TikZ & PGF 手册 (3.1.3) 笔记

2019 年 6 月 21 日

TikZ & PGF 是什么？ 请自行阅读手册的 Introduction 部分。

TikZ & PGF 能做什么？ 这个问题不好回答，简单地说，TikZ 主要用于图形方面的工作；而 PGF 除了用在图形方面外，还有其它的用处，例如它的数学引擎和 key 机制可以单独使用。很多其它宏包都用到 TikZ & PGF. TikZ & PGF 也在不断更新，新近的版本中增加了一些有趣的内容，例如制作 SVG 动画。

TikZ 能画什么图形？ 这个我也说不好，你可以翻看手册，查看各种例子，就能初步、大概地有个答案。反正科技资料中那些主要使用点、线构图的图形一般都可以用 TikZ 画出来，利用线条和色彩作装饰的图形也可以画，表格、阵列形式的内容也可以画，能画什么图形还得看使用者自己的创意。

这个笔记适合谁？ 适合那些懒得读英文的初学者。

这个笔记有错误吗？ 当然有，我自己只是 TikZ & PGF 的初级使用者，我学到的也很有限。如果你有 TikZ & PGF 方面的问题，可以去 QQ 群 (91940767) 问一下，我从群友们那里学到不少知识。也可以去网站 http://www.latexstudio.net/question/index 的问答栏提问。

绘图的程序有很多，该怎样选择？ 这个问题在网络上有很多解释，我的建议是：

- 如果你主要使用 LaTeX 编辑文档，而你又比较在意细节的处理，那建议你使用与 LaTeX 相容性较好的绘图工具。初学 LaTeX 时主要精力放在基本文类，基本命令、环境的学习上，此时采用插入外部图形的办法比较方便一些。当熟悉 LaTeX 的基本操作后，对图形的要求也会越来越高，此时使用与 LaTeX 相容性较好的绘图工具才更加方便。但是如果你对图形没有那么高的要求，那就无所谓了，你喜欢用什么就用什么。有的几何教材中的图形全部使用尺规手绘，这是一种很特别的特色，更是一种情怀，已经不仅仅是“绘图”了。
- 如果你大量使用与 TikZ & PGF 相关的工具，如 pgfplots 宏包，circuitikz 宏包，tcolorbox 宏包，tkz-euclide 宏包，那么建议你了解一下 TikZ & PGF.
- 如果你想自定义一个或一套用于绘图的命令，那非常建议你“仔细”了解一下 TikZ & PGF.
- 如果你需要的图形很难用 TikZ 画出来，那就使用别的工具画。

画图的工具很多，但一个人的精力有限，以有涯随无涯，殆已。无论你使用什么画图工具，除了应付日常需要之外，能从“画图”中找到乐趣也是很重要的！

TikZ 难学吗？ 不难学。像其他技能一样，熟能生巧。

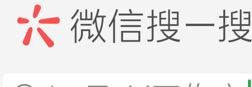
学习 TikZ & PGF 有哪些好资料？ 好资料应该有很多，你可以去 QQ 群 91940767 问一下，或者去网站 http://www.latexstudio.net 搜一下。我觉得最好的学习资料应该是《TikZ & PGF Manual》以及宏包的源代码，当然这个资料有点“大”，而且源代码也不容易读——这恰好表明，正因为诸位宏包作者和维护者的长期、出色工作，才有这么一款优秀的工具。

向 TikZ & PGF 的作者和维护者表示敬意！

这个笔记中的示例环境是用 tcolorbox 宏包做的。用 tcolorbox 宏包能制作风格多样、非常漂亮的盒子，推荐读者使用。tcolorbox 宏包与 TikZ & PGF 有很好的相容性，如果想发挥 tcolorbox 宏包的优点，那么了解一下 TikZ & PGF 是非常必要的。

也向 tcolorbox 宏包的作者表示敬意！

如果使用 tcolorbox 宏包的选项 `/tcb/listing engine=minted`, 并且在 xelatex 下编译，那么可能要使用编译选项 `--shell-escape-8bit`(参考 minted 宏包的手册)。如果你不想总是去命令行编译，可以把这两个选项添加到编辑器的当前排版工具中，例如，对于 TeXworks, 在“编辑–首选项–排版”中设置这两个选项。然后只要点击一下排版按钮就可以编译了。这个办法来自群友的提示。在群里总能学到东西，所以还是推荐读者加 QQ 群 (91940767), 可以扫码加群:



上面的微信二维码连接到 LaTeX 工作室的公众号，公众号会推送各种学习资料，精美的 LaTeX 作品，非常精美，爱不释手！推荐读者关注公众号，如果你有作品想跟大家分享，也可以向公众号投稿。

要是读者觉得这份中文材料值得打赏，请扫描下面的支付宝二维码，非常感谢读者的打赏！

目录

11 Design Principles 23

12 Hierarchical Structures:
Package, Environments, Scopes, and Styles 23
12.1 载入宏包和程序库 . . . 23
12.2 创建一个 picture . . . 24
12.2.1 用环境创建一个 picture . . . 24
12.2.2 用命令创建一个 picture . . . 25
12.2.3 Handling Catcodes and the Babel Package . . . 25
12.2.4 Adding a Background . . . 26
12.3 使用 scope . . . 26
12.3.1 scope 环境 . . . 26
12.3.2 scope 环境的简写形式—scopes 库 . . . 27
12.3.3 scope 命令 . . . 28
12.3.4 在路径之内插入 scopes . . . 28
12.4 使用图形选项 . . . 28
12.4.1 如何处理图形选项 . . . 28
12.4.2 使用 style . . . 29

13 设置坐标 30
13.1 Overview . . . 30
13.2 坐标系统 . . . 31
13.2.1 Canvas, XYZ, and Polar Coordinate Systems . . . 31
13.2.2 质心坐标系统 . . . 34
13.2.3 node 坐标系统 . . . 35
13.2.4 tangent 坐标系统 . . . 37
13.2.5 自定义坐标系 . . . 37
13.3 交点坐标 . . . 38
13.3.1 水平线与竖直线的交点：perpendicular 坐标系统 . . . 38
13.3.2 任意路径的交点 . . . 39
13.4 相对坐标，增量坐标 . . . 43
13.4.1 指定相对坐标 . . . 43
13.4.2 旋转的相对坐标——曲线上一点处的坐标系 . . . 44
13.4.3 相对坐标与当前点的局部化 . . . 45
13.5 坐标计算 . . . 46
13.5.1 一般句法 . . . 46
13.5.2 数乘坐标（向量） . . . 46
13.5.3 比例–角度定点句法 . . . 47
13.5.4 距离–角度定点句法 . . . 48

13.5.5 正射影—角度定点句法 49

14 设置路径的语句 49

14.1 Move-To 操作 51
14.2 Line-To 操作 52
14.2.1 线段 52
14.2.2 横线和竖线 53
14.3 Curve-To 操作 53
14.4 矩形操作 54
14.5 Rounding Corners 54
14.6 创建圆、椭圆 55
14.7 Arc 操作 56
14.8 Grid 操作 56
14.9 Parabola 操作 58
14.10 Sine 和 Cosine 操作 59
14.11 SVG 操作 60
14.12 Plot 操作 60
14.13 To Path 操作 60
14.14 Foreach 操作 62
14.15 Let 操作 62
14.16 Scoping 操作 65
14.17 Node and Edge 操作 65
14.18 Graph 操作 65
14.19 Pic 操作 65
14.20 Attribute Animation 操作 65
14.21 PGF-Extra 操作 65
14.22 在 Ti*k*Z 中使用软路径 66

15 Actions on Paths 66

15.1 Overview 66
15.2 指定颜色 67
15.3 画路径 67
15.3.1 Line Width, Line Cap, and Line Join 67
15.3.2 Dash Pattern 69
15.3.3 线条透明度 71
15.3.4 Double Lines and Bordered Lines 71
15.4 在路径上添加箭头 72
15.5 填充路径 72
15.5.1 图形参数：填充 Pattern 72
15.5.2 图形参数：非零规则和奇偶规则 73

15.5.3 图形参数：填充透明度 . . . 73
15.6 用任意图像填充路径 . . . 74
15.7 用颜色渐变填充路径 . . . 75
15.8 调整边界盒子 . . . 75
15.9 剪切 . . . 79
15.10 对一个路径执行多重操作 . . . 80
15.11 装饰路径 . . . 81
15.12 在起点或终点处截去一段路径 . . . 81

16 Arrows **82**
16.1 Overview . . . 82
16.2 如何添加箭头 . . . 82
16.3 设置箭头的外观 . . . 84
16.3.1 箭头的“特征尺寸” . . . 84
16.3.2 箭头的缩放 . . . 86
16.3.3 圆弧箭头 . . . 86
16.3.4 倾斜 . . . 86
16.3.5 Reversing, Halving, Swapping . . . 87
16.3.6 箭头颜色 . . . 87
16.3.7 线型 . . . 88
16.3.8 Bending and Flexing . . . 89
16.4 Arrow Tip Specifications . . . 93
16.4.1 句法 . . . 93
16.4.2 Specifying Paddings . . . 94
16.4.3 Specifying the Line End . . . 95
16.4.4 定义箭头的简写形式 . . . 95
16.4.5 Scoping of Arrow Keys . . . 96
16.5 Reference: Arrow Tips . . . 97

17 Nodes and Edges **99**
17.1 Overview . . . 99
17.2 Nodes and Their Shapes . . . 99
17.2.1 Node 命令的句法 . . . 99
17.2.2 预定义的形状 . . . 103
17.2.3 一般选项 . . . 104
17.3 Multi-Part Nodes . . . 107
17.4 node 中的文字 . . . 108
17.4.1 文字参数：颜色、不透明度 . . . 108
17.4.2 文字参数：字体 . . . 108
17.4.3 文字参数：文字换行、对齐方式、文字行宽 . . . 108

17.4.4 文字参数：文字的高度和深度 . . . 110
17.5 Positioning Nodes . . . 111
17.5.1 利用 anchor 来确定 node 的位置 . . . 111
17.5.2 基本的平移选项 . . . 112
17.5.3 高级平移选项 . . . 113
17.5.4 排布 node 的高级方法 . . . 117
17.6 Fitting Nodes to a Set of Coordinates . . . 117
17.7 变换 . . . 117
17.8 在直线段或曲线上显式地摆放 node . . . 117
17.9 在直线段会曲线上隐式地摆放 node . . . 120
17.10 label 和 pin 选项 . . . 121
17.10.1 Overview . . . 121
17.10.2 label 选项 . . . 121
17.10.3 The Pin Option . . . 123
17.10.4 引用句法 . . . 124
17.11 Connecting Nodes: Using Nodes as Coordinates . . . 127
17.12 Connecting Nodes: 用 edge 操作 . . . 127
17.12.1 edge 操作的基本句法 . . . 127
17.12.2 Edges 路径上的标签: Quotes Syntax . . . 129
17.13 Referencing Nodes Outside the Current Picture . . . 130
17.13.1 Referencing a Node in a Different Picture . . . 130
17.13.2 引用 Current Page Node——绝对位置 . . . 132
17.14 Late Code and Late Options . . . 132
17.15 遇到的问题 . . . 133
17.15.1 关于 auto 选项 . . . 133
17.15.2 关于 edge, to 的引用句法标签 . . . 133
17.15.3 把 {tikzpicture} 环境作为 node 的内容 . . . 133

18 Pics: Small Pictures on Paths 134
18.1 Overview . . . 134
18.2 The Pic Syntax . . . 134
18.2.1 指定所用的 pic type . . . 134
18.2.2 指定 pic 图形的位置 . . . 135
18.2.3 选项的有效与无效 . . . 135
18.2.4 定义 pic code . . . 135
18.2.5 pic 的选项的传递 . . . 136
18.2.6 指定 pic 图形的遮挡次序 . . . 136
18.2.7 设置每个 pic 图形的样式 . . . 137
18.2.8 设置 pic 图形中 node 名称的前缀并引用它 . . . 137
18.2.9 用 pic 制作动画 . . . 138

18.2.10 引用句法 138
18.3 定义 pic type 138
18.4 遇到的问题 140

41 angles 库 **142**

54 fit 程序库 **143**

73 topaths 程序库 **147**
73.1 直线 147
73.2 Move-To 147
73.3 曲线 147
73.4 Loops 151
73.5 关于 curve to 选项的系数 152

74 through 程序库 **153**

20 矩阵及其对齐方式 **153**
20.1 Overview 153
20.2 Matrices are Nodes 153
20.3 元素图形 154
20.3.1 元素图形的对齐方式 154
20.3.2 调整行距和列距 155
20.3.3 设置元素图形样式的选项 157
20.4 矩阵的位置 159
20.5 自定义分列符 160

59 matrix 库 **160**
59.1 矩阵中的 node 160
59.2 换行符号与矩阵行的结束符号 162
59.3 定界符 163

22 函数绘图 **164**
22.1 Overview 164
22.2 plot 路径操作 164
22.3 连点成线 164
22.4 从外部文件中读取数据绘图 165
22.5 用函数表达式绘图 166
22.6 调用 gnuplot 绘制函数图形 167
22.7 给 plot 路径上的样本点加标记 171
22.8 直线、曲线、柱状图、条形图等 173
22.9 遇到的问题 177

64 图柄库 **179**

64.1 曲线图柄 . . . 179
64.2 Constant 图柄 . . . 180
64.3 Comb 图柄 . . . 181
64.4 Bar 图柄 . . . 182
64.5 Gapped 图柄 . . . 183
64.6 Mark 图柄 . . . 184

65 Plot Mark 库 **186**

23 透明度 **187**

23.1 Overview . . . 187
23.2 为图形、路径、文字设定透明度 . . . 187
23.3 混色模式 . . . 188
23.4 颜色淡入、淡出——fading . . . 189
23.4.1 创建 fading . . . 189
23.4.2 创建 fading 路径 . . . 191
23.4.3 Fading a Scope . . . 193
23.5 Transparency Groups . . . 196
23.6 遇到的问题 . . . 198

53 fadings 库 **199**

24 装饰路径 **199**

24.1 Overview . . . 199
24.2 用 decorate 操作装饰子路径 . . . 202
24.3 装饰整个路径 . . . 203
24.4 调整装饰路径的外观 . . . 204
24.4.1 调整装饰路径与原被装饰路径的相对位置 . . . 204
24.4.2 调整装饰路径的始端与终端的形态 . . . 205

50 Decoration 库 **206**

50.1 公共选项 . . . 206
50.2 修饰路径的装饰类型 . . . 209
50.2.1 由直线段构成的装饰路径 . . . 209
50.2.2 由曲线构成的装饰路径 . . . 211
50.3 替换路径的装饰类型 . . . 213
50.4 标记装饰 . . . 216
50.5 自选标记装饰 . . . 217
50.5.1 程序库 decorations.markings . . . 217
50.5.2 脚印标记 . . . 222

50.5.3 形状装饰 . . . 223
50.6 文字装饰 . . . 228
50.6.1 装饰类型 `text along path` . . . 228
50.6.2 装饰类型 `text effects along path` . . . 232
50.7 分形装饰 . . . 240

25 变换 241
25.1 各种坐标系统 . . . 241
25.1.1 变换选项的作用次序以及作用方式 . . . 242
25.1.2 各种标架及其作用 . . . 243
25.2 标架变换：指定标架 . . . 249
25.3 坐标变换 . . . 254
25.3.1 坐标变换选项 . . . 254
25.3.2 注意的问题 . . . 258
25.3.3 平面上的轴对称 . . . 260
25.4 画布变换 . . . 268

40 三维绘图库 269
40.1 坐标系统 . . . 269
40.2 坐标平面 . . . 271
40.2.1 转换到任意平面 . . . 271
40.2.2 预定义的平面 . . . 272
40.3 例子 . . . 273

63 三点透视图程序库 274
63.1 坐标系统 . . . 275
63.2 设置视角 . . . 275
63.3 自定义透视 . . . 276
63.4 缺点 . . . 277
63.5 例子 . . . 277
63.6 代码实现 . . . 277

26 动画 281
26.1 Introduction . . . 281
26.2 创建动画 . . . 283
26.2.1 Animate 选项 . . . 283
26.2.2 时间线条目 . . . 284
26.2.3 指定对象 . . . 284
26.2.4 指定属性 . . . 285
26.2.5 指定 ID . . . 285
26.2.6 指定时刻 . . . 286

26.2.7 属性的值 . . . 288
26.2.8 Scopes . . . 289
26.3 各种句法 . . . 289
26.3.1 指定对象和属性的句法 . . . 289
26.3.2 关于 myself 的动画 . . . 290
26.3.3 关于时刻的句法 . . . 291
26.3.4 引号与属性值 . . . 292
26.3.5 时间表 . . . 292
26.4 可用于动画的属性 . . . 293
26.4.1 Animating Color, Opacity, and Visibility . . . 294
26.4.2 Animating Paths and their Rendering . . . 297
26.4.3 动态变换: Relative Transformations . . . 299
26.4.4 Animating Transformations: Positioning . . . 303
26.4.5 Animating Transformations: Views . . . 304
26.5 调控时间线 . . . 304
26.5.1 Before and After the Timeline: Value Filling . . . 304
26.5.2 Beginning and Ending Timelines . . . 305
26.5.3 Repeating Timelines and Accumulation . . . 307
26.5.4 Smoothing and Jumping Timelines . . . 308
26.6 Snapshots . . . 309
26.7 一个例子 . . . 311

78 Views Library **312**

49 Circuits 程序库 **314**
49.1 简介 . . . 314
49.1.1 一个例子 . . . 315
49.1.2 符号 . . . 316
49.1.3 Symbol Graphics . . . 316
49.1.4 Annotations . . . 317
49.2 circuits 程序库 . . . 318
49.2.1 Symbol 的尺寸 . . . 318
49.2.2 声明新的 symbols . . . 319
49.2.3 让 symbol 指向某个方向 . . . 321
49.2.4 Info 标签 . . . 322
49.2.5 创建、使用 annotation . . . 324
49.2.6 调整 Symbols 的外观 . . . 325
49.2.7 符号图形的变体 . . . 327
49.3 逻辑电路 . . . 329
49.3.1 Overview . . . 329

49.3.2 逻辑门符号 335
49.3.3 逻辑门符号的形状 335
49.3.4 US 风格的逻辑门符号形状 337
49.3.5 IEC 风格的逻辑门符号形状 339
49.4 电子工程电路 341
49.4.1 Overview 341
49.4.2 指示电流方向的符号 348
49.4.3 Symbols: Basic Elements 348
49.4.4 Symbols: Diodes 348
49.4.5 Symbols: Contacts 348
49.4.6 Symbols: Measurement devices 348
49.4.7 Units 348
49.4.8 Annotations 348
49.4.9 EE-Symbols 的形状 348
49.4.10 IEC 风格的 EE-Symbols 形状 351
49.5 遇到的问题 353

55 定点算术程序库 353

55.1 Overview 354
55.2 在 PGF 和 TikZ 中使用定点算术 354

56 浮点单元程序库 355

56.1 Overview 355
56.2 用法 356
56.3 与定点算术程序库的比较 358
56.4 命令与编程参考 359
56.4.1 浮点数的创建与转换 359
56.4.2 符号舍入操作 362
56.4.3 数学运算命令 363
56.4.4 用于编程的原始数学程序 365
56.4.5 例子 365
56.5 遇到的问题 366

57 Lindenmayer System 分形图 367

57.1 Overview 370
57.1.1 声明一个 L-S 370
57.2 使用 L-S 373
57.2.1 在 PGF 中使用 L-S 373
57.2.2 在 TikZ 中使用 L-S 373

58 数学程序库 **375**

58.1 Overview . . . 375
58.2 赋值语句 . . . 376
58.3 声明变量类型 . . . 377
58.4 循环语句 . . . 380
58.5 条件语句 . . . 381
58.6 声明函数 . . . 381
58.7 在命令 \tikzmath 的辖域内执行代码 . . . 382

69 shadings 程序库 **384**

70 shadows 程序库 **388**

70.1 Overview . . . 388
70.2 一般的阴影选项 . . . 389
70.3 预定义的阴影 . . . 389
70.3.1 Drop Shadows . . . 389
70.3.2 Copy Shadows . . . 390
70.4 针对圆形的阴影 . . . 391

72 Spy 程序库：将图形的局部放大 **392**

72.1 将图形的某个局部放大 . . . 393
72.2 spy scopes . . . 394
72.3 spy scopes . . . 394
72.4 预定义的 spy 样式 . . . 397
72.5 例子 . . . 397

79 数据可视化简介 **398**

79.1 数据点 . . . 398
79.2 可视化管线（visualization Pipeline） . . . 398

80 数据可视化的基本概念 **399**

80.1 Overview . . . 399
80.2 数据点与数据格式 . . . 399
80.3 轴，刻度线，网格 . . . 400
80.4 显像器（visualizer） . . . 401
80.5 样式表和图例 . . . 401
80.6 用法 . . . 402
80.7 在数据可视化过程中执行用户自定义的代码 . . . 406
80.8 创建新对象 . . . 407

81 用于数据可视化的数据格式 407
81.1 Overview . . . 407
81.2 简介 . . . 407
81.3 内置格式 . . . 407
81.4 函数格式 . . . 410
81.5 数据处理过程 . . . 412
81.6 定义新数据格式 . . . 413

82 坐标轴 415
82.1 Overview . . . 415
82.2 轴的基本设置 . . . 415
82.2.1 用法 . . . 416
82.2.2 与轴对应的变量 . . . 417
82.2.3 变量值的范围 . . . 417
82.2.4 轴对数据的变换 . . . 418
82.2.5 对数轴 . . . 421
82.2.6 设置坐标轴的长度和单位长度 . . . 421
82.2.7 坐标轴的标签 . . . 423
82.2.8 将变量值与页面上的坐标系相关联 . . . 423
82.3 轴系统 . . . 424
82.3.1 用法 . . . 425
82.3.2 Scientific Axis Systems . . . 425
82.3.3 School Book Axis Systems . . . 427
82.3.4 底层的笛卡尔坐标系 . . . 428
82.4 坐标轴的刻度和网格 . . . 430
82.4.1 概略 . . . 430
82.4.2 刻度和网格的主要选项 . . . 430
82.4.3 计算刻度线和网格线位置的半自动机制 . . . 431
82.4.4 计算刻度线和网格线位置的自动机制 . . . 432
82.4.5 手工确定刻度线和网格线的位置 . . . 434
82.4.6 刻度与网格线的样式：概略 . . . 436
82.4.7 刻度与网格线的样式：style 与 node Style . . . 436
82.4.8 网格线的样式 . . . 437
82.4.9 刻度线与刻度值标签的样式 . . . 439
82.4.10 设置个别刻度的样式 . . . 441
82.4.11 其它刻度值标签选项 . . . 441
82.4.12 交错叠放刻度值标签 . . . 443
82.4.13 自动添加刻度的策略 . . . 446
82.4.14 定义新的添加刻度的策略 . . . 446
82.5 创建新的轴系统 . . . 446

82.5.1 创建一个轴系统 . . . 447
82.5.2 坐标轴的可视化 . . . 450
82.5.3 可视化网格线 . . . 455
82.5.4 刻度线、刻度值标签的可视化 . . . 457
82.5.5 坐标轴标签的可视化 . . . 460
82.5.6 完整的定义代码 . . . 462
82.5.7 专用于创建新坐标系统的 key . . . 466
82.6 遇到的问题 . . . 467

83 Visualizers 468
83.1 Overview . . . 468
83.2 用法 . . . 468
83.2.1 使用一个显像器 . . . 468
83.2.2 使用多个显像器 . . . 469
83.2.3 设置显像器的外观效果 . . . 471
83.3 基本的显像器 . . . 474
83.3.1 直线段或曲线显像器 . . . 474
83.3.2 散点显像器 . . . 476
83.4 创建新的显像器 . . . 477

84 样式表与图例 477
84.1 Overview . . . 477
84.2 Style Sheets 的例子 . . . 477
84.3 Legends 的例子 . . . 478
84.4 Style Sheet 的用法 . . . 479
84.4.1 引入一个 Style Sheet . . . 479
84.4.2 创建新的样式表 . . . 480
84.4.3 创建新的颜色样式表 . . . 483
84.5 预定义的线型样式表 . . . 485
84.6 预定义的散点样式表 . . . 487
84.7 预定义的颜色样式表 . . . 488
84.8 显像器的标签 . . . 489
84.8.1 给一组数据点设置标签 . . . 490
84.8.2 给一组数据点设置大头针标签 . . . 493
84.9 为数据点组创建图例 . . . 495
84.9.1 创建图例，图例中的条目 . . . 496
84.9.2 图例中条目的行列排布 . . . 500
84.9.3 确定图例位置的一般方法 . . . 503
84.9.4 在绘图区域之外放置图例 . . . 504
84.9.5 在绘图区域之内放置图例 . . . 506

84.9.6 图例条目的一般样式 507
84.9.7 图例条目中的文字标签 508
84.9.8 条目中文字标签与图示标签的相对位置 509
84.9.9 手工添加条目 510
84.9.10 条目显像器 511

85 极坐标系 516

85.1 Overview 516
85.2 Scientific Polar Axis System 517
85.2.1 角度轴的刻度线 518
85.2.2 角度轴的角度范围 519
85.3 创建新的极坐标系统 520

86 The Data Visualization Backend 521

87 Key Management 521

87.1 简介 521
87.1.1 与其它类似宏包的比较 521
87.1.2 快速引导 521
87.2 The Key Tree 523
87.3 执行 Keys 526
87.3.1 首字符句法检测 536
87.3.2 默认参数值 538
87.3.3 方法：定义“执行某个命令的键”并使用之 539
87.3.4 Keys That Store Values 546
87.3.5 定义手柄键 546
87.3.6 设置未知键的提示信息 551
87.3.7 搜索键的前缀路径，搜索手柄 553
87.4 手柄 553
87.4.1 设置键路径的手柄 553
87.4.2 设置键的默认值的手柄 554
87.4.3 定义键所储存的代码 555
87.4.4 定义样式的手柄 557
87.4.5 Defining Value-, Macro-, If- and Choice-Keys 560
87.4.6 键值的展开，多重键值 563
87.4.7 键路径的转换 565
87.4.8 测试键的手柄 570
87.4.9 解释键的手柄 571
87.5 提示错误的键 572
87.6 键筛选 574

88 重复操作：foreach 句法 **574**
88.1 ⟨*commands*⟩ 的句法 575
88.2 ⟨*list*⟩ 中的省略号 575
88.3 在 ⟨*list*⟩ 中使用花括号包裹列举条目 576
88.4 在路径中使用 foreach 语句 576
88.5 多个相互关联的变量 576
88.6 针对变量的选项 578
88.7 命令 \pgfplotsforeachungrouped 580

91 扩展颜色支持 **582**

92 解析器模块 **584**
92.1 Parser 模块的选项 589
92.2 例子 589

93 数学引擎概略 **589**

94 数学表达式 **590**
94.1 解析一个表达式 591
94.1.1 命令 591
94.1.2 长度单位的“显”、“隐” 594
94.2 数学表达式中的算子 595
94.3 数学表达式中的函数 598
94.3.1 基本算术函数 598
94.3.2 舍入函数 601
94.3.3 几个整数运算函数 602
94.3.4 三角函数 602
94.3.5 比较函数与逻辑函数 604
94.3.6 伪随机函数 606
94.3.7 基本的转换函数 606
94.3.8 其它函数 607

95 其它数学命令 **608**
95.1 基本算术函数 608
95.2 比较与逻辑函数 608
95.3 整数的进位制转换 609
95.4 角度计算 611

96 用数学引擎自定义函数 **611**

97 输出数值的格式 615
97.1 基本的命令与选项 . . . 615
97.2 输出数值的样式以及标点符号 . . . 620

99 基本层（basic layer）概略 625
99.1 内核和模块 . . . 625
99.2 基本层的宏 . . . 625
99.3 以路径为核心的构图方式 . . . 626
99.4 坐标变换与画布变换 . . . 626

100 层级结构：宏包，环境，子环境，文字 626
100.1 Overview . . . 626
100.1.1 宏包的层级结构 . . . 626
100.1.2 图形的层级结构 . . . 626
100.2 宏包的层次 . . . 627
100.2.1 内核宏包 . . . 627
100.2.2 模块 . . . 628
100.2.3 程序库宏包 . . . 628
100.3 图形的层级 . . . 628
100.3.1 主要的环境 . . . 628
100.3.2 绘图子环境 . . . 631
100.3.3 插入文字和图形 . . . 634
100.4 Object Identifiers . . . 635
100.4.1 创建图形对象的命令 . . . 635
100.4.2 设置、引用 identifier . . . 636
100.5 Resource Description Framework Annotations (RDFa) . . . 638
100.6 错误信息与警告 . . . 638

101 指定坐标 638
101.1 Overview . . . 638
101.2 基本的坐标命令 . . . 638
101.3 XY-坐标系统中的坐标 . . . 639
101.4 三维坐标 . . . 640
101.5 用已有坐标构建新的坐标 . . . 641
101.5.1 基本的坐标计算 . . . 641
101.5.2 直线或曲线上的点 . . . 641
101.5.3 矩形或椭圆边界上的点 . . . 643
101.5.4 两直线的交点 . . . 644
101.5.5 两个圆的交点 . . . 644
101.5.6 两个路径的交点 . . . 644
101.6 坐标分量 . . . 645

101.7 坐标点命令的工作方式 646

102 构建路径 647

102.1 Overview 647
102.2 Move-To 路径操作 647
102.3 Line-To 路径操作 648
102.4 Curve-To 路径操作 649
102.5 Close 路径操作 650
102.6 Arc, Ellipse, Circle 路径操作 650
102.7 Rectangle 路径操作 654
102.8 Grid 路径操作 654
102.9 Parabola 路径操作 655
102.10 Sine 和 Cosine 路径操作 655
102.11 Plot 路径操作 656
102.12 圆角（Rounded Corners） 656
102.13 跟踪路径或图形的边界盒子 658

103 路径装饰 660

103.1 Overview 660
103.2 装饰自动化（Decoration Automata） 660
103.2.1 约定路径名称 661
103.2.2 片段（segment）与状态（state） 662
103.3 自定义装饰路径 663
103.4 {pgfdecoration} 环境 674
103.5 Meta-Decorations 678
103.5.1 定义一个 Meta-Decorations 678
103.5.2 预定义的 Meta-decorations 681
103.5.3 {pgfmetadecoration} 环境 681

104 使用路径 681

104.1 Overview 681
104.2 画出路径 682
104.2.1 图形参数：线宽 Line Width 682
104.2.2 图形参数：线冠 Caps 与交接 Joins 683
104.2.3 图形参数：线型 Dashing 684
104.2.4 图形参数：线条颜色 684
104.2.5 线条透明度 684
104.2.6 双线的内线 684
104.3 给路径加箭头 685
104.4 填充路径 686
104.4.1 图形参数：判断内部点的规则 686

104.4.2 图形参数：填充色 687
104.4.3 图形参数：填充色的不透明度 687
104.5 剪切路径 687
104.6 将路径用作边界盒子 687

105 定义新的箭头 687
105.1 Overview 687
105.2 有关术语 688
105.3 PGF 处理箭头的一般过程 688
105.4 自定义箭头 689
105.5 关于箭头的选项 695
105.5.1 尺寸选项 695
105.5.2 True–False 选项 695
105.5.3 `setup code` 中不能引用的选项 696
105.5.4 自定义箭头选项 696

106 Nodes and Shapes 704
106.1 Overview 704
106.1.1 创建与索引 node 704
106.1.2 锚 Anchors 705
106.1.3 shape 的“层”Layers 705
106.1.4 Node Parts 705
106.2 创建 node 705
106.2.1 创建简单 node 705
106.2.2 创建 Multi-Part Nodes 706
106.2.3 另一种添加 node 的方法 710
106.3 使用锚位置 Anchors 712
106.3.1 在一个图形中引用锚位置 713
106.3.2 跨图引用 node 的锚位置 714
106.4 特殊 node 715
106.5 定义新的 shape 717
106.5.1 一个 shape 具备的要素 717
106.5.2 Normal Anchors 与 Saved Anchors 717
106.5.3 定义新 shape 的命令 717
106.5.4 一个例子 725

107 矩阵 744
107.1 Overview 744
107.2 矩阵元素的对齐方式 744
107.3 矩阵命令 745
107.4 行间距与列间距 747

107.5 调用命令 748

108 坐标变换，画布变换，非线性变换 749

108.1 Overview 749
108.2 坐标变换 750
108.2.1 坐标变换矩阵 750
108.2.2 坐标变换命令 750
108.2.3 其它变换 754
108.2.4 保存或使用某个变换矩阵 754
108.2.5 坐标变换中的调整 755
108.3 画布变换 756
108.3.1 创建 View Boxes 757
108.4 非线性变换 758
108.4.1 导引 758
108.4.2 定义并载入一个非线性变换 759
108.4.3 将非线性变换用于一个点 760
108.4.4 将非线性变换用于一个路径 761
108.4.5 将非线性变换用于文字 762
108.4.6 用线性变换近似非线性变换 762
108.4.7 非线性变换程序库 763

109 图样 Patterns 765

109.1 Overview 765
109.2 声明一个图样 766
109.3 使用图样 770

110 声明、使用外部图形 770

110.1 Overview 770
110.2 声明外部图形 770
110.3 使用外部图形 771
110.4 给图形“带面具” 772

112 创建 Plots 773

112.1 Overview 773
112.2 创建图流 773
112.2.1 图流的基本结构 774
112.2.2 生成图流的命令 776
112.3 图柄 779
112.4 定义新图柄 780

113 图层 785
113.1 Overview . . . 785
113.2 声明图层 . . . 785
113.3 在图层上绘图 . . . 786

114 颜色渐变 787
114.1 Overview . . . 787
114.1.1 颜色渐变中使用的颜色模式 . . . 787
114.2 声明渐变样式 . . . 788
114.2.1 横向渐变与纵向渐变 . . . 788
114.2.2 辐射渐变 . . . 788
114.2.3 函数渐变 . . . 789
114.3 使用颜色渐变 . . . 794
114.4 关于 type 4 函数的补充 . . . 796

115 透明度 800
115.1 指定不透明度 . . . 801
115.2 指定混色模式 . . . 801
115.3 Fading 效果 . . . 802
115.4 透明度组 . . . 804

116 Animations 805
116.1 Overview . . . 805
116.2 Animating an Attribute . . . 805
116.2.1 主要命令 . . . 805
116.2.2 时间线选项 . . . 807
116.2.3 快照 . . . 808
116.3 Animating Color, Opacity, Visibility, and Staging . . . 809
116.4 Animating Paths and their Rendering . . . 810
116.5 Animating Transformations and Views . . . 811
116.6 Commands for Specifying Timing: Beginnings and Endings . . . 812
116.7 Commands for Specifying Timing: Repeats . . . 813

117 临时寄存器 813

118 快速命令 815
118.1 快速坐标命令 . . . 815
118.2 快速创建路径的命令 . . . 816
118.3 快速使用路径的命令 . . . 816
118.4 快速文字盒子命令 . . . 817

119 系统层的设计 **817**

119.1 驱动文件 . . . 817
119.2 公共的定义文件 . . . 818

120 系统命令 **818**

120.1 系统命令流的开启与结束 . . . 818
120.2 构建子环境的系统命令 . . . 820
120.3 构建路径的系统命令 . . . 820
120.4 做画布变换的系统命令 . . . 821
120.5 画、填充、剪切路径的系统命令 . . . 822
120.6 设置图形状态选项的系统命令 . . . 823
120.7 设置颜色的系统命令 . . . 824
120.8 关于图样的系统命令 . . . 826
120.9 插入外部图形的系统命令 . . . 826
120.10 关于颜色渐变的系统命令 . . . 826
120.11 关于透明度的系统命令 . . . 826
120.12 关于动画的系统命令 . . . 826
120.13 关于 Object Identification 的系统命令 . . . 826
120.14 可重复利用对象的系统命令 . . . 833
120.15 使得路径“不可见”的系统命令 . . . 833
120.16 关于页面尺寸的系统命令 . . . 834
120.17 跟踪页面上某个位置的系统命令 . . . 834
120.18 尺寸转换命令 . . . 834

121 软路径子系统 **835**

121.1 创建路径的过程 . . . 835
121.2 保存、调用一个软路径 . . . 835
121.3 创建软路径的命令 . . . 836
121.4 软路径数据结构 . . . 836

122 Protocol 子系统 **837**

123 动画的系统层 **837**

11 Design Principles

TikZ 的设计主要有以下几个方面

1. 指定坐标的句法。
2. 指定路径的句法。
3. 对路径的操作。
4. 图形参数 (graphic parameters) 的 Key-value 句法。
5. 指定 node 的句法。
6. 指定 tree 的句法。
7. 指定 graph 的句法。
8. 图形参数的分组。
9. 坐标变换系统。

12 Hierarchical Structures: Package, Environments, Scopes, and Styles

12.1 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 TeX 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.tex` 前使用命令

```
\def\pgfsysdriver{pgfsys-dvipdfm.def}
```

`\usetikzlibrary`{⟨*list of libraries*⟩}

当载入 tikz 后，就可以用这个命令载入 libraries. 列出的 libraries 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）。

12.2 创建一个 picture

12.2.1 用环境创建一个 picture

tikz 的最外层绘图区域是 {tikzpicture} 环境，所有绘图命令（除了命令 \tikzset）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。遇到这个环境时，⟨*options*⟩ 会被解析，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）。然后环境的内容被处理，图形命令会被放入盒子中。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，因为这会扰乱驱动的位置计算系统。

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。每遇到一个坐标 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）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. 76 来调整边界盒子。

/tikz/baseline=⟨*dimension or coordinate of default*⟩ (default 0pt)

通常，图形的最下端会被放在图形周围文字的基线上。使用本选项可以把图形中的水平直线 $y =$ ⟨*value*⟩ 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。直到绘图过程的末尾处才会计算这个 ⟨*coordinate*⟩，所以 ⟨*coordinate*⟩ 可以是图形中 node 的坐标。

Hello world

```
Hello
\tikz[baseline=(X.base)]
  \node [cross out,draw] (X) {world};
```

Top align:

```
Top align:
\tikz[baseline=(current bounding box.north)]
  \draw (0,0) rectangle (1cm,1ex);
```

/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，本选项必须针对环境 {tikzpicture} 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}
```

`/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，本选项必须针对环境 `{tikzpicture}` 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）会被添加到每个图形的开始处。

```
\tikzset{every picture/.style={draw=red,semithick}}
```

`\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。

12.2.2 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。

12.2.3 Handling Catcodes and the Babel Package

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`，

这个库可以与其它（全局地改变符号类别的）宏包一起使用。`babel` 库的工作是在 `{tikzpicture}` 环境的开头重设符号类别，并在 node 的开头处重载符号类别设置。

12.2.4 Adding a Background

在默认下图形没有背景，如果要添加背景，可参考 backgrounds 库。

12.3 使用 scope

在 `{tikzpicture}` 环境里可以用 `{scope}` 环境创建域(scope)。`{scope}` 环境只能用在 `{tikzpicture}` 环境里。

12.3.1 scope 环境

`\begin{scope}`⟨*animations spec*⟩`[`⟨*options*⟩`]`
⟨*environment content*⟩
`\end{scope}`

环境选项 ⟨*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。

`/tikz/every scope` (style, initially empty)

这个 style 会添加到每个 `{scope}` 环境的开头。

`/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。

`/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。

12.3.2 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. 26 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开启一个 {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⟩
}
```

当遇到 {[⟨options⟩] 时会插入 \begin{scope}[⟨options⟩]，之后遇到闭花括号 } 时会插入 \end{scope}，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三种地方：分号之后（分号表示一个路径的结束），{scope} 环境之后，{scope} 环境的开头。在使用这种简写形式时要注意检查这种形式是否真地如你所期望，观察下面的例子：

```
\tikz{{ [red] \draw (0,0)--(1,1); }}
```

上面这个简写的 {scope} 环境无效。

U

```
\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。

另外在 `\tikzmath{}` 之后使用简写 `{scope}` 环境时也要注意是否有效。

12.3.3 scope 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。

12.3.4 在路径之内插入 scopes

命令 `\path` 用于创建一个路径（path），此命令可以带有图形选项（graphic options），这些选项只对本路径有效。使用简写形式的 scope 可以在路径内部插入一个 scope：

```
\tikz \draw (0,0) -- (1,1)
  {[rounded corners, red] -- (2,0) -- (3,1)}
  -- (3,0) -- (2,1);
```

上面例子中，选项 `rounded corners` 的作用范围受到花括号的限制，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。

12.4 使用图形选项

12.4.1 如何处理图形选项

很多 tikz 的命令、环境都接受选项 (options), 这些选项被称为 key 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。

12.4.2 使用 style

参考本手册的 pgfkeys 宏包。

一个 style（样式）就是一组选项或代码、操作，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，或者在某个时机执行代码、操作。有许多预定义的样式，例如规定了颜色和线宽，效果如下：

```
\tikz[help lines]{
  \draw (0,0)--(2,0);
  \fill (0,0.5) rectangle (2,2);
}
```

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，所以也可以把定义样式的代码放在命令、环境的选项列表中。

```
\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。

13 设置坐标

13.1 Overview

一个坐标就是图形画布（canvas）上的一个点。tikz 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 tikz 遇到开圆括号“(”时就倾向于认为它遇到了一个坐标。指定坐标的一般句法是：

```
([⟨options⟩]⟨coordinate specification⟩)
```

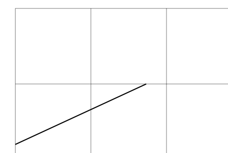
⟨*coordinate specification*⟩ 利用某种坐标系统来指定一个坐标。有数种坐标系统，例如 Cartesian coordinate system，polar coordinates，spherical coordinates，无论使用哪一种坐标系统，最终都会对应到画布上的某一个点。有两种选定坐标系统的方式：

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⟩)
```

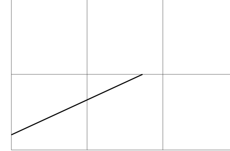
Implicitly 显式地指定坐标的句法格式有点繁琐，“隐式地”格式会简洁一些。“隐式地”指的是你不需要给出坐标系统的名称，tikz 会根据你的输入格式自动确定所对应的坐标系统。例如 `(0,0)` 对应笛卡尔坐标，`(30:2)` 对应极坐标（其中 30 代表角度）。

对比下面两个例子：



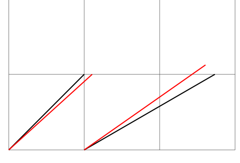
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (canvas cs:x=0cm,y=2mm)
    -- (canvas polar cs:radius=2cm,angle=30);
\end{tikzpicture}
```

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：

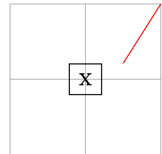


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0cm,2mm) -- (30:2cm);
\end{tikzpicture}
```

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}
```

13.2 坐标系统

对于几何图形来说，坐标系是比较直观的。但在这里，一个“坐标系统”实际上是个计算体系，能够实现特定的计算目的。有的计算体系专门计算 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。

13.2.1 Canvas, XYZ, and Polar Coordinate Systems

Coordinate system `canvas`

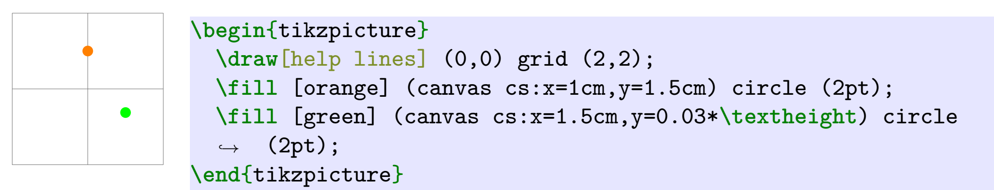
这是画布坐标系统，x 轴正向向右，y 轴的正向向上，使用选项 x=d_x 和 y=d_y 设置坐标数据，数据是带长度单位的尺寸，也可以是（展开为长度的）宏。

`/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，也可以是（展开为长度的）宏，也可以是算式，但算式内的运算项应该带长度单位，例如 `1cm+2pt`，数学引擎会处理算式得到结果。

`/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}
```

指定画布坐标的“隐式”形式是 `(2cm,30pt)` 这种带单位的形式，或者是与之等效的形式，如 `(2cm,\textheight)`。

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=，y= ，z= 给出，坐标数据不带长度单位。各个坐标轴的单位长度默认为 1cm，而 z 轴的单位向量默认是 $(-3.85\text{mm}, -3.85\text{mm})$。坐标轴的单位长度可以用选项 x=⟨*dimension*⟩，y=⟨*dimension*⟩ 来修改 (⟨*dimension*⟩ 要带上长度单位)，例如 x=-2cm 就把 $(-2\text{cm}, 0)$ 作为 x 轴的单位向量。

/tikz/cs/x=⟨*factor*⟩ (no default, initially 0)

注意，作为 xyz 坐标系统下的坐标，这里的 ⟨*factor*⟩ 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。如果 ⟨*factor*⟩ 带上长度单位就变成 canvas 坐标系统下的坐标了。

/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}
```

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}
```

给出的隐式坐标 (⟨*x*⟩,⟨*y*⟩)，如果其坐标数据都带长度单位，则将其解释到 canvas 坐标系统中；如果坐标数据都不带长度单位，则将其解释到 xyz 坐标系统中；如果有的数据带长度单位、有的数据不带长度单位，处理规则是：

- (2pt,3) 等效于 (2pt,0pt)+(0,3)，也就是说，把 canvas 坐标系统的 x 轴与 xyz 坐标系统的 y 轴拿来组合成一个坐标系统，就是 (2pt,3) 对应的坐标系统。
- (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) 等效于 (5pt,9pt)。

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。

/tikz/cs/radius=⟨*dimension*⟩ (no default)

本选项指定坐标点的极径，⟨*dimension*⟩ 是带长度单位的尺寸。

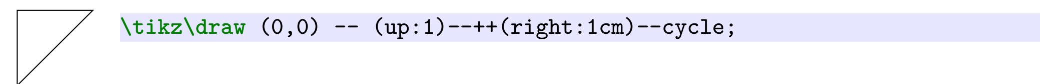
/tikz/cs/x radius=⟨*dimension*⟩ (no default)

/tikz/cs/y radius=⟨*dimension*⟩ (no default)

选项 `x radius=`d_x 与选项 `y=`d_y 确定一个椭圆，其横半轴长度是 d_x，纵半轴长度是 d_y，d_x 与 d_y 都是带长度单位的尺寸。这两个选项配合选项 `angle=`θ，就把椭圆上方向角为 θ 的点确定下来。

`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

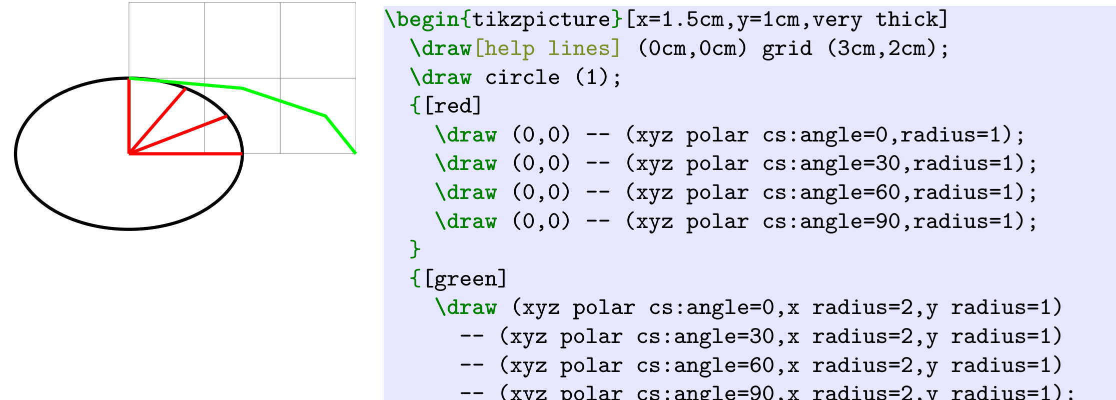


```
\tikz\draw (0,0) -- (up:1)--++(right:1cm)--cycle;
```

Coordinate system xyz polar

使用选项 `angle`，`radius`，`x radius`，`y radius`，`z radius` 提供坐标数据，然后翻译到 `xyz` 坐标系统中。如果不提供 `z radius` 数据就是二维的情况。对于二维的情况，按如下方式确定一个点：

假设 x 轴的单位向量是 v_x，y 轴的单位向量是 v_y。在默认下 $v_x = (1\text{cm}, 0\text{cm})$，$v_y = (0\text{cm}, 1\text{cm})$，不过坐标轴的单位向量是可以通过变换选项修改的。给出选项 `angle=`θ（$-360 \leqslant \theta \leqslant 720$），`x radius=`$f_x$，`y radius=`$f_y$，这里 θ，f_x，f_y 都是数值。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。然后将 E 的横半轴变为 $f_x \cdot v_x$，将纵半轴变为 $f_y \cdot v_y$，得到椭圆 E'。这个变化可能会导致椭圆的形状有所变化，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)

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。

Coordinate system `xy polar`

这是 `xyz polar` 的二维情况。

13.2.2 质心坐标系统

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

$$\frac{a_1 v_1 + \cdots + a_n v_n}{a_1 + \cdots + a_2},$$

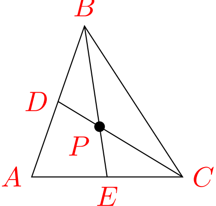
如果把点 v_i 看作质点，把数值 a_i 看作 v_i 的质量，上式就是这个质点组的质心。

Coordinate system `barycentric`

对应这个坐标系统的坐标格式是

`([⟨options⟩]barycentric cs:⟨node name⟩=⟨number⟩,⟨node name⟩=⟨number⟩...)`

注意其中 ⟨*node name*⟩ 的前后不能有空格。⟨*node name*⟩ 是 node 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);)。⟨*number*⟩ 是纯数值（可以是负值），看作是相应质点的“质量”。




```
\begin{tikzpicture}[every node/.style={red}]
  \coordinate (A) at (0,0);
  \coordinate (B) at (0.7,2);
  \coordinate (C) at (2,0);
  \coordinate (D) at ($(A)!0.5!(B)$);
  \coordinate (E) at ($(A)!0.5!(C)$);
  \node [left] at (A) {$A$};
  \node [above] at (B) {$B$};
  \node [right] at (C) {$C$};
  \node [left] at (D) {$D$};
  \node [below] at (E) {$E$};
  \draw (A) -- (B) -- (C) -- cycle;
  \draw (B) -- (E) (C) -- (D);
  \fill (barycentric cs:A=-1,B=-1,C=-1) circle (2pt);
  \node [below left]at (barycentric cs:A=1,B=1,C=1) {$P$};
\end{tikzpicture}
```

13.2.3 node 坐标系统

Coordinate system node

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/anchor=⟨*anchor*⟩ (no default)

指定 node 的 ⟨*anchor*⟩ 位置。注意这个选项的路径是 /tikz，不是 /tikz/cs。

a node

```
\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*⟩ 是射线，射线与 node 边界线的交点就是本选项确定的点。

```
\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 tikz 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 tikz 不能确定“恰当的”点，就会把 node 的中心点作为线条的端点，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 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

`--(node cs:name=b)--`

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)`。

13.2.4 tangent 坐标系统

Coordinate system `tangent`

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 就是路径。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)

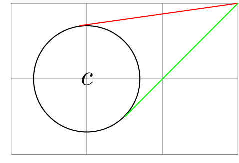
指定所需的 node。

`/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。

13.2.5 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*⟩`)`

其中的 ⟨*arguments*⟩ 会被传递给 ⟨*code*⟩ 来处理，也就是说，⟨*arguments*⟩ 对应 ⟨*code*⟩ 中的参数 `#1`. 执行 ⟨*code*⟩ 的代码实现一个计算过程，执行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（这两个寄存器是 pgf 自己定义并处理的），这两个寄存器分别作为 x 分量和 y 分量一起确定一个 `canvas` 坐标系统中点—记为 ⟨*target*⟩—这样就由 ⟨*arguments*⟩ 得到了 ⟨*target*⟩. 通常，在 ⟨*code*⟩ 中通过 key 机制来处理 ⟨*arguments*⟩, 也可以用其它方式处理。
例如在文件《tikz.code.tex》中对 canvas 坐标系统，xyz 坐标系统的定义是：

```
\tikzdeclarecoordinatesystem{canvas}
{%
  \tikzset{cs/.cd,x=0pt,y=0pt,#1}%
  \pgfpoint{\tikz@cs@x}{\tikz@cs@y}%
}%
%......
\tikzdeclarecoordinatesystem{xyz}
{%
  \tikzset{cs/.cd,x=0,y=0,z=0,#1}%
  \pgfpointxyz{\tikz@cs@x}{\tikz@cs@y}{\tikz@cs@z}%
}%
%......
\tikzset{cs/x/.store in=\tikz@cs@x}%
\tikzset{cs/y/.store in=\tikz@cs@y}%
\tikzset{cs/z/.store in=\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}, 参考命令 `\pgfpointxyz`→P. 640.

