=10mm] node {Hi};
  \end{scope} }
```