在程序库 perspective 的文件《tikzlibraryperspective.code.tex》中，用此命令定义了坐标系统 `three point perspective`.

`\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。

13.3 交点坐标

13.3.1 水平线与竖直线的交点：perpendicular 坐标系统

Coordinate system `perpendicular`

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⟩)
```

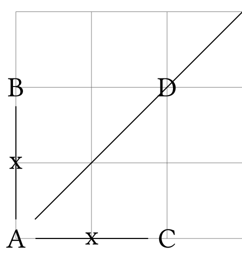
例如 `(2,1 |- 3,4)` 与 `(3,4 -| 2,1)` 表示同一个点，这里注意：

(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；

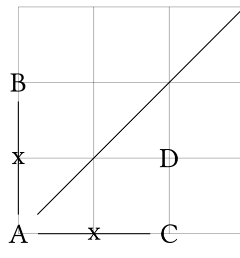
(c) ⟨*p*⟩ 或 ⟨*q*⟩ 可以是运算式，如果算式比较复杂或者算式中含有圆括号，那最好把整个算式用花括号括起来。

(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。但如果这个序列中有用花括号括起来的复杂算式，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。

13.3.2 任意路径的交点

TikZ Library `intersections`

```
\usetikzlibrary{intersections} % LaTeX and plain TeX
\usetikzlibrary[intersections] % ConTeXt
```

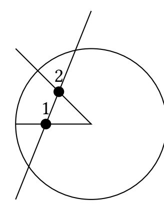
这个库能计算任意两个路径的交点。由于 TeX 的计算精度所限，所处理的路径不能太复杂。特别地，如果一个路径由很多小的线段构成（如装饰路径），那最好不要计算它与别的路径的交点。

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，路径名称所指的范围只有该路径本身，不包括添加到此路径上的 node。但是如果路径的名称是 ⟨*name*⟩，此路径上的 node 的名称也是 ⟨*name*⟩，那么 ⟨*name*⟩ 所指的就是“此路径”加上“此 node 的形状路径”，是二者的之总和（但是，看下面的测试）。




```
\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}
```

`/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。交点坐标前缀 `intersection` 可以修改，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。

`/tikz/intersection/name`=⟨*prefix*⟩ (no default, initially `intersection`)

这个选项指定交点名称的前缀。

`/tikz/intersection/total`=⟨*macro*⟩ (no default)

这个选项会创建宏 ⟨*macro*⟩，并把交点的总个数保存在 ⟨*macro*⟩ 中。

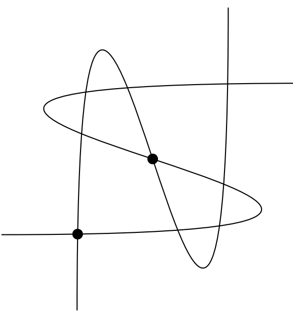
```
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1]
    (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw [name path=curve 2]
    (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);
  \fill [name intersections={of=curve 1 and curve 2, name=i, total=
↪   \t}]
    [red, 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)

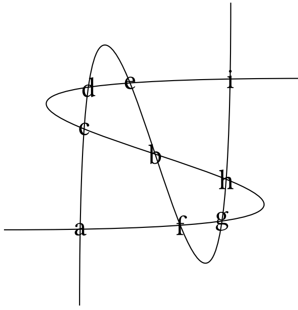
这个 key 的作用是给交点起“别名”，或者对交点执行某些操作。⟨*comma-separated list*⟩ 是一个用逗号分隔的列表，这个列表会被 `\foreach` 语句处理，所以列表的形式是 `\foreach` 语句所允许的形式。⟨*comma-separated list*⟩ 的格式是

[⟨*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）交点个数，其中第一个项目对应第一个交点，第二个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，使二者自动匹配。
- ⟨*name i*⟩ 是可选的，代表第 ⟨*i*⟩ 个交点的别名，也就是说第 ⟨*i*⟩ 个交点可以有两个名称，即（默认之下的）`intersection-`⟨*i*⟩ 和 ⟨*name i*⟩；
- [⟨*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。



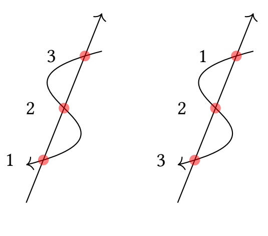
```
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1] (-2,-1) .. controls (8,-1) and
↪   (-8,1) .. (2,1);
  \draw [name path=curve 2] (-1,-2) .. controls (-1,8) and
↪   (1,-8) .. (1,2);
  \fill [name intersections={of=curve 1 and curve 2, by={a,b}}]
    (a) circle (2pt)
    (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}
```

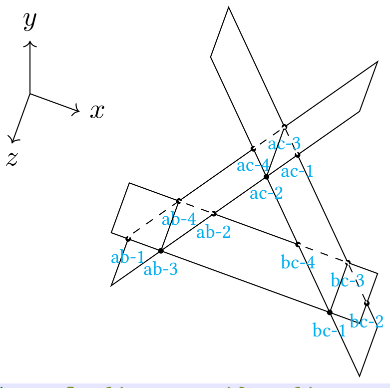
`/tikz/intersection/sort by`=⟨*path name*⟩ (no default)

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，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}
```

下面的例子用 3 维坐标创建路径，然后计算路径交点：




```
\tikz[z={(-110:1cm)},x={(-20:1cm)},scale=0.7]{
  \begin{scope}[shift={(-2,2)}]
    \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 [name path=a,]
    (0,0,0)coordinate(a1)--++(0,0,1)coordinate(a2)
    --++(5,5,0)coordinate(a3)--++(0,0,-1)coordinate(a4)--cycle;
  \draw [name path=b,] (0,1,0)coordinate(b1)--++(0,0,1)coordinate(b2)
    --++(5,0,0)coordinate(b3)--++(0,0,-1)coordinate(b4)--cycle;
  \draw [name path=c,] (2,5,0)coordinate(c1)--++(0,0,1)coordinate(c2)
    --++(3,-5,0)coordinate(c3)--++(0,0,-1)coordinate(c4)--cycle;
  {[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}};
    \fill [name intersections={of=a and c,name=ac,sort by=a,total=\t}]
      \foreach \i in {1,...,\t} {(ac-\i) circle(1.5pt) node[below]{ac-\i}};
    \fill [name intersections={of=b and c,name=bc,sort by=b,total=\t}]
      \foreach \i in {1,...,\t} {(bc-\i) circle(1.5pt) node[below]{bc-\i}};
  }

  \draw (ab-3)--(ab-4) (ac-2)--(ac-3) (bc-1)--(bc-3);
```

```
  \draw [dashed,draw=white,line width=0.6pt]
    (ab-1)--(a1)--(ab-4)--(ab-2) (bc-4)--(bc-3)--(bc-2) (ac-4)--(ac-3)--(ac-1);
}
```

上面图形表明，在计算交点时，用 3 维坐标创建的路径其实仍然是 2 维平面的路径。

13.4 相对坐标，增量坐标

13.4.1 指定相对坐标

在坐标前面加前缀“++”或“+”，例如 `++(1,0)`，就成为相对坐标（relative coordinates）。“相对”有“参考、参照”的意思，“++”和“+”都是相对于当前的参照点来进行操作。与此相关联的概念是“当前点”，比如，用画笔画曲线，笔尖所在的位置点就是当前点。参照点是画笔移动时的参照，当前点是画笔的落笔点。“++”和“+”实际上都是确定当前点和当前参照点的机制。

首先，不带前缀“++”或“+”的普通坐标点总会成为当前点、当前参照点。

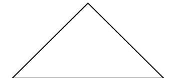
对于 `++(B)` 来说，`++` 有两个作用：第一，让 `(A)+(B)` 成为当前点（其中 `(A)` 为当前参照点）；第二，让 `(A)+(B)` 成为当前参照点。

```
\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

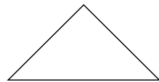
```
\tikz \draw (0,0) --  (1,0) --  (1,1) --  (0,1) -- cycle;
```

对于 `+(B)` 来说，`+` 有两个作用：第一，让 `(A)+(B)` 成为当前点（其中 `(A)` 为当前参照点）；第二，保持 `(A)` 为当前参照点。



```
\tikz \draw (0,0) -- +(1,0) -- +(0,1) -- +(-1,0) -- cycle;
```

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)`，`+(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}
```

13.4.2 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。

13.4.3 相对坐标与当前点的局部化

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，并不能将当前点也限制在内。当前点属于路径，而选项不属于路径，选项只是影响了路径的表现形式。下面的选项可以将当前点临时（局部地）局部化。

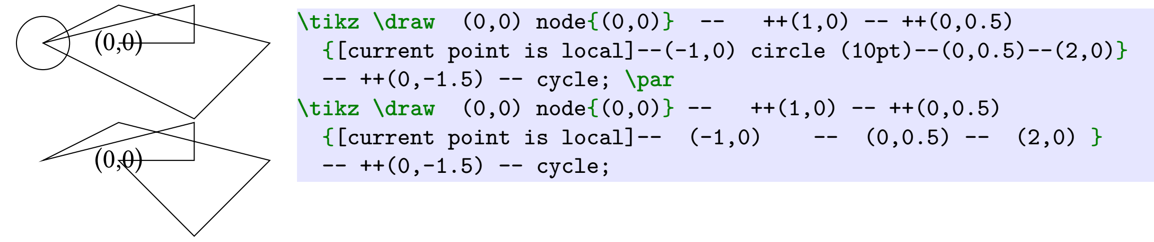
`/tikz/current point is local`=⟨*boolean*⟩ (no default, initially `false`)

当本选项的值是 `false` 时，路径内的 scope 对当前点没有影响。在路径内部创建一个 scop，假设这个 scop 开始时（读取开花括开花括号 {时）的当前点是 p；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），检查本选项的值是否为 true，如果是就把当前点设置为 p。

```
\tikz \draw (0,0) -- ++(1,0) -- ++(0,0.5)
    -- ++(-1,0) -- ++(0,0.5) -- ++(2,0)
    -- ++(0,-1.5) -- cycle; \par
\tikz \draw[green] (6,0) --   ++(1,0) -- ++(0,0.5)
    {[current point is local]-- ++(-1,0) -- ++(0,0.5) -- ++(2,0) }
    -- ++(0,-1.5) -- cycle;
```

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; \par
\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;
```

13.5 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。

13.5.1 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。

13.5.2 数乘坐标（向量）

⟨*factor*⟩*⟨*coordinate*⟩

当按照这个格式写出一串符号后，需要判断这串符号的哪一部分属于 ⟨*factor*⟩，为了避免无法判断的错误，注意以下几点：

- ⟨*factor*⟩* 这一部分是可选的，如果没有 ⟨*factor*⟩ 那就不要有 *。
- 如果 tikz 认为你提供了 ⟨*factor*⟩，那么在第一个符号组合 *(之前的那些符号就是 ⟨*factor*⟩。因此，如果你提供的 ⟨*factor*⟩ 中含有符号组合 *(，那就应当用花括号把整个 ⟨*factor*⟩ 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 ⟨*factor*⟩ 之后、⟨*coordinate*⟩ 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 ⟨*factor*⟩ 内部的 * 两侧可以有空格。

- 提供 ⟨*factor*⟩ 后，宏 `\pgfmathparse` 会解析 ⟨*factor*⟩，所以 ⟨*factor*⟩ 可以是复杂的算式，但要注意使用花括号包裹 ⟨*factor*⟩，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 ⟨*factor*⟩，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);
```

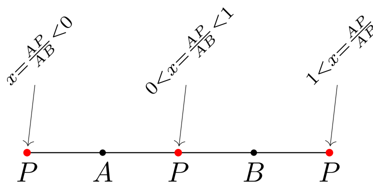
13.5.3 比例—角度定点句法

⟨*coordinate*⟩`!`⟨*number*⟩`!`⟨*angle*⟩`:`⟨*second coordinate*⟩

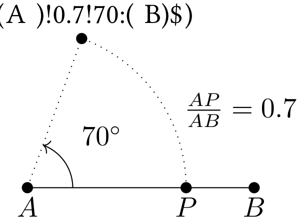
其中 ⟨*number*⟩ 是任意实数, ⟨*angle*⟩ 是角度制下的数值, ⟨*angle*⟩: 是可选的。⟨*number*⟩ 会被 `\pgfmathparse` 解析，所以 ⟨*number*⟩ 可以是比较复杂的算式。

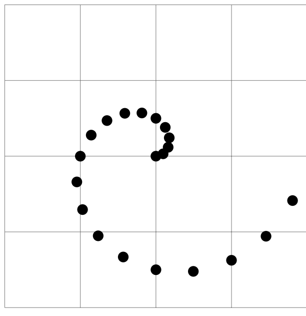
例如 `($(A)!x!(B)$)`，等价于 `($ {(1-x)}*(A)+x*(B)$)`. 记 `($(A)!x!(B)$)` 为 P，则 x 与 P 的对应关系如下图所示

$P =$`($(A)!x!(B)$) = ($ {(1-x)}*(A)+x*(B)$)`



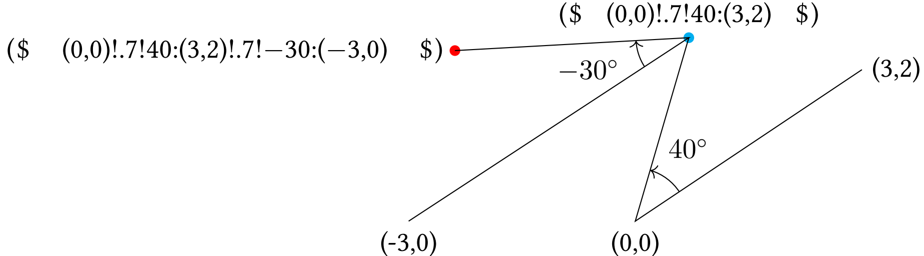
记 `($(A)!x!y:(B)$)` 为 Q，Q 这样得到：先计算 `($(A)!x!(B)$)`（记为 P），然后将线段 AP 绕点 A 旋转角度 y，点 P 变成 Q. 例如，`($(A)!0.7!70:(B)$)` 如下图所示





```
\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] {\lstinline&($(0,0)!.7!40:(3,2)$)&}
     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] {\lstinline&($(0,0)!.7!40:(3,2)!.7!-30:(-3,0)$)&}
     pic [draw, <-,  "$-30^\circ$", angle radius=7mm, angle eccentricity=2] {angle =c--a--d};
\end{tikzpicture}
```

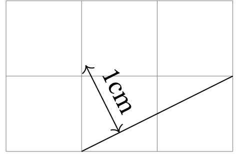
13.5.4 距离—角度定点句法

⟨*coordinate*⟩!⟨*dimension*⟩!⟨*angle*⟩:⟨*second coordinate*⟩

其中 ⟨*dimension*⟩ 是带单位的长度，可以是负值。⟨*angle*⟩：是可选的。

`(A)!`⟨*dimension*⟩`!(B)` 是以点 `(A)` 为起点，沿着向量 $\overrightarrow{(A)(B)}$ 的方向，移动 ⟨*dimension*⟩ 确定的点，⟨*dimension*⟩ 可以是负值尺寸。

`(A)!`⟨*dimension*⟩`!`⟨*angle*⟩`:(B)` 是将 `(A)!`⟨*dimension*⟩`!(B)` 绕点 `(A)` 旋转角度 ⟨*angle*⟩ 确定的点。



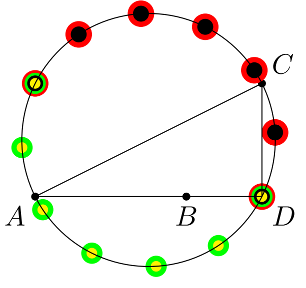

```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \coordinate (a) at (1,0);
  \coordinate (b) at (3,1);
  \draw (a) -- (b);
  \coordinate (c) at ($ (a)!.25!(b) $);
  \coordinate (d) at ($ (c)!1cm!90:(b) $);
  \draw [<->] (c) -- (d) node [sloped,midway,above] {1cm};
\end{tikzpicture}
```

13.5.5 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。

`(A)!(C)!`⟨*angle*⟩`:(B)` 的意思不是很明显，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}
```

上面图形中的圆以 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。

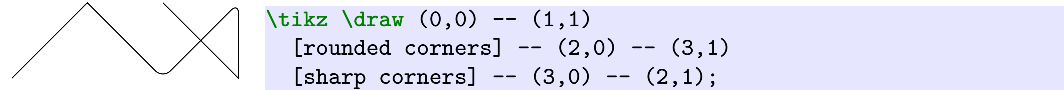
14 设置路径的语句

`\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。选项的发挥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。



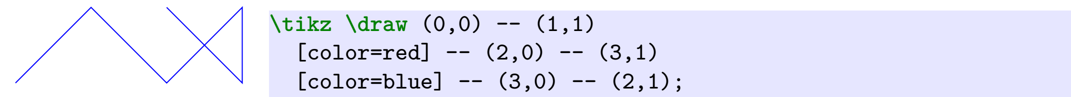
```
\tikz \draw (0,0) -- (1,1)
  [rounded corners] -- (2,0) -- (3,1)
  [sharp corners] -- (3,0) -- (2,1);
```

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。



```
\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. 39 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：

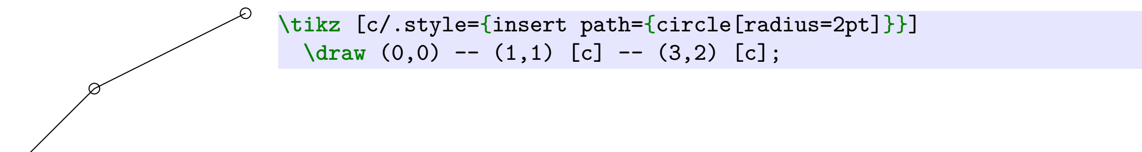
`/tikz/every path` (style, initially empty)

整个 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*⟩ (no default)

这里的 ⟨*path*⟩ 也是那些可以用在“\path”之后的、构成（描绘）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*⟩。这个选项可以用作 node 的选项。如果多次使用这个选项，那么它们会依次产生作用。

foo

```
\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。

`/tikz/prefix after command`=⟨*path*⟩ (no default)

与上一选项类似，只是 ⟨*path*⟩ 被插入到路径的开始处。如果多次使用这个选项，那么它们都有效，但它们的作用次序是：后给出选项先作用，先给出的选项后作用。

foo

```
\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};
```

上面例子中，先给出的红色 node 遮挡了后给出的绿色 node。

14.1 Move-To 操作

`\path`...⟨*coordinate*⟩...

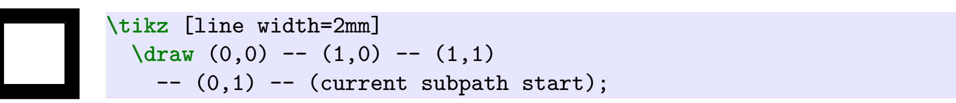
move-to 操作（operation）通常出现在两种地方，例如

```
\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”），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，它不是由一个子路径（它不是 move-to 操作的“落脚点”），但是属于当前路径，包含在当前路径的边界盒子中。

(current subpath start)

这是个预定义的坐标点，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”。

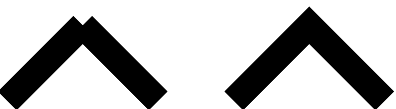
```
\tikz [line width=2mm]
  \draw (0,0) -- (1,0) -- (1,1)
    -- (0,1) -- (current subpath start);
```

14.2 Line-To 操作

14.2.1 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。因为在计算图形的边界盒子时并不考虑线条的线宽，所以若线宽较大就会使得线条明显突出图形边界盒子之外，此时要适当扩展图形的边界盒子来容纳线宽。所以上例中使用命令 \useasboundingbox 把点 (0,1.5) 纳入图形的边界盒子，从而扩展图形的边界盒子。

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。

14.2.2 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}
```

14.3 Curve-To 操作

curve-to 操作创建 Bézier 曲线。

\path⟨···⟩..controls⟨*c*⟩and⟨*d*⟩..⟨*y or cycle*⟩⟨···⟩;

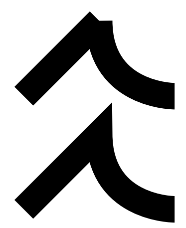
这个操作以当前点为起点创建一段 3 次 Bézier 曲线来延伸当前路径。假设当前点是 ⟨*x*⟩，则 ⟨*x*⟩ 是起点，⟨*c*⟩ 是第二点，⟨*d*⟩ 是第三点，⟨*y*⟩ 是终点。

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}
```

14.4 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。

14.5 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=5pt] -- 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，此时使用尖角会比较好。

/tikz/sharp corners (no value)

本选项关闭圆角功能，使用尖角。

14.6 创建圆、椭圆

\path...circle[⟨*options*⟩]...;

这个命令在路径上添加一个圆，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。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。

/tikz/y radius=⟨*value*⟩ (no default)

类似 x radius。

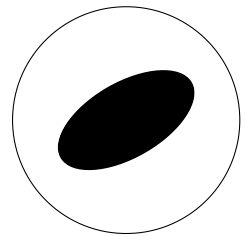
/tikz/radius=⟨*value*⟩ (no default)

将 x radius 和 y radius 都设为 ⟨*value*⟩。

/tikz/at=⟨*coordinate*⟩ (no default)

本选项可以设置圆的圆心位置。

在 ⟨*options*⟩ 中也可以使用 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}
```

/tikz/every circle (style, no value)

这个 style 针对每个圆。

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。

指定半径的旧句法

注意，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。

`\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。

14.7 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，其中起始角度、终止角度、角度差这三个参数是相关的。这些参数对应下面的选项。

`/tikz/start angle=`⟨*degrees*⟩ (no default)

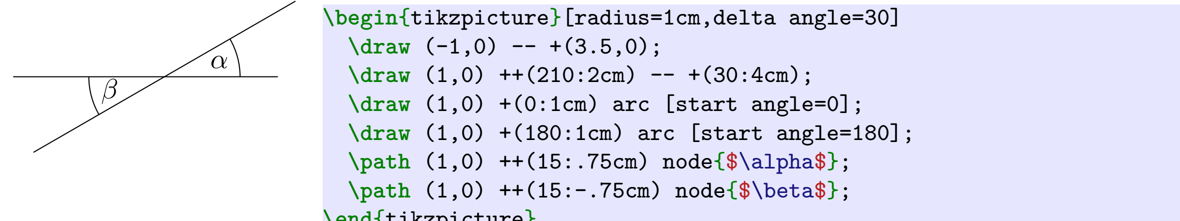
设置起始角度。

`/tikz/end angle=`⟨*degrees*⟩ (no default)

设置终止角度。

`/tikz/delta angle=`⟨*degrees*⟩ (no default)

设置角度差。




```
\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*⟩`)`

14.8 Grid 操作

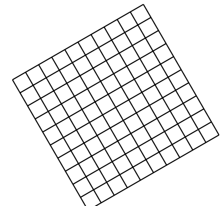
`\path...grid[⟨options⟩]⟨corner or cycle⟩...;`

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)

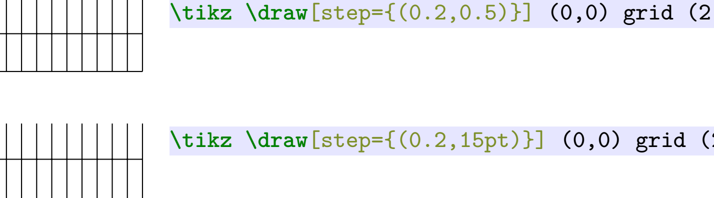
设置网格在 x 轴方向和 y 轴方向的步长。

- 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。



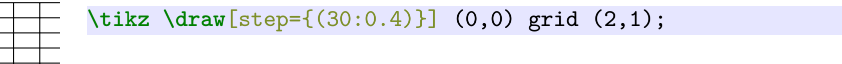
```
\tikz[rotate=30] \draw[step=0.2] (0,0) grid (2,2);
```

- 如果本选项的值是坐标 (a,b)，则网格在 x 轴方向的步长是 a，而在 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);
```



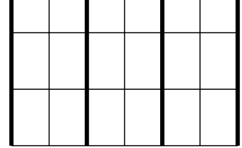
```
\tikz \draw[step={(30:0.4)}] (0,0) grid (2,1);
```

- 如果某个轴向的步长是 0 或者是负值，则该轴上没有网格线。

```
\tikz \draw[step={(0,0.5)}] (0,0) grid (2,1);
```

/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}
```

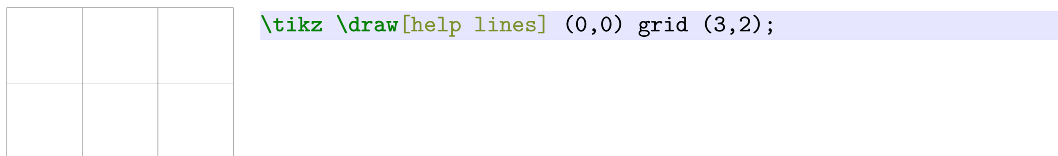
/tikz/ystep=⟨*dimension or number*⟩ (no default, initially 1cm)

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);
```

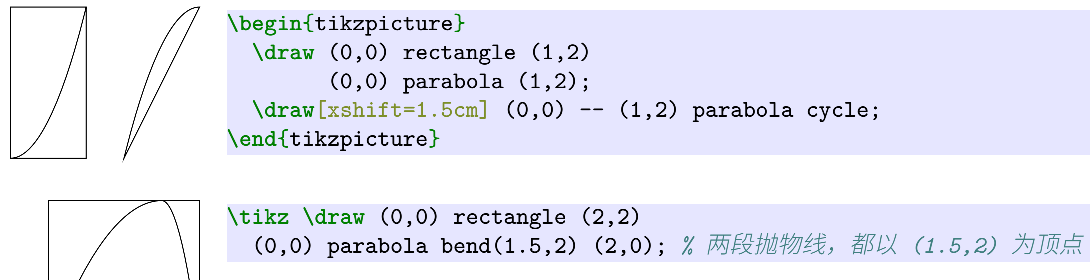
14.9 Parabola 操作

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（或两段首位相接的抛物线），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，可以创建两段首位相接的抛物线，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)。因为 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，先在线段 ⟨*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。

`/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)

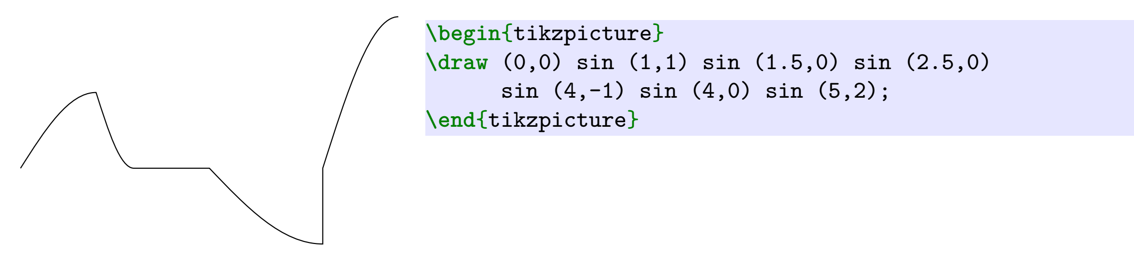
等效于选项 `bend pos=1,bend={+(0,0)}`.

14.10 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}
```

14.11 SVG 操作

14.12 Plot 操作

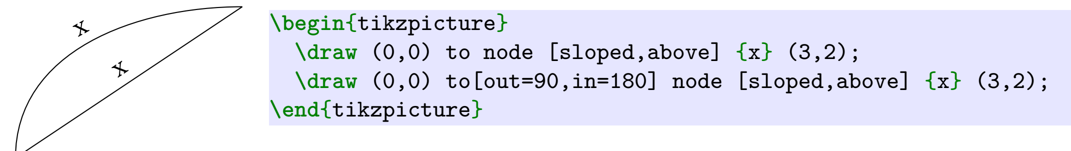
14.13 To Path 操作

\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。

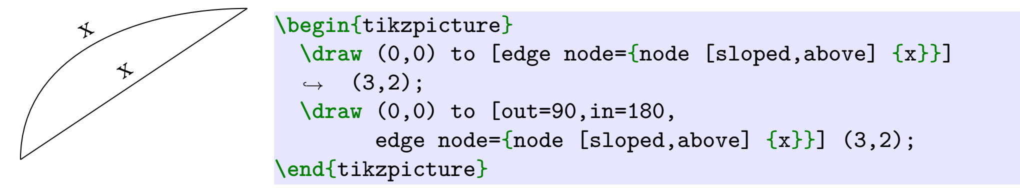



```
\begin{tikzpicture}
  \draw (0,0) to node [sloped,above] {x} (3,2);
  \draw (0,0) to[out=90,in=180] node [sloped,above] {x} (3,2);
\end{tikzpicture}
```

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。



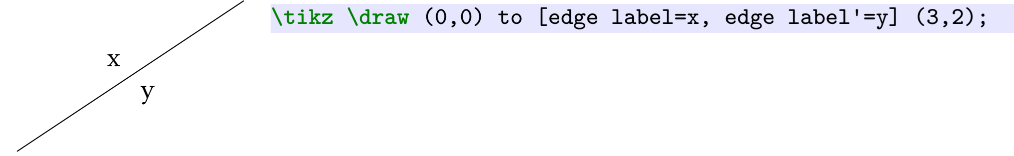

```
\begin{tikzpicture}
  \draw (0,0) to [edge node={node [sloped,above] {x}}]
  ↪  (3,2);
  \draw (0,0) to [out=90,in=180,
        edge node={node [sloped,above] {x}}] (3,2);
\end{tikzpicture}
```

/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。

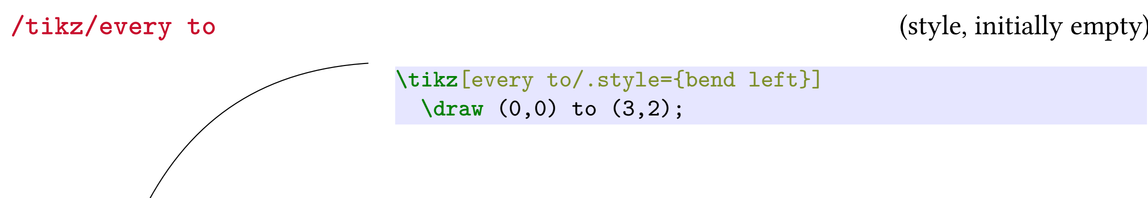



```
\tikz \draw (0,0) to [edge label=x, edge label'=y] (3,2);
```

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);
```

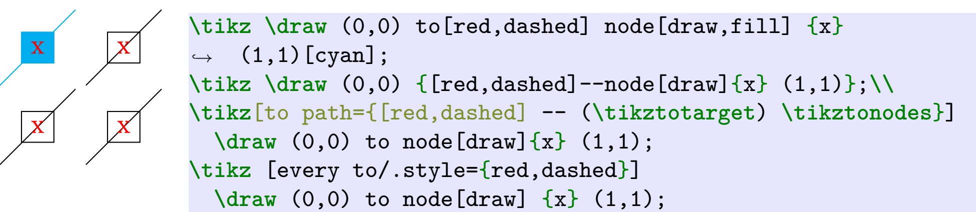
选项 选项 `to path` 用于定义 `to` 操作所画的路径。在默认下，`to` 路径被定义为线段，可以改成曲线。

`/tikz/to path=`⟨*path*⟩ (no default)

`to` 操作所创建的路径就是 ⟨*path*⟩，实际上，`to` 操作会在路径上插入一个 scope：

`{[every to,`⟨*options*⟩`]` ⟨*path*⟩`}`

其中的 ⟨*options*⟩ 是针对 `to` 操作的选项，⟨*path*⟩ 是本选项设置的内容。这个 scope 内的选项一般只有局部的作用。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);
```

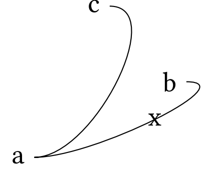
上面例子中，处于 `to` 操作辖制范围内的选项 `red` 只对 node 中的文字有效，对 node 的背景线颜色、填充色，对 scope 内的子路径颜色都无效；node 的背景线颜色、填充色、scope 内的子路径颜色使用的是“主路径”的设置（被主路径的颜色设置覆盖了）。而选项 `dashed` 则没有任何作用。

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，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，tikz 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*⟩ 会在执行 to 操作前被执行，可以用来添加其它路径或者做某些计算。

`/tikz/execute at end to`=⟨*code*⟩ (no default)

代码 ⟨*code*⟩ 会在执行 to 操作后被执行。

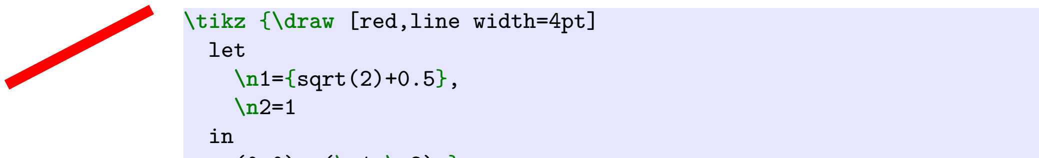
```
\tikz[every to/.style={draw,dashed}]
  \draw (0,0) to (2,0);\\
\tikz[every to/.style={append after command={[draw,dashed]}}]
  \draw (0,0) to (2,0);
```

```
\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);
```

14.14 Foreach 操作

14.15 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。

`\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程序自动完成。

以上宏的值存储在专门的寄存器（register）中，这是专属于 Tikz 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。

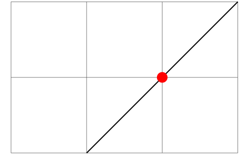
\p⟨*point register*⟩={⟨*formula*⟩}
\p⟨*point register*⟩

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) 展开为坐标形式 (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。

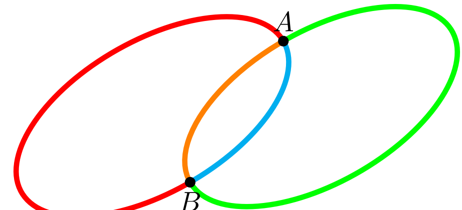
以上各个宏只能用在 let 操作中，但是 let 操作内所规定的 coordinate 在整个 {tikzpicture} 环境内有效，如下例：



```
\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}
```

14.16 Scoping 操作

14.17 Node and Edge 操作

14.18 Graph 操作

14.19 Pic 操作

14.20 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）。

14.21 PGF-Extra 操作

每个绘图命令，例如 \draw，都有自己的使用句法，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);
```

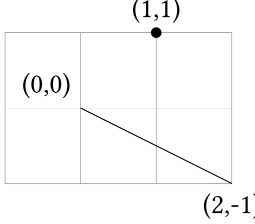
在路径中直接使用宏 \def 来做定义，也会导致上面的错误信息。不过在路径中的适当位置插入 \foreach 命令则是可以接受的：

```
\tikz
  \draw [red,very thick]
    \foreach \i in {0,1,2}
      \foreach \j in {0,1,2}
      {(\j,\i)--(\i,\j)};
```

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，⟨*code*⟩ 的有效范围也限于路径之内。




```
\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`

如果 `\pgfextra` 之后的 ⟨*code*⟩ 没有用花括号括起来，就需要用 `\endpgfextra` 结束 `\pgfextra` 的作用。

14.22 在 Ti*k*Z 中使用软路径

关于软路径可参考 §121.

`/tikz/save path`=⟨*macro*⟩ (no default)

将当前的软路径保存在宏 ⟨*macro*⟩ 中。

`/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。

```
\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}
```

15 Actions on Paths

15.1 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。

15.2 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。选项 draw=⟨*color name*⟩, fill=⟨*color name*⟩ 比本选项更优先。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*⟩.

15.3 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。

15.3.1 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，但头是方的），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}
```

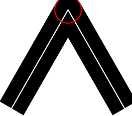
可见帽子会凸出路径端点之外，`round`（圆形）帽子的直径是线宽，`rect`（方形）帽子的凸出长度是半个线宽。（参考 PDF Reference 的 Line cap styles.）

`/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）.

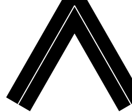
```
\begin{tikzpicture}[line width=10pt]
  \draw[line join=round] (0,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[line join=bevel] (1.25,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[line join=miter] (2.5,0) -- ++(.5,1) -- ++(.5,-1);
  \useasboundingbox (0,1.5);
\end{tikzpicture}
```

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:



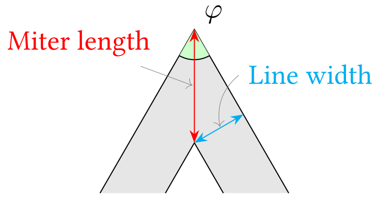
```
\begin{tikzpicture}
  \draw [line join=bevel,line width=5pt,double distance=0.4pt]
    (0,0)--++(60:1.5)--++(-60:1.5);
\end{tikzpicture}
```

/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`。

```
\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}
```

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.



15.3.2 Dash Pattern

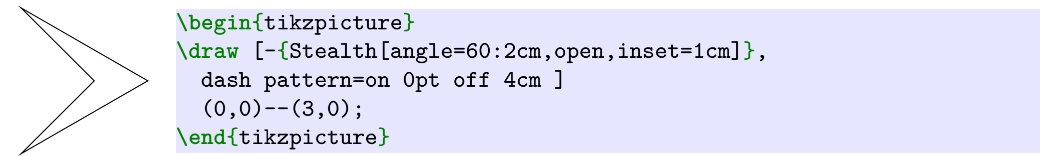
/tikz/dash pattern=⟨*dash pattern*⟩ (no default)

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曲线，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。

/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。

15.3.3 线条透明度

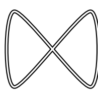
参考选项 `/tikz/draw opacity`→ P. 187.

15.3.4 Double Lines and Bordered Lines

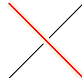
`/tikz/double=`⟨*core color*⟩ (default `white`)

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~
```

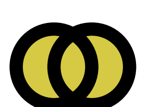
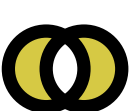
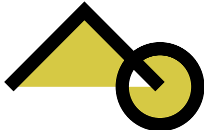
15.4 在路径上添加箭头

15.5 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)

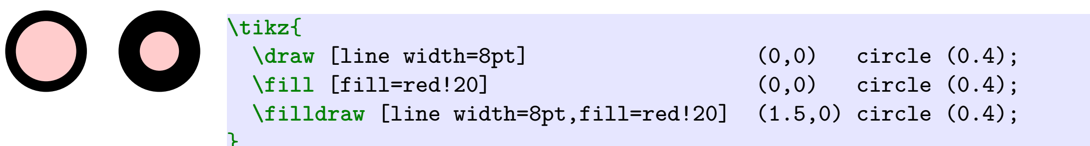
这个选项使得路径被填充。填充色或者是 ⟨*color*⟩，或者是用 `color=` 设置的颜色。如果 `fill=none`，则取消填充。

```
\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。这是因为线宽是“图形状态参数”，并不属于路径，填充路径的行为并不考虑线宽，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);
}
```

15.5.1 图形参数：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，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。

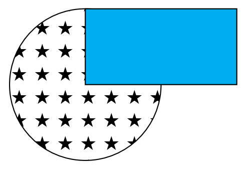
Tiling patterns 分两种：不变色的（inherently colored patterns）和可变色的（form-only patterns）。pattern 缺少可变性，它不会因放缩、旋转变换而变化。

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。

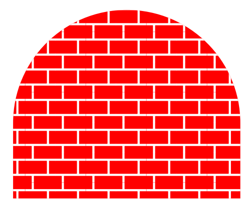
pattern 和 fill 的功能类似，当这两个选项并列时，无论哪个在前，后面的总会抑制前面的。



```
\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}
```

/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}
```

15.5.2 图形参数：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。

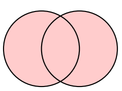
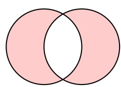
由操作 rectangle，circle 创建的矩形和圆的方向都是逆时针方向。

/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);
```

15.5.3 图形参数：填充透明度

参考选项 /tikz/fill opacity →P. 187.

15.6 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图形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*⟩；然后结束局部 scope；然后画出路径（如果路径带有 `draw` 选项的话）。

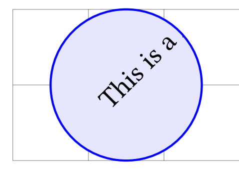
如 `fill`，`draw` 选项，该选项用在路径之内才有效。每当新路径开始时，path picture 都会被清空。⟨*code*⟩ 可以是 Tikz 的绘图命令，如由 `\draw`，`\node` 等；也可以插入外部图像。如果要插入外部图形，必须把 graphicx 宏包的 `\includegraphics` 命令放在 `\node` 命令中。

Predefined node `path picture bounding box`

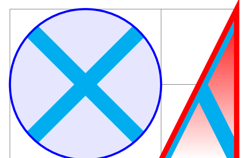
每个路径都有自己的边界盒子（bounding box），在默认下这个边界盒子是恰好装下该路径的矩形盒子，并且每当向路径中添加新的坐标点（路径被延伸）时，路径的边界盒子都会被刷新。也就是说，在路径创建过程的不同位置（时间点）上有不同的边界盒子，PGF 能实时地跟踪路径的边界盒子。当创建一个路径时，PGF 会自动生成一个名称为 `current path bounding box` 的矩形 node，这个 node 的尺寸、位置与该路径的、当前时间点的 bounding box 相同。也就是说，在路径创建过程的不同位置上有不同的 `current path bounding box`。作为一个 node，`current path bounding box` 有自己的 node 坐标系统。

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。

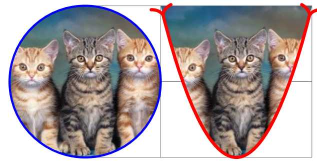
当路径带有 `path picture=`⟨*code*⟩ 选项时，一个名称为 `path picture bounding box` 的 node 会被自动创建，它的尺寸和位置与 (执行 ⟨*code*⟩ 之前的) `current path bounding box` 一样。也就是说，在执行 ⟨*code*⟩ 之前就已经创建了 `path picture bounding box`，所以在 ⟨*code*⟩ 中可以引用 `path picture bounding box` 的坐标系统中位置点。




```
\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}
```

15.7 用颜色渐变填充路径

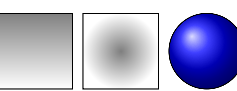
`/tikz/shade` (no value)

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`，`fill` 选项，则 `fill` 无效。

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. 788.



```
\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。

15.8 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. 131
- `/tikz/use as bounding box`
- `\useasboundingbox`→ P. 77
- `\pgfresetboundingbox`→ P. 658
- `\pgf@relevantforpicturesizefalse`→ P. 659
- `\pgf@relevantforpicturesizetrue`→ P. 660

`/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计入图形的 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.

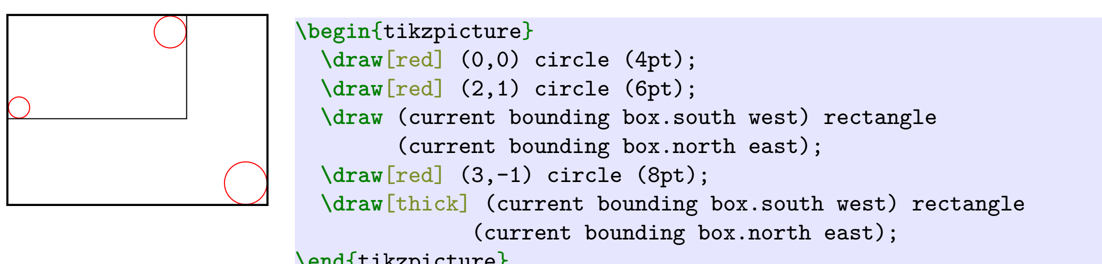
```
Left of picture
\begin{tikzpicture}
  \draw [red] (3,0.5) circle (0.5cm);
  {\draw[use as bounding box] (2,0) rectangle (3,1);}
  \draw (1,0) -- (4,.75);
\end{tikzpicture}
right of picture.
```

`\useasboundingbox`

这个命令是 `\path[use as bounding box]` 的缩写，注意其中没有 `draw` 选项。

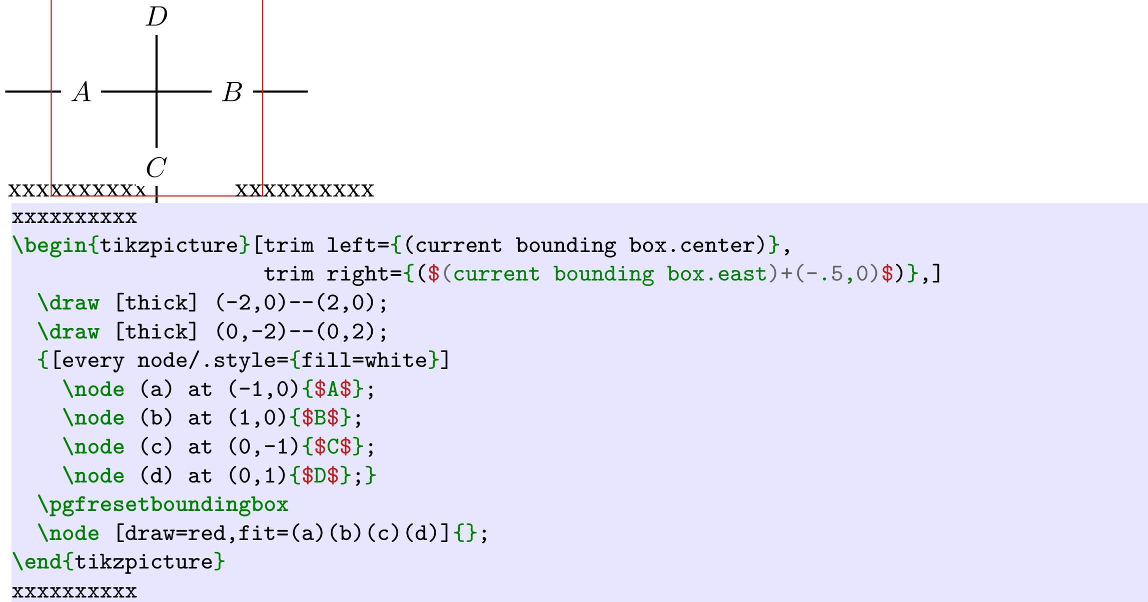
Predefined node `current bounding box`

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
```

`/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 TikZ 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}
```

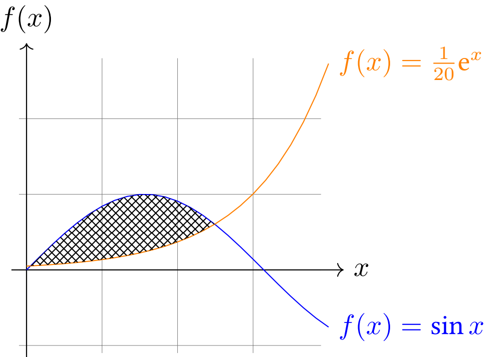
```
\begin{tikzpicture}
  \node [draw, clip, text height=0pt, text depth=0pt, text width=0pt,
          inner sep=0pt, minimum size=1cm]
    {\tikz[baseline=0cm, trim left={(current bounding box.center)}]
     \draw [red] (-1,-1)--(1,1);};
\end{tikzpicture}
```

15.9 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。

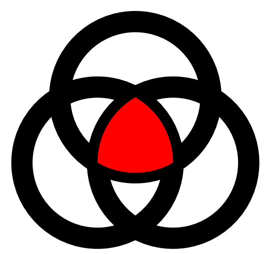



```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,2.8);
  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,3) node[above] {$f(x)$};
  \draw[color=blue] plot (\x,{sin(\x r)}) node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot (\x,{0.05*exp(\x)})
    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。

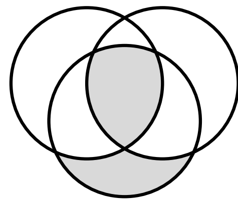


```
\begin{tikzpicture}[line width=8pt]
  \draw (0,0)circle(1cm) (1,0)circle(1cm)
        (60:1)circle(1cm);
  \path[clip] (0,0) circle (1cm);
  \path[clip] (1,0) circle (1cm);
  \fill[red] (60:1)circle(1cm);
\end{tikzpicture}
```

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);
}
```

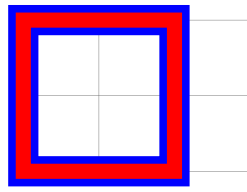
上面例子中，用一个 scope 环境限制剪切范围；为了避免填充色覆盖半个线宽，故在最后画出 3 个圆。

15.10 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。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。

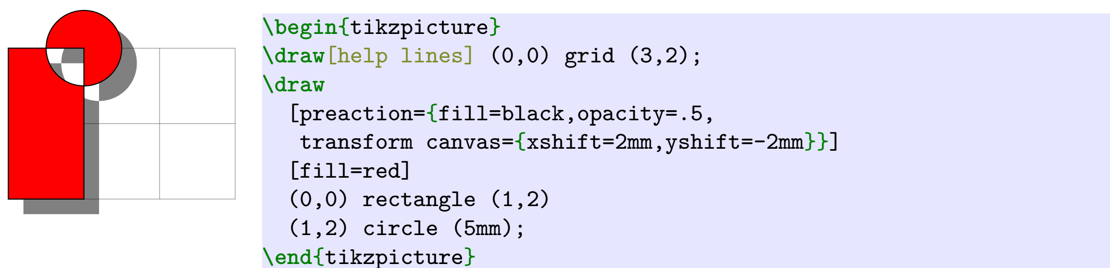


```
\begin{tikzpicture}
\draw[help lines] (0,0) grid (3,2);
\draw
  [preaction={draw,line width=4mm,blue}]
  [line width=2mm,red]
  (0,0) rectangle (2,2);
\end{tikzpicture}
```

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。

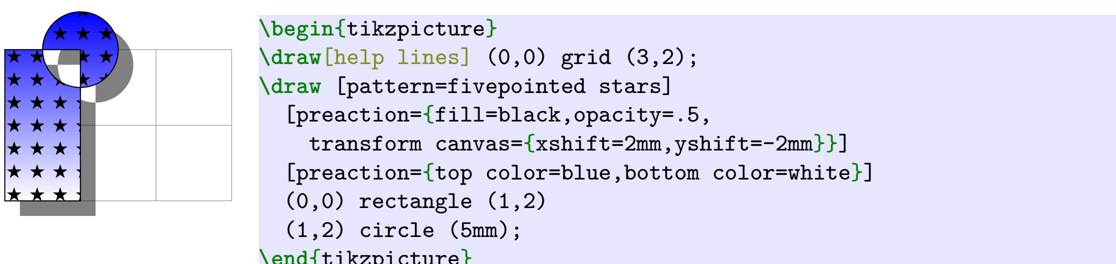
但 `preaction` 指定的 ⟨*options*⟩ 中可以有画布变换选项。



```
\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，程序会参照每个 `preaction` 选项的 ⟨*options*⟩ 将路径画一次，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}
```

`/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。

15.11 装饰路径

参考 `/pgf/decoration` → P. 202.

15.12 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 tikz 选项:

```
\tikzoption{shorten <}{\pgfsetshortenstart{#1}}
\tikzoption{shorten >}{\pgfsetshortenend{#1}}
```

其中的命令 `\pgfsetshortenstart` → P. 686 能够在路径的开头处把路径裁掉一段; 命令 `\pgfsetshortenend` → P. 686 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}
```

16 Arrows

16.1 Overview

库 `arrows.meta` 提供 3.0 版的 TikZ 的新箭头，库 `arrows`，`arrows.spaced` 提供旧的箭头。

TikZ 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. 689.

本节所介绍的箭头特性都属于 TikZ 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。

16.2 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` 结尾的路径加箭头。

`/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：

- 没有设置箭头算子，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);
```

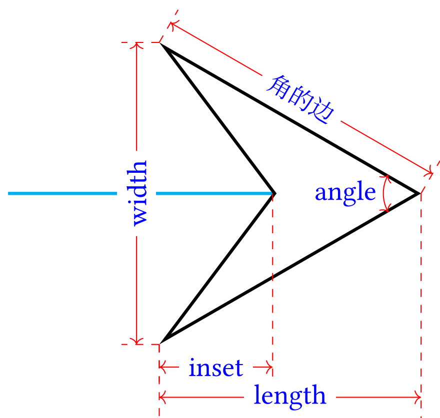
箭头的初始指向都是指向右侧的，箭头尖端的最末一个像素的位置是路径的端点。箭头不接受 `rotate` 选项。当箭头添加到路径上时，程序会自动将箭头绕着箭头尖端旋转，使得箭头的指示方向与路径（在端点附近的）方向一致。程序在确定箭头旋转状态时要参考箭头轮廓线与路径的公共点（见 §16.3.8），如果路径长度太短造成公共点缺失就导致程序计算失败，结果可能出人意料。所以被添加箭头的路径的长度与箭头尺寸要匹配，最好不要太短。

16.3 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]`.

16.3.1 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：

10mm

```
\tikz{
  \draw [line width=1mm, -{Stealth[length=10mm, open]}]
    (0,0) -- (2,0);
  \draw [|<->|] (2,.6) -- node[above=1mm] {10mm} ++(-10mm,0);
  }
```

10mm

```
\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。

16.3.2 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。

16.3.3 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);
}
```

16.3.4 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);
  }
```

16.3.5 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。

16.3.6 箭头颜色

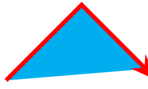
箭头是个路径，例如 Stealth 这种箭头路径，它有自己的路径线条颜色、路径内部的填充色，这两种颜色通常与箭头所在路径的 `draw` 颜色一致。

```
\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。

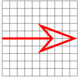


```
\tikz [ultra thick]
  \draw [draw=red, fill=cyan,
    arrows = {-Stealth[length=10pt]}]
    (0,0) -- (1,1) -- (2,0);
```

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)

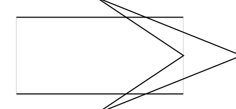
这个选项指定箭头内部的填充色。fill=none 表示不填充颜色（注意这不同于填充白色）。



```
\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);
```

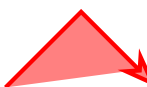
上面例子中，路径是双线，由于 inset 过深且双线间距过大，造成双线突出箭头外部。

/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);
```

16.3.7 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。

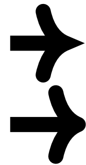
/pgf/arrow keys/line cap=⟨*round or butt*⟩ (no default)

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)

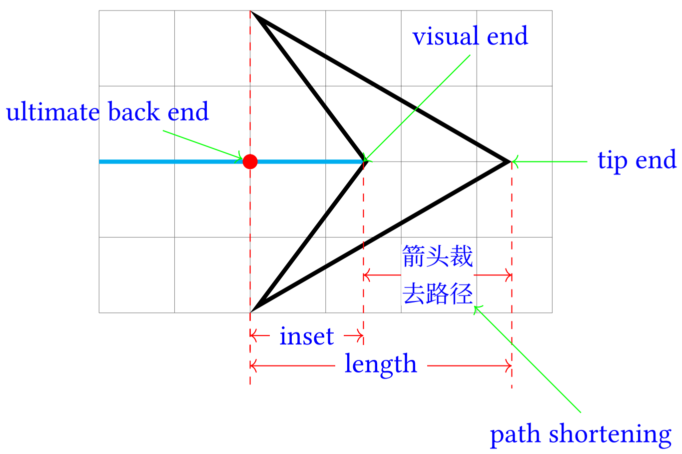
本选项设置箭头路径的线宽，各个参数的意义与选项 /pgf/arrow keys/length→P. 84 的相同。如果设置本选项为 0pt，那么对于封闭的箭头路径来说，就只能填充箭头内部。当箭头带有 bend 选项时，设置箭头线宽为 0pt 可能会比较好。

/pgf/arrow keys/line width'=⟨*dimension*⟩⟨*length factor*⟩ (no default)

本选项设置箭头路径的线宽。

16.3.8 Bending and Flexing

首先规定几个名词，参照下图



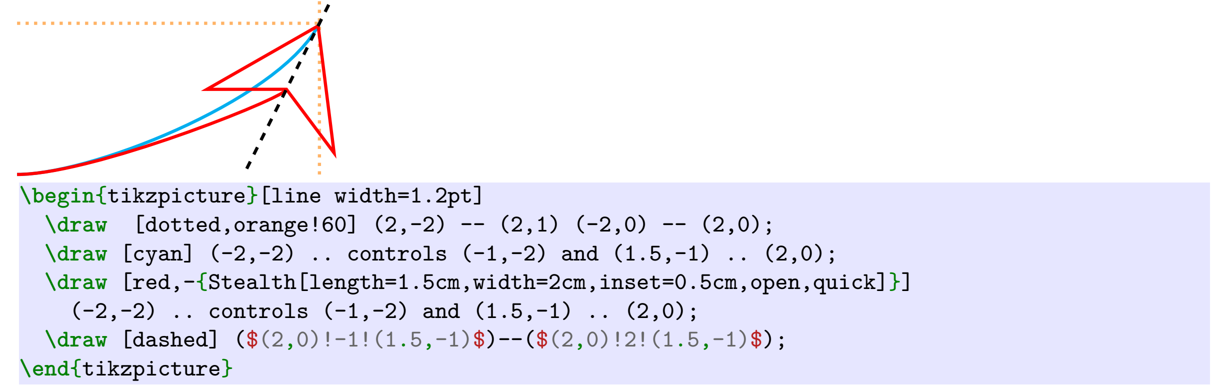
上面图中，首先画出网格，然后画出路径和箭头，箭头使用了 open 选项。箭头没有遮挡网格，却把处于箭头内部的那一段路径裁去了，被裁去的路径长度是 length - inset，这种裁去一段路径的行为称为 path shortening，即“裁线”。箭头最前端的那个像素是原来的路径端点，称之为 tip end. 路径与箭头轮廓线的公共点（图中是箭头内凹的顶点）称作 visual end.

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)

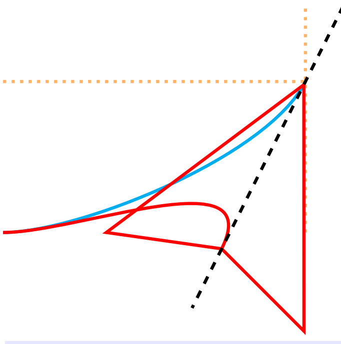
下面以一段控制曲线为例来说明这个选项的作用。为了明显，箭头的尺寸设置得比较大。



```
\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。

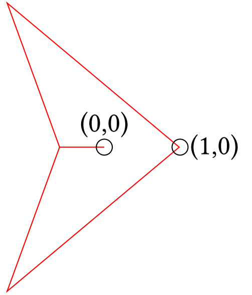
将上面例子中箭头的尺寸修改一下，得到下面比较极端的例子：



```
\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，这个库提供选项 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）。

`/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。

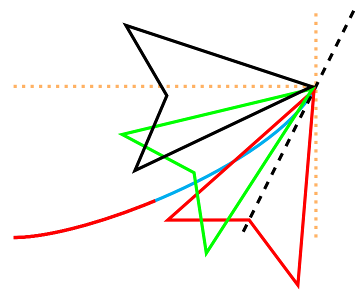
下面例子是 flex 取负值的情况：

```
\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);
```

16.4 Arrow Tip Specifications

16.4.1 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，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。

16.4.2 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。

16.4.3 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。

16.4.4 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*⟩`/.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 [foo /.tip = {Stealth[sep] Latex[sep]},
    bar /.tip = {Stealth[length=10pt,open]}]
  \draw [{foo[red] . bar}-{foo[red] . bar}] (0,0) -- (2,0);
```

上面例子中，`foo[red]` 的效果等效于 `Stealth[sep,red] Latex[sep,red]`，也就是说当 `foo[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 [foo /.tip = {Stealth[sep] Latex[sep]},
       bar /.tip = {foo[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。

`/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);
```

16.4.5 Scoping of Arrow Keys

`/tikz/arrows`=⟨*arrow keys*⟩ (no default)

当某个环境带有这个选项后，如果 ⟨*arrow keys*⟩ 是给路径添加箭头的选项（要注意选项的句法格式），那么环境内的任何一个路径命令都会带上选项 ⟨*arrow keys*⟩，例如：

```
\tikz[arrows=Bar-{Stealth[sep=10pt]Bar}]{
  \draw (0,0) to [bend left] (1,1);
  \draw (0,-.5) -- ++(1,0);
}
```

如果 ⟨*arrow keys*⟩ 是设置箭头外观、间隔等的选项，这些选项要用方括号括起来，而方括号之外还要用花括号括起来，这会使得环境内的所有路径命令中的任何一个箭头都会带上 ⟨*arrow keys*⟩，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);
}
```

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。
2. `arrows=[`⟨*keys*⟩`]` 中的选项设置。
3. 由 `/.tip={}` 定义的样式。
4. 箭头序列前面方括号里的选项设置。
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。

16.5 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 TikZ 兼容。当然，arrows.meta 可以与旧的 arrows，arrows.spaced 库同时加载使用。

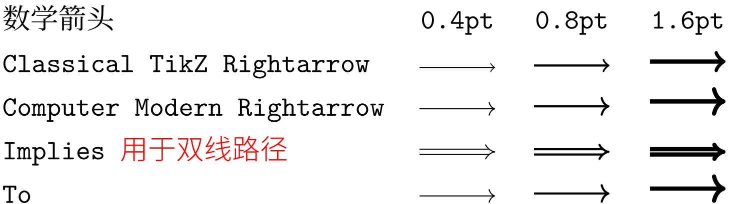
arrows.meta 定义的箭头可以分为以下几类：

- 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。

钩形箭头	0.4pt	0.8pt	1.6pt
Arc Barb			
Bar			
Bracket			
Hooks			
Parenthesis			
Straight Barb			
Tee Barb			

- 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。

数学箭头	0.4pt	0.8pt	1.6pt
`Circle`			
`Diamond`			
`Ellipse`			
`Kite`			
`Latex`			
`Rectangle`			
`Square`			
`Stealth`			
`Triangle`			
`Turned Square`			

- 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:

Cap 箭头	1ex	1em
`Butt Cap`		
`Fast Round`		
`Fast Triangle`		
`Round Cap`		
`Triangle 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.

17 Nodes and Edges

17.1 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, 在形状内部可以放置文字、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.

17.2 Nodes and Their Shapes

17.2.1 Node 命令的句法

\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属于这个 node 的辖域内。

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（(n2)）有效；如果给出两个“at(⟨*coordinate*⟩)”，则后一个坐标有效. 但如果给出多个“[⟨*options*⟩]”，它们的作用会累计。

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, 句法解析过程会在遇到闭方括号“]”时停止解析，也就是说，此时闭方括号 “]” 之后, 分号 “;” 之前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。

A B C D

```
\tikz {
  \path (0,0) node [red] {A}
        (1,0) node [blue] {B}
        (2,0) node [green, node contents=C]
        (3,0) node [node contents=D] [draw=red];
                                % [draw=red] 被忽略
}
```

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，这艘船有多个锚，每个锚都有自己的抛锚口；把一个锚抛到水底固定后，锚所抓住的点是 “锚定点”(anchor point)；尽管一个 node 有多个锚，但只能使用一个锚，所以只能有一个锚定点；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，即放出船外的锚链长度是 0；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”，而不是抛出船外的锚。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 TEX 错误。

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. 80.

```
\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. 785.

/tikz/in front of path (no value)

这个 key 的效果与 behind path 相反，这是 tikz 的默认行为。

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。

`/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. 132 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}
```

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。

```
\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}
```

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. 705. 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. 717.

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*⟩.

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。

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. 574.

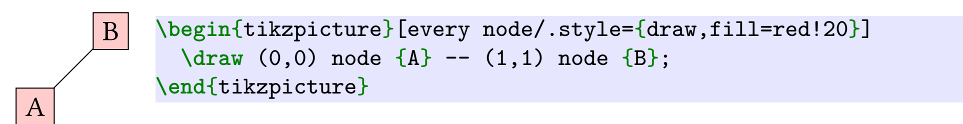
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
1,1	2,1	3,1	4,1

```
\tikz \node foreach \x in {1,...,4}
  foreach \y in {1,2,3}
  [draw] at (\x,\y) {\x,\y};
```

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)

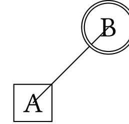
在这个 style 的有效范围内，它会把它所含的选项添加到所有 node 的开头。node 的选项会按照次序排成一个“选项序列”，依次执行。这个 style 所含的选项总是处于选项序列的开头。




```
\begin{tikzpicture}[every node/.style={draw,fill=red!20}]
  \draw (0,0) node {A} -- (1,1) node {B};
\end{tikzpicture}
```

/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}
```

/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}
```

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)。

A — B
A — B

```
\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}
}
```

其中选项 name prefix=PREFIX 的作用范围仅仅是 \node 命令，超出这个范围引用此 node 要在名称前加前缀。

A —— B

```
\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。

\path ... coordinate[⟨*options*⟩](⟨*name*⟩)at(⟨*coordinate*⟩){};

等价于

```
\node[shape=coordinate][⟨options⟩](⟨name⟩)at(⟨coordinate⟩){};
```

\node

这是 \path node 的简写.

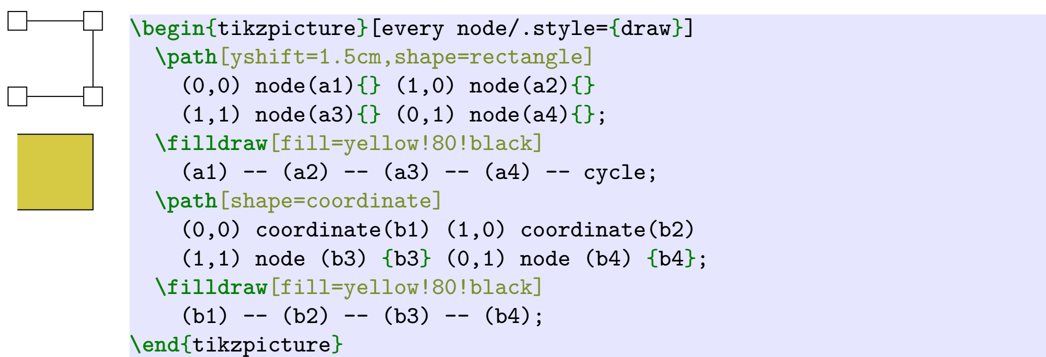
\coordinate

这是 \path coordinate 的简写.

17.2.2 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:

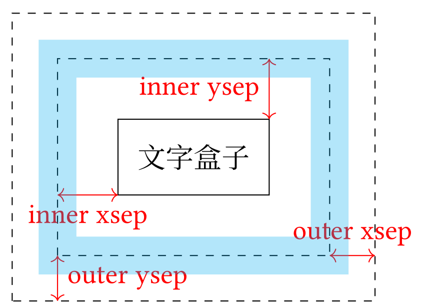


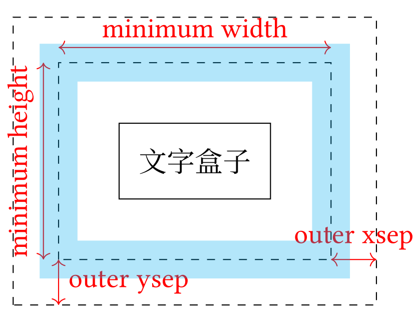
```
\begin{tikzpicture}[every node/.style={draw}]
  \path[yshift=1.5cm,shape=rectangle]
    (0,0) node(a1){} (1,0) node(a2){}
    (1,1) node(a3){} (0,1) node(a4){};
  \filldraw[fill=yellow!80!black]
    (a1) -- (a2) -- (a3) -- (a4) -- cycle;
  \path[shape=coordinate]
    (0,0) coordinate(b1) (1,0) coordinate(b2)
    (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 `\fill` 命令没有填充效果。

17.2.3 一般选项

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 “接口”。例如，对于形状为 `rectangle` 的 node, 它的特征尺寸如下图所示：



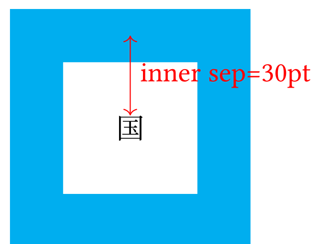


上面图形中的青色粗线代表 node 的背景路径 (即 shape 路径)，在默认下，`outer sep` 的初始值是半个线宽，即恰好达到青色线的外缘。

`/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。

↔ 10pt

连接点呢?

```
\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.

A

A

```
\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. 755, \pgfverticaltransformationadjustment→P. 755.
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)

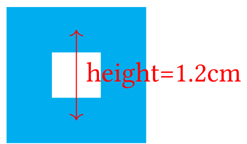
/tikz/outer ysep

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

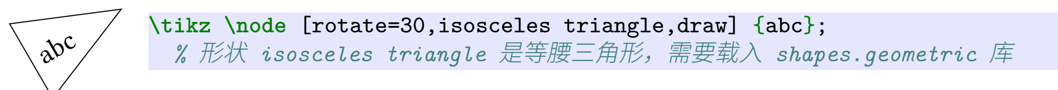
这是 /pgf/minimum size 的别名, 本选项把形状 (shape) 的最小宽度值, 高度值都指定为 ⟨*dimension*⟩.

/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 $\frac{\text{宽度}}{\text{高度}} = \langle \textit{aspect ratio} \rangle$.

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：



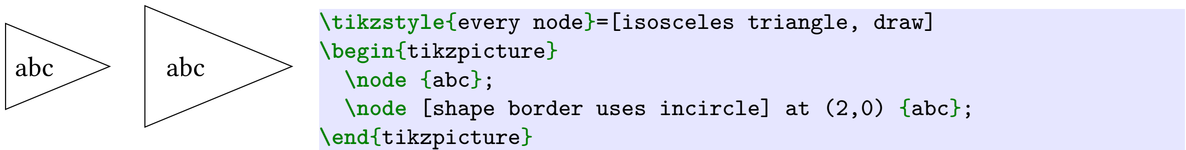
```
\tikz \node [rotate=30,isosceles triangle,draw] {abc};
  % 形状 isosceles triangle 是等腰三角形，需要载入 shapes.geometric 库
```

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*⟩. 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。

17.3 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.

17.4 node 中的文字

17.4.1 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。

red red red

```
\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. 188 设置。

17.4.2 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。

tiny small

```
\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。

tiny small

```
\tikz \node [font=\tiny, minimum height=3em, draw] {tiny};
\tikz \node [font=\small, minimum height=3em, draw] {small};
```

17.4.3 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] ...
```

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。注意不能直接将列格式定义 `\hfil\hfil##\hfil\cr` 作为该选项的值，而必须用这种曲折的方式。

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。

aaaaaaaa
aaaaaaaa

```
\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。

17.4.4 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`.

inner sep=5mm
text height=10mm
y

```
\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}
```

y y y

```
\tikz \node[draw] {y};
\tikz \node[draw,text height=10pt] {y};
\tikz \node[draw,text height=-5pt] {y};
```

`/tikz/text height`=⟨*dimension*⟩ (no default)

本选项指定文字盒子的深度。如果 ⟨*dimension*⟩ 空置，则使用文字盒子的“自然高度”。

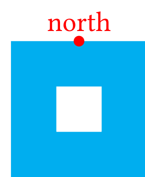
设置文字的高度和深度可以实现某种对齐效果。

17.5 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。

17.5.1 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：

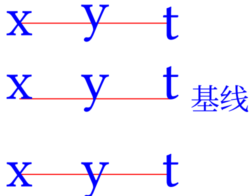
- 带有方向名词的位置：`north`，`east`，`north east` 等。
- 与内容文字的基线 `base` 有关的位置：`base`，`base west`，`base east`.
- 与 node 形状的竖直方向的中点 `mid` 有关的位置：`mid`，`mid west`，`mid east`.

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：

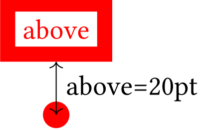



```
\begin{tikzpicture}[scale=2,red,text=blue,transform shape]
  \draw[anchor=center] (0,1) node{x} -- (0.5,1) node{y} -- (1,1) node{t};
  \draw[anchor=base] (0,.5) node{x} -- (0.5,.5) node{y} -- (1,.5) node{t}
                                                       node[anchor=west]{\tiny 基线
                                                       ↪  };
  \draw[anchor=mid] (0,0) node{x} -- (0.5,0) node{y} -- (1,0) node{t};
\end{tikzpicture}
```

17.5.2 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.

17.5.3 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 等会变得不一样。

/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){};
```

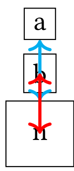
如果不指定平移距离，即等号“=”后没有算式，例如 `above= of somenode.north`, 那么平移距离就由选项 `node distance` 规定。

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。



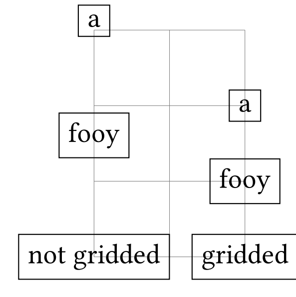

```
\tikz {\node [draw,inner sep=10pt] (n){n};
  \node (a)[above=8mm of n,draw]{a};
  \node (b)[above=8mm,anchor=center,draw] at(n){b};
  \draw [<->,cyan,very thick] (n.north)--++(0,8mm);
  \draw [<->,red,very thick] (n.center)--++(0,8mm);
}
```

`/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]{};
```

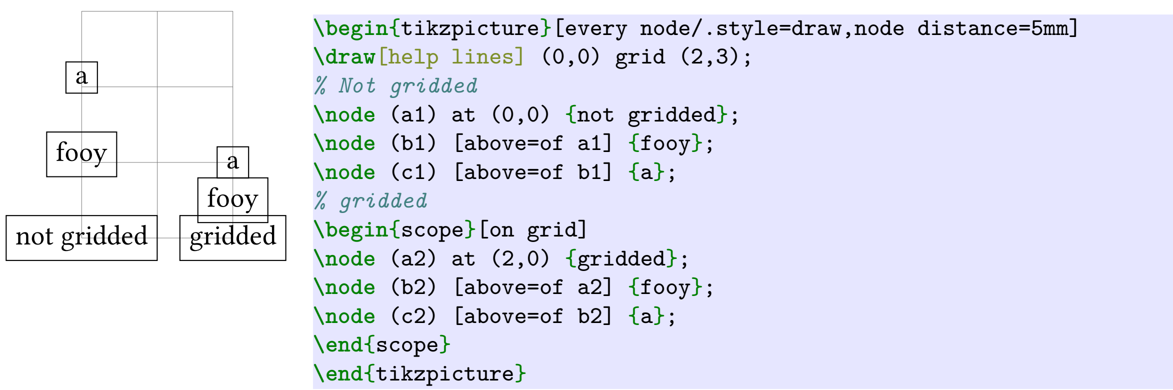
使得 `(a)` 的中心 `center` 位置位于 `(somenode)` 的中心 `center` 之上 5mm 处。




```
\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}
```

`/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)

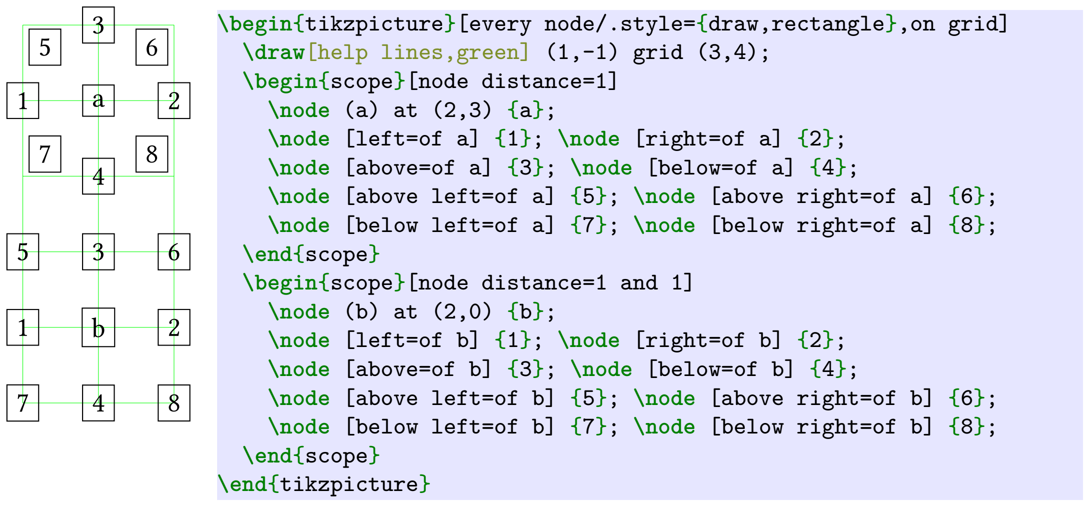
类似 `above`.

/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)

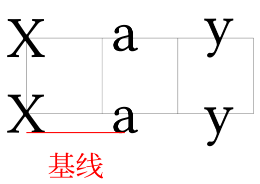
类似 `above left`.

`/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`.

17.5.4 排布 node 的高级方法

参考 graphdrawing 库和 matrix 库。

17.6 Fitting Nodes to a Set of Coordinates

参考 fit 库。

17.7 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}
```

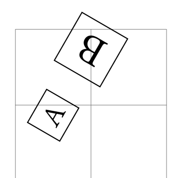
可以给 node 加变换选项来实现 node 的放缩、旋转、平移等变换，node 的旋转和平移都是相对于它的锚定点的：

```
\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}
```

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*⟩ (no default, 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。

`/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. 759.

17.8 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

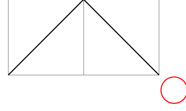
作创建的路径（子路径）的终点之后，例如 `(0,0)--(1,1)node{a}`. 如果 node 语句位于这些操作符号之后，例如 `(0,0)--node{a}(1,1)`, 则是“隐式地（implicitly）”。

`/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。

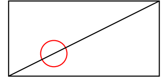

```
\tikz \draw (0,0) .. controls +(right:2) and +(right:2) .. (0,2)
  node foreach \p in {-0.5,-0.25,...,1.5} [pos=\p]{\small \p};
```

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}
```

/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。

/tikz/swap (no value)

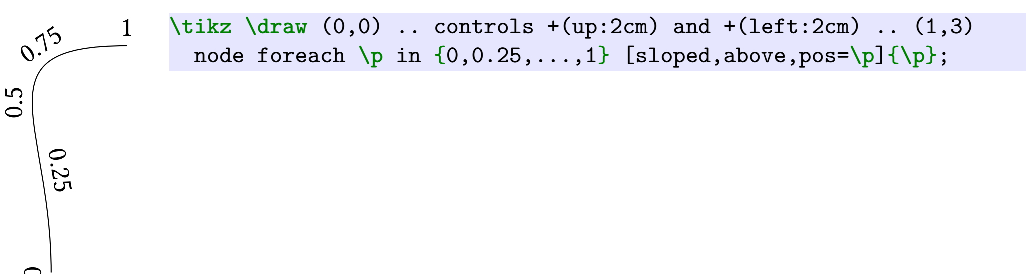
如果 `auto` 选项决定了 `left` 一侧，那么 `swap` 选项就把 node 转换到 `right` 一侧，即它翻转路径的左右侧。

/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”。

kkk

```
\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}
```

kkk

```
\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.

17.9 在直线段会曲线上隐式地摆放 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 ⟨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);
```

17.10 label 和 pin 选项

17.10.1 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。

17.10.2 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。

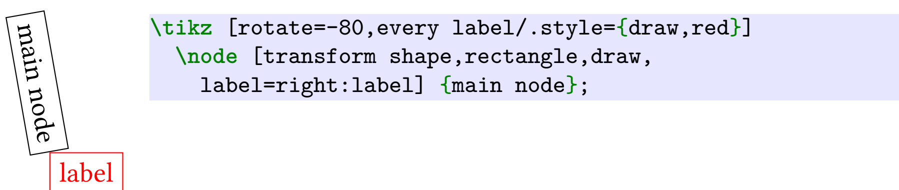
   如果不给出 ⟨*angle*⟩, 那么就默认使用下一个选项的值：

   `/tikz/label position=`⟨*angle*⟩ (no default, initially above)

   设置 label node 的默认位置。

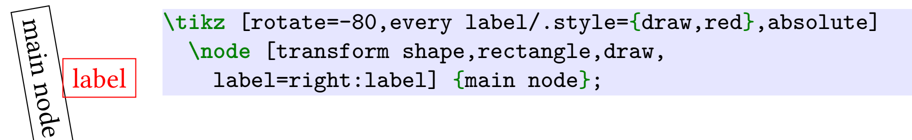
`/tikz/absolute`=⟨*true or false*⟩ (default true)

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。

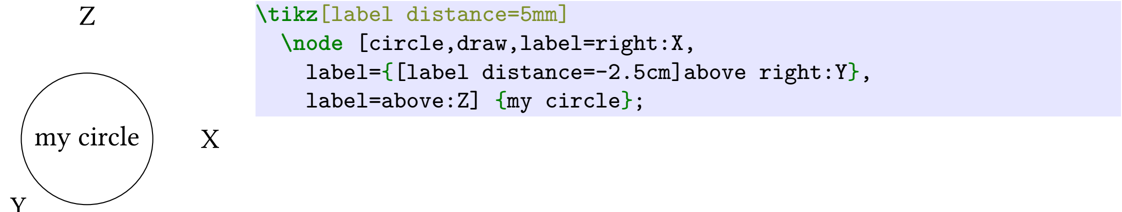
```
\begin{tikzpicture}
  \node [circle,draw,label={[name=label node]above left:$a,b$}] {};
  \draw (label node) -- +(1,1);
\end{tikzpicture}
```

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.

17.10.3 The Pin Option

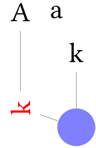
/tikz/pin=[⟨*options*⟩]⟨*angle*⟩:⟨*text*⟩ (no default)

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`)

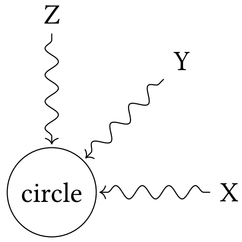
设置每个 pin 标签的样式。

`/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};
```

17.10.4 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 tikz 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。

90°

180° circle

```
\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。

1

2 circle

```
\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);
  }
```

17.11 Connecting Nodes: Using Nodes as Coordinates

17.12 Connecting Nodes: 用 edge 操作

17.12.1 edge 操作的基本句法

edge 操作的用法类似 to 操作和 node 操作，edge 操作会暂时中断当前主路径的构建过程并构建一个新的路径；在画主路径后，再画出这个新路径。edge 操作可以带有诸多选项来设定其所创建的路径外观。

edge 操作的基本句法是：

\path...edge[⟨*options*⟩]⟨*node*⟩(⟨*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 /tikz/to path →P. 61 规定的路径。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)）—这一点与 to 操作、node 操作是类似的—在 edge 操作完成后，继续主路径的构建过程；画出主路径后，再画出 to path 路径。但与 to 操作不同，对于 edge 操作来说，(\tikztostart) 会在 ⟨*path*⟩ 之前被添加（对于 to 操作来说，(\tikztostart) 是主路径上的一个点，没必要重复添加这个点）。

edge 操作与 to 操作有几个区别。

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：

(1, 1)

a

(0, 0) (1, 0)

```
\tikz{
  \draw (0,0) node(a)[above]{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:

$(0,1)$ $(1,1)$ $(0,0)$ $(1,0)$

```
\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，此时主路径的当前点还是点 (0,0)，于是主路径就是 (0,0)--(1,0). 对比下面 to 操作的例子:

to 2 to 1 $(0,0)$ $(1,0)$

```
\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] {$(1,0)$};
}
```

to 2 to 1 $(0,0)$ $(1,0)$

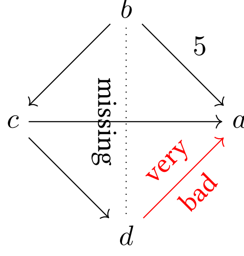
```
\tikz{
  \draw (0,0) edge[red,"to 1"left](0,1) to[cyan,"to
    2"](1,1)--(1,0);
  \fill [font=\scriptsize]
    (0,0) circle (2pt) node [below] {$(0,0)$}
    (1,0) circle (2pt) node [below] {$(1,0)$};
}
```

区别之三 edge 操作能继承主路径的选项，这与 to 操作一样:

b c a d

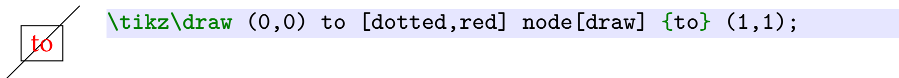
```
\begin{tikzpicture}
  \node foreach \name/\angle in {a/0,b/90,c/180,d/270}
    (\name) at (\angle:1) {$\name$};
  \path[->] (b) edge (a)
            edge (c)
            edge [-,dotted] (d)
        (c) edge (a)
            edge (d)
        (d) edge (a);
\end{tikzpicture}
```

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`)

针对每个 edge 操作的样式。

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}
```

17.12.2 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. 60, `/tikz/edge label`→P. 60 选项，但不能直接使用 `label`，`pin` 选项。
- 调用 quotes 程库，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。

left

```
\tikz \draw (0,0) edge ["left", ->] (2,0);
```

right

```
\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：

mid

```
\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);
```

17.13 Referencing Nodes Outside the Current Picture

17.13.1 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，tikz 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，TikZ 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 Big circle xxxxxx
xxxxxxxxxxxxxxx

```
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
Big circle
xxxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx

```
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 TikZ 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:

Big circle

```
\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。

17.13.2 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，它的锚位置 north east 是当前页面的右上角。给环境或命令加上 remember picture 和 overlay 选项后，就可以在该环境或命令中引用 current page 坐标系统中的位置。

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：

楞嚴經：注释注释注释注释注释注释

大佛頂如來密因修證了義諸菩薩萬行首楞嚴經文句

```
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
```

17.14 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 可能会帮助。

This is an absolutely positioned text in the lower left corner. No shipout-hackery is used.

Example

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、最近创建的 node.

`/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：

world
Hello

```
\begin{tikzpicture}
  \node [draw,circle] (a) {Hello};
  \path [late options={name=a, label=above:world}];
\end{tikzpicture}
```

17.15 遇到的问题

17.15.1 关于 auto 选项

见 `\pic`→P. 134 的内容。

17.15.2 关于 edge, to 的引用句法标签

内容 180: 内容

```
\tikz\draw (0,0)-- node ["180: 内容"] {}(0,0.5);\hspace{5mm}
\tikz\draw (0,0) edge ["180: 内容"] (0,0.5);
```

上面例子中，引用句法标签分别用作 `node` 与 `edge` 的选项，其效果不一样。

17.15.3 把 {tikzpicture} 环境作为 node 的内容

有时候，一个 node 的内容是个 `{tikzpicture}` 环境，而这个 `{tikzpicture}` 环境中也有 node，引用这个 `{tikzpicture}` 环境中的 node 时也会出现问题，例如：

a5 b5

```
\begin{tikzpicture}
  \node (a) {\tikz\node(aaaaa){a5};};
  \node (b)[right=1cm of a]{\tikz\node(bbbbb){b5};};
  \draw [red,->] (aaaaa) to[bend left] (b);
  \draw (aaaaa)--(bbbbb);
\end{tikzpicture}
```

其中第二个 `\draw` 命令没有预期的结果，添加 `remember picture` 选项也没有效果：

```
\begin{tikzpicture}[remember picture]
  \node (a) {\tikz[remember picture]\node(aaaaa){a5};};
  \node (b)[right=1cm of a]{\tikz[remember picture]\node(bbbbb){b5};};
  \draw [red,->] (aaaaa) to[bend left] (b);
  \draw (aaaaa)--(bbbbb);
\end{tikzpicture}
```

18 Pics: Small Pictures on Paths

18.1 Overview

pic 是 picture 的省写，代表的“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 tikz 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};
```

18.2 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，tikz 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*⟩`]` 中的选项如下文。

18.2.1 指定所用的 pic type

`/tikz/pic type=`⟨*pic type*⟩ (no default)

本选项指定当前 `pic` 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这个选项，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. 99，实际上这两个选项是可以相互替换使用的。

```
\tikz {
  \path (0,0) pic [pic type = seagull]
        (1,0) pic {seagull};
}
```

```
\tikz {
  \path (0,0) pic [node contents = seagull]
        (1,0) pic {seagull};
}
```

18.2.2 指定 pic 图形的位置

pic 图形的锚定点，或者是 `pic` 之前的点，或者是由 `at(`⟨*coordinate*⟩`)` 这一部分指定，或者是用选项 `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。

18.2.3 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。

18.2.4 定义 pic code

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}}}}
};
```

18.2.5 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)

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];}}
};
```

18.2.6 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. 100 或 /tikz/in front of path→P. 100 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)

18.2.7 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。

18.2.8 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. 102 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);
}
```

18.2.9 用 pic 制作动画

这与用 node 制作动画类似。

18.2.10 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。

18.3 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*⟩`/.pic=`⟨*some 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 ⟨*key*⟩ 加路径前缀 `/tikz/`，然后 `pic` 会用 `/tikz/pic/` 替换 `/tikz/`。这个手柄会把 ⟨*key*⟩ 定义为一个样式（style），此样式所保存的内容就是 `code=`⟨*some code*⟩.

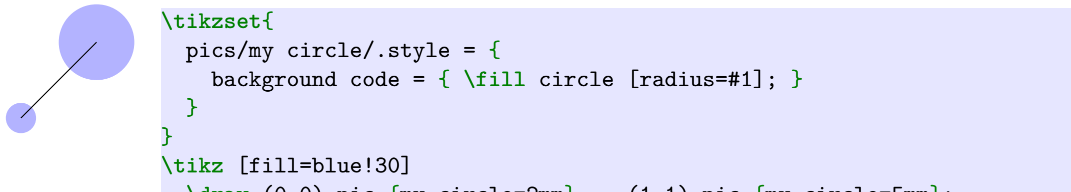
```
\tikzset{
  seagull/.pic = { % 定义名称为 seagull 的 pic type
    \draw (...) ... ;
    ...
  }
}
```

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 (...) ... ;
    }
  }
}
```

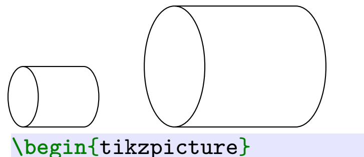
手柄 `/.style` 最多可以使用 1 个变量，如果需要使用 2 个或更多变量，可以用手柄 `/.style 2 args`, `/.style args`, `/.style n args`.

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}
```

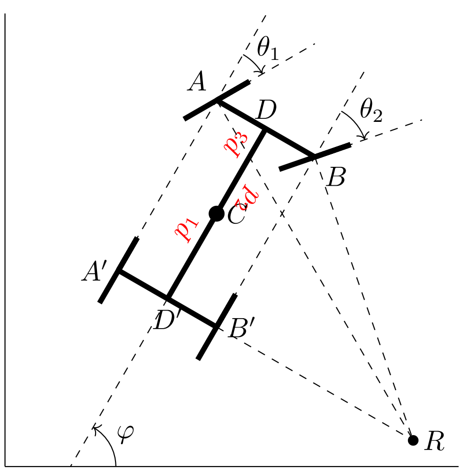
18.4 遇到的问题

这个问题是关于 pic 图形与 node 的选项 `/tikz/auto`→P. 119 的关系的。

定义一个 pic type：

```
% 定义一个 pic 图形，其中点的相对位置是
% A---D---B
%     |
%     |
%     |
%     |
%A'---D'---B'
\tikzset{% #1 轮距，#2 轴距，#3 左前轮转向角（顺时针为负，逆时针为正）
  pics/carturn/.style n args={3}{
    code={
      \coordinate (-A') at (0,0);
      \coordinate (-B') at (#1,0);
      \coordinate (-A)  at (0,#2);
      \coordinate (-B)  at (#1,#2);
      \coordinate (-D)  at ($(-A)!0.5!(-B)$);
      \coordinate (-D') at ($(-A')!0.5!(-B')$);
      \coordinate (-C)  at ($(-D)!0.5!(-D')$); % 质心
      \coordinate (-R) at ({-#2*cos(#3)/sin(#3)},0); % 转向中心，曲率中心
  \path
    let \p1=($(-R)-(-B)$)
    in coordinate (-rightturningangle) at({atan(\y1/\x1)+90}:1);
  \draw (-A)--(-B);
  \draw (-A')--(-B');
  \draw (-D)--(-D');
  \draw (-A)--+(90+#3:0.5)--+(90+#3+180:0.5);
  \draw (-B)--+($0.5*(-rightturningangle)$)--+($-0.5*(-rightturningangle)$);
  \draw (-A')--+(90:0.5)--+(-90:0.5);
  \draw (-B')--+(90:0.5)--+(-90:0.5);
  \fill (-C) circle (3pt);
}}}
```

用 carturn 画一个图形：




```
% \usepackage{xifthen}
\begin{tikzpicture}
  \draw [name path=x](0,0)--(6,0);
  \draw (0,0)--(0,6);
  \def\leftturningangle{-30} % 左前轮转向角
  \begin{scope}[rotate=-30,]
    \pic (1)[transform shape,line width=2pt]at(0,3){carturn={1.5}{2.6}{\leftturningangle}};
    \node [above left] at(1-A){$A$};
    \node [below right] at(1-B){$B$};
    \node [above] at(1-D){$D$};
    \node [left] at(1-A'){$A'$};
    \node [right] at(1-B'){$B'$};
    \node [below] at(1-D'){$D'$};
    \node [right] at(1-C){$C$};
% 注意下面 3 个 \path 路径所添加的标签 $p_1$, $p_2$, $p_3$, 就是令人疑惑的地方
    {[samestyle/.style={transform shape,midway,sloped,auto=left,text=red}]
    \path (1-D')--(1-C) node[samestyle]{$p_1$};
    \path (1-C)--(1-D) node[samestyle]{$p_2$};
    \path (1-C)--(1-D) node[samestyle,allow upside down=true]{$p_3$};
    }

    \fill (1-R) circle(2pt) node [right] {$R$};
    \draw [dashed](1-B')--(1-R);
    \draw [dashed](1-B)--(1-R);
    \draw [dashed](1-A)--(1-R);

    \coordinate (T1) at ($(1-A')!1.5!(1-A)$);
    \coordinate (T2) at ($(1-A)+(90+\leftturningangle:1.5)$);
    \draw [dashed](1-A')--(T1);
    \draw [dashed](1-A)--(T2);
    \ifthenelse{\leftturningangle>0}
    {\pic [draw, "$\theta_1$", angle radius=7mm, angle eccentricity=1.4,][->]{angle=T1--1-A--T2}}
    {\pic [draw, "$\theta_1$", angle radius=7mm, angle eccentricity=1.4,][<-]{angle=T2--1-A--T1}};

    \coordinate (T3) at ($(1-B')!1.5!(1-B)$);
    \coordinate (T4) at ($($1.5*($(1-rightturningangle)-(1-A')$)$)+(1-B)$);
    \draw [dashed](1-B')--(T3);
    \draw [dashed](1-B)--(T4);

    \ifthenelse{\leftturningangle>0}
    {\pic [draw, "$\theta_2$", angle radius=7mm, angle eccentricity=1.4,][->]{angle=T3--1-B--T4}}
    {\pic [draw, "$\theta_2$", angle radius=7mm, angle eccentricity=1.4,][<-]{angle=T4--1-B--T3}};
  \end{scope}

```

```
45   \path [name path=DD',overlay] (1-D)--($(1-D)!2!(1-D')$);
46   \draw [name intersections={of=DD' and x,by=T5},dashed] (1-D')--(T5);
47
48   \coordinate (T6) at ($(T5)+(5mm,0)$);
49   \pic [draw, "$\varphi$", angle radius=6mm, angle eccentricity=1.4,][->]{angle=T6--T5--1-D};
50 \end{tikzpicture}
```

上面代码中的第 16, 17, 18, 19, 20 这几行添加标签，但 `auto=left` 的作用有点费解。

41 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`

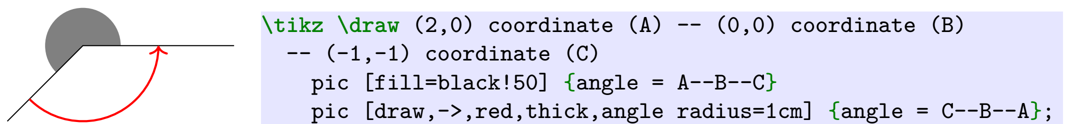
`angle=`⟨*A*⟩`--`⟨*B*⟩`--`⟨*C*⟩

`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。

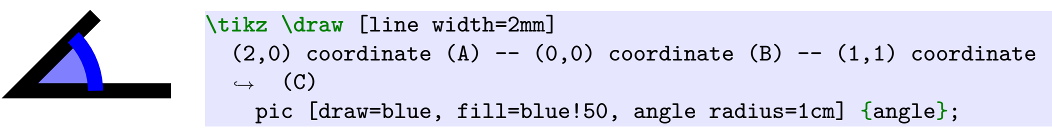
`/tikz/angle radius=`⟨*dimension*⟩ (no default, initially 5mm)

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：

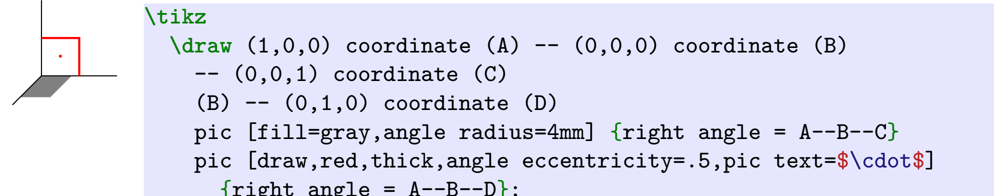
`/tikz/angle eccentricity`=⟨*factor*⟩ (no default, initially `0.6`)

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) ["$\alpha$", 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};
```

54 fit 程序库

TikZ Library `fit`

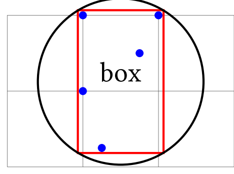
```
\usetikzlibrary{fit} % LaTeX and plain TeX
\usetikzlibrary[fit] % ConTeXt
```

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}
```

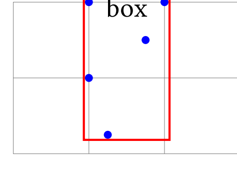
`/tikz/every fit` (style, initially empty)

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。

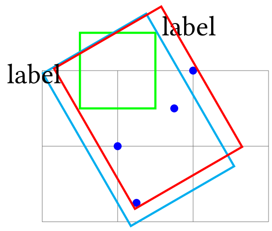
由于 fit node 的文字盒子的对齐方式是固定不变的，因此要想调整其文字盒子里的文字内容的位置就需要变通的方法，如下面的例子所示，另作一个包含文字的 node 添加到图形中：




```
\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}
```

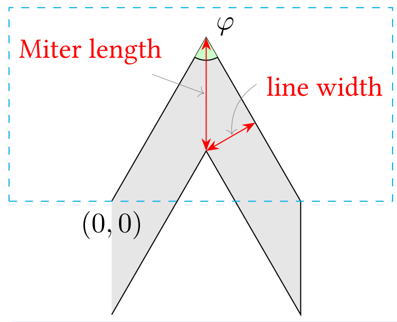
`/tikz/rotate fit=`⟨*angle*⟩ (no default, initially 0)

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)]{};
}
```

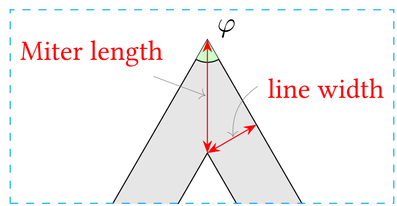
可能会觉得实际上只需要虚线框内的部分，即 `(fit node)` 内的那一部分。这个情况下首先想到的当然是使用 `clip` 命令，但是 `(fit node)` 是由最后一个 node 命令创建的，给这个 node 命令带上 `clip` 选项并不能达到目的。此时可以变通一下，先把上面的绘图代码保存到某个 key 中：

```
\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. 833 和 \pgfsys@endinvisible →P. 833, 这两个命令之间的绘图命令所画出来的图形是“不可见的”，但的确是画出来了（会被计入图形的 bounding box 之内，在页面上占据一定空间)。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。

73 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》.

73.1 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. 61.

```
\tikz {\draw (0,0) to[line to] (1,1);}
```

73.2 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);
```

73.3 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：

$(x_0\text{pt}, y_0\text{pt})$ .. controls $(x_1\text{pt}, y_1\text{pt})$ and $(x_2\text{pt}, y_2\text{pt})$ .. $(x_3\text{pt}, y_3\text{pt})$

记起点 $P_0(x_0\text{pt}, y_0\text{pt})$, 终点 $P_3(x_3\text{pt}, y_3\text{pt})$, 起点和终点是给定的。按如下的方式确定控制点 $P_1 = (x_1\text{pt}, y_1\text{pt})$ 和 $P_2 = (x_2\text{pt}, y_2\text{pt})$.

记 $Q = (|x_3 - x_0|\text{pt}, |y_3 - y_0|\text{pt}) = (x_Q\text{pt}, y_Q\text{pt})$, $n_Q = (x_{n_Q}\text{pt}, y_{n_Q}\text{pt})$, n_Q 与 Q 同向平行并且 $\|n_Q\| = 1\text{pt}$.

- 如果 $[65536\cdot\max\{x_{nQ},y_{nQ}\}]\neq 0$

$$r_{out}=\min\{\max\{m_{out},r'_{out}\},M_{out}\},\quad r'_{out}=l_{out}\cdot 0.3915\cdot\frac{16^2\cdot\max\{x_Q\texttt{pt},y_Q\texttt{pt}\}}{\left[\frac{[65536\cdot\max\{x_{nQ},y_{nQ}\}]}{255}\right]},$$

$$r_{in}=\min\{\max\{m_{in},r'_{in}\},M_{in}\},\quad r'_{in}=l_{in}\cdot 0.3915\cdot\frac{16^2\cdot\max\{x_Q\texttt{pt},y_Q\texttt{pt}\}}{\left[\frac{[65536\cdot\max\{x_{nQ},y_{nQ}\}]}{255}\right]}.$$

其中的方括号表示数值的整数部分，注意 $\max\{x_{nQ},y_{nQ}\}$ 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，$\left[\frac{[65536\cdot\max\{x_{nQ},y_{nQ}\}]}{255}\right]$ 是由

```
\c@pgf@counta=max{x_Q pt, y_Q pt}%
\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 $[65536\cdot\max\{x_{nQ},y_{nQ}\}]=0$

$$r_{out}=\min\{\max\{m_{out},r'_{out}\},M_{out}\},\quad r'_{out}=l_{out}\cdot 0.3915\cdot x_Q\texttt{pt},$$

$$r_{in}=\min\{\max\{m_{in},r'_{in}\},M_{in}\},\quad r'_{in}=l_{in}\cdot 0.3915\cdot x_Q\texttt{pt}.$$

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*⟩`)`.

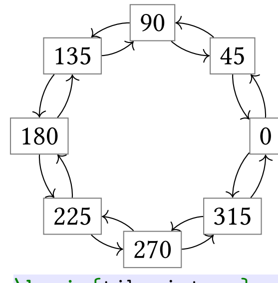
/tikz/relative=⟨*true or false*⟩ (default `true`)

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.



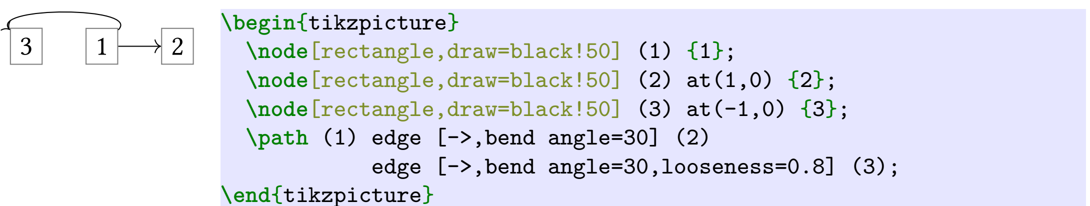

```
\begin{tikzpicture}
  \foreach \angle in {0,45,...,315}
  \node[rectangle,draw=black!50] (\angle) at (\angle:1.5) {\angle};
  \foreach \from/\to in {0/45,45/90,90/135,135/180,180/225,225/270,270/315,315/0}
  \path (\from) edge [->,bend right=22,looseness=0.8] (\to)
                edge [<-,bend left=22,looseness=0.8] (\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}:

$$0.3915\times\frac{16^2\times 28.45274\text{pt}}{\left\lceil\frac{[65536\times\frac{\sqrt{2}}{2}]}{255}\right\rceil}\approx 0.5537237569061\times 28.45274\text{pt}\approx 15.75495808707\text{pt}.$$

`/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*⟩ (default 0pt)

这个选项把参数 m_{out} 设置为 ⟨*distance*⟩.

`/tikz/out max distance`⟨*distance*⟩ (default 10000pt)

这个选项把参数 M_{out} 设置为 ⟨*distance*⟩.

`/tikz/in min distance`⟨*distance*⟩ (default 0pt)

这个选项把参数 m_{in} 设置为 ⟨*distance*⟩.

`/tikz/in max distance`⟨*distance*⟩ (default 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)

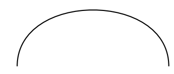
本选项直接指定第一个支撑点 P_1，这里 ⟨*coordinate*⟩ 可以使用相对坐标形式，例如 `+(1,1)`，它相对于起点计算。

`/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。

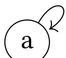


```
\tikz \draw (0,0) to [controls=+(90:1) and +(90:1)] (2,0);
```

73.4 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`.

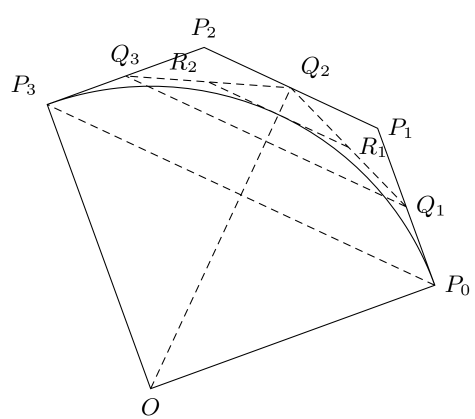
`/tikz/every loop` (style, initially `->,shorten >=1pt`)

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}
```

73.5 关于 curve to 选项的系数

设 $0 < \varphi \leqslant \frac{\pi}{2}$, 考虑单位圆上从点 $P_0 = (\cos\alpha, \sin\alpha)$ 到点 $P_3 = (\cos(\alpha+\varphi), \sin(\alpha+\varphi))$ 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：

$$P_1 = \frac{4}{3}\cdot\frac{1-\cos\frac{\varphi}{2}}{\sin\frac{\varphi}{2}}\cdot\begin{pmatrix}-\sin\alpha\\ \cos\alpha\end{pmatrix}+\begin{pmatrix}\cos\alpha\\ \sin\alpha\end{pmatrix},$$

$$P_2 = \frac{4}{3}\cdot\frac{1-\cos\frac{\varphi}{2}}{\sin\frac{\varphi}{2}}\cdot\begin{pmatrix}\sin(\alpha+\varphi)\\ -\cos(\alpha+\varphi)\end{pmatrix}+\begin{pmatrix}\cos(\alpha+\varphi)\\ \sin(\alpha+\varphi)\end{pmatrix}.$$

当 $\varphi=\frac{\pi}{2}$ 时，系数 $\frac{4}{3}\cdot\frac{1-\cos\frac{\varphi}{2}}{\sin\frac{\varphi}{2}}=\frac{4}{3}(\sqrt{2}-1)\approx 0.5522847498308.$

74 through 程序库

TikZ Library through

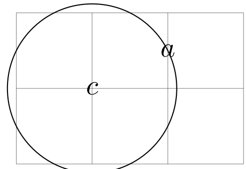
```
\usetikzlibrary{through} % LaTeX and plain TeX
\usetikzlibrary[through] % ConTeXt
```

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}
```

20 矩阵及其对齐方式

20.1 Overview

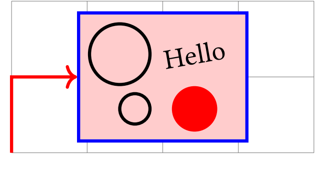
一个 TikZ 的矩阵（matrix）类似于 LaTeX 的 `{tabular}` 或 `{array}` 环境，只不过矩阵的元素（cell）是 TikZ 的绘图代码（或者留空）。矩阵的每一行（包括最后一行）都用 `\\` 结束，一行内的相邻元素之间用 `&` 分隔。注意 `&` 和 `\\` 的后面可以带有方括号选项。

20.2 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。

20.3 元素图形

矩阵的元素是“图形”，元素图形（cell pictures）没有“图层”（layer）的概念。矩阵有确定的行数和列数，如果某一行的元素个数不足就自动用“空元素”（empty cells）填补。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。元素图形不是在 {pgfpicture} 环境中画出的，它有更为“轻量化”的设计。

20.3.1 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。

```
\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。

20.3.2 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：

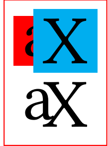
- 是一个长度尺寸，例如 10pt，也可以是负值尺寸，如 -2mm.
- 是一个长度与一个词组的组合,例如 {5mm,between origins},或者 {between borders,-2mm}。

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。

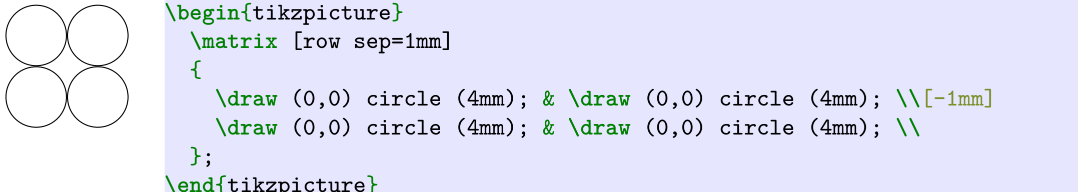



```
\tikz[font=\Huge]
  \matrix [draw=red,column sep=-4mm]
  {
  \node [fill=red]{a}; & \node [fill=cyan]{X}; \\
  \node {a}; & \node {X}; \\
  };
```

`/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。如果上两个元素图形有重叠，则下部的遮挡上部的。

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]
    \draw (0,0) circle (4mm); & \draw (0,0) circle (4mm); \\
  };
\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。

8 1
3 5

```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep=1mm]
  {
    \node {8}; &                      \node{1}; \\
    \node {3}; &[20mm,between origins] \node{5}; \\
  };
\end{tikzpicture}
```

5mm 10mm
8 1 6
3 5 7
4 9 2

```
\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}
```

20.3.3 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 `dra`，`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。

8	1	6
3	5	7
4	9	2

```
\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}
```

123	456	789
12	45	78
1	4	7

```
\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。

8	1	??
3	??	7
??	??	2

```
\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}
```

20.4 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)

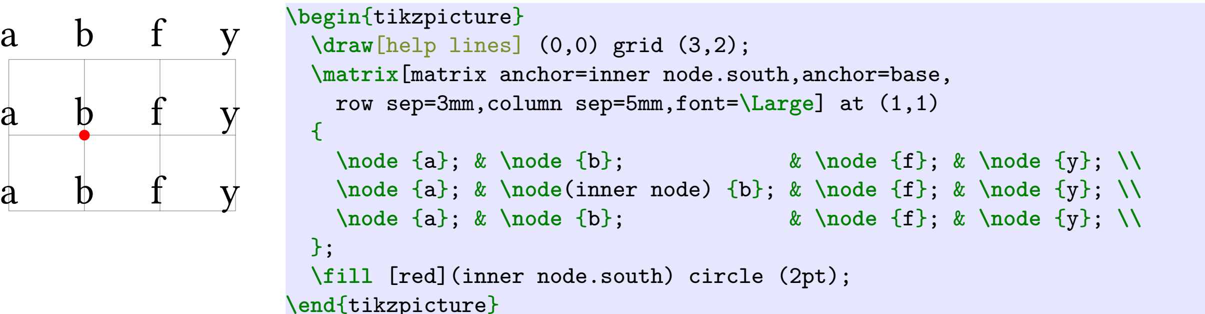
这里的 ⟨*anchor or node.anchor*⟩ 可以是矩阵自己的 anchor 位置，也可以是某个元素图形中的 node 的 anchor 位置。这个选项将 ⟨*anchor or node.anchor*⟩ 位置放在矩阵的锚定点上。矩阵的锚定点就是选项 `at` 指定的位置，或者其它类似的定位形式确定的位置。

123
12
1

```
\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 node 的 ⟨*anchor*⟩ 位置放在它的锚定点上（默认为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` 位置与该图形坐标系的原点重合，这决定了元素的对齐方式。

20.5 自定义分列符

在构造矩阵时，TikZ 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 TikZ 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 TikZ 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。

Hello

```
\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); \\
  };
```

关于矩阵的其它内容参考 matrix 库。

59 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。

59.1 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）。

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 & |[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，tikz 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，但其它作用（即设置锚位置和命名作用）仍然有效。这样就可以单独设置该元素图形的外观。

8	1	6
3	5	7
4	9	2

```
\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.

a_8		a_6
a_3		a_7
a_4	a_9	

```
\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}
```

59.2 换行符号与矩阵行的结束符号

符号 \\ 是 TEX 的换行符号，也是 TikZ 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。

row 1	upper line
lower line	
row 2	hmm

```
\begin{tikzpicture}
  \matrix [matrix of nodes,nodes={text width=16mm,draw}]
  {
    row 1 & upper line \\ lower line \\
    row 2 & hmm \\
  };
\end{tikzpicture}
```

row 1	upper line lower line
row 2	hmm

```
\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`.

59.3 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。但这里要求 node 有各种标准的 anchor 位置（罗盘位置 `north`，`south` 等）。

/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 delimiter              (style, initially empty)
/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)
```

$\left(\int_0^1 x\,\mathrm{d}x\right\}$

```
\begin{tikzpicture}
  \node [fill=red!20,left delimiter=(,right delimiter=\}]
    {$\displaystyle\int_0^1 x\,\mathrm{d}x$};
\end{tikzpicture}
```

$\left(\begin{matrix} a_8 & a_1 & a_6 \\ a_3 & a_5 & a_7 \\ a_4 & a_9 & a_2 \end{matrix}\right\}$

```
\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}
```

$\left\|\begin{matrix} a_8 & a_1 & a_6 \\ a_3 & a_5 & a_7 \\ a_4 & a_9 & a_2 \end{matrix}\right\rmoustache$

```
\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}
```

22 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.

22.1 Overview

对于十分复杂、精细的图形来说, TikZ 的绘图能力不如专业的数学软件(例如 gnuplot, mathematica), 但在 TEX 中使用 TikZ 绘图的优势是它与 TEX 的兼容性。

用 TikZ 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。

22.2 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"，尽管当前路径可能只有这么一段子路径，但 "层次" 不一样。

22.3 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。

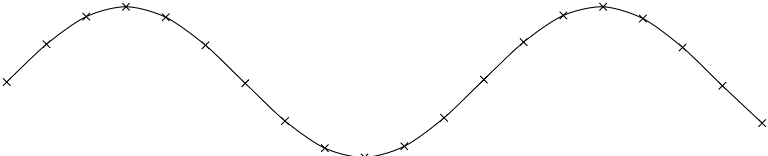
```
\tikz \draw plot coordinates {(0,0) (1,1) (2,0) (3,1) (2,1) (10:2cm)};
```

```
\tikz \draw (0,0) -- (1,1) plot coordinates {(2,0) (3,0)}
  --plot coordinates {(2,1)(3,1)};
```

22.4 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 TikZ 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)
```

22.5 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。

可以设置关于函数的自变量、定义域、样本点等内容。

`/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。

`/tikz/samples`=⟨*number*⟩ (no default, initially 25)

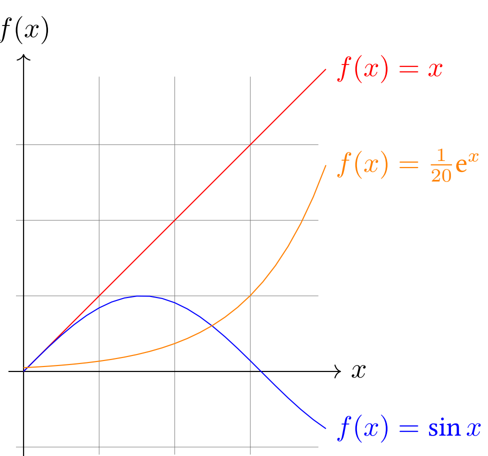
设置变量 ⟨*macro*⟩ 的采样点数目。

`/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。

`/tikz/samples at`=⟨*sample list*⟩ (no default)

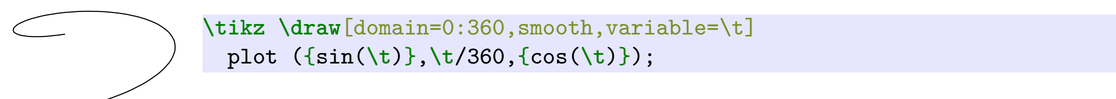
直接指定变量 ⟨*macro*⟩ 的采样点。⟨*sample list*⟩ 是个列表，此列表采用 `\foreach` 句法的列表形式。注意这个选项会抑制 `samples` 和 `domain` 选项。




```
\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}
```

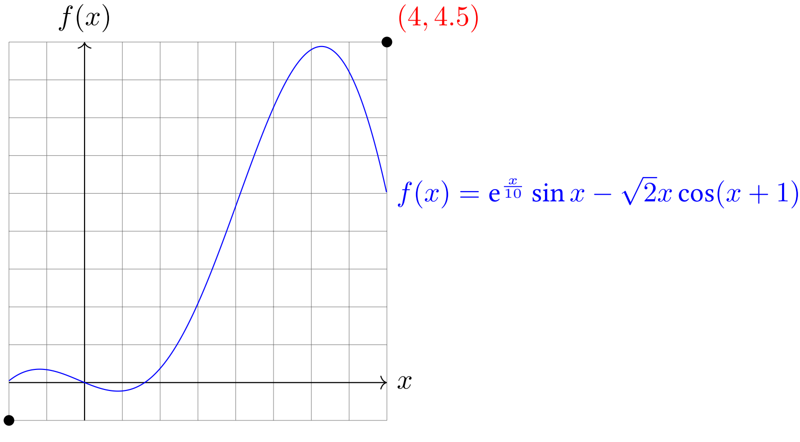
上面例子中，符号 `\x r` 是将角度 `\x` 转换为弧度。



```
\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}
```

当 ⟨*coordinate expression*⟩ 中有变量的幂运算时，例如 x^4 时，要注意区别 `(\x)^4` 与 `\x^4`，比较下图中的两个曲线：

```
\tikz{
  \draw [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。

22.6 调用 gnuplot 绘制函数图形

首先确保安装了 gnuplot. 参考 `\pgfplotgnuplot` →P. 778.

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。

操作 `plot` 调用 gnuplot 绘图需要使用 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 `pdflatex --shell-escape` ⟨*tex* 文件名⟩`.tex` 或 `xelatex --shell-escape` ⟨*tex* 文件名⟩`.tex` 这样的命令。

当 TikZ 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，TikZ 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。这些内部代码能够让 `plot` 操作把某些坐标数据写入另一个文件 ⟨*prefix*⟩`.`⟨*id*⟩`.table` 中。文件 ⟨*prefix*⟩`.`⟨*id*⟩`.table` 会被用于绘图，就像使用语句 `plot file{`⟨*prefix*⟩`.`⟨*id*⟩`.table}` 那样。

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}
```

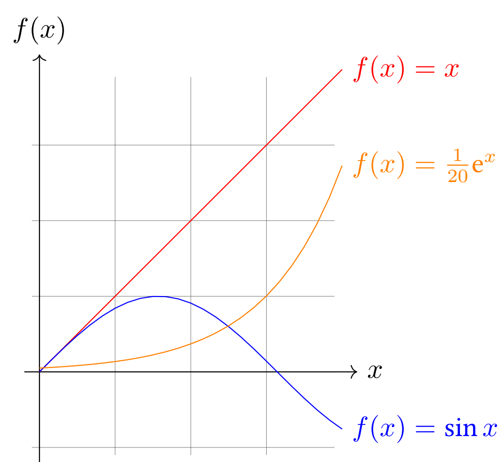
会得到文件

⟨*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 TikZ 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,但仍然能编译 .tex 文件来得到文件 ⟨*prefix*⟩.⟨*id*⟩.gnuplot,然后再用 gnuplot 处理文件 ⟨*prefix*⟩.⟨*id*⟩.gnuplot 得到文件 ⟨*prefix*⟩.⟨*id*⟩.table, 这样这两个文件就都有了，然后 TikZ 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 TikZ 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:

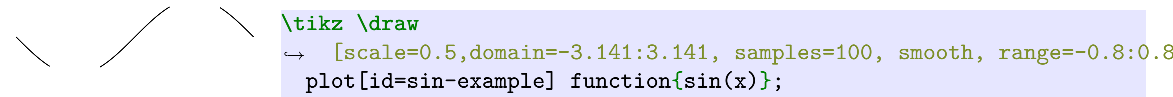
/tikz/parametric=⟨*boolean*⟩ (default true)

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/}}
```

22.7 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. 185 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，那最好不要使用 `ball`，因为它用 PostScript 来渲染，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)

这个选项会使得序号为 `1`，⟨*r*⟩`+1`，`2*`⟨*r*⟩`+1`，`3*`⟨*r*⟩`+1...` 的样本点被标记。

```
\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)
  };
```

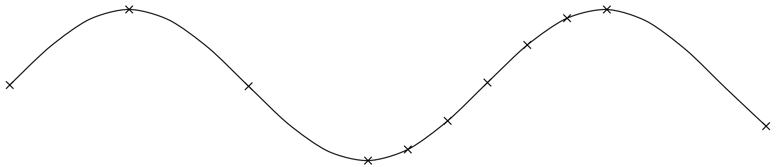
上面的图形中有 20 个样本点，被标记的是第 1, 4, 7, 10, 13, 16, 19 个点，共七个点。

`/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 `metap+`⟨*r*⟩ 个点，然后标记第 `metap+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。如果 ⟨*dimension*⟩ 是负值尺寸则把它当作正值尺寸看待。

`/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。

22.8 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)

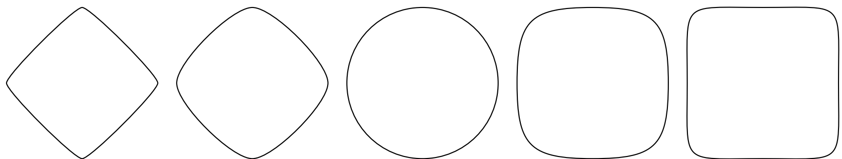
这个选项指示 plot 操作用直线段来连接样本点，这是默认的做法。

/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。

/tikz/tension=⟨*value*⟩ (default 0.5)

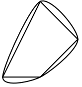
这个选项调节曲线的“张力”，即弯曲状态，数值越大，弯曲愈著。若有 4 个样本点均匀分布于一个圆上且张力值 ⟨*value*⟩ 是 1，则绘制的曲线是个圆。⟨*value*⟩ 的默认值是 0.5.



```
\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。

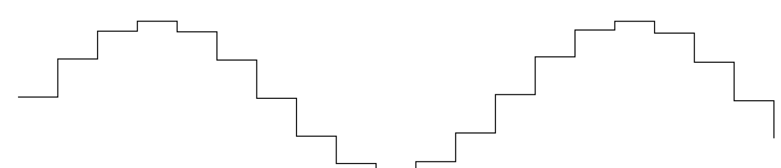


```
\tikz[scale=0.5]
  \draw plot[smooth cycle] coordinates{(0,0) (1,0) (2,1) (1,2)}
    plot coordinates{(0,0) (1,0) (2,1) (1,2)} -- cycle;
```

/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};
```

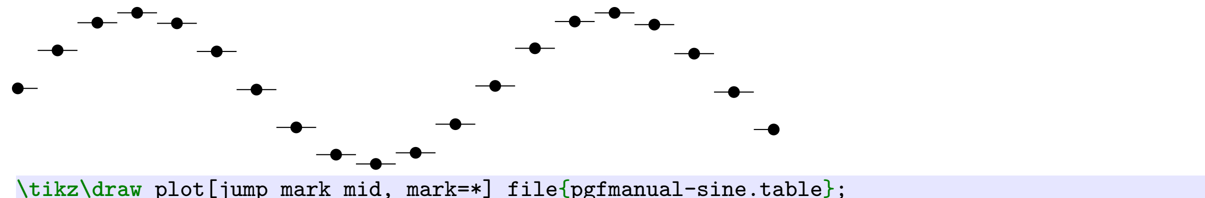
/tikz/jump mark left (no value)

```
\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};
```

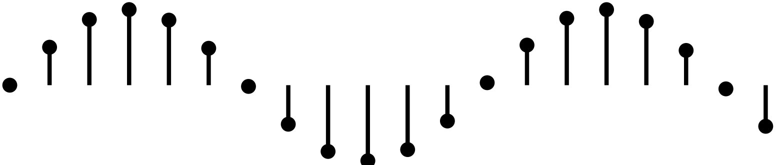
/tikz/jump mark mid (no value)



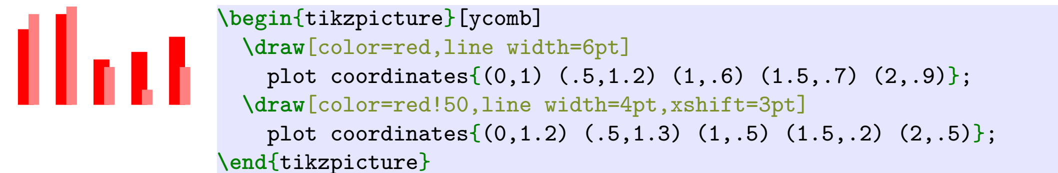
```
\tikz\draw plot[jump mark mid, mark=*] file{pgfmanual-sine.table};
```

/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。

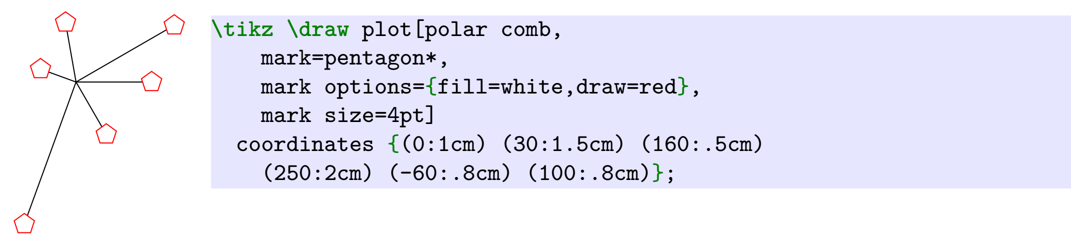
/tikz/xcomb (no value)

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)

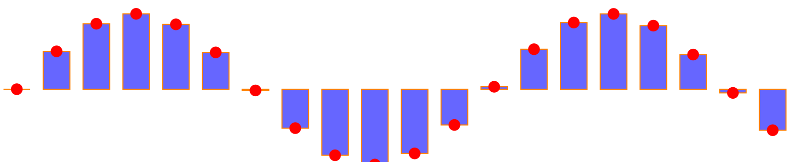
绘制一个放射状的图形，放射中心在 $(0, 0)$.



```
\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。



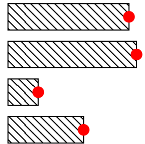
```
\tikz{\draw[draw=orange,fill=blue!60!white] plot[ybar] file{pgfmanual-sine.table};
  \draw plot[mark=*,only marks,mark options={color=red}] file{pgfmanual-sine.table};}
```

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. 182 和 /pgf/bar shift→P. 182 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)

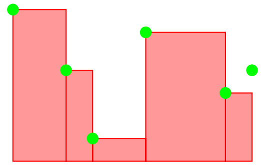
各个 bar 的起点都位于直线 $x = 0$ 上。



```
\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)};}
```

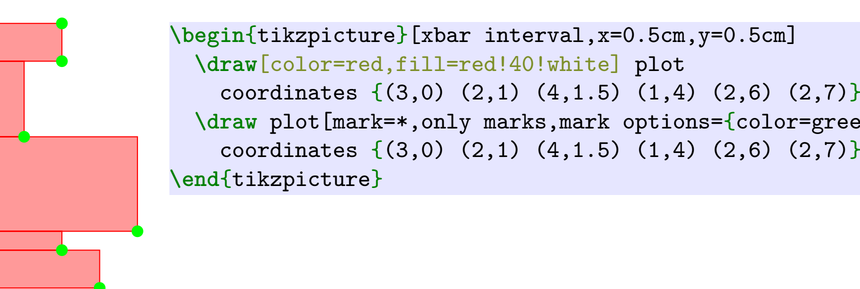
/tikz/ybar interval (no value)

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)

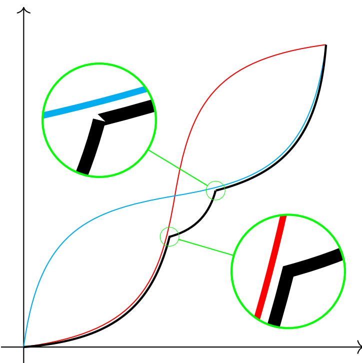


```
\begin{tikzpicture}[xbar interval,x=0.5cm,y=0.5cm]
  \draw[color=red,fill=red!40!white] plot
    coordinates {(3,0) (2,1) (4,1.5) (1,4) (2,6) (2,7)};
  \draw plot[mark=*,only marks,mark options={color=green}]
    coordinates {(3,0) (2,1) (4,1.5) (1,4) (2,6) (2,7)};
\end{tikzpicture}
```

对于 ybar, xbar 类型的柱状图,可以用选项 /pgf/bar width→P. 182 和 /pgf/bar shift→P. 182 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. 183 和 /pgf/bar interval shift→P. 183 调整其外观。

22.9 遇到的问题

观察下面的图形：



```
\begin{tikzpicture}
  [spy using outlines={circle, magnification=6, size=1.5cm, connect spies}]

% 坐标轴
  \draw [->](-0.3,0)--(4.5,0);
  \draw [->](0,-0.3)--(0,4.5);

\tikzmath{
% 红色控制曲线的 7 个点依次是 \first1=(0,0), \first2, ... , \first7,
% 假设 t 是参数，则红色控制曲线的第一段的参数方程是
% x=3*t*(1-t)^2*\firstx2+3*t^2*(1-t)*\firstx3+t^3*\firstx4,
% y=3*t*(1-t)^2*\firsty2+3*t^2*(1-t)*\firsty3+t^3*\firsty4,
  coordinate \first;
  \first1=(0,0);
  \first2=(1.5,0.2);
  \first3=(1.8,0.8);
  \first4=(2,2);
  \first5=($(\first3)!2!(\first4)$);
  \first6=($(\first2)!2!(\first4)$);
  \first7=($(\first1)!2!(\first4)$);
% 青色控制曲线的 7 个点依次是 \second1=(0,0), \second2, ... , \second7
% 假设 t 是参数，则青色控制曲线的第二段的参数方程是
% x=(1-t)^3*\secondx4+3*t*(1-t)^2*\secondx5+3*t^2*(1-t)*\secondx6+t^3*\secondx7,
% y=(1-t)^3*\secondy4+3*t*(1-t)^2*\secondy5+3*t^2*(1-t)*\secondy6+t^3*\secondy7,
  coordinate \second;
  \second1=(0,0);
  \second2=($(\first1)!(\first2)!(\first4)!-1!(\first2)$);
  \second3=($(\first1)!(\first3)!(\first4)!-1!(\first3)$);
  \second4=(\first4);
  \second5=($(\first1)!(\first5)!(\first4)!-1!(\first5)$);
  \second6=($(\first1)!(\first6)!(\first4)!-1!(\first6)$);
  \second7=(\first7);
}

% 画红色曲线，顺便定义一个坐标点 (labelfirst)
  \draw [red](\first1)..controls(\first2)and(\first3)..(\first4)..controls(\first5)and(\first6)..(
  ↪  \first7) coordinate [pos=0.7] (labelfirst);
% 画青色曲线，顺便定义一个坐标点 (labelsecond)
  \draw [cyan](\second1)..controls(\second2)and(\second3)..(\second4) coordinate [pos=0.2]
  ↪  (labelsecond)..controls(\second5)and(\second6)..(\second7);

%%%%%%%% 黑色粗曲线的第一部分
% 设置偏移尺寸
```

```
\tikzmath{
  coordinate \pianyiA;
  \pianyiA2=(-0.21,-0.1);
  \pianyiA3=(0,-0.2);
  \pianyiA4=(0,-0.2);
% 设置黑色粗曲线的第一部分的控制点
  coordinate \thirdA;
  \thirdA1=(0,0);
  \thirdA2=(\first2)+(\pianyiA2);
  \thirdA3=(\first3)+(\pianyiA3);
  \thirdA4=(\first4)+(\pianyiA4);
}
% 规定黑色粗曲线的第一部分，定义其右端点 (duandianA)
  \path plot[variable=\t,domain=0:0.9,samples=50] (
    {3*\t*(1-\t)^2*(\thirdAx2)+3*\t^2*(1-\t)*(\thirdAx3)+\t^3*(\thirdAx4)},
    {3*\t*(1-\t)^2*(\thirdAy2)+3*\t^2*(1-\t)*(\thirdAy3)+\t^3*(\thirdAy4)}) coordinate
    ↪ (duandianA);

%%%%%%%%% 黑色粗曲线的第三部分
% 设置偏移尺寸
\tikzmath{
  coordinate \pianyiB;
  \pianyiB2=($(-0.1,0.1)!(\pianyiA4)!(0.1,-0.1)!-1!(\pianyiA4)$);
  \pianyiB3=($(-0.1,0.1)!(\pianyiA3)!(0.1,-0.1)!-1!(\pianyiA3)$);
  \pianyiB4=($(-0.1,0.1)!(\pianyiA2)!(0.1,-0.1)!-1!(\pianyiA2)$);
% 设置黑色粗曲线的第三部分的控制点
  coordinate \thirdC;
  \thirdC1=(\second4)+(\pianyiB2);
  \thirdC2=(\second5)+(\pianyiB3);
  \thirdC3=(\second6)+(\pianyiB4);
  \thirdC4=(\second7);
}
% 规定黑色粗曲线的第三部分的左端点 (duandianB)
  \path plot[variable=\t,domain=0:0.1] (
    {(1-\t)^3*\thirdCx1+3*\t*(1-\t)^2*\thirdCx2+3*\t^2*(1-\t)*\thirdCx3+\t^3*\thirdCx4},
    {(1-\t)^3*\thirdCy1+3*\t*(1-\t)^2*\thirdCy2+3*\t^2*(1-\t)*\thirdCy3+\t^3*\thirdCy4})
    ↪ coordinate (duandianB);

% 画黑色粗曲线
  \draw [line width=0.8pt]
    plot[variable=\t,domain=0:0.9,samples=50] (
      {3*\t*(1-\t)^2*(\thirdAx2)+3*\t^2*(1-\t)*(\thirdAx3)+\t^3*(\thirdAx4)},
      {3*\t*(1-\t)^2*(\thirdAy2)+3*\t^2*(1-\t)*(\thirdAy3)+\t^3*(\thirdAy4)})
    to [bend right] (duandianB)
    --plot[variable=\t,domain=0.1:1,samples=50] (
      {(1-\t)^3*\thirdCx1+3*\t*(1-\t)^2*\thirdCx2+3*\t^2*(1-\t)*\thirdCx3+\t^3*\thirdCx4},
      {(1-\t)^3*\thirdCy1+3*\t*(1-\t)^2*\thirdCy2+3*\t^2*(1-\t)*\thirdCy3+\t^3*\thirdCy4});

  \spy [green] on (duandianA) in node at(3.5,1);
  \spy [green] on (duandianB) in node at(1,3);

\end{tikzpicture}
```

上面代码中的最后一个 `\draw` 命令绘制这条黑色粗曲线, 可见曲线在右尖点处是 line-to 的, 不是 move-to 的, 所以不应该出现裂缝。在这个 `\draw` 命令的方括号选项中添加选项 `line join=round` 可以填充这个裂缝；使用选项 `miter limit=55` 也可以填充这个裂缝，但结果令人意外，参考 `/tikz/line join` →P. 68, `/tikz/miter limit` →P. 69.

64 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. 779 等。

TikZ 会自动加载这个库。

用 \pgfsetmovetofirstplotpoint→P. 779 或 \pgfsetlinetofirstplotpoint→P. 779 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
```

64.1 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. 780.

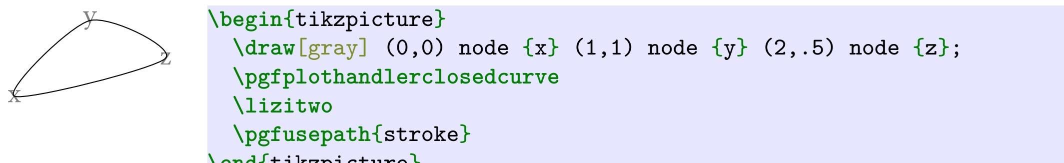
```
\begin{tikzpicture}
  \draw[green] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlercurveto
  \lizitwo
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfsetplottension{⟨value⟩}`

本命令的作用如前述，默认值是 0.5. 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。

`\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}
```

64.2 Constant 图柄

`\pgfplothandlerconstantlineto`

这个图柄的作用类似 `plot[const plot]`.

```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizione
  \pgfplothandlerconstantlineto
  \lizione
  \pgfusepath{stroke}
\end{tikzpicture}
```

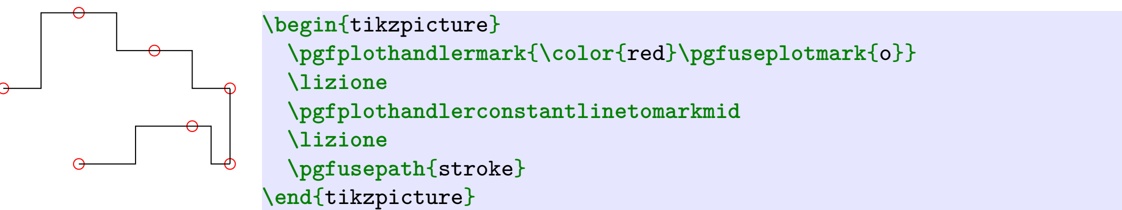
`\pgfplothandlerconstantlinetomarkright`

这个图柄的作用类似 `plot[const plot mark right]`.

```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizione
  \pgfplothandlerconstantlinetomarkright
  \lizione
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplothandlerconstantlinetomarkmid`

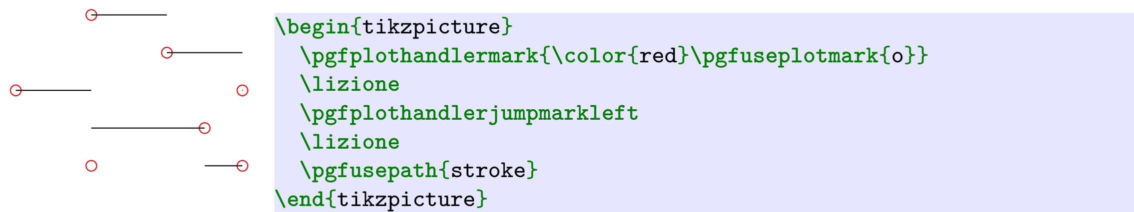
这个图柄的作用类似 `plot[const plot mark mid]`.



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizione
  \pgfplothandlerconstantlinetomarkmid
  \lizione
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplothandlerjumpmarkleft`

这个图柄的作用类似 `plot[jump mark left]`.



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizione
  \pgfplothandlerjumpmarkleft
  \lizione
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplothandlerjumpmarkright`

这个图柄的作用类似 `plot[jump mark right]`.

`\pgfplothandlerjumpmarkmid`

这个图柄的作用类似 `plot[jump mark mid]`.

64.3 Comb 图柄

`\pgfplothandlerxcomb`

这个图柄的作用类似 `plot[xcomb]`.

`\pgfplothandlerycomb`

这个图柄的作用类似 `plot[ycomb]`.

`\pgfplothandlerpolarcomb`

这个图柄的作用类似 `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：

3 月
2 月
1 月

```
\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：

3 月
2 月
1 月

```
\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。

64.4 Bar 图柄

`\pgfplothandlerybar`

这个图柄的作用类似 plot[ybar]。

```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizitwo
\pgfplotxzerolevelstreamconstant{1cm}
  \pgfplothandlerybar
  \lizitwo
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplothandlerxbar`

这个图柄的作用类似 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`

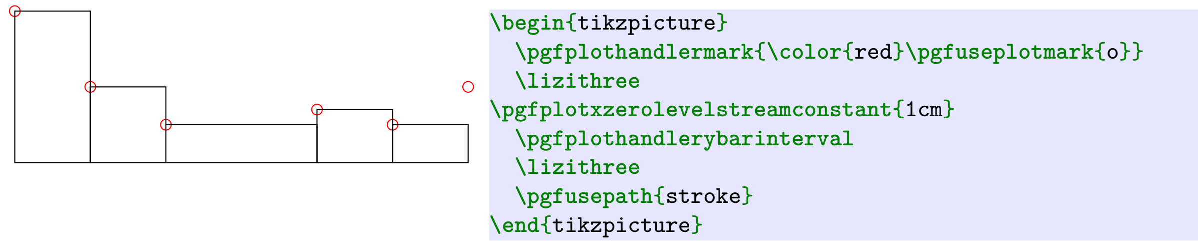
这个选项的作用类似平移选项 `xshift`，`yshift`，但只对柱状图中的 bar 有效。对于 `ybar` 图形，`bar shift` 指的是各个 bar 在 x 轴方向的平移；对于 `xbar` 图形，`bar shift` 指的是各个 bar 在 y 轴方向的平移。⟨*dimension*⟩ 可以是带有长度单位的表达式，会被数学解析器解析。

`\pgfplotbarwidth`

这个宏的展开值是选项 `/pgf/bar width` 的值。

`\pgfplothandlerybarinterval`

这个图柄的作用类似 `plot[ybar interval]`.



```
\begin{tikzpicture}
  \pgfplothandlermark{\color{red}\pgfuseplotmark{o}}
  \lizithree
\pgfplotxzerolevelstreamconstant{1cm}
  \pgfplothandlerybarinterval
  \lizithree
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplothandlerxbarinterval`

这个图柄的作用类似 `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 $x_i + \langle factor\rangle \cdot (x_{i+1} - x_i)$；矩形宽度（水平方向的尺寸）是 $\langle scale\rangle \cdot (x_{i+1} - x_i)$；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 $y_i + \langle factor\rangle \cdot (y_{i+1} - x_i)$；矩形宽度（竖直方向的尺寸）是 $\langle scale\rangle \cdot (y_{i+1} - y_i)$；矩形的“水平长度”是 $\|x_i\|$；图流的最后一个点“落单”。

64.5 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}
```

64.6 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*⟩}

这个命令的效果类似 `plot[mark repeat=`⟨*r*⟩`]`.

\pgfsetplotmarkphase{⟨*phase*⟩}

这个命令的效果类似 `plot[mark phase=`⟨*p*⟩`]`.

\pgfplothandlermarklisted{⟨*mark code*⟩}{⟨*index list*⟩}

这个命令的效果类似 `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。

y ⓩ ⓧ

```
\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。

y z x

```
\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}
```

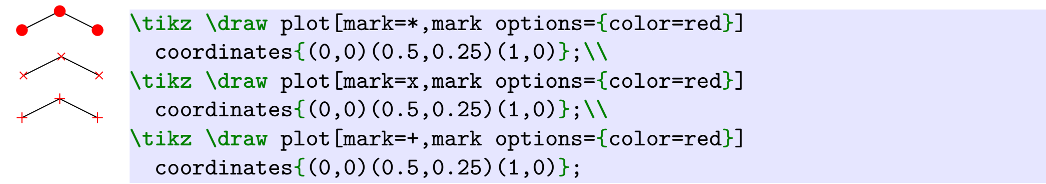
\pgfsetplotmarksize{⟨*dimension*⟩}

这个命令的作用类似 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)};
```

65 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。

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)

这个选项设置点标记的颜色，颜色是填充的。如果选项值留空，则无填充色。如果选项值是 none，则取消填充。有的点标记只能填充其内部的一半区域。

/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)

这个选项决定由选项 text mark={⟨*text*⟩} 给出的点标记 ⟨*text*⟩ 是否转为 node.

/pgf/text mark style={⟨*options for* `mark=text`⟩} (no default)

这个选项设置由选项 `text mark={⟨text⟩}` 给出的点标记 ⟨*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。

23 透明度

23.1 Overview

通常，TikZ 画出的图形都是不透明的。

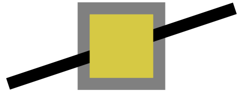
pdfTEX 对透明度效果的支持最好。

23.2 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)

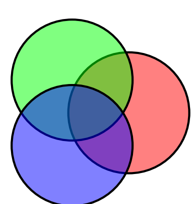
这个选项设置路径线条的 “不透明度”，若 ⟨*value*⟩ 是 1，则线条完全不透明；若 ⟨*value*⟩ 是 0，则线条完全透明，即不可见。



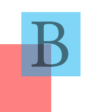
```
\begin{tikzpicture}[line width=1ex]
  \draw (0,0) -- (3,1);
  \filldraw [fill=yellow!80!black,draw opacity=0.5] (1,0) rectangle
  ↪ (2,1);
\end{tikzpicture}
```

/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。

23.3 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。

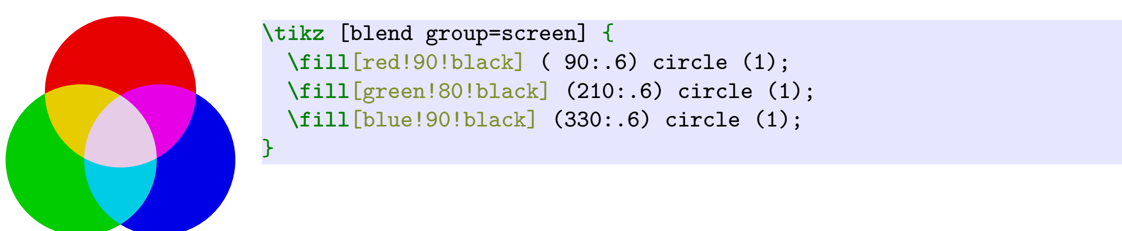
如果这个选项用作环境选项，则对环境内的各个图形有效，即按混色模式画出图形的重叠部分。如果这个选项用作某个绘图命令的选项，则该命令绘出的图形与其它图形出现混色效果。

```
\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)

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。

23.4 颜色淡入、淡出——fading

颜色的淡入、淡出指的是颜色透明度的平滑过渡，即 fading，soft masks，opacity masks，masks，soft clips.

参考 §110.

23.4.1 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，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}`

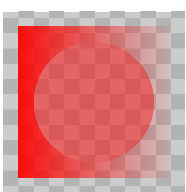
这个环境定义灰度图，但不可见，环境中的绘图命令是通常的绘图命令。

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：

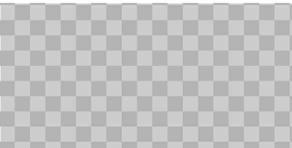
`/tikz/name=`⟨*name*⟩ (no default)

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

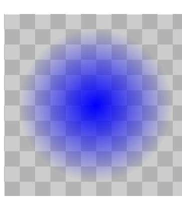
The ConTEXt version of the environment.

\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。



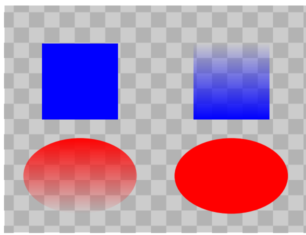
```
  % 规定灰度的变化
\tikzfading[name=fade out,
  inner color=transparent!0,
  outer color=transparent!100]
  % 下面在 {tikzpicture} 环境中画出 fading 图
\begin{tikzpicture}
  \fill [black!20] (-1.2,-1.2) rectangle (1.2,1.2);
  \path [pattern=checkerboard,pattern color=black!30]
    (-1.2,-1.2) rectangle (1.2,1.2);
  \fill [blue,path fading=fade out] (-1,-1) rectangle (1,1);
\end{tikzpicture}
```

23.4.2 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}
```

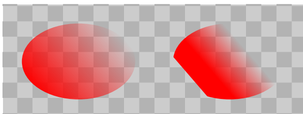
`/tikz/fit fading`=⟨*boolean*⟩ (default true, initially true)

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。

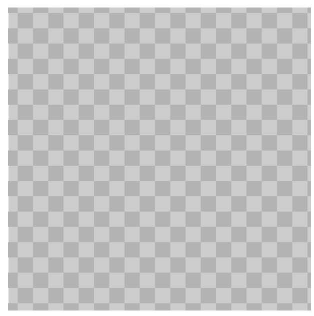


```
\begin{tikzpicture}[path fading=south]
  \fill [black!20] (0,0) rectangle (4,1.5);
  \path [pattern=checkerboard,pattern color=black!30]
    (0,0) rectangle (4,1.5);
  \fill [red,path fading,fading transform={rotate=120}]
    (1,0.75) ellipse (.75 and .5);
  \fill [red,path fading,fading transform={rotate=120,yscale=0.4}]
    (3,0.75) ellipse (.75 and .5);
\end{tikzpicture}
```

`/tikz/fading angle`=⟨*degrees*⟩ (no default)

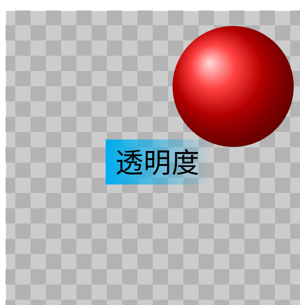
等价于 `fading transform={rotate=`⟨*degrees*⟩`}`.

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}
```

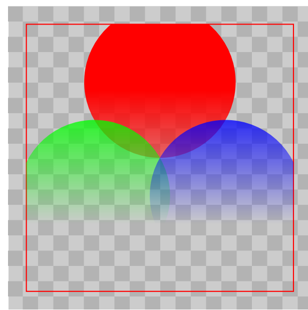
23.4.3 Fading a Scope

`/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。

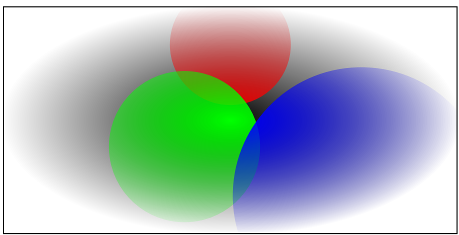
对于带有 `scope fading` 选项的路径来说，选项 `fit fading` 和 `fading transform` 默认是有效的，也就是说，带有 `scope fading` 选项的路径决定灰度图的尺寸、位置，从而决定了在图形的什么地

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。



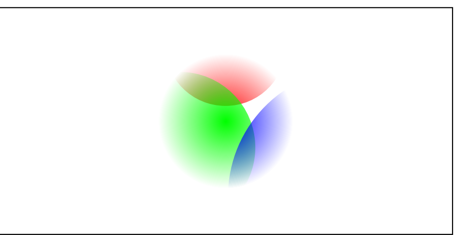
```
\begin{tikzpicture}
  \tikzfading[name=fade out,inner color=transparent!0,outer color=transparent!100]
  \begin{scope}[overlay]
  \fill [scope fading=fade out,inner color=cyan,outer color=black]
    (-3,-1.5) rectangle (3,1.5);
  \fill[red] ( 90:1) circle (0.8);
  \fill[green] (210:0.7) circle (1);
  \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}
```

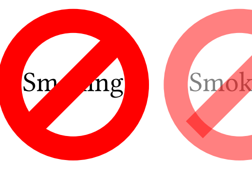
如果 `scope fading` 用作 node 的选项，则 node 的背景和文字都具有 fading 效果，这与选项 `path fading` 不同。

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.

```
\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.
};
```

23.5 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。

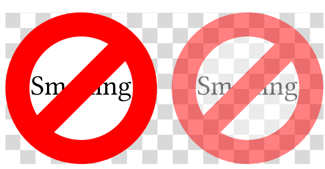



```
\begin{tikzpicture}
\node [forbidden sign,line width=2ex,draw=red,fill=white]
  at (0,0) {Smoking};
\node [opacity=.5,forbidden sign,line width=2ex,draw=red,fill=white]
  at (2.2,0) {Smoking};
\end{tikzpicture}
```

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.

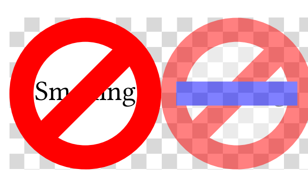
`/tikz/transparency group=[⟨options⟩]` (no default)

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。



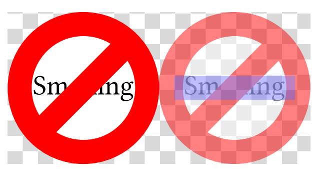

```
\begin{tikzpicture}
  \pattern[pattern=checkerboard,pattern color=black!15]
    (-1,-1) rectangle (3.2,1);
  \node [forbidden sign,line width=2ex,draw=red,fill=white]
    at (0,0) {Smoking};
  \begin{scope}[transparency group,opacity=.5]
    \node [forbidden sign,line width=2ex,draw=red,fill=white]
      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}
```

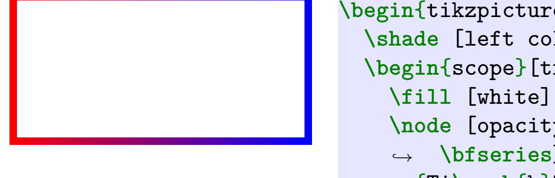
观察下面图形：

```
\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。

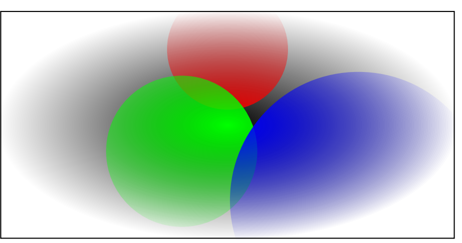
`isolated=false` 在默认下，transparency group 是被隔离的。将 ⟨*options*⟩ 设为 `isolated=false` 则取消隔离，详情参考 PDF Reference, six edition, §7.3.4.

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。

23.6 遇到的问题

观察下面图形与上文的间距。先看使用命令 \tikzfading 的情况：

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

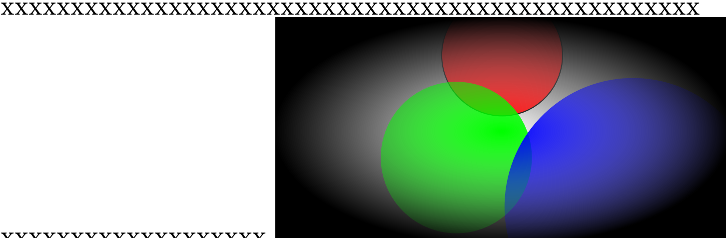
xxxxxxxxxxxxxxxxxxxxxxx  xxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxx
\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] (-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}
  \path[draw,use as bounding box] (-3,-1.5) rectangle (3,1.5);
\end{tikzpicture}
xxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

再看使用 {tikzfadingfrompicture} 环境的情况：

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxx  xxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxx
\begin{tikzpicture}
  \begin{tikzfadingfrompicture}[name=Fade Out]
    \shade [inner color=transparent!0,outer color=transparent!100] (-3,-1.5) rectangle (3,1.5);
  \end{tikzfadingfrompicture}
  \shade [inner color=transparent!0,  outer color=transparent!100] (-3,-1.5) rectangle (3,1.5);
  \begin{scope}[overlay,fit fading=false]
    \filldraw[red,draw=black,scope fading=Fade Out] ( 90:1) circle (0.8);
```

```
    \fill[green] (210:0.7) circle (1);
    \fill[blue] (330:2) circle (1.7);
  \end{scope}
\end{tikzpicture}
xxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

53 fadings 库

fadings 库，即文件《pgflibraryfadings.code.tex》（这个文件只有 58 行代码），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 $(25-21.5)/25=15\%$.

24 装饰路径

24.1 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

当调用任何一个以 decorations 为名称前缀的库后，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：

Saved from trash

```
\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：

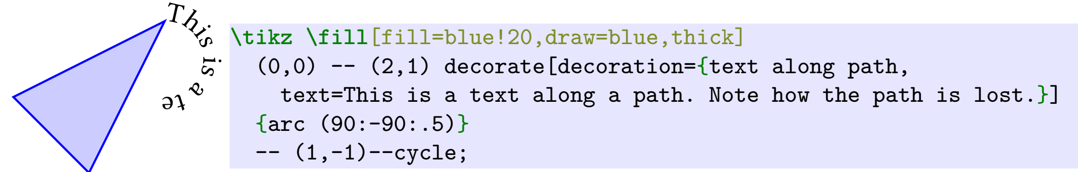
1. path morphing，即路径变体，这种装饰线型由程序库 decorations.pathmorphing 提供，例如 zigzag，bumps，coil，snake，saw 等线型。

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程序“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. 663 时，应该清楚，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。

24.2 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，等的。可以在 ⟨*subpath*⟩ 中使用 node，但 node 只是添加到 ⟨*subpath*⟩ 上的，不属于该 ⟨*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 是与此类型相配的：

around

```
\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。

24.3 装饰整个路径

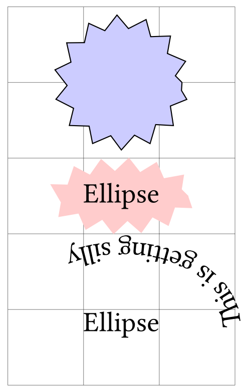
下面两个形式等效：

```
\path decorate[⟨options⟩]{⟨path⟩};
\path [decorate,⟨options⟩] \meta{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)

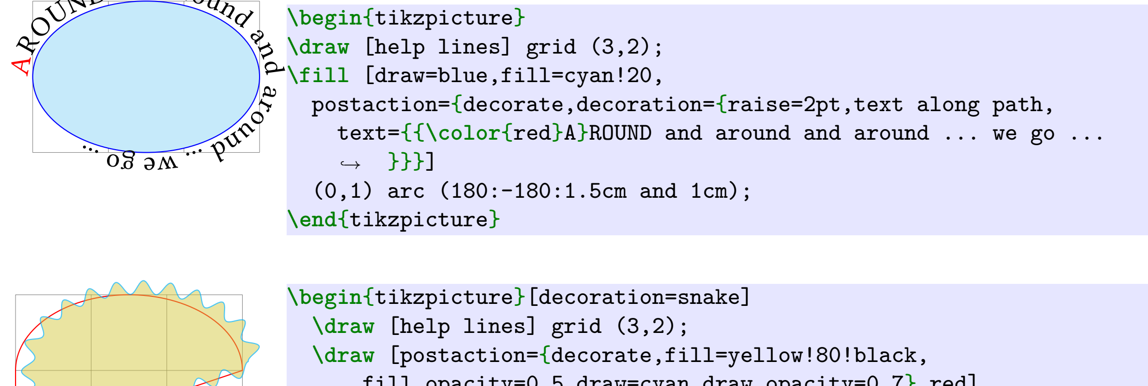
当某个路径带有整个选项时，对该路径启动装饰功能，但具体怎么装饰还得用选项 decoration 指定，否则没有装饰。使用该选项两次不意味着套嵌装饰操作，因为作为选项的 decorate 只是意味着逻辑值 true，这与作为操作（动词）的 decorate 不一样。




```
\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}
```

24.4 调整装饰路径的外观

24.4.1 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 TikZ 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。

24.4.2 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），而两点间的联系方式由 ⟨*decoration*⟩ 指定，两点间的联系长度由选项 `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}
```

50 Decoration 库

50.1 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/`，因此它们都作选项 `decoration={`⟨*options*⟩`}` 的值。

/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。

在程序库 decorations.pathmorphing 的源文件《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。

50.2 修饰路径的装饰类型

TikZ Library `decorations.pathmorphing`

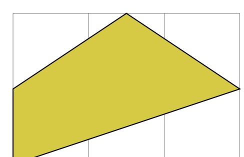
```
\usepgflibrary{decorations.pathmorphing} % LaTeX and plain TeX and pure pgf
\usepgflibrary[decorations.pathmorphing] % ConTeXt and pure pgf
\usetikzlibrary{decorations.pathmorphing}
↪ % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.pathmorphing] % ConTeXt when using TikZ
```

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。

50.2.1 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}
```

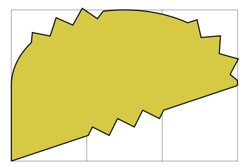
Decoration `straight zigzag`

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）。



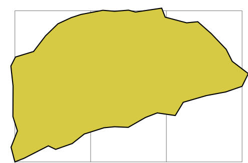
```
\begin{tikzpicture}[decoration={straight zigzag,
    meta-segment length=1.1cm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

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。



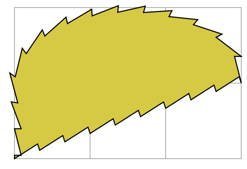
```
\begin{tikzpicture}
    [decoration={random steps,segment length=2mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

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。



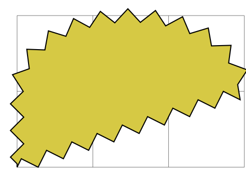
```
\begin{tikzpicture}[decoration=saw]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

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}
```

50.2.2 由曲线构成的装饰路径

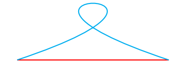
Decoration bent

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：

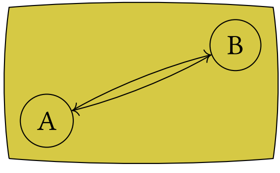
⟨*current point*⟩..controls (t*r, a) and ((1-t)*r, a)..(r,0)

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}
```



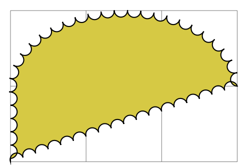

```
\begin{tikzpicture}[decoration={bent,aspect=.3}]
  \draw [decorate,fill=yellow!80!black]
    (0,0) rectangle (3.5,2);
  \node[circle,draw] (A) at (.5,.5) {A};
  \node[circle,draw] (B) at (3,1.5) {B};
  \draw[->,decorate] (A) -- (B);
  \draw[->,decorate] (B) -- (A);
\end{tikzpicture}
```

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}
```

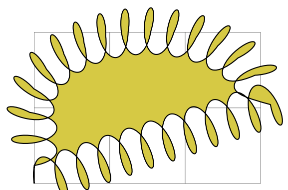
Decoration coil

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。



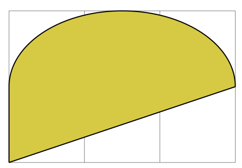
```
\begin{tikzpicture}[decoration={coil,aspect=0.4,
    segment length=3mm,amplitude=3mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

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，不要被它的名称“curve”误导。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。



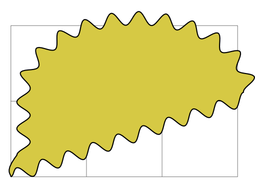
```
\begin{tikzpicture}[decoration=curveto]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

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}
```

50.3 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。

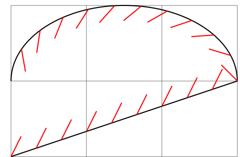
Decoration border

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。



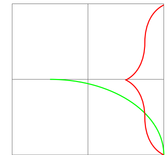
```
\begin{tikzpicture}[decoration={border,amplitude=3mm}]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate,draw,red}]
    (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

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.



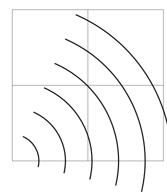
```
\begin{tikzpicture}[decoration={brace,amplitude=5mm}]
  \draw [help lines] grid (-2,2);
  \draw [postaction={decorate,draw,red}][green]
    (0,0) arc (0:90:1.5 and 1);
\end{tikzpicture}
```

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

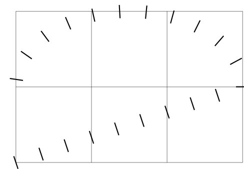
这个装饰类型是 decoration 模块定义的,它直接跳到被装饰路径的终点,常用在选项 pre=moveto 或 post=moveto 中。

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}
```

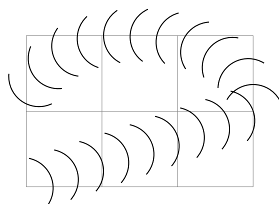
Decoration waves

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。

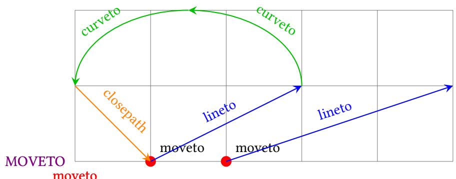
radius 这个选项的值是圆弧的半径。



```
\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 TikZ 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 TikZ 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 TikZ 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 TikZ 命令中。

50.4 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。

50.5 自选标记装饰

50.5.1 程序库 decorations.markings

TikZ 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。

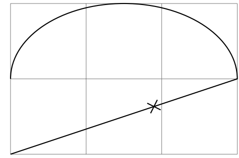
Decoration markings

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

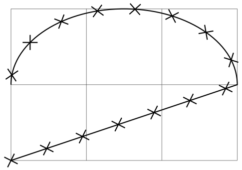
```
\draw (-2pt,-2pt) -- (2pt,2pt);
\draw (2pt,-2pt) -- (-2pt,2pt);
```

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”，tikz 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*⟩ 用于决定标记在路径上的位置，⟨*pos*⟩ 可以有 4 种形式：

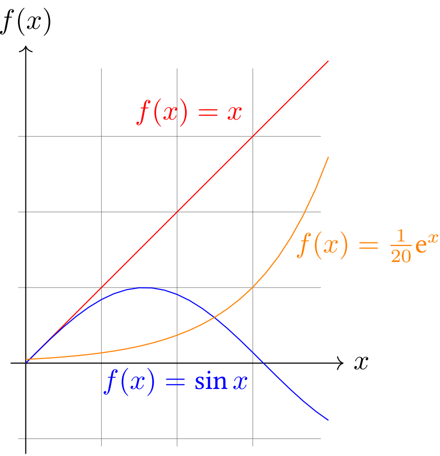
1. ⟨*pos*⟩ 可以是非负值的尺寸，例如 `0pt`，或者 `5cm/2`. 此时程序会从被装饰路径的起点开始，沿着被装饰路径行进 ⟨*pos*⟩ 指定的长度从而确定点 p，这个点 p 就是放置标记的位置。
2. ⟨*pos*⟩ 可以是负值的尺寸，例如 `-2pt`，或者 `-1sp`. 此时程序会从被装饰路径的终点开始，沿着被装饰路径的反方向行进 ⟨*pos*⟩ 指定的尺寸长度从而确定点 p，这个点 p 就是放置标记的位置。
3. ⟨*pos*⟩ 可以是非负数字或数字运算表达式，例如 `1/2`，或者 `0.33+2*0.1`. 此时程序会从被装饰路径的起点开始计算其总长度 L，然后由 ⟨*pos*⟩ 与 L 的乘积确定一个长度从而确定一个点 p，这个点 p 就是放置标记的位置。如果 ⟨*pos*⟩ 是 `0.5`，那么标记就放在被装饰路径的中间（$L/2$）处。
4. ⟨*pos*⟩ 可以是负的数字或数字运算表达式，例如`-1/2`，或者 `-0.33-2*0.1`. 此时程序会从被装饰路径的终点开始计算其总长度 L，然后由 ⟨*pos*⟩ 与 L 的乘积确定一个长度从而确定一个点 p，这个点 p 就是放置标记的位置。

注意如果 ⟨*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}
```

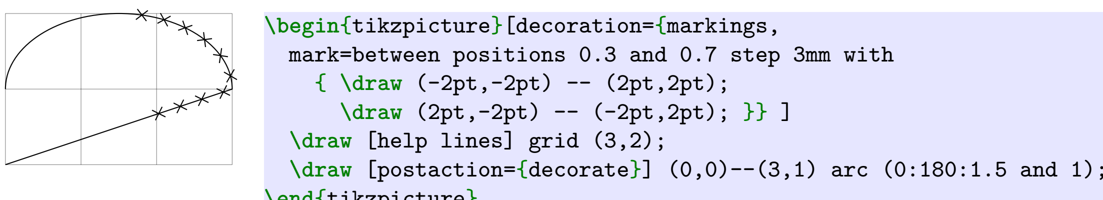
下面的例子展示了如何用 markings 装饰类型给路径加标签。




```
\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}
```

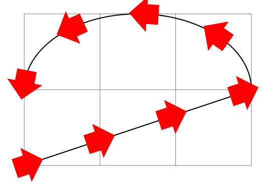
/pgf/decoration/mark=`betweenpositions` ⟨*start pos*⟩ `and` ⟨*end pos*⟩ `step` ⟨*stepping*⟩ `with` ⟨*code*⟩ (no default)

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*⟩. ⟨*start pos*⟩ 和 ⟨*end pos*⟩ 分别指定被装饰路径上的点 P_1 和 P_2，从 P_1 到 P_2 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。



```
\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:

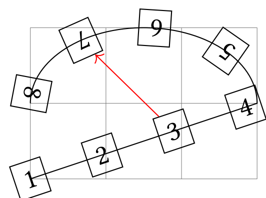


```
\begin{tikzpicture}[decoration={markings,
  mark=between positions 0 and 1 step 1cm with
    {\node [single arrow,fill=red,
      single arrow head extend=3pt, transform shape] {};}}]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0)--(3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

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)

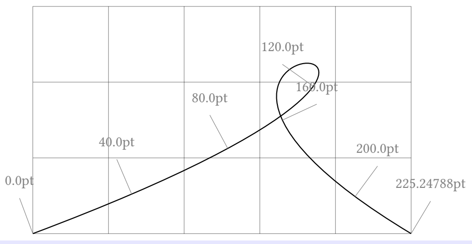
当用前两个 mark 选项为装饰路径设置一个或数个标记时，tikz 会按照这些标记被添加的次序为它们编号。添加的第一个标记的编号是 1，添加的第二个标记的编号是 2⋯⋯当前标记的编号就保存在这个 key 中，可以使用命令 \pgfkeysvalueof{/pgf/decoration/mark info/sequence number} 来引用或者输出当前标记的编号。




```
\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}
```

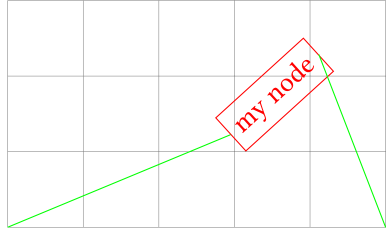
在选项 decoration 的值中可以使用以下选项。

`/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。

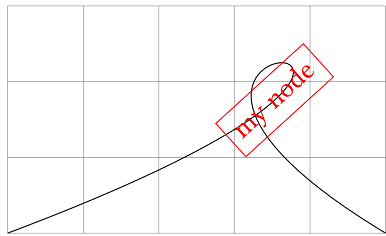
`/pgf/decoration/mark connection node=`⟨*node name*⟩ (no default, initially empty)

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”：

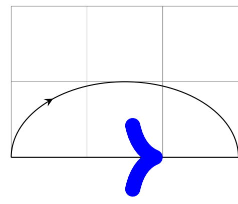



```
\begin{tikzpicture}[decoration={markings,
    mark connection node=my node,
    mark=at position .5 with
    {\node [draw,red,transform shape] (my node) {my node};}}]
  \draw [help lines] grid (5,3);
  \draw [postaction={decorate,green}] (0,0) .. controls (8,3) and (0,3) .. (5,0);
\end{tikzpicture}
```

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*⟩}

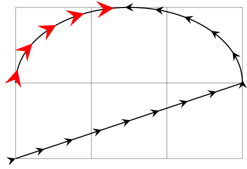
其中 ⟨*arrow end tip*⟩ 是箭头类型的名称，⟨*options*⟩ 是箭头选项。在 ⟨*code*⟩ 的坐标系中，箭头的尖点位于坐标系原点，箭头方向指向右侧，添加箭头后，箭头的尖点位于指定的标记位置点上，箭头方向沿着路径的切线方向。在使用 tikz 的环境、命令时，选项 ⟨*options*⟩ 才有效。箭头和 ⟨*options*⟩ 都会被放入一个 scope 中来执行。



```
\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}
```

50.5.2 脚印标记

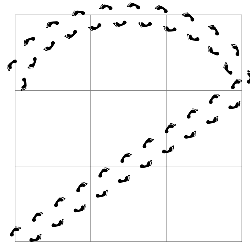
TikZ Library decorations.footprints

```
\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。

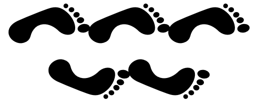


```
\begin{tikzpicture}[decoration={footprints,
    foot length=5pt,stride length=10pt}]
  \draw [help lines] grid (3,3);
  \fill [decorate] (0,0) -- (3,2) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

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，就是“严重外八字脚”。

```
\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,另外还有“矮人”(gnome),“鸟”(bird),“猫”(felis silvestris）这 3 种可选。

```
\begin{tikzpicture}[decoration={footprints,
    foot of=felis silvestris}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

50.5.3 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. 717 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。

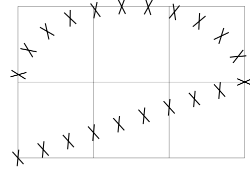
Decoration crosses

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}
```

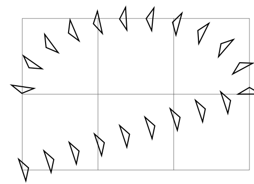
Decoration triangles

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}
```

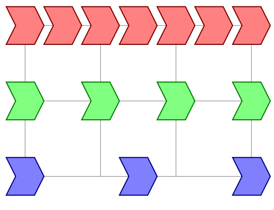
用作装饰的 shape 没有文字，当把装饰 shape 放到路径上之后，它有自己的默认尺寸。针对 shape 的变换有效，但是像 `inner sep`，`minimum size` 这种针对 node 的选项对装饰 shape 无效。

/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`)

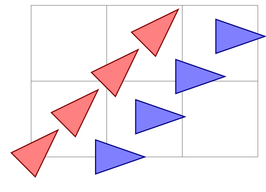
这个选项能取消（overrides）选项 `shape sep` 的设置。本选项在被装饰路径上放置 ⟨*number*⟩ 个装饰 shape，并使它们均匀分布。如果 ⟨*number*⟩ 小于 1，那么没有 shape；如果 ⟨*number*⟩ 等于 1，那么一个 shape 被放在路径中间。还有两个可选的关键词 `by centers` 和 `by borders`，这两个参数确定两个相邻 shape 路径的间距的计算方式，即“中心到中心”和“边界到边界”，默认 `by centers`.

```
\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)

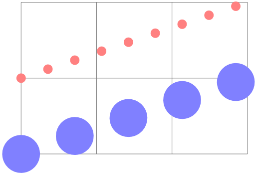
本选项同时设置 `shape end width` 和 `shape end height`.

`/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}
```

50.6 文字装饰

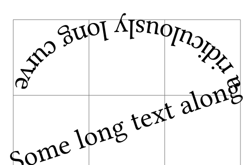
TikZ Library `decorations.text`

```
\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`.

50.6.1 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}
```

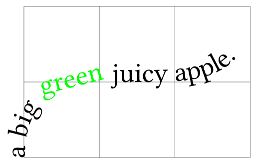
关于这个装饰类型要注意以下几点:

- 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。

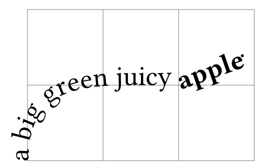
这个装饰类型有以下选项可用。

/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。



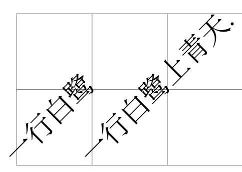

```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate,decoration={text along path,
      text={a big {\color{green}green} juicy apple.}}]
    (0,0) .. controls (0,2) and (3,0) .. (3,2);
\end{tikzpicture}
```




```
\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：

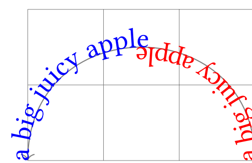
- 定界符的符号类别是 11 或 12；
- 定界符要成对使用；
- 在一对定界符内使用那些设置文字外观的命令（字体、字号、文字颜色等），定界符不能套嵌使用；
- 把某些命令放在一对定界符内，这些命令就对之后的文字起作用；
- 使用一对内容为空（即不含任何命令）的定界符将文字还原为默认状态；
- 加号“+”的作用是把之前一对定界符内的命令添加到“+”所在位置，因此“+”要放在定界符内。

`/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。



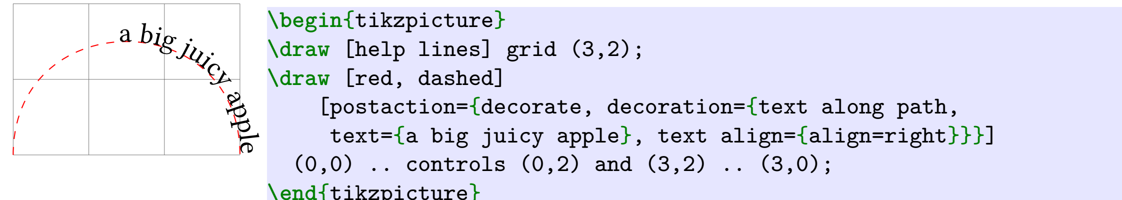

```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [gray, ->]
    [postaction={decorate, decoration={text along path,
     text={a big juicy apple}, text color=red}}]
    [postaction={decorate, decoration={text along path,
     text={a big juicy apple}, text color=blue, reverse path}}]
  (3,0) .. 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)

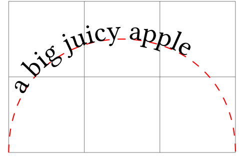
本选项等效于 `/pgf/decoration/text align/align=right`.

`/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`)

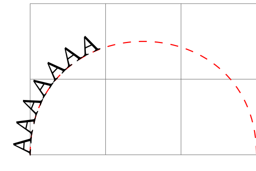
本选项使得文字左侧有一段（沿着路径）长度为 ⟨*length*⟩ 的空白（类似文字的缩进），其效果相当于改变了的被装饰路径的起点位置，即在被装饰路径的开始处裁掉一段路径。




```
\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={AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA},
     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}
```

50.6.2 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}
```

```
\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)

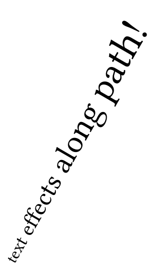
当被装饰路径只含有一个点时 P，将本选项的值设为 `true`，那么 PGF 会计算装饰文字的总宽度 d，并且假设一个水平向右的直线段 $|PQ| = d$，把 PQ 当作被装饰路径，因此装饰文字就沿着水平向右的方向显示。

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)

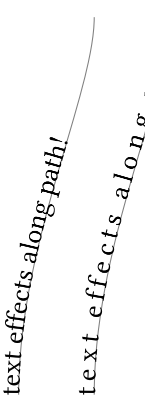
这个选项与前一个选项 `path from text` 配合使用，本选项会使得假想的被装饰直线段 PQ 围绕起点 P 旋转 ⟨*angle*⟩ 角度。




```
\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}
```

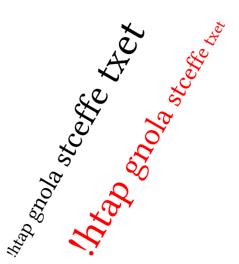
/pgf/decoration/text effects/scale text to path=⟨*true or false*⟩ (default true)

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)

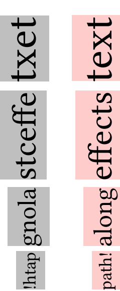
这个选项使得各个字符 node 按照倒序排布，如果原来的装饰文字是“从左向右”阅读的，那么使用本选项后，装饰文字是“从右向左”阅读的，不过 PGF 先将各个字符 node 倒序排布后再对它们做处理，因此各个字符 node 的编号仍然是“从左向右”的，选项的作用也是“从左向右”的。注意，装饰文字开头的‘soft’spaces 会被忽略。




```
\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。



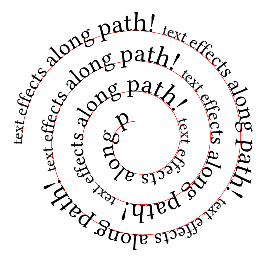

```
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!},
    text effects/.cd,
      path from text, path from text angle=90,
      every word separator/.style={fill=none},
      character count=\i, character total=\n,
      characters={text along path, fill=gray!50, 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*⟩

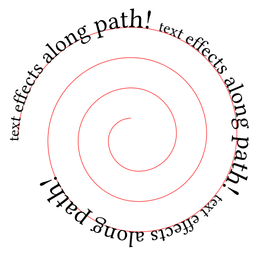
如果被装饰路径过长，而装饰文字过短，那么装饰文字就只能装饰一部分路径，此时可使用这个选项，让装饰文字沿着被装饰路径重复。如果使用选项 `repeat text`，那么装饰文字会不断重复直到把被装饰路径全部装饰完毕。如果使用选项 `repeat text=`⟨*times*⟩，那么在首次添加完毕装饰文字后，再重复添加 ⟨*times*⟩ 次，故装饰文字总共被添加 ⟨*times*⟩ + 1 次。在重复装饰文字时，字符、字母、单词的编号都会重新编排，针对装饰文字的各种选项也会重新计算。

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。

text$_1$ effects$_2$ along$_3$ path!$_4$

```
\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}
```

50.7 分形装饰

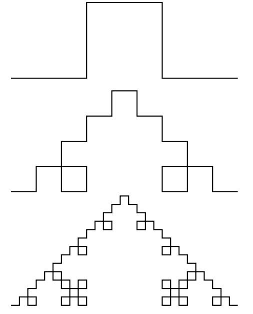
TikZ Library decorations.fractals

```
\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}
```

25 变换

25.1 各种坐标系统

绘图时，提供一个坐标，如 `(1,2)`，然后指定对该坐标的变换，直到最后在屏幕上这个位置，是一个很长的处理过程。其中可能涉及以下操作：

1. PGF 将 `(1,2)` 解释为它的 xy 坐标系统中的位置 P_1.

2. PGF 对 P_1 应用一个变换，得到位置 P_2.
3. 驱动程序（如 `dvips` 或 `pdftex`）将 P_2 变换为 TEX 的页面坐标系中的位置 P_3.
4. PDF （或 PostScript）对 P_3 应用画布变换将它对应到页面上的位置 P_4.
5. 预览器将 P_4 变换为显示器上的像素位置 P_5.

在 TikZ 中只涉及以上步骤中的前两个：xy 坐标系统以及坐标变换矩阵。PGF 也支持画布变换，但是一旦作出画布变换，PGF 就不再正确地计算坐标位置，例如与 node 有关的形状、锚位置，各种与 bounding box 相关的位置，在将来这一点可能会改进。

本节所讲的变换选项对应几何中的仿射变换以及平移，对于平面上的点 $\binom{x}{y}$，将点写成齐次坐标形式，并把变换矩阵写成扩展矩阵形式，可以把仿射变换和平移变换写成一个矩阵乘积形式：

$$\begin{pmatrix} x \\ y \end{pmatrix} \longrightarrow \begin{pmatrix} a & b & s \\ c & d & t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax+by+s \\ cx+dy+t \\ 1 \end{pmatrix} \longrightarrow \begin{pmatrix} ax+by+s \\ cx+dy+t \end{pmatrix},$$

其中 $\left(\begin{smallmatrix} a & c \\ b & d \\ 0 & 0 \end{smallmatrix}\right)$ 的作用是个仿射变换，而 $\left(\begin{smallmatrix} s \\ t \\ 1 \end{smallmatrix}\right)$ 的作用是平移。但实际上，矩阵 $\left(\begin{smallmatrix} a & c \\ b & d \end{smallmatrix}\right)$ 可以是退化矩阵。

25.1.1 变换选项的作用次序以及作用方式

各种变换选项选项可以用作环境选项，也可以做绘图命令选项。环境选项中的变换选项会被加到每个绘图命令的方括号选项序列中，并且排在方括号内的其它（命令本身带有的）变换选项之前。各个变换选项按顺序依次起作用，变换效果是累计的。例如

$(0,0)$

```
\begin{tikzpicture}[scale=0.8]
  \draw [help lines,dotted] (-1,0) grid (1,2);
  {[rotate=30]
    \draw [yshift=1.2cm] [blue] (0,0) rectangle (1,0.5);
  }
  \fill circle (2pt) node [below]{$(0,0)$};
\end{tikzpicture}
```

把上面代码中第一个 `\draw` 绘制的网格看作是不变的参照系。代码中的选项 `scale=0.8` 是 `{tikzpicture}` 环境的选项，其作用是将图形缩小为原来的 0.8 倍（以原点为位似中心）；`rotate=30` 是 `{scope}` 环境的选项，其作用是将图形绕原点旋转 30°；`yshift=1.2cm` 是第二个 `\draw` 自带的选项，其作用是将图形沿着 y 轴方向平移 1.2cm. 这些选项都会按次序传递给第二个 `\draw`，所以第二个 `\draw` 中的变换选项就是"`scale=0.8`，`rotate=30`，`yshift=1.2cm`"，下面猜测一下这些选项的作用过程。

把默认的初始坐标系——标架 U 写成矩阵形式：$U = \left(\begin{smallmatrix} \text{1cm} & \text{0cm} & \text{0cm} \\ \text{0cm} & \text{1cm} & \text{0cm} \\ 0 & 0 & 1 \end{smallmatrix}\right)$，其中第一列、第二列分别是标架 U 的第一、第二基向量，第 3 列是标架 U 的原点坐标。设绘图命令有 n 个变换选项 $\langle opt\, i \rangle$，$i = 1, 2, \cdots, n$，页面上的点 **P** 在标架 U 下的坐标是 P_U，这些变换选项把点 **P** 变换为页面上的点 **P′**，点 **P′** 的坐标（在标架 U 下）记为 P'_U.

先给出的变换先作用，后给出的变换后作用。猜测变换的对象是标架，而不是坐标数据。下面用同一符号来表示变换及变换对应的矩阵。把第 i 个变换选项确定的变换（矩阵）记为 T_i，先计算 $U \cdot T_1$，再计算 $U \cdot T_1 \cdot T_2$，……最后得到 $\overline{U} = U \cdot T_1 \cdots T_n$，然后在标架 $\overline{U}$ 中描出坐标为 P_U 的点。这个计算过程表达为：

$$UT_1 \cdots T_n \cdot P_U = \overline{U} \cdot P_U.$$

以上面的图形为例，首先选项 `scale=0.8` 作用，然后 `rotate=30` 作用，最后 `yshift=1.2cm` 作用。

首先，选项 `scale=0.8` 将默认标架的基向量的长度变成原来的 0.8 倍，得到的标架记为 C_1，在标架 U 之下表达 C_1 如下：

$$C_1 = U \cdot T_1 = \begin{pmatrix} 0.8\text{cm} & 0\text{cm} & 0\text{cm} \\ 0\text{cm} & 0.8\text{cm} & 0\text{cm} \\ 0 & 0 & 1 \end{pmatrix}.$$

然后选项 `rotate=30` 的作用得到标架 C_2，在标架 U 之下表达 C_2 如下：

$$C_2 = C_1 \cdot T_2 = \begin{pmatrix} \frac{\sqrt{3}}{5}\text{cm} & \frac{-1}{5}\text{cm} & 0\text{cm} \\ \frac{1}{5}\text{cm} & \frac{\sqrt{3}}{5}\text{cm} & 0\text{cm} \\ 0 & 0 & 1 \end{pmatrix},$$

这个变换的意思是将 C_1 绕 C_1 的原点旋转 30°.

然后选项 `yshift=1.2cm` 的作用得到标架 C_3，在标架 U 之下表达 C_3 如下：

$$C_3 = C_2 \cdot T_3 = \begin{pmatrix} \frac{\sqrt{3}}{5}\text{cm} & \frac{-1}{5}\text{cm} & 0\text{cm} \\ \frac{1}{5}\text{cm} & \frac{\sqrt{3}}{5}\text{cm} & 1.2\text{cm} \\ 0 & 0 & 1 \end{pmatrix},$$

这个变换的意思是将 C_2 沿着它的 y 轴方向平移 1.2cm，得到标架 C_3.

最后，在标架 C_3 中画出路径 `(0,0) rectangle (1,0.5)`，标架变换步骤如下所示：

$U \longrightarrow C_1\ U \longrightarrow C_2\ U \longrightarrow C_3\ U \longrightarrow C_3\ U$

总结起来，变换过程的基本思路是：(a) 先给出的变换先作用，后给出的变换后作用；(b) 变换的对象是标架；(c) 变换的参照系不是固定不变的；(d) 最后一个变换作用后所得到的标架用于绘图。在平面上，3 个不共线的点 A, B, C 可以作为一个 2 维标架，对这 3 个点做变换就让相应的对标架也发生变化，下面提到对标架做变换时指的是对构成标架的点做变换。

如果一个绘图命令的变换选项分属数个方括号，例如

```
\draw [shift={(1,2)}] [x={(-60:1)},y={({-sqrt(3)},-2)}] [shift={(-1,-2)}] ...;
```

那么这些选项仍然按次序起作用。

25.1.2 各种标架及其作用

TikZ 中使用的 xy 坐标系统和 xyz 坐标系统类似“仿射标架”，由基向量 v_x, v_y, v_z 和某个点 O（作为原点）构成。坐标 `(1,2,3)` 就代表点 $O + v_x + 2v_y + 3v_z$. 当给出一个坐标数据 `(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)` 代表页面上的不同点。对坐标点的变换也要参照标架进行。

借用 3 个名词：固定标架、平动标架、固连标架，这 3 个概念原本是描述刚体运动的：

(1) 固定标架：当绘图命令工作时，假想命令会先创建一个固定标架 U：

$$U = \{u_1, u_2; O\} = \begin{pmatrix} 1\text{cm} & 0\text{cm} & 0\text{cm} \\ 0\text{cm} & 1\text{cm} & 0\text{cm} \\ 0 & 0 & 1 \end{pmatrix},$$

其中 O 是页面上的某个点，把 O 作为标架 U 的原点，它的的坐标是 (0,0)；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，即点 O 在页面上的位置，基向量的方向、长度都是不变的。固定标架 U 是单位正交标架。

在绘图命令工作之初，固定标架 U 被作为默认的初始标架，是第一个变换所参照的标架，也是第一个变换的变换对象。

在 U 内坐标为 $\alpha_U = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$ 的点可以表示为：

$$U \cdot \alpha_U = \begin{pmatrix} 1\text{cm} & 0\text{cm} & 0\text{cm} \\ 0\text{cm} & 1\text{cm} & 0\text{cm} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x\text{cm} \\ y\text{cm} \\ 1 \end{pmatrix}.$$

(2) 平动标架：设点 P 是页面上任意一点，在标架 U 下点 P 的坐标是 $\begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$. 以标架 U 为参照系，以 $\overrightarrow{OP} = \begin{pmatrix} p_1 \\ p_2 \\ 1 \end{pmatrix}$ 为平移向量，对 U 做一个平移所得到的标架 $V_P = \{u_1, u_2; P\}$ 是一个平动标架，记号 V_P 的下标指示该标架的原点。可以假想在页面上任意一点处都有一个平动标架，平动标架之间的区别仅在于标架原点的位置不同。固定标架 U 也可以看作是平动标架 V_O，即平移向量为 0 向量时的平动标架。

在固定标架 U 下，平动标架 V_P 的基矩阵是 $V_P = \begin{pmatrix} 1\text{cm} & 0\text{cm} & p_1\text{cm} \\ 0\text{cm} & 1\text{cm} & p_2\text{cm} \\ 0 & 0 & 1 \end{pmatrix}$，若页面上的点 A 在 V_P 内的坐标为 $\alpha_{V_P} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$，则点 A 可以表达为：

$$V_P \cdot \alpha_{V_P} = \begin{pmatrix} 1\text{cm} & 0\text{cm} & p_1\text{cm} \\ 0\text{cm} & 1\text{cm} & p_2\text{cm} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x\text{cm} + p_1\text{cm} \\ y\text{cm} + p_2\text{cm} \\ 1 \end{pmatrix}.$$

(3) 固连标架：(i) 设点 P 是页面上任意一点，以点 P 处的平动标架 V_P 为参照标架对 V_P 自己做一个变换，所得到标架 $W_Q = \{w_1, w_2; Q\}$ 是个固连标架，其中 Q 代表 W_Q 的原点。W_Q 与 V_P 的原点可能重合，也可能不重合，W_Q 可能是正交标架，也可能不是正交标架，也可能是退化的标架——这都取决于对 V_P 做的变换。任意一个平动标架可以看作是固连标架，即恒等变换下的固连标架。(ii) 设 W_P 是点 P 处的一个固连标架，以 W_P 为参照标架对 W_P 自己做一个变换，所得到标架 $W_Q = \{w'_1, w'_2; Q\}$ 也是个固连标架，其中 Q 代表 W_Q 的原点。

在点 P 处的固连标架组成一个集合：

$$\mathscr{W}_P = \{W_P\}.$$

在固定标架 U 下，固连标架 W_P 的基矩阵是 $W_P = \begin{pmatrix} w_1^1\text{cm} & w_2^1\text{cm} & p_1\text{cm} \\ w_1^2\text{cm} & w_2^2\text{cm} & p_2\text{cm} \\ 0 & 0 & 1 \end{pmatrix}$，若页面上的点 A 在 W_P 内的坐标为 $\alpha_{W_P} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$，则点 A 可以表达为：

$$W_P \cdot \alpha_{W_P} = \begin{pmatrix} w_1^1\text{cm} & w_2^1\text{cm} & p_1\text{cm} \\ w_1^2\text{cm} & w_2^2\text{cm} & p_2\text{cm} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot w_1^1\text{cm} + y \cdot w_2^1\text{cm} + p_1\text{cm} \\ x \cdot w_1^2\text{cm} + y \cdot w_2^2\text{cm} + p_2\text{cm} \\ 1 \end{pmatrix}.$$

变换对应的标架 在变换过程中，任意一个变换 T 都对应两种标架：画布标架和 xy 标架，这两个标架的原点始终重合；每个标架都有 x 轴和 y 轴两个轴，所以就有 4 个轴。也可以认为变换 T 引起一个二元映射：

$$f_T \colon (W_P, I_P) \to (W'_{P'}, I'_{P'}),$$

其中 f_T 的下标表示该二元映射是由变换 T 引起的；P 和 P' 代表页面上的两个点，这两个点的相对位置取决于变换 T；W_P 和 $W'_{P'}$ 是画布标架；I_P 和 $I'_{P'}$ 是 xy 标架；f_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。

约定：W_P 是 f_T 的当前固连标架，$W'_{P'}$ 是 f_T 的标定固连标架，I_P 是 f_T 的输入标架，$I'_{P'}$ 是 f_T 的输出标架。

由于每个标架都有两个轴，所以也可以认为变换 T 引起一个四元映射：

$$f_T\colon (cx_P, cy_P, x_P, y_P) \to (cx'_{P'}, cy'_{P'}, x'_{P'}, y'_{P'}),$$

其中 cx_P, cy_P 分别是标架 W_P 的 x 轴和 y 轴，而 x_P, y_P 分别是标架 I_P 的 x 轴和 y 轴。由轴 cx_P, y_P 组成的标架记为 $C_{\text{固连-输出}}$；由轴 x_P, cy_P 组成的标架记为 $C_{\text{输出-固连}}$.

I_P x_P W_P cx_P cy_P P y_P f_T $I'_{P'}$ $x'_{P'}$ $cy'_{P'}$ $y'_{P'}$ P' $W'_{P'}$ $cx'_{P'}$

设 T_1 和 T_2 是前后相继出现的两个变换，则有

$$(W_{P_0}, I_{P_0}) \xrightarrow{f_{T_1}} (W_{P_1}, I_{P_1}) \xrightarrow{f_{T_2}} (W_{P_2}, I_{P_2}),$$

W_{P_0} 是 T_1 的当前固连标架；I_{P_0} 是 T_1 的当前输入标架；W_{P_1} 是 T_1 的标定固连标架，也是 T_2 的当前固连标架；I_{P_1} 是 T_1 的输出标架，也是 T_2 的当前输入标架；W_{P_2} 是 T_2 的标定固连标架；I_{P_2} 是 T_2 的输出标架。

约定以下标架名称：

- 参照标架：当一个变换在作用时需要有个参照系，即要确定相对于哪个标架做变换。为变换做参照系的标架就是与该变换对应的参照标架。当前变换的参照标架是“当前参照标架”。
- 输入标架：变换的操作对象就是该变换的“输入标架”。第一个变换的输入标架与固定标架 U 重合。其它变换的输入标架是前一个变换的输出标架。当前变换的输入标架是“当前输入标架”。当前输入标架的原点总是与当前参照标架的原点重合。
- 输出标架：一个变换将其输入标架变成该变换的输出标架。
- 最终输出标架：最后一个变换作用完毕后的输出标架，是正式用于绘图的标架。
- 当前平动标架：每个当前变换都对应一个当前平动标架，若当前输入标架的原点是页面上的点 P，则当前平动标架是 V_P.
- 当前固连标架：每个当前变换都对应一个当前固连标架，若当前输入标架的原点是页面上的点 P，则当前固连标架是 $\mathscr{W}_P$ 的某个元素 W_P.
- 标定固连标架：记变换的输出标架的原点是 P，变换会在 $\mathscr{W}_P$ 中选取一个固连标架 W_P，称之为“标定固连标架”（选取的方式见后文）。当前变换选取的标定固连标架是下一个变换的当前固连标架。
- 最终标定固连标架：最后一个变换选取的标定固连标架，其原点与最终输出标架的原点相同。

- 描点标架:有的变换选项的参数是坐标,例如 `shift={(1.5,0.5cm)}`,`rotate around={30:(0.5cm,1)}`, `x={(0.5cm,1)}`，需要把其中的坐标解释为页面上的某个点（在页面上"描点"），这就需要把该坐标看作是某个标架中的坐标，这个标架就是该坐标对应的"描点坐标"。另外，在得到最终输出标架和最终标定固连标架后，再逐个读取路径中的坐标。对于路径上的任意一个坐标，需要确定该坐标所对应的点是页面上的哪个点（相当于"描点"），这也需要把该坐标看作是某个标架中的坐标，这个标架是该坐标所对应的"描点标架"。也就是说，每个坐标都对应自己的"描点标架"。在描点时，不同形式的坐标对应不同的描点标架，按照坐标数据是否带长度单位，坐标的形式有 4 种，如 `(1,2)`，`(1cm，2cm)`，`(1，2cm)`，`(1cm，2)`，`(30:1cm)`，`(30:1 and 2)`，所以相应的描点标架有 4 种，见后文。

标架名目有点多，其实不过是对画布标架，xy 标架，$C_{固连\text{-}输出}$ 和 $C_{输出\text{-}固连}$ 这 4 种标架的利用。

各种当前标架的确定 确定一个变换对应的各个当前标架是理解变换效果的关键，不过这有点复杂：

♦ 当前参照标架：

(a) 第一个变换的当前参照标架与固定标架 U（或者平动标架 V_O）重合。

(b) 若变换选项含有比例、旋转或反射成分，例如 `scale=0.8`，`xscale=-2`，`yscale=2`，`rotate=30`，`rotate around={30:(30:1cm)}`，或者是选项 `cm`，那么当前参照标架是当前固连标架。

(c) 若变换选项的值用画布坐标形式给出,例如 `x={(1cm,2cm)}`,`x=2cm`,`shift={(1.5cm,1cm)}`,`yshift=1.5cm`，那么当前参照标架是当前固连标架。

(d) 若变换选项的值用 xy 坐标形式给出,且该选项不包含旋转或反射成分,例如 `x={(1,2)}`,`shift={(1.5,1)}`，那么当前参照标架是当前输入标架。

(e) 其它相关情况参考后文关于"描点标架"的解释。

♦ 当前输入标架：第一个变换的当前输入标架与固定标架 U 重合。其它变换的当前输入标架就是前一个变换的输出标架。

♦ 当前平动标架：即当前输入标架的原点处的平动标架，第一个变换的当前平动标架与固定标架 U 重合。

♦ 当前固连标架：第一个变换的当前固连标架与固定标架 U 重合，其它变换的当前固连标架是前一个变换选取的"标定固连标架"。变换的输出标架与其标定固连标架的原点重合，所以当前固连标架的原点位置与当前输入标架的原点位置相同。把变换 T_n 对应的当前固连标架记为 W_{P_n}，其中的下标 P_n 是变换 T_n 的当前输入标架的原点，下面看一下 W_{P_n} 的基向量。

考虑当前变换 T_{i+1} 的前一个变换 T_i：

(a) 若 T_i 是由指定标架的选项所确定的变换,例如选项 `x=2cm`,`y=10`,`y={(1cm,0.5cm)}`,`x={(1cm,0.5)}`，那么标架 $W_{P_{i+1}}$ 与标架 W_{P_i} 完全相同（原点也一样）。

(b) 若 T_i 是含有旋转、反射、比例变化成分的选项，或者选项 `cm`，则 $W_{P_{i+1}} = T_i(W_{P_i}) = W_{P_i} \cdot T_i$，其中用 T_i 同时代表变换和变换的矩阵，这个式子的意思是，变换 T_i 以标架 W_{P_i} 为参照系对 W_{P_i} 自己做变换得到标架 $W_{P_{i+1}}$.

(c) 若 T_i 是平移变换选项，且输出标架的原点是 P'，则 $P_{i+1} = P'$，$W_{P_{i+1}}$ 与 W_{P_i} 的方向（两个基向量的方向）相同。

参照标架的影响 记变换 T 的当前固连标架是 W_P，当前输入标架是 I_P，看一下 T 的标定固连标架和输出标架。

参照标架与输入标架相同 此时的情况是：变换选项的值用 xy 坐标形式给出，且该选项不包含旋转或反射成分，例如 x={(1,2)}，shift={(1.5,1)}. 此时变换 T 的标定固连标架是

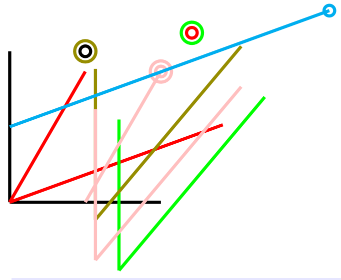
$$W'_{P'} = W_P + I_P \cdot T \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

也就是说，当变换 T 具有平移作用时才会让 W_P 与 $W'_{P'}$ 有差别，而且差别仅是原点不同。变换 T 的输出标架是 $I'_{P'} = I_P \cdot T$.

参照标架是当前固连标架 此时变换 T 的标定固连标架是 $W'_{P'} = W_P \cdot T$. 变换 T 的输出标架是 $I'_{P'} = W'_{P'} \cdot \mathrm{scalar}(I_P) = W_P \cdot T \cdot \mathrm{scalar}(I_P)$，其中的 $\mathrm{scalar}(I_P)$ 是把 I_P 中的长度单位去掉后得到纯数值矩阵。

描点标架——变换选项的参数坐标 有的变换选项的参数含有坐标,例如 rotate around={30:(0.5,1)}，其中的坐标是不带长度单位的 xy 坐标形式；再如 scale around={30:(0.5cm,1cm)}，其中的坐标是带长度单位的画布坐标形式。

- 若坐标是不带长度单位的 xy 坐标形式，则该坐标是当前输入标架中的坐标。
- 若坐标是带长度单位的画布坐标形式，则该坐标是当前固连标架中的坐标。
- 若坐标是 (2mm,1) 这种第一个坐标分量带长度单位、第二个坐标分量不带长度单位的形式, 那么用当前固连标架的 x 轴和当前输出标架的 y 轴作成一个新的标架 $C_{\text{固连-输出}}$，把 (2mm,1) 作为 $C_{\text{固连-输出}}$ 下的坐标。
- 若坐标是 (2,1mm) 这种第一个坐标分量不带长度单位、第二个坐标分量带长度单位的形式, 那么用当前输出标架的 x 轴和当前固连标架的 y 轴作成一个新的标架 $C_{\text{输出-固连}}$，把 (2,1mm) 作为 $C_{\text{输出-固连}}$ 下的坐标。



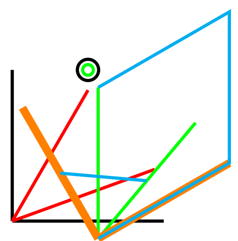
```
\begin{tikzpicture}[scale=2,very thick]
  \draw (0,0)--(1,0) (0,0)--(0,1);
% 使用 xy 坐标形式，其中 (0.5,1) 是红色标架中的点坐标
  {[x={(20:1.5cm)},y={(60:1cm)}]
   \draw [red] (0,0)--(1,0) (0,0)--(0,1) (0.5,1) circle (1pt);
   \draw [green,rotate around={30:(0.5,1)}]
     (0,0)--(1,0) (0,0)--(0,1) (0.5,1) circle (2pt);
% 使用画布坐标形式，其中 (0.5cm,1cm) 是黑色标架中的点坐标
   \draw (0.5cm,1cm) circle (1pt);
   \draw [olive,rotate around={30:(0.5cm,1cm)}]
     (0,0)--(1,0) (0,0)--(0,1) (0.5cm,1cm) circle (2pt);
% 使用混合坐标形式，其中 (0.5cm,1) 对应粉色点
   \draw [pink](0.5cm,1) circle (1pt)--(0.5cm,0);
   \draw [pink,rotate around={30:(0.5cm,1)}]
     (0,0)--(1,0) (0,0)--(0,1) (0.5cm,1) circle (2pt);
% 使用混合坐标形式，其中 (1.5,0.5cm) 对应青色点
```

```
  \draw [cyan,shift={(1.5,0.5cm)}]
    (0,0) circle (1pt)--([shift={(-1.5,-0.5cm)}]0,0.5cm);
  }
\end{tikzpicture}
```

上面例子中，黑色线段代表最初的固连标架，即固定标架 U；选项 `x={(20:1cm)},y={(60:1cm)}` 的输出标架用红色线段代表；坐标 `(0.5,1)`（红色小圆圈）的描点标架是红色标架；坐标 `(0.5cm,1cm)`（黑色小圆圈）的描点标架是黑色标架；黑色标架的 x 轴与红色标架的 y 轴组成的标架是粉色点 `(0.5cm,1)` 的描点标架；红色标架的 x 轴与黑色标架的 y 轴组成的标架是青色点 `(1.5,0.5cm)` 的描点标架；

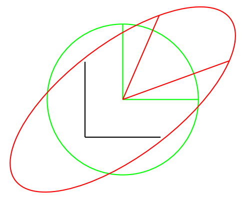
描点标架——路径上的坐标 最后一个变换作用完毕后，得到最终输出标架和最终标定固连标架，这两个标架的原点相同。然后在这两个标架中确定路径上的点。绘图命令的路径由一些点坐标构成，点的坐标形式不同，其解释也不同：

- 若坐标是不带长度单位的 xy 坐标形式，如 `(1,2)`，则该坐标是最终输出标架中的坐标。
- 若坐标是带长度单位的画布坐标形式，如 `(1cm,2cm)`，则该坐标是最终标定固连标架中的坐标。
- 若坐标是 `(2mm,1)` 这种第一个坐标分量带长度单位、第二个坐标分量不带长度单位的形式，那么用最终标定固连标架的 x 轴和最终输出标架的 y 轴作成一个新的标架 $C_{\text{固连-输出}}$，把 `(2mm,1)` 作为 $C_{\text{固连-输出}}$ 下的坐标。
- 若坐标是 `(2,1mm)` 这种第一个坐标分量不带长度单位、第二个坐标分量带长度单位的形式，那么用最终输出标架的 x 轴和最终标定固连标架的 y 轴作成一个新的标架 $C_{\text{输出-固连}}$，把 `(2,1mm)` 作为 $C_{\text{输出-固连}}$ 下的坐标。



```
\begin{tikzpicture}[scale=2,very thick]
  \draw (0,0)--(1,0) (0,0)--(0,1); % 画出黑色标架
  \draw (0.5cm,1cm) circle (2pt); % 标记黑色标架中的坐标
  {[x={(20:1cm)},y={(60:1cm)}] % 指定红色的标架
   \draw [red](0,0)--(1,0) (0,0)--(0,1);
   \draw [green](0.5cm,1cm) circle (1pt); % 标记黑色标架中的坐标
    {[rotate around={30:(0.5cm,1cm)}] % 确定最终输出标架（绿色），最终标定固连标架（橙色）
   \draw [green,]
     (0,0)--(1,0) (0,0)--(0,1);
   \draw [orange,line width=3pt,] % 画最终的标定固连标架
     (0cm,0cm)--(1cm,0cm) (0cm,0cm)--(0cm,1cm);
   \draw [cyan,] % 注意本命令中的坐标点如何确定
     (0,0)--(1cm,0)--(1cm,1)--(0,1)
     (0.5,0cm)--(0,0.5cm);
  }}
\end{tikzpicture}
```

在标架变换下，画圆的操作 `circle (1cm)` 与 `circle (1)` 也有不同表现：



```
\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) 使用了带单位的半径尺寸，所以这个圆上点的坐标是当前固连标架（绿色）下的坐标，所以画出的是圆。操作 circle (1) 使用了不带单位的半径尺寸，所以这个圆上点的坐标是当前输入标架（红色）下的坐标，所以画出的是椭圆。

对于画矩形操作有如下效果：

```
\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);
```

25.2 标架变换：指定标架

下面的选项用于指定标架，它们的参照标架是当前固连标架或当前输入标架，它们把输入标架变成指定的标架。但是不认为它们具有旋转、反射作用，所以它们的标定固连标架与其当前固连标架完全一样。

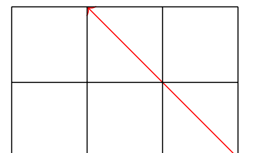
/tikz/x=⟨*value*⟩ (no default, initially 1cm)

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。

如果 ⟨*value*⟩ 是带单位的尺寸（可以是负值尺寸），例如 x=-20mm，则本选项的当前参照标架是当前固连标架。本选项把当前固连标架中的向量 (-20mm,0pt) 作为其输出标架的 x 轴的单位向量，其输出标架的 y, z 轴的单位向量以及原点都与当前固连标架相同。

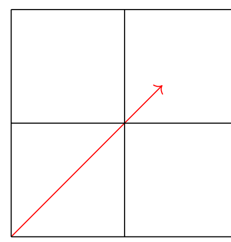
如果 ⟨*value*⟩ 是不带单位的数值，例如 x=10，则它等价于 x={(10pt,0pt)}，也就是说 PGF 会自动给 ⟨*value*⟩ 加上长度单位 pt.

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)}, 详情参照后文。



```
\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，这个长度在当前固连标架内确定，而变换选项 x=-1.5cm 并不改变固连标架，故网格不接受 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 改变固连标架，所以网格的外观与前面的例子不同。

设当前固连标架 W_P 的基矩阵是 $W_P = \begin{pmatrix} w_1^1\text{cm} & w_2^1\text{cm} & p_1\text{cm} \\ w_1^2\text{cm} & w_2^2\text{cm} & p_2\text{cm} \\ 0 & 0 & 1 \end{pmatrix}$，则选项 x={a cm} 的输出标架是：

$$W_P \cdot \begin{pmatrix} \frac{a}{\sqrt{{w_1^1}^2+{w_1^2}^2}} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

/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 1cm)

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。

```
\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}
```

注意在以上选项中，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)}，则这个选项的当前参照标架是当前固连标架，选项参数 (10pt,20mm) 是当前固连标架之下的坐标。这个选项把当前固连标架中的向量 (10pt,20mm) 作为其输出标架的 x 轴的单位向量；把当前固连标架的 y, z 轴的单位向量分别作为其输出标架的 y, z 轴的单位向量；其输出标架的原点与当前固连标架的原点相同。

若 ⟨value⟩ 是不带单位的 xy 坐标形式，如 x=(30:2), x={(1.5,2)}，则这个选项的当前参照标架是当前输入标架，选项参数 (1.5,2) 是当前输入标架中的坐标。这个选项把当前输入标架中坐标为 (1.5,2) 的向量作为其输出标架的 x 轴的单位向量；把当前输入标架的 y, z 轴的单位向量分别作为其输出标架的 y, z 轴的单位向量；其输出标架的原点与当前输入标架的原点相同。

指定标架的选项的参数中，不同形式的坐标对应不同的描点标架。设当前固连标架 W_P 的基矩阵是 $W_P = \begin{pmatrix} w_1^1\text{cm} & w_2^1\text{cm} & p_1\text{cm} \\ w_1^2\text{cm} & w_2^2\text{cm} & p_2\text{cm} \\ 0 & 0 & 1 \end{pmatrix}$，当前输入标架 I_P 的基矩阵是 $I_P = \begin{pmatrix} i_1^1\text{cm} & i_2^1\text{cm} & p_1\text{cm} \\ i_1^2\text{cm} & i_2^2\text{cm} & p_2\text{cm} \\ 0 & 0 & 1 \end{pmatrix}$：

- 选项 `x={(a cm, b cm)}` 对应的描点标架是当前固连标架，本选项的输出标架是：

$$W_P \cdot \begin{pmatrix} \frac{a}{\sqrt{{w_1^1}^2+{w_1^2}^2}} & 0 & 0 \\ \frac{b}{\sqrt{{w_2^1}^2+{w_2^2}^2}} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix};$$

  两个选项 `x={(a cm, b cm)}, y={(c cm, d cm)}` 的输出标架是：

$$W_P \cdot \begin{pmatrix} \frac{a}{\sqrt{{w_1^1}^2+{w_1^2}^2}} & \frac{c}{\sqrt{{w_1^1}^2+{w_1^2}^2}} & 0 \\ \frac{b}{\sqrt{{w_2^1}^2+{w_2^2}^2}} & \frac{d}{\sqrt{{w_2^1}^2+{w_2^2}^2}} & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

- 选项 `x={(a, b)}` 对应的描点标架是当前输出标架，本选项的输出标架是：

$$I_P \cdot \begin{pmatrix} a & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix};$$

  而两个选项 `x={(a, b)}, y={(c, d)}` 的输出标架是：

$$I_P \cdot \begin{pmatrix} a & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & c & 0 \\ 0 & d & 0 \\ 0 & 0 & 1 \end{pmatrix} = I_P \cdot \begin{pmatrix} a & ac & 0 \\ b & bc+d & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

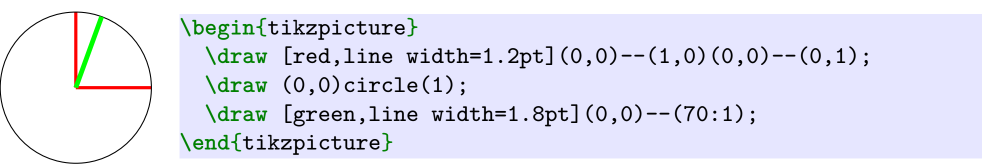
- 选项 `x={(a, b cm)}` 对应的描点标架的 x 轴是当前输入标架 I_P 的 x 轴，描点标架的 y 轴是当前固连标架 W_P 的 y 轴，所以描点标架就是

$$S_P = \begin{pmatrix} i_1^1\text{cm} & w_2^1\text{cm} & p_1\text{cm} \\ i_1^2\text{cm} & w_2^2\text{cm} & p_2\text{cm} \\ 0 & 0 & 1 \end{pmatrix} = I_P \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} + W_P \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

  本选项的输出标架是

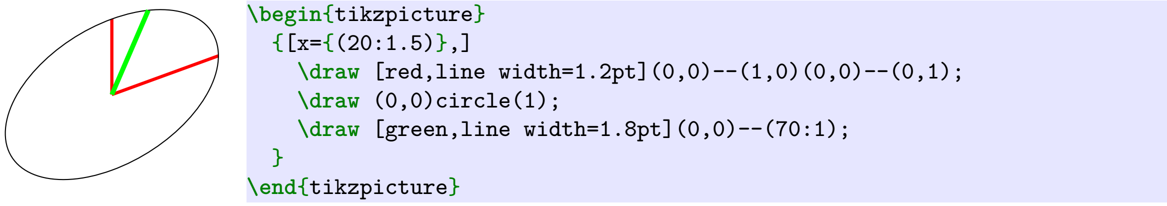
$$S_P \cdot \begin{pmatrix} a & 0 & 0 \\ \frac{b}{\sqrt{{w_2^1}^2+{w_2^2}^2}} & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} + I_P \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} = I_P \cdot \begin{pmatrix} a & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} + W_P \cdot \begin{pmatrix} 0 & 0 & 0 \\ 0 & \frac{b}{\sqrt{{w_2^1}^2+{w_2^2}^2}} & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

再看一下选项 `x={(30:2)}` 的输出标架如何确定。选项 `x={(30:2)}` 的当前参照标架是当前输入标架，坐标 `(30:2)` 是当前参照标架中的坐标。若当前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：



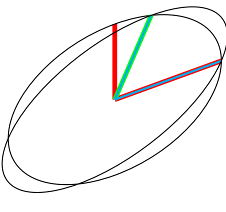
```
\begin{tikzpicture}
  \draw [red,line width=1.2pt](0,0)--(1,0)(0,0)--(0,1);
  \draw (0,0)circle(1);
  \draw [green,line width=1.8pt](0,0)--(70:1);
\end{tikzpicture}
```

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，使得绘图标架变为 {(0,0);(20:1.5cm),(0cm,1cm)}，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}
```

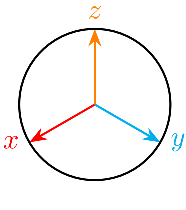
可以猜测，在仿射标架 {(0,0);(20:1.5cm),(0cm,1cm)} 下极坐标 (70:1) 代表的向量（绿色线段）由之前图形中的绿色线段变换而来。这就是在非单位正交标架中解释极坐标的方法，即通过变换来解释。



```
\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}
```

观察下面两个例子。

下面画坐标轴 x, y, z 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)} 把当前固连标架 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，从而得到输出标架 C，即把固定标架 U 变成下面的标架：

$$C=\left\{\begin{array}{ll}\text{原点} & : O\to O,\\ x\ \text{轴单位向量} & : (1\text{cm},0\text{pt})\to(\frac{-\sqrt{3}}{2}\text{cm},\frac{-1}{2}\text{cm}),\\ y\ \text{轴单位向量} & : (0\text{pt},1\text{cm})\to(\frac{\sqrt{3}}{2}\text{cm},\frac{-1}{2}\text{cm}),\\ \text{z 轴单位向量} & : (-3.58\text{mm},-3.85\text{mm})\to(0\text{cm},1\text{cm}).\end{array}\right\}\text{固定标架 } U \text{ 下的坐标}$$

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)}` 以当前输入标架 C_1 为当前参照标架，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 Y，并以 OY 为 y 轴的单位向量，而 z 轴的单位向量还是不变。点 Y 在 C_1 中的坐标为 $(\frac{\sqrt{3}}{2}, \frac{-1}{2})$，由此计算出点 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)}` 以当前输入标架 C_2 为当前参照标架对 C_2 做变换，即在 C_2 中找出坐标为 $(0,1)$ 的点 Z，并以 OZ 为 z 轴的单位向量。在 C_2 中坐标为 $(0,1)$ 的点就是点 Y，实际上点 Z 与点 Y 重合。因此本选项的输出标架 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”。

再看一个例子。

```
\begin{tikzpicture}[line width=2pt]
  \draw (0,0)--(1,0) (0,0)--(0,1);
  \draw [red]
        [x={(60:1cm)},y={(150:1cm)}]
        [shift={(1,1)},xscale=-1]
        [x={(-20:1cm)},y={(30:1cm)}]
    (0,0)--(1,0) (0,0)--(0,1);
\end{tikzpicture}
```

在这个例子中，选项 `x={(60:1cm)},y={(150:1cm)}` 在当前固连标架——即固定标架 U 中找出坐标分别

为 (60:1cm)，(150:1cm) 的点 A, B，并以 O 为原点，以 $\overrightarrow{OA}$ 为 x 轴的单位向量，以 $\overrightarrow{OB}$ 为 y 轴的单位向量得到输出标架 E:

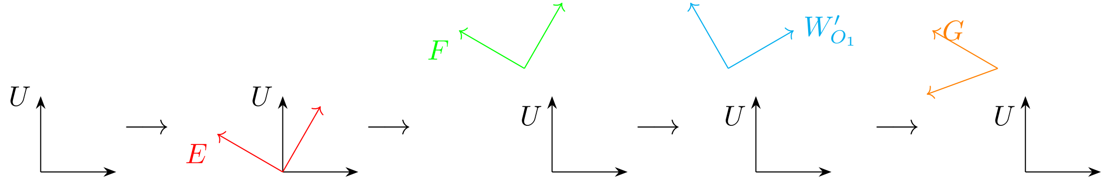
$$E = \begin{cases} \text{原点} & : O, \\ x \text{ 轴单位向量} & : (\frac{1}{2}\text{cm}, \frac{\sqrt{3}}{2}\text{cm}), \\ y \text{ 轴单位向量} & : (\frac{-\sqrt{3}}{2}\text{cm}, \frac{1}{2}\text{cm}). \end{cases} \text{固定标架 } U \text{ 下的坐标}$$

然后选项 shift={(1,1)} 以当前输入标架 E 为当前参照标架来平移 E，得到的输出标架记为 F. 标架 F 的原点 O_1 在标架 E 中的坐标是 (1,1)，在固定标架 U 下的坐标是 $(\frac{1-\sqrt{3}}{2}, \frac{1+\sqrt{3}}{2})$.

然后是选项 xscale=-1 规定的变换，因为之前没有旋转或反射变换，所以本选项的当前固连标架是 $W_{O_1} = V_{O_1}$，并以 W_{O_1} 为当前参照标架，这个选项一方面翻转 W_{O_1} 的 x 轴（以 W_{O_1} 自身为参照系），使之成为左手系 W'_{O_1}（标定固连标架），另一方面创建一个与 W'_{O_1} 一样的输出标架。

然后是选项 x={(-20:1cm)},y={(70:1cm)} 规定的变换，本选项的当前固连标架是 W'_{O_1}，并以 W'_{O_1} 为当前参照标架。在 W'_{O_1} 中找出坐标分别为 (-20:1cm)，(30:1cm) 的点 X, Y，并以 O_1 为原点，以 $\overrightarrow{O_1X}$ 为 x 轴的单位向量，以 $\overrightarrow{O_1Y}$ 为 y 轴的单位向量得到最终输出标架 G，然后在标架 G 中画出路径。

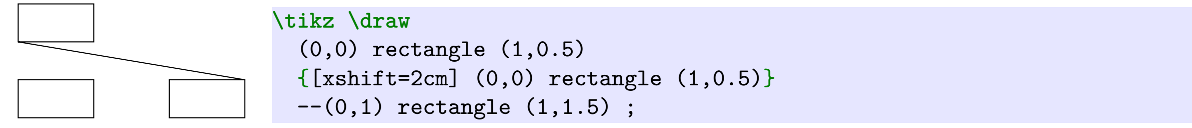
标架变换的步骤如下所示:



25.3 坐标变换

如 (1,0)，(1cm,1pt)，(30:2cm) 这样的坐标对应坐标系中的位置，位置是用横向和纵向的有单位的长度确定的。PGF 和 TikZ 可以对坐标应用坐标变换矩阵。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。一般而言，坐标变换矩阵不能用于不涉及坐标的项目，即使该项目间接地涉及坐标。

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 (1,0.5)
  {[xshift=2cm] (0,0) rectangle (1,0.5)}
  --(0,1) rectangle (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。

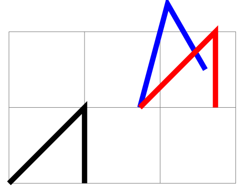
25.3.1 坐标变换选项

/tikz/shift=⟨*coordinate*⟩ (no default)

平移变换，平移向量为 ⟨*coordinate*⟩.

/tikz/shift only (no default)

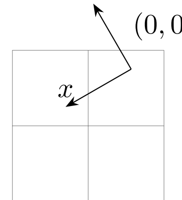
本选项把当前输入标架变成当前平动标架。



```
\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=`⟨*dimension*⟩ (no default)

本选项的当前参照标架是当前固连标架。注意 ⟨*dimension*⟩ 应当带长度单位，否则默认其单位为 `pt`，⟨*dimension*⟩ 的值可以是负的。本选项将当前输入标架沿着当前固连标架的 x 轴方向平移 ⟨*dimension*⟩ 得到输出标架。



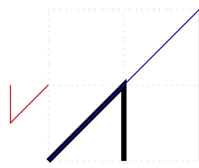

```
\begin{tikzpicture}
  \draw[help lines] (-2,-2) grid (0,0);
  \node [above]{$(0,0)$};
  {[>=Stealth,rotate=30,xscale=-1,xshift=0.5cm]
    \draw [->](0,0)--(1,0)node[above]{$x$};
    \draw [->](0,0)--(0,1);
  }
\end{tikzpicture}
```

`/tikz/yshift=`⟨*dimension*⟩ (no default)

本选项的当前参照标架是当前固连标架。沿着当前固连标架的 y 轴方向来平移当前输入标架，得到其输出标架。⟨*dimension*⟩ 的值可以是负的。注意 ⟨*dimension*⟩ 应当带长度单位，否则默认其单位为 `pt`.

`/tikz/scale=`⟨*factor*⟩ (no default)

本选项的当前参照标架是当前固连标架。以当前固连标架的原点为中心做位似变换，即放缩，⟨*factor*⟩ 是放缩比例（放缩后的尺寸比上放缩前的尺寸）。如果 ⟨*factor*⟩ 是负值，就先将图形以原点为中心做中心对称，再以原点为中心放缩。



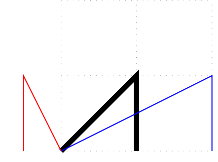
```
\begin{tikzpicture}
\draw[help lines,dotted] (0,0) grid (2,2);
\draw [line width=2pt](0,0) -- (1,1) -- (1,0);
\draw[scale=2,blue] (0,0) -- (1,1) -- (1,0);
\draw[yshift=1cm,scale=-0.5,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

`/tikz/scale around={`⟨*factor*⟩`:`⟨*coordinate*⟩`}` (no default)

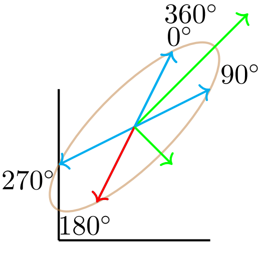
与 `scale` 类似，只是变换的中心改为 ⟨*coordinate*⟩.

`/tikz/xscale=`⟨*factor*⟩ (no default)

本选项的当前参照标架是当前固连标架。以当前固连标架的原点为中心，沿着当前固连标架的 x 轴方向对当前输入标架做放缩，⟨*factor*⟩ 为放缩比例。如果 ⟨*factor*⟩ 是负值，则先翻转当前固连标架的 x 轴方向，再以原点为中心做放缩。



```
\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}
```



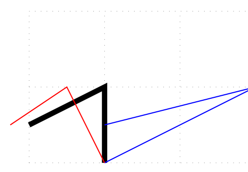

```
\begin{tikzpicture}[thick]
  \draw (0,0)--(2,0) (0,0)--(0,2);
  \begin{scope}[cm={0.5,1,1,0.5,(1,1.5)}]
    \draw [brown,,opacity=0.5](0,0)circle(1);
    \draw [->,green](0,0)--(1,1); % 绿色线将变成下面的绿色线
    \foreach \ang / \dis in {0/2mm,90/4mm,180/3mm,270/4mm,360/5mm}
      {\draw [->,cyan](0,0)--(\ang:1);
        \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)

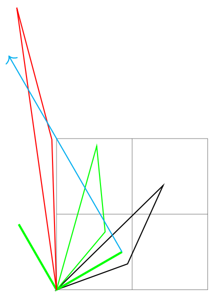
本选项的当前参照标架是当前固连标架, 沿着当前固连标架的 x 轴方向做变换, 这个变换是 $(x,y) \to (x + y \cdot \langle factor \rangle, y)$.



```
\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)

本选项的当前参照标架是当前固连标架, 沿着当前固连标架的 y 轴方向做变换, 这个变换是 $(x,y) \to (x, y + x \cdot \langle factor \rangle)$.



```
\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);
    }}}
```

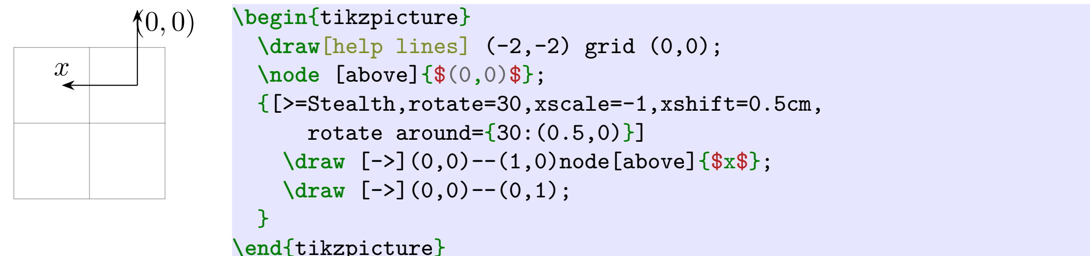
上面图形中，青色箭头指示选项 `yslant=1.5` 的倾斜方向。

`/tikz/rotate=`⟨*degree*⟩ (no default)

本选项的当前参照标架是当前固连标架。以当前固连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。

`/tikz/rotate around={`⟨*degree*⟩`:`⟨*coordinate*⟩`}` (no default)

本选项的当前参照标架是当前固连标架。以 ⟨*coordinate*⟩ 为中心做旋转。




```
\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/rotate around x=`⟨*angle*⟩ (no default)

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。

`/tikz/rotate around y=`⟨*angle*⟩ (no default)

围绕 y 轴做旋转，以右手螺旋为正。

`/tikz/rotate around z=`⟨*angle*⟩ (no default)

围绕 z 轴做旋转，以右手螺旋为正。

`/tikz/cm={`⟨*a*⟩`,`⟨*b*⟩`,`⟨*c*⟩`,`⟨*d*⟩`,`⟨*coordinate*⟩`}` (no default)

这个变换是一般的变换形式，其中的 ⟨*a*⟩, ⟨*b*⟩, ⟨*c*⟩, ⟨*d*⟩ 最好是不带长度单位的数值，不同形式的坐标 ⟨*coordinate*⟩ 对应不同的描点标架。假设 ⟨*coordinate*⟩ 是向量 $\binom{t_x}{t_y}$，针对点 $\binom{x}{y}$ 做变换，则变换结果是

$$\begin{pmatrix} a & c \\ b & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}.$$

这个选项的当前参照标架是当前固连标架，变换的对象是当前输入标架。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固连标架，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)}
```

不过，把作用次序看作是“先平移标架、再用矩阵积变换标架”可能会方便一些。

```
\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}
```

上面例子中，第二个 cm 选项的当前参照标架是它的当前固连标架，即红色线段标识的向量。

`/tikz/reset cm` (no value)

将变换矩阵设为单位矩阵。取消当前分组内的所有变换，继承自外层环境的变换也被取消。

25.3.2 注意的问题

关于坐标变换要注意以下问题。

- 变换对预定义的 `rectangle` 路径只有横向、纵向的作用。
- 按前面的猜测，变换的对象是“标架”，即构成标架的 3 个点（两个定点向量），有的情况下可能需要一定量的手工计算来确定变换数据。举例说，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

$$\left\{\begin{pmatrix}\frac{1}{2}\\-\frac{\sqrt{3}}{2}\end{pmatrix},\begin{pmatrix}-\frac{\sqrt{3}}{2}\\-\frac{1}{2}\end{pmatrix}\right\}=\left\{\begin{pmatrix}\cos(-60^\circ)\\\sin(-60^\circ)\end{pmatrix},\begin{pmatrix}\sin(-60^\circ)\\-\cos(-60^\circ)\end{pmatrix}\right\},$$

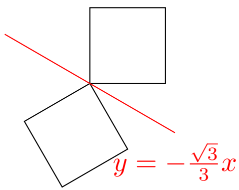
可以先在标架

$$\left\{\begin{pmatrix}\frac{1}{2}\\-\frac{\sqrt{3}}{2}\end{pmatrix},\begin{pmatrix}0\\1\end{pmatrix}\right\}$$

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)}]
```

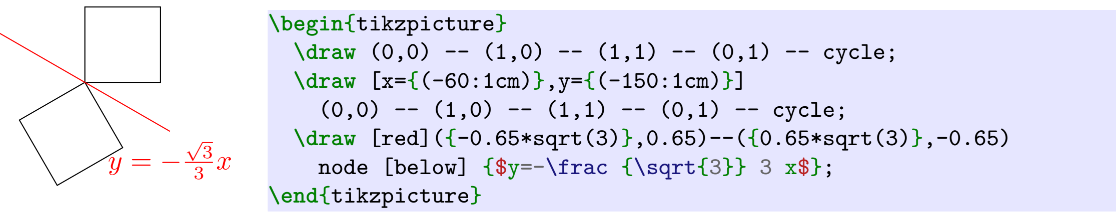
重新画上面的图形：




```
\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;
```

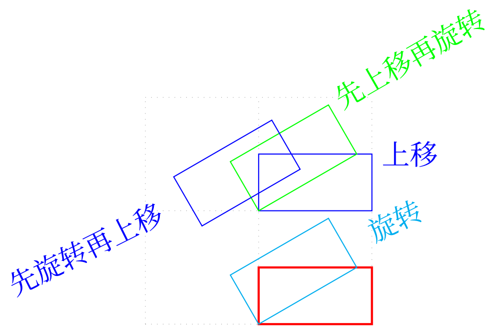
所做的标架变换是：

$$(0,0)\to(0,0),\quad (1,0)\to(\frac{5}{4},\frac{\sqrt{3}}{4}),\quad (1,1)\to(\frac{5-2\sqrt{3}}{4},\frac{\sqrt{3}-2}{4}),\quad (0,1)\to(-\frac{\sqrt{3}}{2},-\frac{1}{2}),$$

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}
```

- 前面提到，坐标变换矩阵不能用于不涉及坐标的项目，即使该项目间接地涉及坐标。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);
}
```

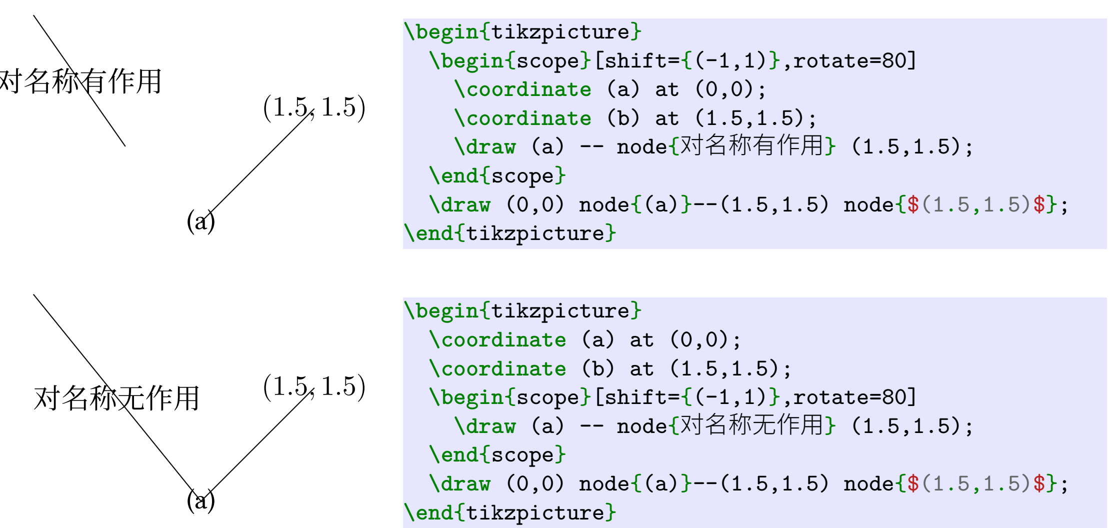
尽管其中的名称 (a), (\b), (\c) 代表坐标点，但是坐标变换对“名称”无效，也就是说，如果路径中用到了名称 (a)，该名称代表的点 (1,1) 在坐标变换下保持不动；其它的凡是以 (x, y) 或 (d:l) 形式，或者以坐标运算形式 ($ ... $)、($...!...!...$) 提供的点，都接受变换选项的作用，其中 x, y, d, l 可以是命令 \tikzmath{} 提供的实数型数值名称（下面有这种例子）。

只对 (1.5, 1.5) 起作用　(1.5, 1.5)

(a)

```
\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}
```

25.3.3 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 $\mathbf{P}'$：

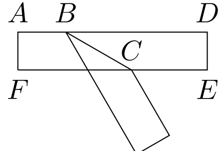
$$\begin{aligned}\mathbf{P}'&=\mathbf{P}-\mathbf{A}-2\left(\mathbf{e},\mathbf{P}-\mathbf{A}\right)\mathbf{e}+\mathbf{A}\\&=\left(\mathbf{I}-2\left(\mathbf{e},\mathbf{e}^T\right)\right)\left(\mathbf{P}-\mathbf{A}\right)+\mathbf{A}\\&=\frac{1}{(a_1-b_1)^2+(a_2-b_2)^2}\begin{pmatrix}(a_1-b_1)^2-(a_2-b_2)^2 & 2(a_1-b_1)(a_2-b_2)\\ 2(a_1-b_1)(a_2-b_2) & (a_2-b_2)^2-(a_1-b_1)^2\end{pmatrix}\begin{pmatrix}p_1-a_1\\p_2-a_2\end{pmatrix}+\begin{pmatrix}a_1\\a_2\end{pmatrix}\\&=\begin{pmatrix}a & c\\ b & d\end{pmatrix}\begin{pmatrix}p_1-a_1\\p_2-a_2\end{pmatrix}+\begin{pmatrix}a_1\\a_2\end{pmatrix}=\begin{pmatrix}p_1'\\p_2'\end{pmatrix}\end{aligned}$$

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。

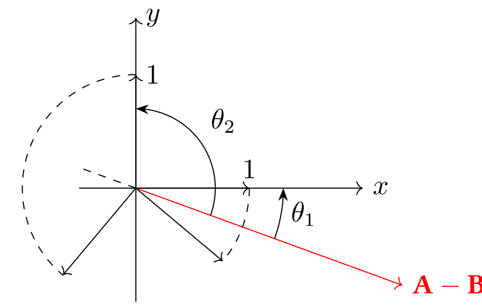
参数直线 $t \cdot (\mathbf{A} - \mathbf{B})$ 的方向是 $\mathbf{A} - \mathbf{B}$，如果能求得这个方向与 xOy 系的单位向量之间的夹角，用对称的办法就能容易地知道系 $x'Oy'$ 的单位向量的坐标。设从方向向量 $\mathbf{A} - \mathbf{B}$ 到单位向量 $(1, 0)$ 的夹角是 θ_1，从方向向量 $\mathbf{A} - \mathbf{B}$ 到单位向量 $(0, 1)$ 的夹角是 θ_2，$|\theta_i| \leqslant 180^\circ, i = 1, 2$，计算外积

$$(\mathbf{A} - \mathbf{B}) \times (1, 0) = |\mathbf{A} - \mathbf{B}| \sin\theta_1 = b_2 - a_2$$
$$(\mathbf{A} - \mathbf{B}) \times (0, 1) = |\mathbf{A} - \mathbf{B}| \sin\theta_2 = a_1 - b_1$$

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

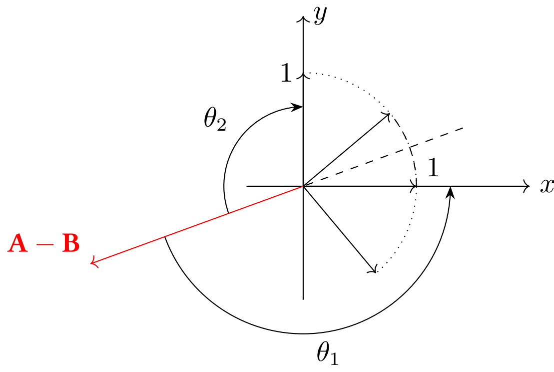
$$(\cos(-2\theta_1), \sin(-2\theta_1)) = (\cos 2\theta_1, -\sin 2\theta_1)$$

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

$$\theta_1 = 180° - \arcsin \frac{b_2 - a_2}{|\mathbf{A} - \mathbf{B}|}, \quad \theta_2 = \theta_1 - 270°$$



x 轴的单位向量 $(1,0)$ 变为 x' 轴的单位向量

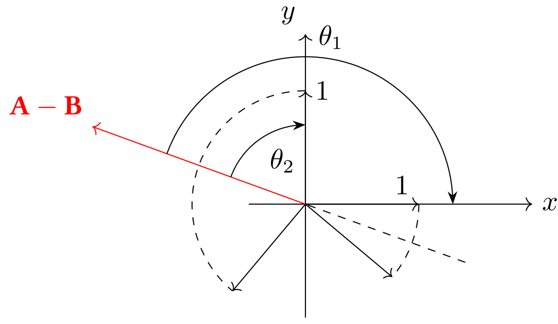
$$(\cos 2(180° - \theta_1), \sin 2(180° - \theta_1)) = (\cos 2\theta_1, -\sin 2\theta_1)$$

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

$$\theta_2 = \arcsin \frac{a_1 - b_1}{|\mathbf{A} - \mathbf{B}|}, \quad \theta_1 = \theta_2 - 90° = -180° - \arcsin \frac{b_2 - a_2}{|\mathbf{A} - \mathbf{B}|}$$



x 轴的单位向量 $(1,0)$ 变为 x' 轴的单位向量

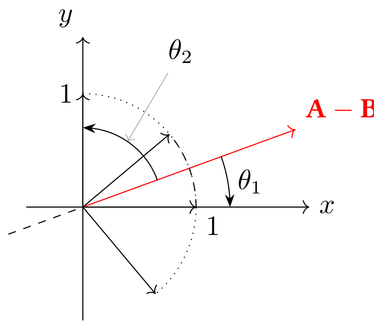
$$(\cos(-2(180° + \theta_1)), \sin(-2(180° + \theta_1))) = (\cos 2\theta_1, -\sin 2\theta_1)$$

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	$\theta_1 = \arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$	$-90^\circ \leqslant \arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|} \leqslant 0^\circ$	$-2\arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$
第二象限	$\theta_1 = -180^\circ - \arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$	$-90^\circ \leqslant \arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|} \leqslant 0^\circ$	$2\arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$
第三象限	$\theta_1 = 180^\circ - \arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$	$0^\circ \leqslant \arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|} \leqslant 90^\circ$	$2\arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$
第四象限	$\theta_1 = \arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$	$0^\circ \leqslant \arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|} \leqslant 90^\circ$	$-2\arcsin \frac{b_2-a_2}{|\mathbf{A}-\mathbf{B}|}$

并且总有

$$\begin{array}{ccc} xOy\text{系的标架} & & x'Oy'\text{系的标架} \\ (1,0) & \longrightarrow & (\cos 2\theta_1, -\sin 2\theta_1) = (-2\theta_1 : 1) \\ (0,1) & \longrightarrow & (-\sin 2\theta_1, -\cos 2\theta_1) = (-2\theta_1 - 90^\circ : 1) \end{array}$$

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

$$
\begin{array}{ccc}
xOy\text{系的标架} & & x'Oy'\text{系的标架} \\
(1,0) & \longrightarrow & (\cos 2\xi, -\sin 2\xi) = (-2\xi : 1) \\
(0,1) & \longrightarrow & (-\sin 2\xi, -\cos 2\xi) = (-2\xi - 90^\circ : 1)
\end{array}
$$

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

$$T^{-1}\begin{pmatrix} -\sin 2\theta_1 \\ -\cos 2\theta_1 \end{pmatrix} = \begin{pmatrix} -\frac{\sin 2\theta_1}{\cos 2\theta_1} \\ -\frac{\sin^2 2\theta_1}{\cos 2\theta_1} - \cos 2\theta_1 \end{pmatrix} = \frac{-1}{\cos 2\theta_1}\begin{pmatrix} \sin 2\theta_1 \\ 1 \end{pmatrix} = \frac{-1}{\cos 2\xi}\begin{pmatrix} \sin 2\xi \\ 1 \end{pmatrix},$$

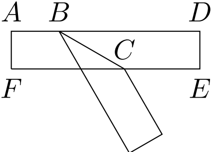
即

$$\left(-2\frac{a_1 - b_1}{|a_1 - b_1|}\frac{(a_1-b_1)(a_2-b_2)}{(a_1-b_1)^2 - (a_2-b_2)^2}, -\frac{(a_1-b_1)^2 - (a_2-b_2)^2}{(a_1-b_1)^2 + (a_2-b_2)^2}\right).$$

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={}`.

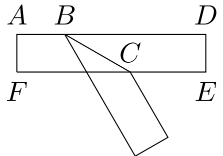
继续沿用前面设定的符号。用下面的思路找出点 $\mathbf{P}$ 的对称点 $\mathbf{P}'$：

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 $\{\left(\begin{smallmatrix}0\\1\end{smallmatrix}\right),\left(\begin{smallmatrix}0\\-1\end{smallmatrix}\right)\}$ 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：

$$
\begin{array}{ccc}
xOy\text{系的标架} & & x'Oy'\text{系的标架} \\
(1,0) & \longrightarrow & (\cos 2\xi, -\sin 2\xi) = (-2\xi : 1) \\
(0,1) & \longrightarrow & (-\sin 2\xi, -\cos 2\xi) = (-2\xi - 90^\circ : 1)
\end{array}
$$

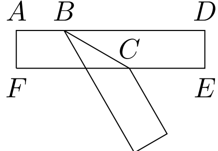
计算 ξ 的值并保存：

```
\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。

25.4 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
U
文字文字 文字文字
文字文字文字文字文字文字

```
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
```

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，标架变换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。

40 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. 249, `/tikz/y`→P. 250, `/tikz/z`→P. 250 来设置各个坐标轴的单位向量。

40.1 坐标系统

Coordinate system `xyz cylindrical`

这是圆柱坐标系统。参考命令 `\pgfpointcylindrical`→P. 640.

下面的选项用于决定一个圆柱坐标点：

- 参考 `/tikz/cs/radius`→P. 34.

/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。

- 参考 /tikz/cs/angle→P. 270.

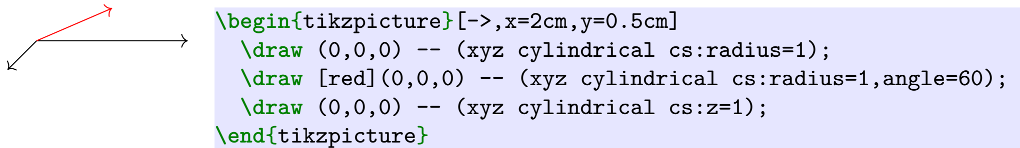
/tikz/cs/angle=⟨*degrees*⟩

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。

- 参考 /tikz/cs/z→P. 275.

/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。



```
\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}
```

Coordinate system xyz spherical

这是球坐标系统。参考命令 \pgfpointspherical→P. 641.

/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。

/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。

/tikz/cs/latitude=⟨*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。

/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}
```

40.2 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，Tikz 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。

40.2.1 转换到任意平面

为了确定一个三维空间中的一个平面，只需要 3 个不共线点 P_0, P_1, P_2 即可。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，此时点 P_0, P_1, P_2 都是有 3 个坐标分量的三维点。实际上，在平面上绘三维图时三维向量是用二维向量来模拟的，在这种模拟的情景下，可以把点 P_0, P_1, P_2 都看作是有 2 个坐标分量的二维点。点 P_0, P_1, P_2 由下面的选项指定：

/tikz/plane origin=⟨*point*⟩ (no default, initially (0,0))

本选项指定平面上的点 P_0, 用作平面标架的原点。⟨*point*⟩ 可以是二维点或三维点。

/tikz/plane x=⟨*point*⟩ (no default, initially (1,0))

本选项指定平面上的点 P_1, 用于计算平面标架的 x 轴的单位向量。⟨*point*⟩ 可以是二维点或三维点。

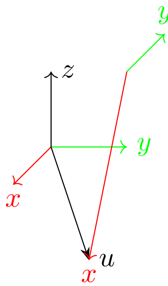
/tikz/plane y=⟨*point*⟩ (no default, initially (0,1))

本选项指定平面上的点 P_2, 用于计算平面标架的 y 轴的单位向量。⟨*point*⟩ 可以是二维点或三维点。

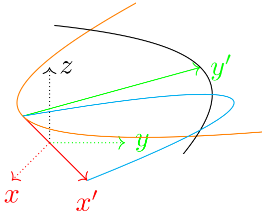
/tikz/canvas is plane (no value)

用前面的选项指定平面上的三个点——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选项之后。

```
\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}
```

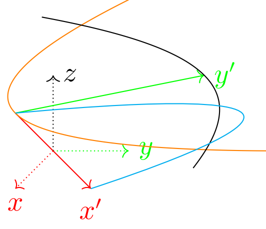



```
\begin{tikzpicture}[x=(-135:{0.5*sqrt(2)}),y=(0:1cm),z=(90:1cm)]
  \draw [->,red](0,0,0)--(1,0,0) node [below] {$x$};
  \draw [->,green](0,0,0)--(0,1,0) node [right] {$y$};
  \draw [->](0,0,0)--(0,0,1) node [right] {$z$};
  \draw [-Stealth](0,0,0)--(1,1,-1) node [right] {$u$};
  \begin{scope}[plane 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}
```

40.2.2 预定义的平面

`/tikz/canvas is xy plane at z=`⟨*dimension*⟩ (no default)

本选项设置:

- `plane origin={(0,0,`⟨*dimension*⟩`)}`
- `plane x={(1,0,`⟨*dimension*⟩`)}`
- `plane y={(0,1,`⟨*dimension*⟩`)}`

`/tikz/canvas is yx plane at z=`⟨*dimension*⟩ (no default)

本选项设置:

- `plane origin={(0,0,`⟨*dimension*⟩`)}`
- `plane x={(0,1,`⟨*dimension*⟩`)}`
- `plane y={(1,0,`⟨*dimension*⟩`)}`

`/tikz/canvas is xz plane at y=`⟨*dimension*⟩ (no default)

本选项设置:

- `plane origin={(0,`⟨*dimension*⟩`,0)}`
- `plane x={(1,`⟨*dimension*⟩`,0)}`
- `plane y={(0,`⟨*dimension*⟩`,1)}`

/tikz/canvas is zx plane at y=⟨*dimension*⟩ (no default)

本选项设置:

- plane origin={(0,⟨*dimension*⟩,0)}
- plane x={(0,⟨*dimension*⟩,1)}
- plane y={(1,⟨*dimension*⟩,0)}

/tikz/canvas is yz plane at x=⟨*dimension*⟩ (no default)

本选项设置:

- plane origin={(⟨*dimension*⟩,0,0)}
- plane x={(⟨*dimension*⟩,1,0)}
- plane y={(⟨*dimension*⟩,0,1)}

/tikz/canvas is zy plane at x=⟨*dimension*⟩ (no default)

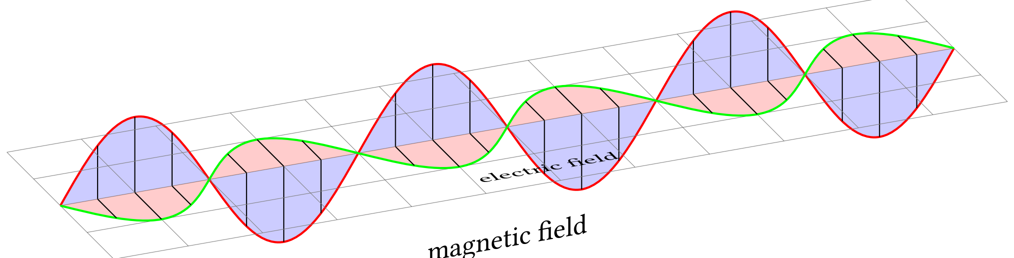
本选项设置:

- plane origin={(⟨*dimension*⟩,0,0)}
- plane x={(⟨*dimension*⟩,0,1)}
- plane y={(⟨*dimension*⟩,1,0)}

40.3 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}
```

63 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。

63.1 坐标系统

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。

`/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。

Coordinate system `tpp`

这是坐标系统 `three point perspective` 的别名。

63.2 设置视角

`/tikz/3d view`={⟨*azimuth*⟩}{⟨*elevation*⟩} (default {-30}{15})

⟨*azimuth*⟩ 代表经度，经度是围绕 z 轴旋转的角度，绕 z 轴右旋为正，y 轴负方向的经度是 0. ⟨*elevation*⟩ 代表纬度，北纬为正，南纬为负。本选项使得原点，点 (⟨*azimuth*⟩,⟨*elevation*⟩)，“眼睛”这三点共线，从而确定视角。

```
\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)

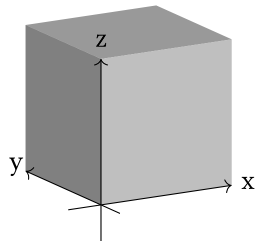
本选项选定一个特殊的视角，即 isometric view, 正等轴测图，等价于 `3d view={-45}{35.26}`, 其中 $35.26 \approx \arctan(1/\sqrt{2})$.

```
\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 条线，那么这 3 条线等分圆周角。

63.3 自定义透视

先定义一个立体图：

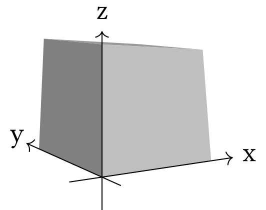



```
\newcommand\simplecuboid[3]{%
  \fill[gray!80!white] (tpp cs:x=0,y=0,z=#3)
    -- (tpp cs:x=0,y=#2,z=#3)
    -- (tpp cs:x=#1,y=#2,z=#3)
    -- (tpp cs:x=#1,y=0,z=#3) -- cycle;
  \fill[gray] (tpp cs:x=0,y=0,z=0)
    -- (tpp cs:x=0,y=0,z=#3)
    -- (tpp cs:x=0,y=#2,z=#3)
    -- (tpp cs:x=0,y=#2,z=0) -- cycle;
  \fill[gray!50!white] (tpp cs:x=0,y=0,z=0)
    -- (tpp cs:x=0,y=0,z=#3)
    -- (tpp cs:x=#1,y=0,z=#3)
    -- (tpp cs:x=#1,y=0,z=0) -- cycle;}
\newcommand{\simpleaxes}[3]{%
  \draw[->] (-0.5,0,0) -- (#1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-0.5,0) -- (0,#2,0) node[pos=1.1]{y};
  \draw[->] (0,0,-0.5) -- (0,0,#3) node[pos=1.1]{z};}

\begin{tikzpicture}[3d view]
\simplecuboid{2}{2}{2}
\simpleaxes{2}{2}{2}
\end{tikzpicture}
```

可以规定 3 个没影点，它们的名称分别是 `p`，`q`，`r`。

`/tikz/perspective`=⟨*vanishing points*⟩ (default `p={(10,0,0)},q={(0,10,0)},r={(0,0,20)}`)

本选项指定 3 个没影点。本选项的默认效果如下：




```
\begin{tikzpicture}[3d view,perspective]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```

`/tikz/perspective/p={(⟨x⟩,⟨y⟩,⟨z⟩)}` (no default, initially (0,0,0))

本选项设置没影点 p, 它的值必须是像 (1,2,3) 这样的 xyz 坐标形式。

`/tikz/perspective/q={(⟨x⟩,⟨y⟩,⟨z⟩)}` (no default, initially (0,0,0))

本选项设置没影点 q, 它的值必须是像 (1,2,3) 这样的 xyz 坐标形式。

`/tikz/perspective/r={(⟨x⟩,⟨y⟩,⟨z⟩)}` (no default, initially (0,0,0))

本选项设置没影点 r, 它的值必须是像 (1,2,3) 这样的 xyz 坐标形式。

63.4 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。

63.5 例子

63.6 代码实现

本程序库主要由文件《tikzlibraryperspective.code.tex》实现，此文件的主要内容如下：

```
\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}
```

以上代码规定单位矩阵

$$\begin{bmatrix} aa & ab & ac & 0 \\ ba & bb & bc & 0 \\ ca & cb & cc & 0 \\ da & db & dc & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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。

```
\def\pgfpointperspectivexyz#1#2#3{%
  \pgfmathsetmacro\pgf@pp@w%
    { \pgf@H@tpp@da*(#1) + \pgf@H@tpp@db*(#2) + \pgf@H@tpp@dc*(#3) + 1}
  \pgfmathsetmacro\pgf@pp@x%
    {(\pgf@H@tpp@aa*(#1) + \pgf@H@tpp@ab*(#2) + \pgf@H@tpp@ac*(#3))/\pgf@pp@w}
  \pgfmathsetmacro\pgf@pp@y%
    {(\pgf@H@tpp@ba*(#1) + \pgf@H@tpp@bb*(#2) + \pgf@H@tpp@bc*(#3))/\pgf@pp@w}
  \pgfmathsetmacro\pgf@pp@z%
    {(\pgf@H@tpp@ca*(#1) + \pgf@H@tpp@cb*(#2) + \pgf@H@tpp@cc*(#3))/\pgf@pp@w}
%
  \pgf@x=\pgf@pp@x\pgf@xx%
  \advance\pgf@x by \pgf@pp@y\pgf@yx%
  \advance\pgf@x by \pgf@pp@z\pgf@zx%
  \pgf@y=\pgf@pp@x\pgf@xy%
  \advance\pgf@y by \pgf@pp@y\pgf@yy%
  \advance\pgf@y by \pgf@pp@z\pgf@zy%
}
```

以上代码定义命令 `\pgfpointperspectivexyz`, 此命令有三个作用：规定乘积和齐次化操作：

$$\begin{bmatrix} aa & ab & ac & 0 \\ ba & bb & bc & 0 \\ ca & cb & cc & 0 \\ da & db & dc & 1 \end{bmatrix} \begin{pmatrix} \#1 \\ \#2 \\ \#3 \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ w \end{pmatrix} \xrightarrow{\text{齐次化}} \begin{pmatrix} ppx \\ ppy \\ ppz \\ 1 \end{pmatrix}$$

还规定 `\pgf@x` 和 `\pgf@y` 的值，也就是

$$\begin{pmatrix} pgfx \\ pgfy \end{pmatrix} = ppx \cdot \begin{pmatrix} xx \\ xy \end{pmatrix} + ppy \cdot \begin{pmatrix} yx \\ yy \end{pmatrix} + ppz \cdot \begin{pmatrix} zx \\ zy \end{pmatrix}$$

其中的 $\left(\begin{smallmatrix} xx \\ xy \end{smallmatrix}\right)$, $\left(\begin{smallmatrix} yx \\ yy \end{smallmatrix}\right)$, $\left(\begin{smallmatrix} zx \\ zy \end{smallmatrix}\right)$ 分别是 x, y, z 轴的单位向量。`\pgf@x` 和 `\pgf@y` 分别作为横坐标和纵半轴确定平面上的一个点。

在文件《pgfcorepoints.code.tex》中有：

```
% Internal registers
\newdimen\pgf@xx
\newdimen\pgf@xy
\newdimen\pgf@yx
\newdimen\pgf@yy
\newdimen\pgf@zx
\newdimen\pgf@zy
%......
\def\pgfsetxvec#1{%
  \pgf@process{#1}%
  \pgf@xx=\pgf@x%
  \pgf@xy=\pgf@y%
  \ignorespaces}
%......
\def\pgfsetyvec#1{%
  \pgf@process{#1}%
  \pgf@yx=\pgf@x%
```

```
  \pgf@yy=\pgf@y%
  \ignorespaces}
%......
\def\pgfsetzvec#1{%
  \pgf@process{#1}%
  \pgf@zx=\pgf@x%
  \pgf@zy=\pgf@y%
  \ignorespaces}
% Default values
\pgfsetxvec{\pgfpoint{1cm}{0cm}}
\pgfsetyvec{\pgfpoint{0cm}{1cm}}
\pgfsetzvec{\pgfpoint{-0.385cm}{-0.385cm}}
```

所以点 (\pgf@xx,\pgf@xy) 就是 x 轴的单位向量，默认值就是 (1cm,0cm). 点 (\pgf@yx,\pgf@yy) 就是 y 轴的单位向量，默认值就是 (0cm,1cm). 点 (\pgf@zx,\pgf@zy) 就是 z 轴的单位向量，默认值就是 (-0.385cm,-0.385cm). (有转换关系 1cm=28.45274pt).

```
\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}
```

上面代码声明坐标系统 three point perspective, 并且为该坐标系统指定一个别名 tpp, 参考 \tikzdeclarecoordina
当解析 (three point perspective cs:x=1,y=2,z=3) 时，会导致执行

```
\tikzset{cs/.cd,x=0,y=0,z=0,x=1,y=2,z=3}%
\pgfpointperspectivexyz{\tikz@cs@x}{\tikz@cs@y}{\tikz@cs@z}
```

导致

```
\pgfpointperspectivexyz{1}{2}{3}
```

```
\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
    },
    q/.code args={(#1,#2,#3)}{
      \pgfmathparse{ifthenelse(#2,int(1),int(0))}
      \ifnum\pgfmathresult=0\else
        \pgfmathsetmacro\pgf@H@tpp@ab{#1/#2}
        \pgfmathsetmacro\pgf@H@tpp@cb{#3/#2}
        \pgfmathsetmacro\pgf@H@tpp@db{ 1/#2}
      \fi
    },
```

```
    r/.code args={(#1,#2,#3)}{
      \pgfmathparse{ifthenelse(#3,int(1),int(0))}
      \ifnum\pgfmathresult=0\else
        \pgfmathsetmacro\pgf@H@tpp@ac{#1/#3}
        \pgfmathsetmacro\pgf@H@tpp@bc{#2/#3}
        \pgfmathsetmacro\pgf@H@tpp@dc{ 1/#3}
      \fi
    },
}
```

以上代码定义选项 /perspective/p, /perspective/q, /perspective/r, 即定义三个没影点的实际作用是怎样的。

上面代码中的 (#1,#2,#3) 是变量列举格式。\pgfkeys{/perspective/p={(p_x,p_y,p_z)}} 的处理过程是:

- 如果 p_x 等于 0, 则本选项无任何作用。
- 如果 p_x 不等于 0, 则执行

```
\pgfmathsetmacro\pgf@H@tpp@ba{p_y/p_x}
\pgfmathsetmacro\pgf@H@tpp@ca{p_z/p_x}
\pgfmathsetmacro\pgf@H@tpp@da{ 1/p_x}
```

也就是改变矩阵 $\begin{bmatrix} aa & ab & ac & 0 \\ ba & bb & bc & 0 \\ ca & cb & cc & 0 \\ da & db & dc & 1 \end{bmatrix}$ 第一列元素的值。

矩阵 $\begin{bmatrix} aa & ab & ac & 0 \\ ba & bb & bc & 0 \\ ca & cb & cc & 0 \\ da & db & dc & 1 \end{bmatrix}$ 原本是单位矩阵, 以上三个选项会把这个矩阵修改为:

$$\begin{bmatrix} 1 & q_x/q_y & r_x/r_z & 0 \\ p_y/p_x & 1 & r_y/r_z & 0 \\ p_z/p_x & q_z/q_y & 1 & 0 \\ 1/p_x & 1/q_y & 1/r_z & 1 \end{bmatrix}$$

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.

```
\tikzset{
  perspective/.append code={\pgfkeys{/perspective/.cd,#1}},
  perspective/.default={
    p={(10,0,0)},
    q={(0,10,0)},
    r={(0,0,20)}},
}
```

上面代码定义选项 /tikz/perspective 并规定其默认值, 即规定三个没影点的默认值。

```
\tikzset{
  3d view/.code 2 args={
    % Set elevation and azimuth angles
    \pgfmathsetmacro\pgf@view@az{#1}
    \pgfmathsetmacro\pgf@view@el{#2}
    % Calculate projections of rotation matrix
    \pgfmathsetmacro\pgf@xvec@x{cos(\pgf@view@az)}
```

```
    \pgfmathsetmacro\pgf@xvec@y{-sin(\pgf@view@az)*sin(\pgf@view@el)}
    \pgfmathsetmacro\pgf@yvec@x{sin(\pgf@view@az)}
    \pgfmathsetmacro\pgf@yvec@y{cos(\pgf@view@az)*sin(\pgf@view@el)}
    \pgfmathsetmacro\pgf@zvec@x{+0}
    \pgfmathsetmacro\pgf@zvec@y{cos(\pgf@view@el)}
    % Set base vectors
    \pgfsetxvec{\pgfpoint{\pgf@xvec@x cm}{\pgf@xvec@y cm}}
    \pgfsetyvec{\pgfpoint{\pgf@yvec@x cm}{\pgf@yvec@y cm}}
    \pgfsetzvec{\pgfpoint{\pgf@zvec@x cm}{\pgf@zvec@y cm}}
  },
  3d view/.default={-30}{15},
  isometric view/.style={3d view={-45}{atan(1/sqrt(2))}},
}
```

上面代码定义选项 `/tikz/3d view`. 经度保存在 `\pgf@view@az` 中，纬度保存在 `\pgf@view@el` 中。假设经度是 θ, 纬度是 ϕ, 那么 x 轴的单位向量是 $\left(\begin{smallmatrix}\cos\theta \\ -\sin\theta\cdot\sin\phi\end{smallmatrix}\right)$; y 轴的单位向量是 $\left(\begin{smallmatrix}\sin\theta \\ \cos\theta\cdot\sin\phi\end{smallmatrix}\right)$; z 轴的单位向量是 $\left(\begin{smallmatrix}0 \\ \cos\phi\end{smallmatrix}\right)$, 这三个单位向量的长度单位都是 cm. 这三个单位向量就是命令 `\pgfpointperspectivexyz` 定义中的 $\left(\begin{smallmatrix}xx \\ xy\end{smallmatrix}\right)$, $\left(\begin{smallmatrix}yx \\ yy\end{smallmatrix}\right)$, $\left(\begin{smallmatrix}zx \\ zy\end{smallmatrix}\right)$.

26 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。

26.1 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。Ti*k*Z 的 animations 库不是这样工作的。目前，Ti*k*Z 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 Ti*k*Z 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，Ti*k*Z 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 Ti*k*Z 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}
}
```

26.2 创建动画

26.2.1 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 TikZ 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 TikZ 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。

$t = 0$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 [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. 190.

26.2.2 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，TikZ 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.

$t = 0\text{s}$ $t = 0.5\text{s}$ $t = 1\text{s}$ $t = 1.5\text{s}$ $t = 2\text{s}$

Click me Click me Click me Click me Click me

```
\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：

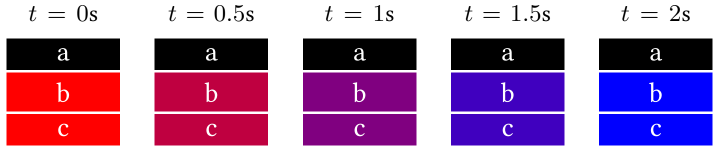
```
\tikz [animate = {
  object = node, attribute = fill, time = 0s, value = red, entry,
                                  time = 2s, value = blue, entry,
                                  begin on = click, entry}]
  \node (node) [fill, text=white] { Click me };
```

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。

26.2.3 指定对象

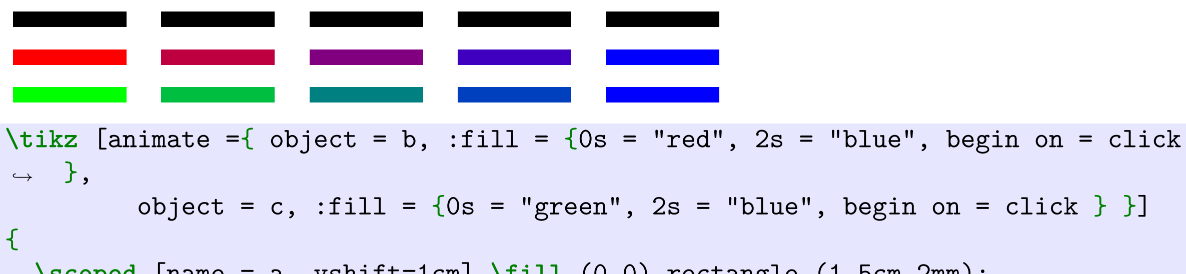
/tikz/animate/object=⟨*list of object*⟩ (no default)

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。

26.2.4 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。

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s

The node The node The node The node The node

```
\tikz [animate = {attribute = fill, n: = { 0s = "red", 2s = "blue", begin on =
↪ click } }]
  \node (n) [fill, text = white] {The node};
```

26.2.5 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

$t = 0s$ $t = 0.5s$ $t = 1s$ $t = 1.5s$ $t = 2s$

The node The node The node The node The node

```
\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 TikZ 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. 304 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。

26.2.6 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→P. 808 解析，这个命令类似 TikZ 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};
```

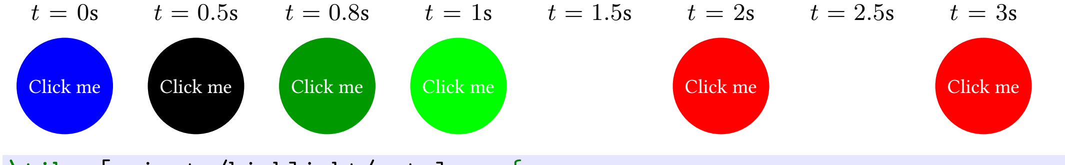
等效于

```
\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. 289 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` $-\varepsilon$ (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
```

26.2.7 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:

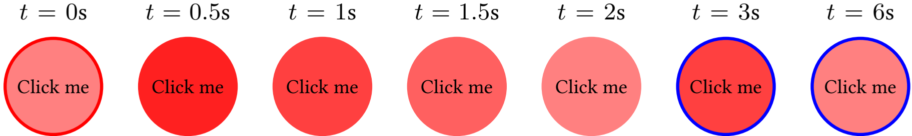
- 如果一个时间线中使用 current value，那么 current value 只能用作第一个时刻的属性值。
- 如果一个时间线中使用 current value，那么这个时间线中只能设置两个时刻。
- 如果一个时间线中使用 current value，那么这个时间线就不支持快照操作（snapshot，参考选项 /tikz/make snapshot of→P. 309）。

26.2.8 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。

26.3 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。

26.3.1 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}
```

26.3.2 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。

26.3.3 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

$$\langle time\rangle\text{=}\langle options\rangle$$

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 }`

26.3.4 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。

26.3.5 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"
    }
  }
}
```

26.4 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`

26.4.1 Animating Color, Opacity, and Visibility

Animation attribute :fill, :draw

这是填充色属性和线条颜色属性。

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s

A B C A B C A B C A B C A B C

```
\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。

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s

A B A B A B A B A B

```
\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

A B A B A B A B A B

```
\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};
}
```

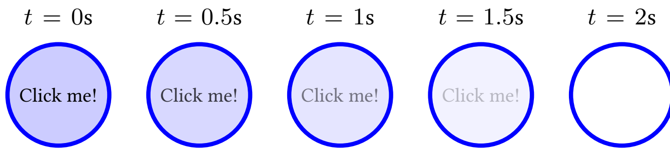
上面代码中的属性 :text 用作 \tikz 的选项，它对 \node 命令无效，node 的文字颜色总是决定于 text = white.

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。




```
\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。

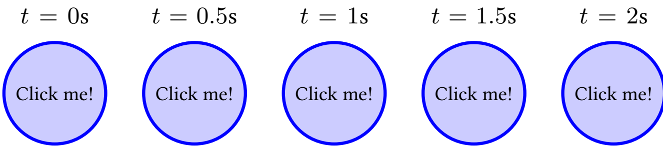
$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s

Click me! Click me! Click me! Click me!

```
\tikz \node :opacity = { 0s="1", 2s="0", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle, font=\footnotesize]
  ↪  {Click me!};
```

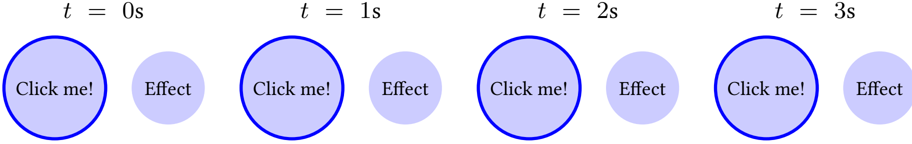
Animation attribute :visible, :stage

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。




```
\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".

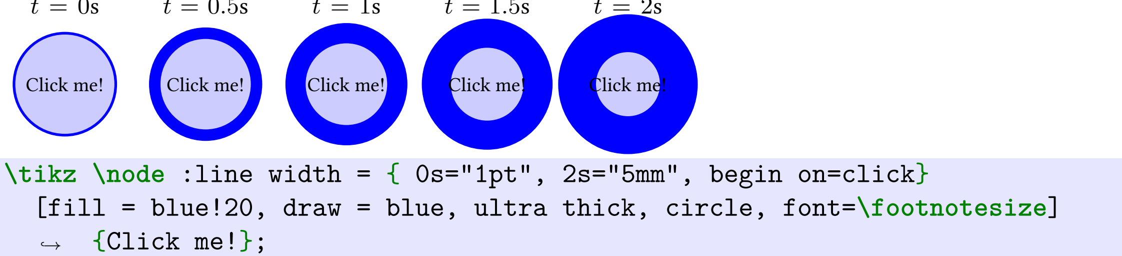



```
\tikz [font=\footnotesize, animate = {example:stage = {
                  begin on = {click, of next=node},
                  0s="true", 2s="true" }}] {
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!
  ↪  };
  \node at (1.5,0) (example) [fill = blue!20, circle] {Effect};
}
```

26.4.2 Animating Paths and their Rendering

Animation attribute :line width

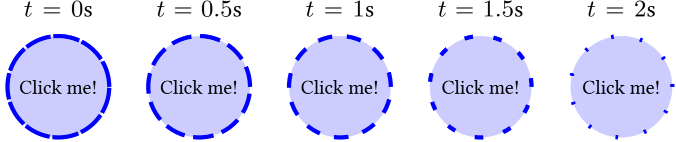
这个属性针对路径的线宽。注意此属性的值必须是带长度单位的数字或表达式,不能使用“thin, thick”这种样式（style）. 注意“线宽”是图形的状态参数，不属于路径本身，一般不会计入图形的边界盒子内。




```
\tikz \node :line width = { 0s="1pt", 2s="5mm", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle, font=\footnotesize]
  ↪ {Click me!};
```

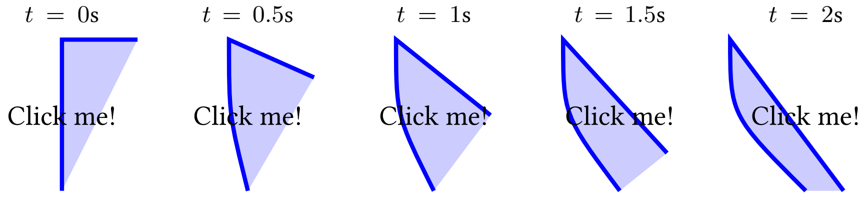
Animation attribute :dash, :dash phase, :dash pattern

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

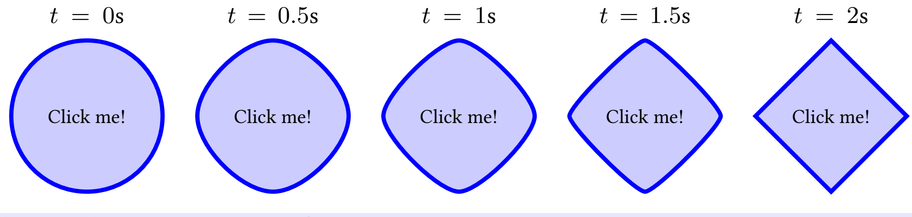



```
\tikz \node :path = {
    0s = "{(0,-1) .. controls (0,0) and (0,0) .. (0,1) -- (1,1)}",
    2s = "{(0,-1) .. controls (-1,0) and (-1,0) .. (-1,1) -- (.5,-1)}",
    ↪ begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

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!};
```

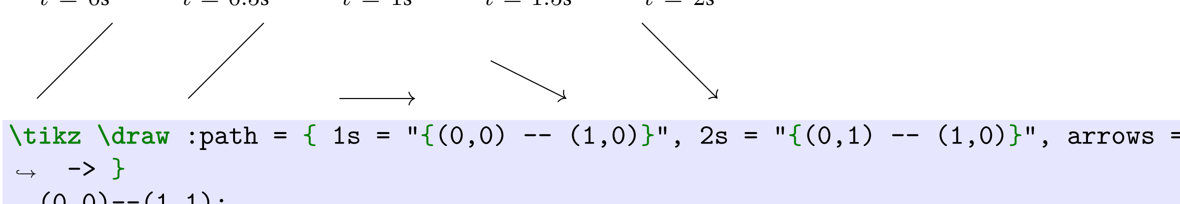
- 若路径被做成动画并且要在该路径上添加箭头，就必须使用选项 `/tikz/animate/arrows` 来设置箭头，不能使用通常的办法加箭头。

`/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：

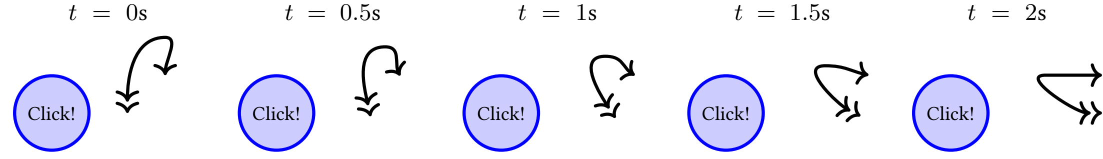



```
\tikz \draw :path = { 1s = "{(0,0) -- (1,0)}", 2s = "{(0,1) -- (1,0)}", arrows =
↪   -> }
  (0,0)--(1,1);
```

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）：

t = 0s t = 0.5s t = 1s t = 1.5s t = 2s

```
\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)

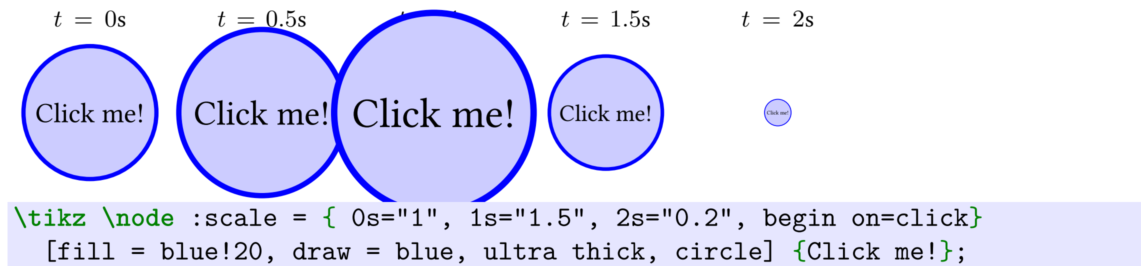
/tikz/animate/shorten >=⟨*dimension*⟩ (no default)

这两个选项只能用在属性 `:path` 中。

26.4.3 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。

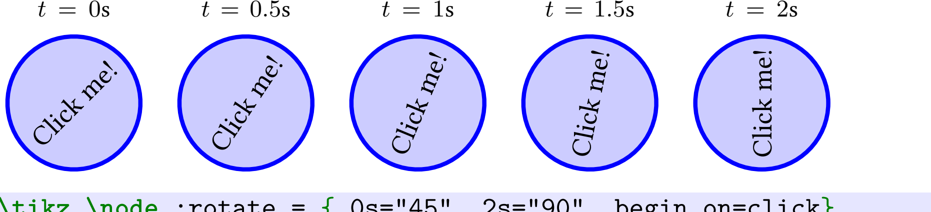
Animation attribute `:scale, :xscale, :yscale`




```
\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. 302 选项来改变旋转的“原点”。

Animation attribute :xskew, :yskew, :xslant, :yslant

$t = 0$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 :xskew = { 0s="0", 2s="45", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

$t = 0$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 :xslant = { 0s="-1", 2s="1", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

Animation attribute :xshift, :yshift

$t = 0$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 :shift = { 0s="{(0,0)}", 2s="{(5mm,-5mm)}",begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

$t = 0$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 :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：

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s

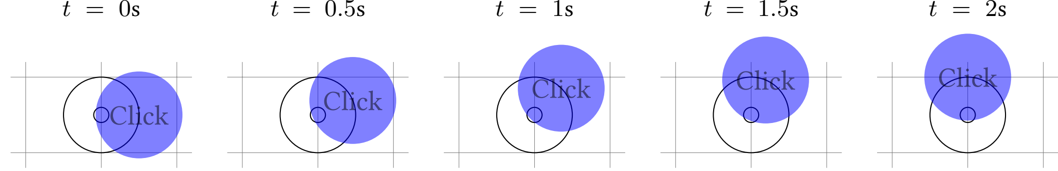
Click me! Click me! Click me! Click me! Click me!

```
\tikz \node :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)

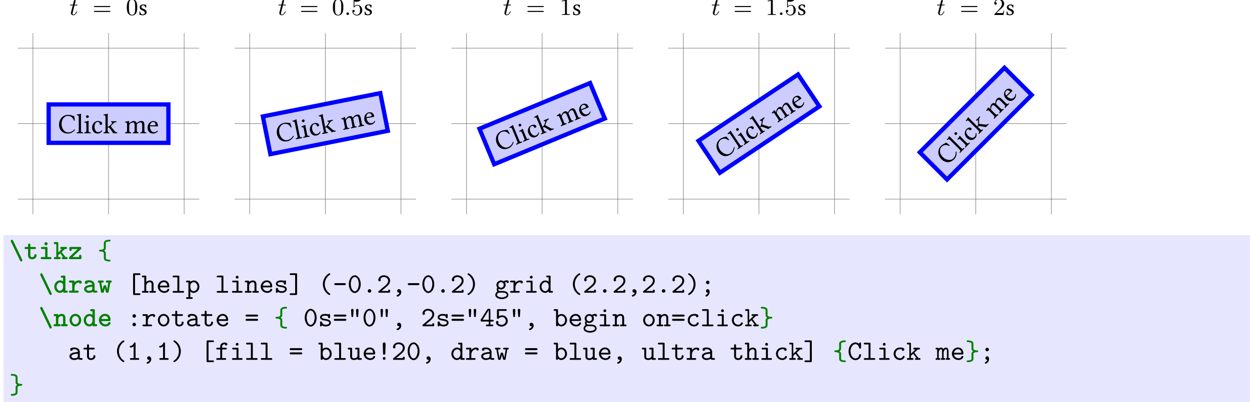
这个选项用在属性 `:shift` 或 `:position` 中，可以使得对象沿着 ⟨*path*⟩ 移动。使用本选项后，编写的属性值就不能再是坐标，而应该是“小数”，数值 0 代表 ⟨*path*⟩ 的起点，数值 1 代表 ⟨*path*⟩ 的终点。⟨*path*⟩ 必须放入花括号中，关键词 `sloped` 或 `upright` 是必选的，而 `in`⟨*time*⟩ 是可选的。如果写出可选项 `in 8s`，则它等效于 `0s="0"`，`8s="1"`，也就是说，在从 `0s` 到 `8s` 的时间内，对象沿着路径，从路径的起点运动到路径的终点。




```
\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};
}
```

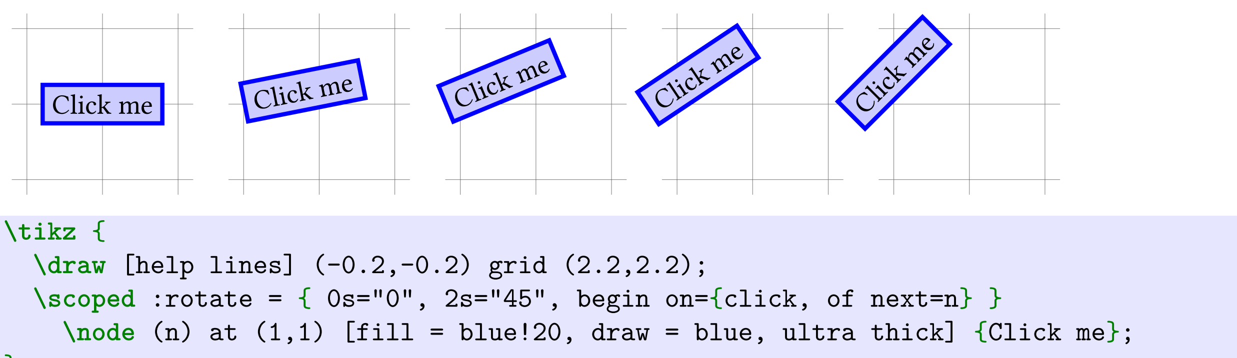
上面例子中，node 的圆心位于坐标点 `(1,0.5)`, 以这个点为原点的坐标系是 animation coordinate system，选项 `along` 规定的路径 `{(0,0) circle[radius=5mm]}` 就是 animation coordinate system 中的路径，node 的中心沿着这个路径移动形成动画效果。

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：



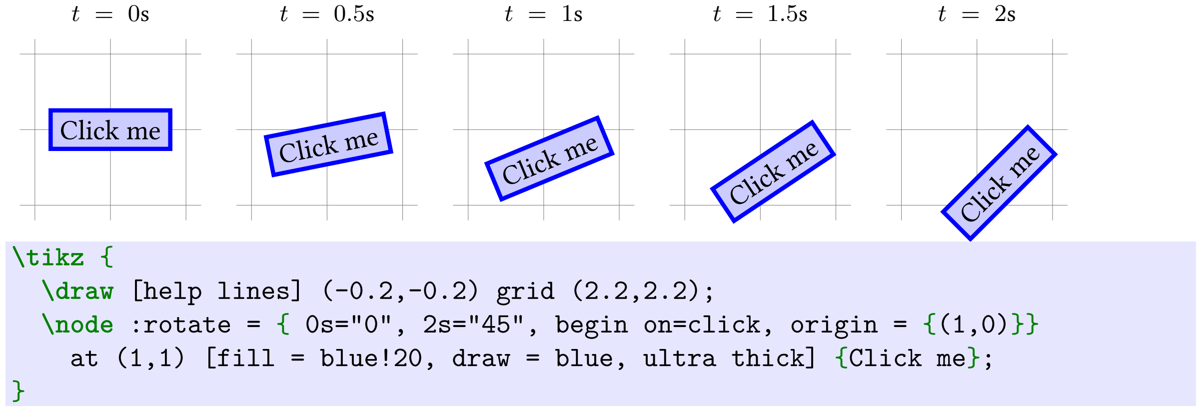

```
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \scoped :rotate = { 0s="0", 2s="45", begin on={click, of next=n} }
    \node (n) at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
}
```

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。

26.4.4 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。

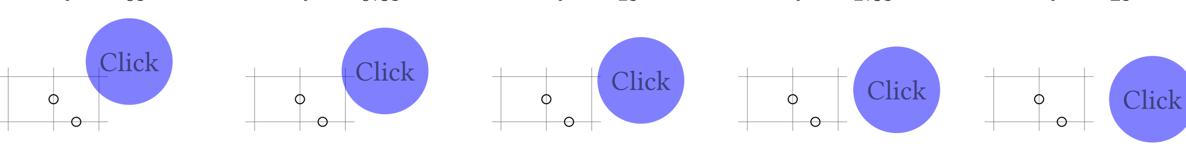
$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s

Click Click Click Click Click

```
\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:

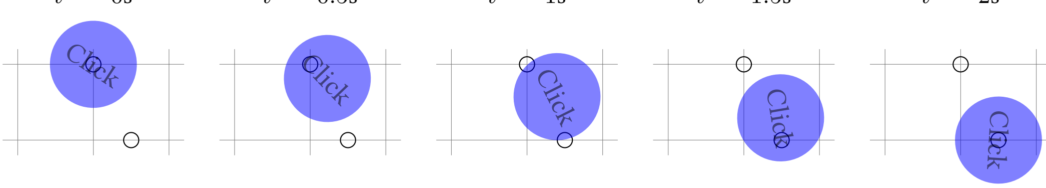
$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s




```
\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`.

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s



```
\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。

26.4.5 Animating Transformations: Views

Animation attribute `:view`

这个属性实施画布变换，参考 views 库。

26.5 调控时间线

26.5.1 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：

$t = 0$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 = 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：

$t = 0$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 = 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：

$t = 0$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 = 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。

$t = 0$s $t = 1$s $t = 2$s $t = 3$s

Click me Click me Click me Click me

```
\tikz \node :fill = { 1s="red", 2s="blue", forever, begin on=click}
  [fill = green!50!black, text = white] {Click me};
```

对比：

$t = 0$s $t = 1$s $t = 2$s $t = 3$s

Click me Click me Click me Click me

```
\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。

26.5.2 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. 309.

/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. 309.

/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. 309.

/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”。

26.5.3 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。

26.5.4 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)

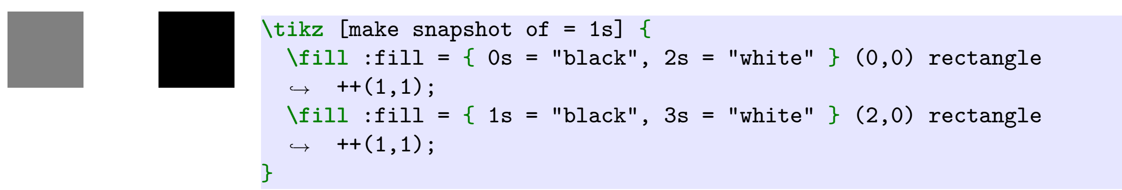
26.6 Snapshots

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。



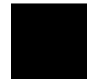
```
\tikz [make snapshot of = 1s] {
  \fill :fill = { 0s = "black", 2s = "white" } (0,0) rectangle
  ↪  ++(1,1);
  \fill :fill = { 1s = "black", 3s = "white" } (2,0) rectangle
  ↪  ++(1,1);
}
```

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);
}
```

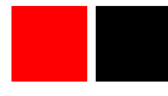
对于快照的计算完全由 Tikz 完成，限于 TEX 的计算能力，对快照的计算可能不是非常精确、快速，另外还要注意以下几点：

- 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. 307.

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 $\langle time \rangle + \epsilon$. 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。

26.7 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。

78 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。

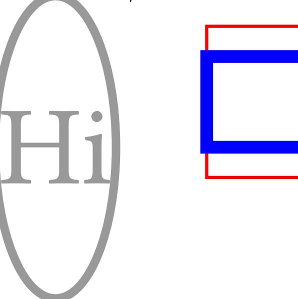
一个 view 就是一个 window, 透过这个窗口可以看到图形（graphic)。为了创建一个 view, 需要创建两个矩形，一个矩形作为“固定窗口”，另一个矩形作为“可变矩形”（称之为 to-be-viewed rectangle)。固定窗口是不变的，tikz 会对可变矩形和其它图形做 画布变换，对其进行平移、放缩，使得可变矩形的

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. 757.

/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。



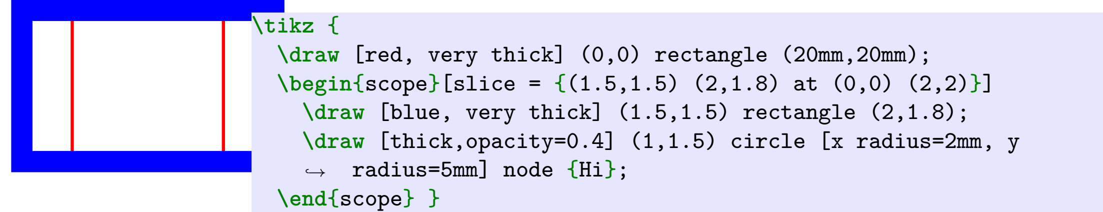

```
\tikz {
  \draw [red, very thick] (0,0) rectangle (20mm,20mm);
  \begin{scope}[meet = {(1.5,1.5) (2,1.8) at (0,0) (2,2)}]
    \draw [blue, very thick] (1.5,1.5) rectangle (2,1.8);
    \draw [thick,opacity=0.4] (1,1.5) circle [x radius=2mm, y
    ↪  radius=5mm] node {Hi};
  \end{scope} }
```

/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} }
```

49 Circuits 程序库

Written and documented by Till Tantau, and Mark Wibrow. Inspired by the work of Massimo Redaelli.

49.1 简介

电路 (circuit) 程序库用于创建各种电子电路和逻辑电路，它并不是一个单一的程序库，它是数个相互关联的程序库的统称；它在创建高质量电路图时，力求在“易用”与“易于扩展”之间取得平衡。

49.1.1 一个例子

```
\begin{tikzpicture}[circuit ee IEC,x=3cm,y=2cm,semithick,
                    every info/.style={font=\footnotesize},
                    small circuit symbols,
                    set resistor graphic=var resistor IEC graphic,
                    set diode graphic=var diode IEC graphic,
                    set make contact graphic= var make contact IEC
                    ↪  graphic]
% Let us start with some contacts:
  \foreach \contact/\y in {1/1,2/2,3/3.5,4/4.5,5/5.5}
  {
    \node [contact] (left contact \contact) at (0,\y) {};
    \node [contact] (right contact \contact) at (1,\y) {};
  }
  \draw (right contact 1) -- (right contact 2) -- (right contact 3)
    -- (right contact 4) -- (right contact 5);
  \draw (left contact 1) to [diode] ++(down:1)
                         to [voltage source={near start,
                                             direction info={volt=3}},
                             resistor={near end,ohm=3}] ++(right:1)
                         to (right contact 1);
  \draw (left contact 1) to [resistor={ohm=4}] (right contact 1);
  \draw (left contact 1) to [resistor={ohm=3}] (left contact 2);
  \draw (left contact 2) to [voltage source={near start,
                                             direction info={<-,volt=8}},
                             resistor={ohm=2,near end}] (right contact
                             ↪  2);
  \draw (left contact 2) to [resistor={near start,ohm=1},
                             make contact={near end,info'={[red]$S_1$}}]
                         (left contact 3);
  \draw (left contact 3) to [current direction'={near start,info=$\iota$
  ↪  },
                             resistor={near end,info={$R=4\Omega$}}]
                         (right contact 3);
  \draw (left contact 4) to [voltage source={near start,
                                             direction info={<-,volt=8}},
                             resistor={ohm=2,near end}] (right contact
                             ↪  4);
  \draw (left contact 3) to [resistor={ohm=1}] (left contact 4);
  \draw (left contact 4) to [resistor={ohm=3}] (left contact 5);
  \draw (left contact 5) to [resistor={ohm=4}] (right contact 5);
  \draw (left contact 5) to [diode] ++(up:1)
                         to [voltage source={near start,
                                             direction info={volt=3}},
                             resistor={near end,ohm=3}] ++(right:1)
                         to (right contact 5);
\end{tikzpicture}
```

上面例子中，`{tikzpicture}` 环境选项中有 `circuit ee IEC`, 整个 key 把环境改造为一个“电路图”环境。

电路程序库的一个重要特点是，电路的外观可以按通常的方式调整，图形中的各种标签都按默认方式自动排布。

49.1.2 符号

一个电路通常包含数种电子元件 (electronic elements)，如用导线 (wire) 连接起来的逻辑门 (logical gate)，电阻 (resistor)，二极管 (diode) 等。在 PGF/TikZ 中，把电子元件做成 node, 用通常的线条来代表导线。TikZ 提供了很多方式来排布、连接 node, 这些方式都可以用在这里。另外，程序库 circuits 定义了一种 `to` 路径，在画线上的电子元件时很有用。为了统一说法，把各种元件称为“符号”(symbols). 一个符号形状 (symbol shape) 就是用命令 `\pgfdeclareshape`→P. 717 声明的 PGF 形状 (PGF shape). 一个 symbol node 就是具有 symbol shape 的 node.

49.1.3 Symbol Graphics

Symbols 可以用命令 `\node[shape=`⟨*some symbol shape*⟩`]` 来创建。为了准确地画出 symbols, 只用标准的 PGF shape 是不够的。例如，多数 symbols 都有默认尺寸 (default size), 但是 symbol shape 的尺寸却依赖当前的参数值，如 `minimum height`，`inner xsep`. 因此电路程序库引入了 symbol graphic 这个概念，它是个样式 (style)，它不仅使得 `\node` 具有准确的形状，也有正确的尺寸。当你写下，例如，`\node[diode]` 时，有一个被称为 `diode graphic` 的样式会被调用，这个样式的效果就类似使用 `shape=diode IEC,draw,minimum height=....` 那样。

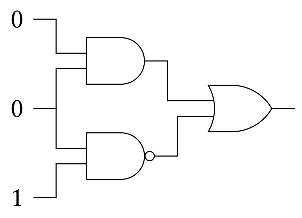
下面是各种电路程序库的总览。

- `circuits`, 这是个 TikZ 程序库，它定义了创建电路图的一般的 keys, 这些 keys 通常用于定义其他更专门的程序库。本程序库没有定义任何 symbol graphic, 因此一般用不到它。
- `circuits.logic`, 这是个 TikZ 程序库，它定义了创建逻辑门的 keys. 本程序库也没有定义任何 symbol graphic, 定义逻辑门的 symbol graphic 的是下面两个程序库：
  - `circuits.logic.US`, 这个程序库定义的逻辑门 symbol graphic 具有“美式风格”(US-style), 它包含了以上两个程序库，可以直接使用它。
  - `circuits.logic.IEC`, 这个程序库定义一种逻辑门 symbol graphic, 所创建的逻辑门是矩形门，而不是圆角的“美式门”(US-gates). 这个程序库可以与以上程序库很好的兼容。
- `cirucits.ee`, 这个程序库定义的 keys 创建电子工程 (electrical engineering) 的符号 (symbols), 例如，电阻 (resistors)，电容 (capacitors). 本程序库没有相关的 symbol graphic, 相关的 symbol graphic 由下面的程序库定义。
  - `circuits.ee.IEC`, 这个程序库定义符合 IEC 规范的 symbol shapes.
- `shapes.gates.*`, 这一类程序库定义 symbol shapes. 通常你不需要直接使用这些 symbol shapes, 使用由这些 symbol shapes 定义的样式会更便利。

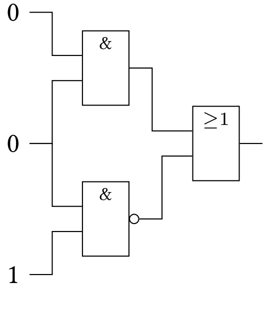
下面再看一个例子。要画一个逻辑电路，先确定使用哪些（哪种）symbol graphics. 如果要用美式风格的，就载入程序库 `circuits.logic.US`; 如果要用 IEC 风格的，就载入程序库 `circuits.logic.IEC`; 如果你不确定使用哪种风格的，那就把这两个程序库都载入。

```
\usetikzlibrary{circuits.logic.US, circuits.logic.IEC}
```

程序库 `circuits.logic.US` 定义的 `and gate` 这个 key 会利用相应的 symbol graphic 创建一个“美式的”与门；程序库 `circuits.logic.IEC` 定义的 `and gate` 这个 key 会利用相应的 symbol graphic 创建一个 IEC 风格的与门。



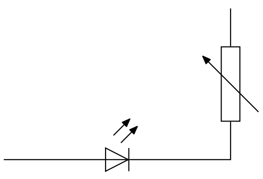

```
\begin{tikzpicture}[circuit logic US]
  \matrix[column sep=7mm]
  {
    \node (i0) {0}; &                               & \\
                    & \node [and gate] (a1) {};  & \\
    \node (i1) {0}; &                               & \node [or gate] (o)
    ↪  {};\\
                    & \node [nand gate] (a2) {}; & \\
    \node (i2) {1}; &                               & \\
  };
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
  \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
  \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
  \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```




```
\begin{tikzpicture}[circuit logic IEC]
  \matrix[column sep=7mm]
  {
    \node (i0) {0}; &                               & \\
                    & \node [and gate] (a1) {};  & \\
    \node (i1) {0}; &                               & \node [or gate] (o)
    ↪  {};\\
                    & \node [nand gate] (a2) {}; & \\
    \node (i2) {1}; &                               & \\
  };
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
  \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
  \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
  \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

49.1.4 Annotations

一个 annotation 是额外绘制的、添加到 symbol 上的小图形。例如，你可能需要在某个电子元件上画两个平行箭头，用来指示该元件是个发光体。代表二极管的 symbol 只有一个 `diode`，但有的二极管是不发光的，有的是发光的，通过给 `diode` 添加 `light emitting` 这个 annotation 参数来得到发光二极管符号，如下例：



```
\tikz [circuit ee IEC]
  \draw (0,0) to [diode={light emitting}] (3,0)
              to [resistor={adjustable}] (3,2);
```

49.2 circuits 程序库

TikZ Library circuits

```
\usetikzlibrary{circuits} % LATEX and plain TEX
\usetikzlibrary[circuits] % ConTEXt
```

这是个基础程序库，它被包含在其它电路程序库内，你不需要直接调用这个程序库，不过它提供了几个很有用的一般性 keys, 本节介绍这些 keys.

`/tikz/circuits` (no value)

如果某个环境中包含绘制电路图的命令，那么此环境的可选项中就应该含有这个 key. 这个 key 会引起某些内部设置 (internal setups). 这个 key 会被更加专门化的 keys 调用，例如 `/tikz/circuit ee IEC`→P. 345, 也就是说，给环境带上 `circuit ee IEC` 后就不必再写出 `circuits`.

49.2.1 Symbol 的尺寸

`/tikz/circuit symbol unit=`⟨*dimension*⟩ (no default, initially 7pt)

这里的 ⟨*dimension*⟩ 是 symbols 的“单位尺寸”，程序库定义的各种 symbols 的尺寸都与这个单位尺寸相关。例如，在 IEC 程序库中，线圈 (`inductor`) 的默认长度是 ⟨*dimension*⟩ 的 5 倍。如果改变 ⟨*diamension*⟩, 那么所有 symbols 的尺寸都会被改变。

可以局部地使用此 key 来改变某个或某些 symbol 的尺寸。

```
\begin{tikzpicture}[circuit ee IEC]
  \draw (0,1) to [resistor] (3.5,1);
  \draw[circuit symbol unit=14pt]
        (0,0) to [resistor] (3.5,0);
\end{tikzpicture}
```

`/tikz/huge circuit symbols` (style, no value)

这个样式设置 `circuit symbol unit=10pt`.

`/tikz/large circuit symbols` (style, no value)

这个样式设置 `circuit symbol unit=8pt`.

`/tikz/medium circuit symbols` (style, no value)

这个样式设置 `circuit symbol unit=7pt`.

`/tikz/small circuit symbols` (style, no value)

这个样式设置 `circuit symbol unit=6pt`.

`/tikz/tiny circuit symbols` (style, no value)

这个样式设置 `circuit symbol unit=5pt`.

/tikz/circuit symbol size=width ⟨*width*⟩ height ⟨*height*⟩ (no default)

这里的 ⟨*width*⟩, ⟨*height*⟩ 都是纯数值，这个 key 把 symbol 的宽度设为 circuit symbol unit 与 ⟨*width*⟩ 的乘积，把 symbol 的高度设为 circuit symbol unit 与 ⟨*height*⟩ 的乘积。这个 key 不仅可以确定电路 symbol 的尺寸，也可以用来确定 node 的尺寸。

```
\begin{tikzpicture}[circuit ee IEC]
  \draw (0,1) to [resistor] (2,1) to[inductor] (4,1);
  \begin{scope}
  [every resistor/.style={circuit symbol size=width 3 height 1}]
    \draw (0,0) to [resistor] (2,0) to[inductor] (4,0);
  \end{scope}
\end{tikzpicture}
```

49.2.2 声明新的 symbols

/tikz/circuit declare symbol=⟨*name*⟩ (no default)

这个 key 用于声明一个新的 symbol. 这个声明仅仅为 ⟨*name*⟩ 创建几个 keys, 用于稍后绘制或设置这个 symbol, 这个声明并不创建 ⟨*name*⟩ 所对应的图形。

具体说，此声明定义的第一个 key 的名称也是 ⟨*name*⟩, 这个 key 应该用作 node 的选项 (option)，或者用作 to 路径的选项。这个 key 可以带有数个选项，以调整相应 symbol graphic 的外观。

看一个例子，假设我们要定义一个名称为 foo 的 symbol, 其形状是个矩形，可以用下面的命令声明这个 symbol:

```
\tikzset{circuit declare symbol=foo}
```

然后这个 symbol 就可以如下使用：

```
\node [foo]        at (1,1) {};
\node [foo={red}] at (2,1) {};
```

尽管上面两个 \node 命令能够顺利编译，但仍然看不到相应的图形，因为还没有指定与 foo 相对应的符号图形。因此，需要使用名称为 set foo graphic 的 key 来为 foo 设置它的 symbol graphic. 一般地，使用

set ⟨*name*⟩ graphic

这个 key 来为符号 ⟨*name*⟩ 设置 symbol graphic. 这个 key 以图形选项 (graphic options) 为参数 (parameter), 这些图形选项规定 ⟨*name*⟩ 对应的图形：

```
\begin{tikzpicture}
  [circuit declare symbol=foo,
   set foo graphic={draw,shape=rectangle,minimum size=5mm}]
  \node [foo]        at (1,1) {};
  \node [foo={red}] at (2,1) {};
\end{tikzpicture}
```

当在 node 中使用 ⟨*name*⟩=⟨*options*⟩ 这种 key 时，就会引发如下操作：

- 选项 /pgf/inner sep→P. 708 被设为 0.5pt.
- 执行下面的样式 (style):

/tikz/every circuit symbol (style, no value)

这个样式针对（其作用范围内的）各个 symbol 做一般性地设置。

- 由 `set ⟨name⟩ graphic` 设置图形选项 (graphic options) 会被使用。
- 执行样式 `every ⟨name⟩`, 可以用这个样式对 symbol 做进一步地设置。
- 执行 ⟨options⟩.

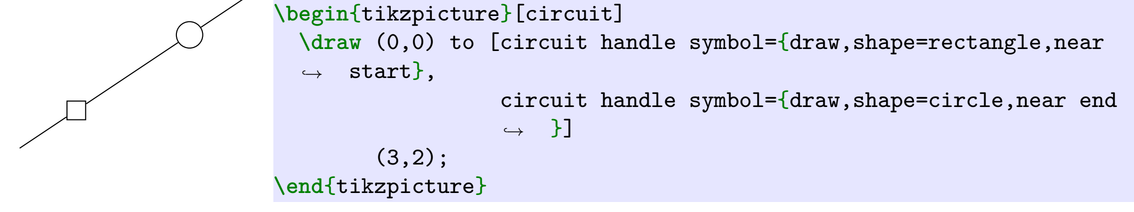
在 `circuit` 环境（带有 `circuit` 选项的环境）中，当 ⟨name⟩ 这个 key 用于 `to` 路径时，它的作用有所不同，此时也会执行以上 5 个步骤的操作，但他们会被传递给 `circuit handle symbol`.

`/tikz/circuit handle symbol=⟨options⟩` (no default)

这个 key 主要用于内部处理过程中，用它渲染 (render) symbol. 这个 key 在 `to` 路径中的作用与在其它地方的作用有所不同。如果这个 key 不是用作 `to` 路径命令的参数，那么就简单地执行 ⟨options⟩. 如果这个 key 用作 `to` 路径命令的参数，那么会发生以下操作：

- `to` 路径会被改造为一个内部路径 (internal path), 多数情况下这个内部路径只是一条线段。
- 过滤 ⟨options⟩ 中的选项，并执行滤得的选项。滤得的选项是这几个：`pos`，`at start`，`very near start`，`near start`，`midway`，`near end`，`very near end`，`at end`, 如果没有滤得这几个选项中的任何一个，就自动使用选项 `midway`.
- 由上一步滤得的选项能决定路径上的一个位置，选项 `pos=0` 对应路径的起点，选项 `pos=1` 对应路径的终点；在这个位置上添加 symbol (即一个 node).
- 所添加的 node 把 ⟨options⟩ 作为它的选项。
- 添加 node 的操作是 `markings` 装饰操作。也就是说，执行一个 `mark` 命令，把 node 作为一个 mark 放置到路径上。
- `markings` 装饰操作会自动分割路径，并把路径的起点与 node 边界上的某个点（这个点位于路径起点与 node 中心点的连线上）用线段连起来，把路径的终点与 node 边界上的某个点（这个点位于路径终点与 node 中心点的连线上）用线段连起来。
- 可以用 `markings` 装饰操作在一个路径上放置多个 marks, 此时相邻两个 node 之间的连线是从边界到边界的。
- `markings` 装饰操作也能旋转绘制 mark 的坐标系，使其 x-axis 指向路径的切线方向。如果使用选项 `transform shape`, 那么 node 就会沿着路径方向而倾斜、旋转。
- 对于 `pos=0`, `pos=1` 的情况会执行某些代码，以避免在路径的起点或终点处出现多余的线段。

简单地说，以上操作的结果就是在路径的指定位置处制造一个缺口，恰好容纳所需要的 node; 可以使用多个 `circuit handle symbol` 在路径上添加数个 node, 最好别让它们重叠。



```
\begin{tikzpicture}[circuit]
  \draw (0,0) to [circuit handle symbol={draw,shape=rectangle,near
  ↪   start},
                 circuit handle symbol={draw,shape=circle,near end
                 ↪   }]
        (3,2);
\end{tikzpicture}
```

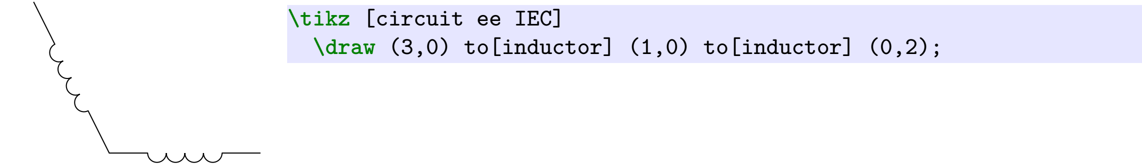
```
\begin{tikzpicture}[transform shape,circuit]
  \draw (0,0) to [circuit handle symbol={draw,shape=rectangle,at
  ↪ start},
                  circuit handle symbol={draw,shape=circle,near end
                  ↪ }]
      (3,2);
\end{tikzpicture}
```

```
\begin{tikzpicture}[transform shape,circuit]
  \draw (0,0) to [circuit handle symbol={draw,shape=rectangle,at
  ↪ start},
                  circuit handle symbol={draw,shape=circle,pos=0.4},
                  circuit handle symbol={draw,shape=circle,pos=0.44
                  ↪ },
                  circuit handle symbol={draw,shape=circle,near end
                  ↪ }]
        (3,2);
\end{tikzpicture}
```

49.2.3 让 symbol 指向某个方向

有两种方法旋转 symbol 使之指向某个方向:

1. 当把一个 symbol 放到 `to` 路径上时，它的 symbol graphic 会被沿着路径方向自动旋转。看下面线圈的例子:



```
\tikz [circuit ee IEC]
  \draw (3,0) to[inductor] (1,0) to[inductor] (0,2);
```

2. 有时把多个 symbols 看作是矩阵的元素 (cell) 会比较方便，即创建一个矩阵（如使用命令 `\matrix` 创建矩阵），把某些个 symbol 作为它的元素，然后用线把这些 symbols 连起来，此时可以给这些 symbols 带上旋转选项（如 `rotate=90`）使之指向正确的方向。下面是几个特别定义的选项:

`/tikz/point up` (no value)

等效于 `rotate=90`.

```
\tikz [circuit ee IEC] \node [diode,point up] {};
```

`/tikz/point down` (no value)

等效于 `rotate=-90`.

```
\tikz [circuit ee IEC] \node [diode,point down] {};
```

`/tikz/point left` (no value)

等效于 `rotate=-180`.

```
\tikz [circuit ee IEC] \node [diode,point left] {};
```

`/tikz/point right` (no value)

这个 key 没有作用（继续保留原来的方向）。

```
\tikz [circuit ee IEC] \node [diode,point right] {};
```

49.2.4 Info 标签

Info labels 是给电路 symbol 添加标签的一个方式，symbol 的标签通常位于它的旁边。当然可以使用选项 `label` 来为 symbol 添加标签，而选项 `info` 就是以选项 `label` 为基础定义的，不过选项 `info` 有一些预定义变量 (predefined variants), 在电路图中使用 `info` 添加标签会比较合适。

`/tikz/info=[`⟨*options*⟩`]`⟨*angle*⟩`:`⟨*text*⟩ (no default)

这个 key 与 `label` 很类似。程序自动把下面的样式应用于这个 key:

`/tikz/every info` (style, no value)

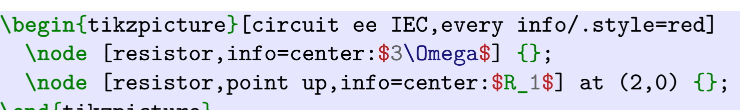
这个样式用于设置 info labels 的外观。这个样式只是针对 info labels, 不影响其它的标签。

这里的 ⟨*options*⟩ 与 ⟨*angle*⟩ 会被直接传递给 `label` 命令。

3Ω

```
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \node [resistor,info=$3\Omega$] {};
\end{tikzpicture}
```

提示：如果要把标签放在 symbol 的中心，就把 ⟨*angle*⟩ 设为 `center`.




```
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \node [resistor,info=center:$3\Omega$] {};
  \node [resistor,point up,info=center:$R_1$] at (2,0) {};
\end{tikzpicture}
```

`/tikz/info'=[`⟨*options*⟩`]`⟨*angle*⟩`:`⟨*text*⟩ (no default)

这个 key 与 `info` 类似。如果不给出 ⟨*angle*⟩, 那么这个 key 决定的位置与 `info` 恰好相对；而 `info` 的默认位置由选项 `/tikz/label position`→P. 121 确定（其默认值为 `above`）。如果 node 被旋转，那么 `info` 确定的位置也会随之旋转。

4Ω R_2 3Ω R_1

```
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \draw (0,0) to[resistor={info={$3\Omega$},info'={$R_1$}}] (3,0)
              to[resistor={info={$4\Omega$},info'={$R_2$}}] (3,2);
\end{tikzpicture}
```

`/tikz/info sloped=[`⟨*options*⟩`]`⟨*angle*⟩`:`⟨*text*⟩ (no default)

这个 key 与 info 类似，不过这个 key 自动给标签带上选项 transform shape, 使得标签随着 main node 的旋转而倾转。

```
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \draw (0,0) to[resistor={info sloped={$3\Omega$}}] (3,0)
              to[resistor={info sloped={$4\Omega$}}] (3,2);
\end{tikzpicture}
```

/tikz/info' sloped=[⟨options⟩]⟨angle⟩:⟨text⟩ (no default)

这是 info' 与 info sloped 的结合。

```
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \draw (0,0) to[resistor={info' sloped={$3\Omega$}}] (3,0)
              to[resistor={info' sloped={$4\Omega$}}] (3,2);
\end{tikzpicture}
```

/tikz/circuit declare unit={⟨name⟩}{⟨unit⟩} (no default)

这个 key 能够声明一组用于创建物理量的 keys, 即在 node 的标签中生成物理量的 keys. 例如，假设设置

circuit declare unit={ohm}{\Omega}

那么选项 ohm=3 等效于选项 info=3Ω; 假设设置

circuit declare unit={siemens}{\S}

那么选项 siemens'=3 等效于选项 info'=$5\mathrm{S}$. 可见这种创建物理量的 key 实际是 info 的简化版。

具体说，本选项导致 4 个 keys 被定义，即

/tikz/⟨name⟩, /tikz/⟨name⟩', /tikz/⟨name⟩ sloped, /tikz/⟨name⟩' sloped

这 4 个 keys 的参数都是 [⟨options⟩]⟨angle⟩:⟨text⟩ 这种形式。实际上，这 4 个 keys 的定义分别与 info, info', info sloped, info' sloped 是非常类似的，在文件《tikzlibrarycircuits.code》中有如下定义：

```
\tikzset{
  info/.code={\pgfutil@ifnextchar[\tikz@lib@circ@lab@plain{
  ↪  \tikz@lib@circ@lab@plain[]}#1\pgf@stop},%}
  info'/.code={\pgfutil@ifnextchar[\tikz@lib@circ@labp@plain{
  ↪  \tikz@lib@circ@labp@plain[]}#1\pgf@stop},%}
  info sloped/.code={\pgfutil@ifnextchar[\tikz@lib@circ@lab@sloped@plain{
  ↪  \tikz@lib@circ@lab@sloped@plain[]}#1\pgf@stop},%}
  info' sloped/.code={\pgfutil@ifnextchar[\tikz@lib@circ@lab@slopedp@plain{
  ↪  \tikz@lib@circ@lab@slopedp@plain[]}#1\pgf@stop},%}
  circuit declare unit/.style 2 args={
    %
    % Defines four styles that can be used to add labels to a node.
    %
```

```
    #1/.code={\pgfutil@ifnextchar[\tikz@lib@circ@lab{\tikz@lib@circ@lab[]}##1
    ↪  \pgf@stop{#2}{#1}},%}
    #1 sloped/.code={\pgfutil@ifnextchar[\tikz@lib@circ@lab@sloped{
    ↪  \tikz@lib@circ@lab@sloped[]}##1\pgf@stop{#2}{#1}},%}
    #1'/.code={\pgfutil@ifnextchar[\tikz@lib@circ@labp{\tikz@lib@circ@labp[]}##1
    ↪  \pgf@stop{#2}{#1}},%}
    #1' sloped/.code={\pgfutil@ifnextchar[\tikz@lib@circ@lab@slopedp{
    ↪  \tikz@lib@circ@lab@slopedp[]}##1\pgf@stop{#2}{#1}}%}
  }
}
```

可见 ⟨*name*⟩ 与 ⟨*unit*⟩ 都用在了这 4 个 keys 的定义中。

49.2.5 创建、使用 annotation

Annotations 很类似于 info labels, 二者的主要区别是，annotations 导致某些图形被画出，而不是添加文字标签。

用下面的 key 声明一个 annotation:

`/tikz/circuit declare annotation={⟨name⟩}{⟨distance⟩}{⟨path⟩}` (no default)

整个 key 声明一个名称为 ⟨*name*⟩ 的 annotation. 声明 ⟨*name*⟩ 后，就可以在 symbol 的选项中使用 ⟨*name*⟩, 这会导致把 ⟨*path*⟩ 添加到 symbol graphic 中（附近）。具体说，本选项产生以下效果:

本选项会定义两个 keys: ⟨*name*⟩ 和 ⟨*name*'⟩, 这两个 keys 所决定的 annotation 位置是相对的；这两个 keys 自己也可以带有参数（如 `info` 选项）。每当使用 ⟨*name*⟩ 时，都会开启一个 local scope, 在整个 scope 中执行以下动作:

1. 样式 `every` ⟨*name*⟩ 被执行。
2. 执行下面的样式:

   `/tikz/annotation arrow` (style, no value)

   这个样式应当把 `>` 设置为某种箭头类型。

   然后执行 `arrows=->`.
3. 坐标系统会被平移，使得坐标系统的原点位于 symbol 的 north anchor（对于 ⟨*name*'⟩ 则把坐标系统的原点平移到 symbol 的 south anchor）。
4. `label distance` 会被局部地设为 ⟨*distance*⟩.
5. ⟨*name*⟩ 自己的参数选项被执行。
6. ⟨*path*⟩ 被执行。

假设要制作一个类似下图的 annotation:

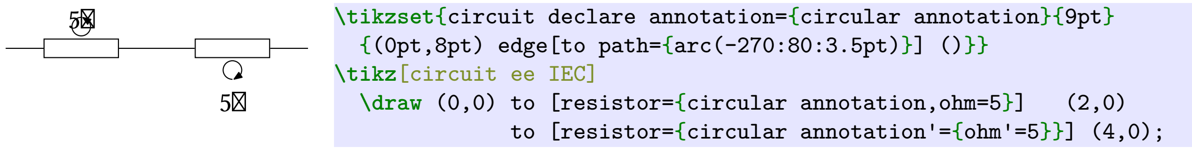
```
\tikz\draw [->](0pt,8pt) arc (-270:80:3.5pt);
```

如下设置:

5⊠

```
\tikzset{circuit declare annotation={circular annotation}{9pt}
  {(0pt,8pt) arc (-270:80:3.5pt)}}
% annotation 的名称是 circular annotation
\tikz[circuit ee IEC]
  \draw (0,0) to [resistor={circular annotation={ohm=5}}] (3,0);
```

注意上图中并没有出现 annotation, 但 annotation 确实已经被创建，只是没有被画出。这是因为代码 (0pt,8pt) arc (-270:80:3.5pt) 中没有画出线条的命令。作如下修改:




```
\tikzset{circuit declare annotation={circular annotation}{9pt}
  {(0pt,8pt) edge[to path={arc(-270:80:3.5pt)}] ()}}
\tikz[circuit ee IEC]
  \draw (0,0) to [resistor={circular annotation,ohm=5}]   (2,0)
            to [resistor={circular annotation'={ohm'=5}}] (4,0);
```

49.2.6 调整 Symbols 的外观

一个 symbol 呈现为一个 symbol graphic, 调整这个 graphic 的外观的办法有多种:

- 使用不同的程序库，并且更换相应的 `circuit ...` 选项，例如，选项 `circuit ee IEC` 与 `circuit ee US` 都定义了与门符号 `and gate`, 但符号图形的外观不一样。
- 使用选项 `circuit size unit` 修改符号的单位尺寸。
- 使用样式 `every circuit symbol` 或 `every` ⟨*symbol name*⟩ 来设置符号图形的外观。

```
\begin{tikzpicture}[circuit ee IEC,
    every circuit symbol/.style={ultra thick},
    every resistor/.style={red}]
  \draw (0,0) to [inductor] ++(right:3) to [resistor] ++(up:2);
\end{tikzpicture}
```

- 使用 `set` ⟨*symbol name*⟩ `graphic` 重设符号的 symbol graphic.
- 使用下文介绍的样式。

`/tikz/circuit symbol open` (style, initially `draw`)

这个样式针对的是这种符号图形：由线条构成，并且线条围出一个封闭区域。

```
\tikz [circuit ee IEC,
   circuit symbol open/.style={thick,draw,fill=yellow}]
  \draw (0,0) to [inductor] ++(right:3) to [resistor] ++(up:2);
```

`/tikz/circuit symbol filled` (style, initially `draw,fill=black`)

这个样式使得符号图形的内部区域被填充颜色，例如 IEC 规定的“变体线圈”（inductor）是具有填充色的矩形块。

```
\tikz [circuit ee IEC, circuit symbol filled]
  \draw (0,0) to [inductor] ++(right:3) to [resistor] ++(up:2);
```

/tikz/circuit symbol lines (style, initially draw)

这个样式针对的是这种符号图形：由线条构成，并且线条没有围出一个封闭区域。

```
\tikz [circuit ee IEC,
    circuit symbol lines/.style={thick,draw=red}]
  \draw (0,0) to [capacitor] ++(right:3) to [resistor] ++(up:2);
```

/tikz/circuit symbol wires (style, initially draw)

这个样式针对的是“由导线构成的符号图形”，例如电路上的开关符号 (make contact)。对比下面两个图形中 circuit symbol wires 与 circuit symbol lines 的区别：

```
\tikz [circuit ee IEC,circuit symbol lines/.style={draw,very thick}]
  \draw (0,0) to [capacitor={near start},
                  make contact={near end}] (3,0);
```

```
\tikz [circuit ee IEC,circuit symbol wires/.style={draw,very thick}]
  \draw (0,0) to [capacitor={near start},
                  make contact={near end}] (3,0);
```

Overview

选项 set ⟨*symbol name*⟩ graphic 的工作机制如下。

在文件《tikzlibrarycircuits.code.tex》中定义了样式 /tikz/circuit declare symbol：

```
\tikzset{
  circuit declare symbol/.style args={#1}{
% 省略若干
    #1/.style={circuit handle symbol={
        inner sep=0.5pt,
        every circuit symbol,
        #1/graphic,
        every #1/.try,
        ##1}
    },
    #1/graphic/.style={},
    set #1 graphic/.style={#1/graphic/.style={##1}}
  },
% 省略若干
}%
```

可见 /tikz/circuit declare symbol 的定义中包含了对 ⟨*symbol name*⟩（即变量 #1）的定义；包含了对 set ⟨*symbol name*⟩ graphic 的定义，而这个定义中又包含了对 /tikz/⟨*symbol name*⟩/graphic 的定义。

在文件《tikzlibrarycircuits.logic.code.tex》中声明了与门符号 and gate：

```
\tikzset{
  circuit declare symbol = and gate,
% 省略若干
```

```
}%
```

这个声明初步定义了⟨*symbol name*⟩, `set and gate graphic` 和 `and gate/graphic` 这 3 个选项。当用户写出选项 `set and gate graphic={`⟨*options in style*⟩`}` 并执行后,就会得到定义 `and gate/graphic/.style = {`⟨*options in style*⟩`}`. 当使用符号选项 `and gate` 时, `and gate/graphic` 中保存的 ⟨*options in style*⟩ 就会被执行。

在文件《tikzlibrarycircuits.logic.IEC.code.tex》中有如下代码:

```
\tikzset{
  circuit logic IEC/.style=
  {
    circuit logic,
    set and gate graphic     = and gate IEC graphic,
% 省略若干
  },
}%
% 省略若干
\tikzset{
  circuit logic IEC make graphic/.style=
  {
    #1 graphic/.style={
      circuit symbol open,
      circuit symbol size=width 2.5 height 4,
      shape=#1,
      inner sep=.5ex
    }
  }
}%


\tikzset{
  circuit logic IEC make graphic=and gate IEC,
% 省略若干
}%
```

上面代码定义选项 `and gate IEC graphic`, 而 `set and gate graphic = and gate IEC graphic` 把与门的符号图形定义为 `and gate IEC graphic`, 这个样式中使用形状 `shape=and gate IEC`, 形状 `and gate IEC` 在文件《pgflibraryshapes.gates.logic.IEC.code.tex》中有定义:

```
\pgfdeclareshape{and gate IEC}{%
% 省略若干
}%
```

以上是对 `set and gate graphic` 的预定义。

49.2.7 符号图形的变体

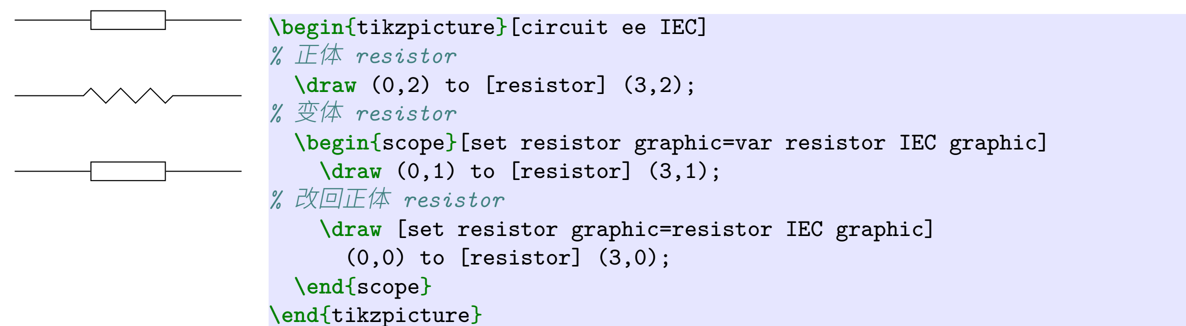
有的程序库中不仅定义了通常的 symbol graphic, 还定义了变体的 symbol graphic, 例如在文件《tikzlibrarycircuits.ee.IEC.code.tex》中有如下代码:

```
\tikzset{
  circuit ee IEC/.style=
  {
    circuit ee,
    set resistor graphic              = resistor IEC graphic,
% 省略若干
}%


%
% Resistors
%

\tikzset{
  resistor IEC graphic/.style={
    circuit symbol open,
    circuit symbol size=width 4 height 1,
    shape=rectangle ee,
    transform shape,
  },
  var resistor IEC graphic/.style={
    circuit symbol lines,
    circuit symbol size=width 4.8 height 0.8,
    shape=var resistor IEC,
    transform shape,
    outer sep=0pt,
    cap=round,
  },
}%
```

上面代码定义了“正体”符号图形 `resistor IEC graphic` 及其“变体”符号图形 `var resistor IEC graphic`；默认符号选项 `resistor` 使用“正体”。在“正体”符号图形名称的前面加上“`var`”就是“变体”符号图形名称。下面代码定义“正体”符号图形 `inductor IEC graphic` 及其“变体”符号图形 `var inductor IEC graphic`：

```
\tikzset{
  inductor IEC graphic/.style={
    circuit symbol lines,
    circuit symbol size=width 4 height .5,
    transform shape,
    shape=inductor IEC,
    outer sep=0pt,
    cap=round,
  },
  var inductor IEC graphic/.style={
    circuit symbol filled,
    circuit symbol size=width 4 height 1,
    transform shape,
    shape=rectangle ee,
  },
}%
```

在 inductor IEC graphic 的定义中使用了选项 circuit symbol lines, 在 var inductor IEC graphic 的定义中使用了选项 circuit symbol filled.



```
\begin{tikzpicture}[circuit ee IEC]
% 正体 resistor
  \draw (0,2) to [resistor] (3,2);
% 变体 resistor
  \begin{scope}[set resistor graphic=var resistor IEC graphic]
    \draw (0,1) to [resistor] (3,1);
% 改回正体 resistor
    \draw [set resistor graphic=resistor IEC graphic]
      (0,0) to [resistor] (3,0);
  \end{scope}
\end{tikzpicture}
```

49.3 逻辑电路

49.3.1 Overview

下面是与逻辑电路相关的程序库,

- circuits.logic, 这个库为以下 3 个库提供一般的选项，但不提供符号图形；这个库会被下面 3 个库调用。
- circuits.logic.IEC, 这个库提供 International Electrotechnical Commission 规定的逻辑门符号, 即 IEC 风格的逻辑门符号，但不包括符号图形的形状路径。
- circuits.logic.US, 这个库提供“美式风格”的逻辑门符号，但不包括符号图形的形状路径。
- circuits.logic.CDH, 这个库提供 A. Croft, R. Davidson 与 M. Hargreaves 在他们的著作《*Engineering Mathematics*》中使用的的逻辑门符号，即 CDH 风格的逻辑门符号；本程序库不包括符号图形的形状路径。
- shapes.gates.logic, 这个库被下面两个库调用，用来定义各种逻辑门符号图形的形状路径。
- shapes.gates.logic.US, 这个库提供“美式风格”的逻辑门符号的形状路径，以及 CDH 风格的与门符号、与非门符号的形状路径。
- shapes.gates.logic.IEC, 这个库提供 IEC 风格的逻辑门符号的形状路径。

在实际使用时，只需载入 circuits.logic.IEC, circuits.logic.US, circuits.logic.CDH 这 3 个库。

TikZ Library circuits.logic

```
\usetikzlibrary{circuits.logic} % LaTeX and plain TeX
\usetikzlibrary[circuits.logic] % ConTeXt
```

这个库声明逻辑门符号，不提供符号图形。这个库提供下面的选项:

/tikz/circuit logic (no value)

这个 key 调用 circuit, 定义 inputs, every circuit logic.

`/tikz/inputs`=⟨*inputs*⟩ (no default)

这个 key 只存在于 `circuit logic` 的有效范围内，本选项等效于 `logic gate inputs`，这个 key 用于文件《pgflibraryshapes.gates.logic.code.tex》中，参考后文关于 `shapes.gates.logic` 库的解释。

⟨*inputs*⟩ 是由字母 i 和 n 组成的一串字母。如果一个逻辑门有 m 个输入点，那么 ⟨*inputs*⟩ 中就应该有 m 个字母，每个字母代表一个输入点；其中的字母 i 表示 “inverted” 输入点（用一个小圆圈标识），字母 n 表示 “normal” 输入点。这 m 个输入点被定义为逻辑门符号形状（shape）上的 anchor 位置，这 m 个位置从上到下依次命名为 `input 1`, `input 2`, ... 如果只有一个输入点，那么这个输入点（anchor 位置）就被命名为 `input`. 参考 `/pgf/logic gate inputs`→P. 335.

每个逻辑门符号都有一个输出点（anchor 位置），被命名为 `output`.

```
\begin{tikzpicture}[circuit logic IEC]
  \node[and gate,inputs={inini}] (A) {};
  \foreach \a in {1,...,5}
    \draw (A.input \a) -| -1,0) -- (A.input \a);
  \draw (A.output) -- ++(right:5mm);
\end{tikzpicture}
```

`/tikz/every circuit logic` (style, no value)

这个样式用于调整逻辑门符号的外观。

这个库声明的逻辑门符号中带字母 “n” 的版本代表 “非门”。

这个库的内容很简短：

```
% Copyright 2008 by Till Tantau and Mark Wibrow
%
% This file may be distributed and/or modified
%
% 1. under the LaTeX Project Public License and/or
% 2. under the GNU Public License.
%
% See the file doc/generic/pgf/licenses/LICENSE for more details.

\usetikzlibrary{circuits}%




%
% Provides a shortcut to the "logic gates inputs" key.
%

\tikzset{
  circuit logic/.style={
    circuit,
    inputs/.style={logic gate inputs={##1}},
```

```
    logic gate inverted radius=.25\tikzcircuitssizeunit,
    every circuit logic/.try,
  }
}%



%
% The default symbols (you need to load a sublib to install the actual rendering).
%

\tikzset{
  circuit declare symbol = and gate,
  circuit declare symbol = nand gate,
  circuit declare symbol = or gate,
  circuit declare symbol = nor gate,
  circuit declare symbol = xor gate,
  circuit declare symbol = xnor gate,
  circuit declare symbol = not gate,
  circuit declare symbol = buffer gate
}%


\endinput
```

可见这个库的主要内容是定义了一些选项、声明逻辑门符号。

TikZ Library `circuits.logic.IEC`

```
\usepgflibrary{circuits.logic.IEC} % LaTeX and plain TeX and pure pgf
\usepgflibrary[circuits.logic.IEC] % ConTeXt and pure pgf
\usetikzlibrary{circuits.logic.IEC} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[circuits.logic.IEC] % ConTeXt when using TikZ
```

这个库提供 International Electrotechnical Commission 规定的逻辑门符号。

`/tikz/circuit logic IEC` (no value)

本程序库定义此 key. 此 key 调用 `circuit logic`, 并把 IEC 风格的逻辑门符号引入当前环境（分组）中。

```
\begin{tikzpicture}[circuit logic IEC, every circuit
↪  symbol/.style={
    logic gate IEC symbol color=black,
    ↪  fill=blue!20,draw=blue,very thick}]
  \matrix[column sep=7mm]
  {
    \node (i0) {0}; &                                & \\
                    & \node [and gate] (a1) {};  & \\
    \node (i1) {0}; &                                & \node [or
    ↪  gate] (o) {};\\
                    & \node [nand gate] (a2) {}; & \\
    \node (i2) {1}; &                                & \\
  };
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
  \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
  \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
  \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

这个库先调用其它的库:

```
\usetikzlibrary{circuits.logic}%
\usepgflibrary{shapes.gates.logic.IEC}%
```

然后定义 `circuit logic IEC`, 为 `set` ⟨*symbol name*⟩ `graphic` 赋值:

```
\tikzset{
  circuit logic IEC/.style=
  {
    circuit logic,
    set and gate graphic    = and gate IEC graphic,
    set nand gate graphic   = nand gate IEC graphic,
    set or gate graphic     = or gate IEC graphic,
    set nor gate graphic    = nor gate IEC graphic,
    set xor gate graphic    = xor gate IEC graphic,
    set xnor gate graphic   = xnor gate IEC graphic,
    set not gate graphic    = not gate IEC graphic,
    set buffer gate graphic = buffer gate IEC graphic,
    %
    and gate IEC symbol    = {$\scriptstyle\char`\&$},
    nand gate IEC symbol   = {$\scriptstyle\char`\&$},
    or gate IEC symbol     = {$\scriptstyle\geq1$},
    nor gate IEC symbol    = {$\scriptstyle\geq1$},
    xor gate IEC symbol    = {$\scriptstyle=1$},
    xnor gate IEC symbol   = {$\scriptstyle=1$},
    not gate IEC symbol    = {$\scriptstyle1$},
    buffer gate IEC symbol = {$\scriptstyle1$},
  },
}%
```

然后定义 `circuit logic IEC make graphic`, 即定义各种逻辑门符号:

```
\tikzset{
  circuit logic IEC make graphic/.style=
  {
    #1 graphic/.style={
      circuit symbol open,
      circuit symbol size=width 2.5 height 4,
      shape=#1,
      inner sep=.5ex
    }
  }
}%


\tikzset{
  circuit logic IEC make graphic=and gate IEC,
  circuit logic IEC make graphic=nand gate IEC,
  circuit logic IEC make graphic=or gate IEC,
  circuit logic IEC make graphic=nor gate IEC,
  circuit logic IEC make graphic=xor gate IEC,
  circuit logic IEC make graphic=xnor gate IEC,
  circuit logic IEC make graphic=not gate IEC,
  circuit logic IEC make graphic=buffer gate IEC,
  circuit logic IEC make graphic=and gate CDH,
  circuit logic IEC make graphic=nand gate CDH,
}%
```

其中的形状路径选项 `shape=#1` 参考 `shapes.gates.logic.IEC` 库的文件《pgflibraryshapes.gates.logic.IEC.code.tex》.

TikZ Library `circuits.logic.US`

```
\usepgflibrary{circuits.logic.US} % LaTeX and plain TeX and pure pgf
\usepgflibrary[circuits.logic.US] % ConTeXt and pure pgf
\usetikzlibrary{circuits.logic.US} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[circuits.logic.US] % ConTeXt when using TikZ
```

这个库提供美式风格的逻辑门符号。这个库的内容与 `circuits.logic.IEC` 类似。

`/tikz/circuit logic US` (no value)

本程序库定义此 key. 此 key 调用 `circuit logic`, 并把美式风格的逻辑门符号引入当前环境（分组）中。

0
0
1

```
\begin{tikzpicture}[circuit logic CDH, tiny circuit symbols,
    every circuit symbol/.style={fill=gray!70,draw}]
  \matrix[column sep=7mm]
  {
    \node (i0) {0}; &                               & \\
                    & \node [and gate] (a1) {};  & \\
    \node (i1) {0}; &      & \node [or gate] (o) {};\\
                    & \node [nand gate] (a2) {}; & \\
    \node (i2) {1}; &                               & \\
  };
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
  \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
  \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
  \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

TikZ Library circuits.logic.CDH

```
\usepgflibrary{circuits.logic.CDH} % LaTeX and plain TeX and pure pgf
\usepgflibrary[circuits.logic.CDH] % ConTeXt and pure pgf
\usetikzlibrary{circuits.logic.CDH} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[circuits.logic.CDH] % ConTeXt when using TikZ
```

这个库提供 A. Croft, R. Davidson 与 M. Hargreaves 在他们的著作 *Engineering Mathematics* 中使用的的逻辑门符号，除了与门符号和与非门符号外，其它符号都与美式风格的符号相同。

`/tikz/circuit logic CDH` (no value)

本程序库定义此 key. 此 key 调用 `circuit logic`, 并把 CDH 风格的逻辑门符号引入当前环境（分组）中。

这个库的内容只是修改了一下 `circuits.logic.US` 的与门符号和与非门符号:

```
\usetikzlibrary{circuits.logic.US}%

\tikzset{
  circuit logic CDH/.style =
  {
    circuit logic US,
    set and gate graphic     = and gate CDH graphic,
    set nand gate graphic    = nand gate CDH graphic,
  }
}%

\tikzset{
  circuit logic US make graphic=and gate CDH,
  circuit logic US make graphic=nand gate CDH,
}%
```

```
\endinput
```

`and gate US` 与 `and gate CDH` 的区别在于它们的 shape 不同,参考文件《pgflibraryshapes.gates.logic.US.code.tex》中的定义命令 `\pgfdeclareshape{and gate CDH}` 与 `\pgfdeclareshape{and gate CDH}`.

49.3.2 逻辑门符号

`/tikz/and gate` (no value)

这个 key 应该用作 node 的选项，将 node 做成一个“与门符号”。符号图形的外观依照风格选项，例如：

```
\tikz [circuit logic IEC] \node [and gate] {$A$};
```

```
\tikz [circuit logic US]{
  \node [and gate,point down] {$A$};
  \node [and gate,point down,info=center:$A$] at (1,0) {};}
```

`/tikz/nand gate` (no value)

`/tikz/or gate` (no value)

`/tikz/nor gate` (no value)

`/tikz/xor gate` (no value)

`/tikz/xnor gate` (no value)

`/tikz/not gate` (no value)

`/tikz/buffer gate` (no value)

49.3.3 逻辑门符号的形状

程序库 `shapes.gates.logic` 提供一些“工具”,然后程序库 `shapes.gates.logic.US` 和 `shapes.gates.logic.IEC` 利用这些“工具”分别定义 US 风格的逻辑门符号形状和 IEC 风格的逻辑门符号形状。

TikZ Library `shapes.gates.logic`

```
\usepgflibrary{shapes.gates.logic} % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes.gates.logic] % ConTeXt and pure pgf
\usetikzlibrary{shapes.gates.logic} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.logic] % ConTeXt when using TikZ
```

这个库中定义了下面 3 个选项。

`/pgf/logic gate inputs=`⟨*input*⟩ (no default, initially `{normal,normal}`)

这个选项与 `/tikz/input` 等效。本选项针对的是逻辑门符号的输入点。⟨*input*⟩ 是个列表，其中可以使用 `inverted`，`normal`，`non-inverted` 这 3 个关键词；`inverted` 代表 inverted input（用一个小圆圈标识）；`normal`，`non-inverted` 代表正常的 input（没有圆圈标识），例如

```
logic gate inputs={inverted, normal, inverted, non-inverted}
```

这些输入点被定义为符号图形形状（shape）的 anchor 位置；这些 anchor 位置从上到下依次命名为 `input 1`，`input 2`，... 如果符号只有一个输入点（anchor 位置），那么这个输入点（anchor 位置）被命名为 `input 1`.

```
\begin{tikzpicture}[minimum height=0.75cm]
  \node[and gate IEC, draw,
    logic gate inputs={inverted, normal, inverted}] (A) {};
  \foreach \a in {1,...,3}
      \draw (A.input \a) -| -1,0) -- (A.input \a);
  \draw (A.output) -- ([xshift=0.5cm]A.output);
\end{tikzpicture}
```

⟨*input*⟩ 中的关键词 `inverted` 可以简化为 `i`, 关键词 `normal`，`non-inverted` 可以简化为 `n`, 同时要去掉 ⟨*input*⟩ 中的逗号。

```
\begin{tikzpicture}[minimum height=0.75cm]
  \node[or gate US, draw,logic gate inputs=inini] (A) {};
  \foreach \a in {1,...,5}
    \draw (A.input \a) -| -1,0) -- (A.input \a);
  \draw (A.output) -- ([xshift=0.5cm]A.output);
\end{tikzpicture}
```

符号图形的高度（height）决定于表达式 $(n+1)\times\max\{2r,s\}$, 其中的 n 是输入点的个数；r 是 inverted input 的标识圆圈的半径；s 是两个相邻输入点的间距。

inverted input, inverted output 的标识圆圈的半径决定于下面的选项。

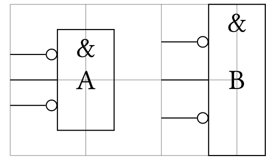
`/pgf/logic gate inverted radius`=⟨*lenght*⟩ (no default, initially 2pt)

本选项确定逻辑门符号的 inverted input 的标识圆圈的半径，同时也确定逻辑门符号的 inverted output 的标识圆圈的半径。

```
\begin{tikzpicture}[minimum height=0.75cm]
  \tikzset{every node/.style={shape=nand gate CDH, draw,
  ↪  logic gate inputs=ii}}
  \node[logic gate inverted radius=2pt] {A};
  \node[logic gate inverted radius=4pt] at (0,-1) {B};
\end{tikzpicture}
```

`/pgf/logic gate input sep`=⟨*lenght*⟩ (no default, initially `.125pt`)

本选项确定逻辑门符号的两个相邻输入点的间距。




```
\begin{tikzpicture}[minimum size=0.75cm]
  \draw [help lines] grid (3,2);
  \tikzset{every node/.style={shape=and gate IEC, draw,
    logic gate inputs=ini}}
  \node[logic gate input sep=0.33333cm] at (1,1)(A) {A};
  \node[logic gate input sep=0.5cm] at (3,1) (B) {B};
  \foreach \a in {1,...,3}
    \draw (A.input \a -| 0,0) -- (A.input \a)
          (B.input \a -| 2,0) -- (B.input \a);
\end{tikzpicture}
```

当逻辑门符号形状的高度变化时，其宽度也会自动变化以保持其宽高比。

49.3.4 US 风格的逻辑门符号形状

TikZ Library shapes.gates.logic.US

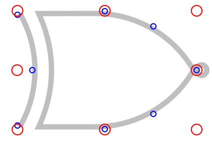
```
\usepgflibrary{shapes.gates.logic.US} % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes.gates.logic.US] % ConTeXt and pure pgf
\usetikzlibrary{shapes.gates.logic.US} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.logic.US] % ConTeXt when using TikZ
```

这个库提供美式风格的逻辑门符号形状 (如 `and gate US`, `or gate US`), 以及 CDH 风格的 `and gate CDH` 与 `nand gate CDH`.

逻辑门符号形状也有自己的“罗盘位置”，即 `north`, `north east` 等位置。下面的选项决定罗盘位置的计算方式:

`/pgf/logic gate anchors use bounding box=`⟨*boolean*⟩ (no default, initially `false`)

如果本选项的值是 `true`, 那么逻辑门符号形状的罗盘位置就是，逻辑门符号形状自己的边界盒子的罗盘位置。本选项只对罗盘位置有效，对“输入点”(input)、“输出点”(output) 无效，但边界盒子把 `outer sep` 包括在内。



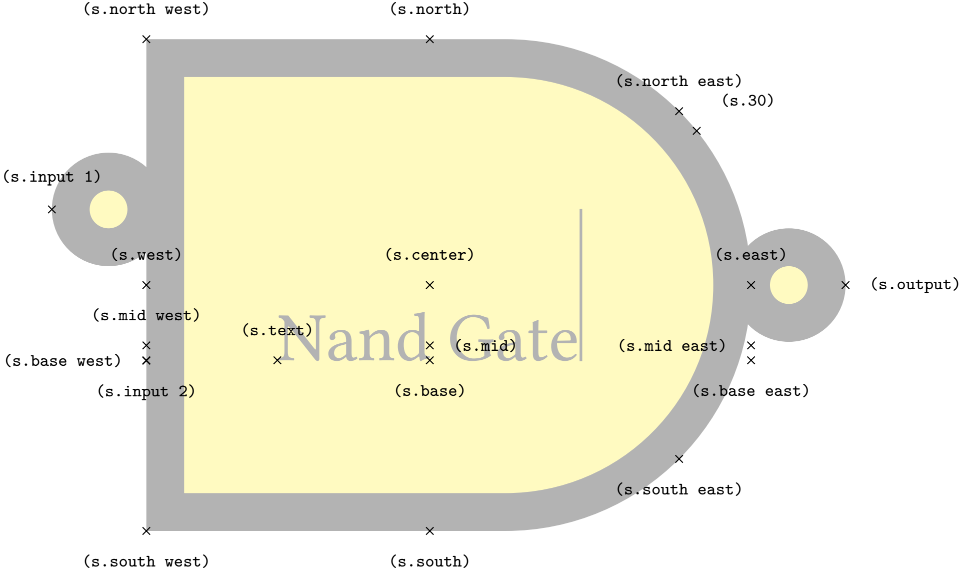
```
\begin{tikzpicture}[minimum height=1.5cm]
  \node[xnor gate US, draw, gray!50,line width=2pt] (A) {};
  \foreach \x/\y/\z in {false/blue/1pt, true/red/2pt}
    \foreach \a in {north, south, east, west, north east,
      south east, north west, south west}
      \draw[logic gate anchors use bounding box=\x, color=\y]
        (A.\a) circle(\z);
\end{tikzpicture}
```

这个库定义了数种逻辑门符号形状，每一形状都有自己所允许的输入点个数:

- `and gate US`, ⩾ 2 个 input
- `and gate CDH`, ⩾ 2 个 input
- `nand gate US`, ⩾ 2 个 input

- nand gate CDH, $\geqslant 2$ 个 input
- or gate US, $\geqslant 2$ 个 input
- nor gate US, $\geqslant 2$ 个 input
- xor gate US, 2 个 input
- xnor gate US, 2 个 input
- not gate US, 1 个 input
- buffer gate US, 1 个 input

下面以 nand gate US 为例，看一下逻辑门符号形状的各种 anchor 位置。




```
\tikzset{
  shape example/.style= {color = black!30, draw,
    fill = yellow!30,
    line width = .5cm,
    inner xsep = 2.5cm,
    inner ysep = 0.5cm}
}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=nand gate US,shape example, inner sep=0cm,
    logic gate inputs={in},
    logic gate inverted radius=.5cm] {Nand Gate\vrule width1pt height2cm};
  \foreach \anchor/\placement in
    {center/above,     text/above,       30/above right,
     mid/right,        mid east/left,    mid west/above,
     base/below,       base east/below,  base west/left,
     north/above,      south/below,      east/above,       west/above,
     north east/above, south east/below, south west/below, north west/above,
     output/right,     input 1/above,    input 2/below}
      \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
        node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.3.5 IEC 风格的逻辑门符号形状

TikZ Library shapes.gates.logic.IEC

```
\usepgflibrary{shapes.gates.logic.IEC} % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes.gates.logic.IEC] % ConTeXt and pure pgf
\usetikzlibrary{shapes.gates.logic.IEC}
↪ % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.logic.IEC] % ConTeXt when using TikZ
```

这个库提供 IEC 风格的逻辑门符号形状，例如 `and gate IEC`, 它们是 International Electrotechnical Commission 推荐的形状。

在默认下，在 `and`，`nand` gates 内部都有符号 &; 在 `or`，`nor` gates 内部都有符号 ≥ 1; 在 `xor`，`xnor` gates 内部都有符号 $= 1$; 在 `not`，`buffer` gates 内部都有符号 1. 这些符号是自动添加的，但这些符号是使用"foreground" path 创建的，并不属于 node 的文字内容。各个 gate 会自动调整自己的尺寸，以确保这些符号处于 node 边界的内部。可以用下面的选项修改这些内部符号：

`/pgf/and gate IEC symbol=`⟨*text*⟩ (no default, initially `\char`\&`)

设置 `and gate` 内部的符号。注意，如果 node 有填充色，那么这个内部符号的颜色也是此填充色，使得这个内部符号不可见；为了避免这问题，可以把 ⟨*text*⟩ 设为 `\color{black}\char`\&`, 或者，把 `logic gate IEC symbol color`(见后文) 的值设为其它颜色。⟨*text*⟩ 的颜色设置所具有的优先地位要高于选项 `logic gate IEC symbol color` 的设置。

在 Tikz 中，当使用选项 `use IEC style logic gates` 时（在这个选项的有效范围内），本选项可以简写为 `and gate symbol`.

`/pgf/nand gate IEC symbol=`⟨*text*⟩ (no default, initially `\char`\&`)

设置 `and gate` 内部的符号。在 Tikz 中，当使用选项 `use IEC style logic gates` 时（在这个选项的有效范围内），本选项可以简写为 `nand gate symbol`.

`/pgf/or gate IEC symbol=`⟨*text*⟩ (no default, initially ≥ 1)

设置 `or gate` 内部的符号。在 Tikz 中，当使用选项 `use IEC style logic gates` 时（在这个选项的有效范围内），本选项可以简写为 `or gate symbol`.

`/pgf/nor gate IEC symbol=`⟨*text*⟩ (no default, initially ≥ 1)

设置 `nor gate` 内部的符号。在 Tikz 中，当使用选项 `use IEC style logic gates` 时（在这个选项的有效范围内），本选项可以简写为 `nor gate symbol`.

`/pgf/xor gate IEC symbol=`⟨*text*⟩ (no default, initially $= 1$)

设置 `xor gate` 内部的符号。在 Tikz 中，当使用选项 `use IEC style logic gates` 时（在这个选项的有效范围内），本选项可以简写为 `xor gate symbol`.

`/pgf/xnor gate IEC symbol=`⟨*text*⟩ (no default, initially $= 1$)

设置 `xnor gate` 内部的符号。在 Tikz 中，当使用选项 `use IEC style logic gates` 时（在这个选项的有效范围内），本选项可以简写为 `xnor gate symbol`.

`/pgf/not gate IEC symbol`=⟨*text*⟩ (no default, initially 1)

设置 `not gate` 内部的符号。在 Tikz 中，当使用选项 `use IEC style logic gates` 时（在这个选项的有效范围内），本选项可以简写为 `not gate symbol`.

`/pgf/buffer gate IEC symbol`=⟨*text*⟩ (no default, initially 1)

设置 `buffer gate` 内部的符号。在 Tikz 中，当使用选项 `use IEC style logic gates` 时（在这个选项的有效范围内），本选项可以简写为 `buffer gate symbol`.

`/pgf/logic gate IEC symbol align`=⟨*align*⟩ (no default, initially `top`)

设置 logic gate symbol 内部的符号的对齐方式。⟨*align*⟩ 是由逗号分隔的列表，列表项出自单词 `top`，`bottom`，`left`，`right`. 符号与 node 的边界的距离由选项 `inner xsep` 和 `inner ysep` 确定。

≥ 1

≥ 1

```
\begin{tikzpicture}[minimum size=1cm, use IEC style logic gates]
  \tikzset{every node/.style={nor gate, draw}}
  \node (A) at (0,1.5) {};
  \node [logic gate symbol align={bottom, right}] (B) at (0,0) {};
  \foreach \g in {A, B}{
    \foreach \i in {1,2}
      \draw ([xshift=-0.5cm]\g.input \i) -- (\g.input \i);
    \draw (\g.output) -- ([xshift=0.5cm]\g.output);
  }
\end{tikzpicture}
```

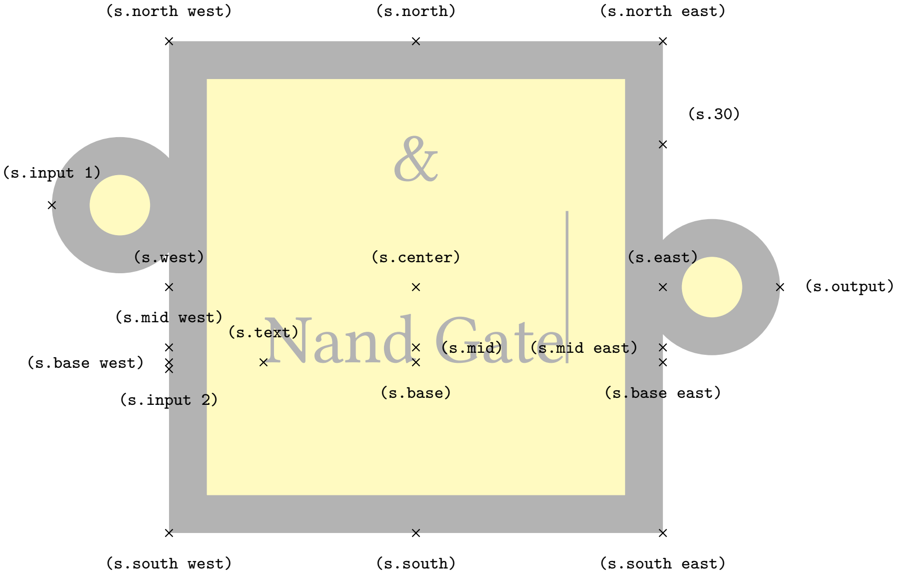
`/pgf/logic gate IEC symbol color`=⟨*color*⟩ (no default)

本选项为各个 logic gate 内部的符号设置颜色，本选项设置的颜色可以被以上选项的设置（⟨*text*⟩）修改。

本程序库定义以下形状，各自有输入点个数限制：

- `and gate IEC`, $\geqslant 2$ 个 input
- `nand gate IEC`, $\geqslant 2$ 个 input
- `or gate IEC`, $\geqslant 2$ 个 input
- `nor gate IEC`, $\geqslant 2$ 个 input
- `xor gate IEC`, 2 个 input
- `xnor gate IEC`, 2 个 input
- `not gate IEC`, 1 个 input
- `buffer gate IEC`, 1 个 input

下面展示 `nand gate IEC` 的各个 anchor 位置。




```
\tikzset{
  shape example/.style= {color = black!30, draw,
    fill = yellow!30,
    line width = .5cm,
    inner xsep = 2.5cm,
    inner ysep = 0.5cm}
}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=nand gate IEC ,shape example, inner xsep=1cm, inner ysep=1cm,
        minimum height=6cm, nand gate IEC symbol=\color{black!30}\char`\&,
        logic gate inputs={in},
        logic gate inverted radius=0.65cm]
    {Nand Gate\vrule width1pt height2cm};
  \foreach \anchor/\placement in
    {center/above,     text/above,       30/above right,
     mid/right,        mid east/left,    mid west/above,
     base/below,       base east/below,  base west/left,
     north/above,      south/below,      east/above,       west/above,
     north east/above, south east/below, south west/below, north west/above,
     output/right,     input 1/above,    input 2/below}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.4 电子工程电路

49.4.1 Overview

一个 electrical engineering circuit （简写为 ee）包含电阻、电容、电源，annotations 等。

程序库 circuitss.ee 声明各种符号、物理单位、annotations、方向箭头等，另外声明两个 symbol graphic（即 current direction ee graphic 和 current direction' ee graphic）.

程序库 circuits.ee.IEC 会调用库 circuitss.ee, 其主要内容是声明各种 symbol graphic.

TikZ Library circuits.ee

```
\usetikzlibrary{circuits.ee} % LaTeX and plain TeX
\usetikzlibrary[circuits.ee] % ConTeXt
```

这个库提供以下选项：

`/tikz/circuit ee` (no value)

此选项包含样式 `every circuit ee`, 调用 `circuit`, 而 `circuit` 包含样式 `every circuit`.

`/tikz/every circuit ee` (style, no value)

此样式会用于 ee circuit.

这个库的主要内容如下。

先调用两个库：

```
\usetikzlibrary{circuits}%
\usepgflibrary{shapes.gates.ee}%
```

然后定义 `circuit ee`：

```
\tikzset{
  circuit ee/.style={
    circuit,
    every circuit ee/.try
  }
}%
```

然后声明符号，顺带“提名”两个 symbol graphic, 即 `current direction ee graphic`，`current direction' ee graphic`：

```
\tikzset{
  circuit declare symbol = resistor,
  circuit declare symbol = inductor,
  circuit declare symbol = capacitor,
  circuit declare symbol = contact,
  circuit declare symbol = ground,
  circuit declare symbol = battery,
  circuit declare symbol = diode,
  circuit declare symbol = Zener diode,
  circuit declare symbol = Schottky diode,
  circuit declare symbol = tunnel diode,
  circuit declare symbol = backward diode,
  circuit declare symbol = breakdown diode,
  circuit declare symbol = bulb,
  circuit declare symbol = current source,
  circuit declare symbol = voltage source,
  circuit declare symbol = current direction,
  circuit declare symbol = current direction',
  circuit declare symbol = make contact,
  circuit declare symbol = break contact,
```

```
  %
  set current direction graphic  = current direction ee graphic,
  set current direction' graphic = current direction' ee graphic,
}%
```

然后声明物理单位:

```
\tikzset{
  circuit declare unit={ampere}{A},
  circuit declare unit={volt}{V},
  circuit declare unit={ohm}{\Omega},
  circuit declare unit={siemens}{S},
  circuit declare unit={henry}{H},
  circuit declare unit={farad}{F},
  circuit declare unit={coulomb}{C},
  circuit declare unit={voltampere}{VA},
  circuit declare unit={watt}{W},
  circuit declare unit={hertz}{Hz},
}%
```

然后定义方向箭头符号 `current direction arrow`:

```
\tikzset{
  % These styles should set the end-arrow.
  %
  % This arrow will generally be used to indicate current directions in a circuit:
  current direction arrow/.style = {
    /utils/exec={\pgfsetarrowoptions{direction ee}{1.3065*.5*\the
    ↪  \tikzcircuitssizeunit+1.3065*.3*\the\pgflinewidth}},
    >=direction ee,
    direction ee arrow = direction ee,
  }
}%
```

其中的 `direction ee` 是个箭头样子的形状 (shape), 见文件《pgflibraryshapes.gates.ee.code.tex》.
其中用到命令 `\pgfsetarrowoptions`, 此命令在文件《pgfcorearrows.code.tex》中定义为:

```
\def\pgfsetarrowoptions#1#2{\expandafter\def\csname pgf@arrow@compat@opt@#1
↪  \endcsname{#2}}
```

即定义 `\def\pgf@arrow@compat@opt@#1{#2}`.
选项 `direction ee arrow` 在文件《pgflibraryshapes.gates.ee.code.tex》中定义为:

```
\pgfkeys{
  /pgf/direction ee arrow/.initial=direction ee,
}%
```

选项 `/utils/exec=`⟨*code*⟩ 就是直接执行 ⟨*code*⟩, 其中的 `1.3065 * .5*\the\tikzcircuitssizeunit + 1.3065*.3 * \the\pgflinewidth` 将被用作符号形状——箭头的长度。

然后规定两个 symbol graphic, 即 `current direction`, `current direction'`, 这两个符号用来指示电流方向, 其形状是 `shape=direction ee`:

```
\tikzset{
  current direction ee graphic/.style = {
    shape=direction ee,
    circuit symbol filled,
    current direction arrow,
    minimum width  = .5*\the\tikzcircuitssizeunit+.3*\the\pgflinewidth,
    minimum height = .5*\the\tikzcircuitssizeunit+.3*\the\pgflinewidth,
    transform shape
  },
  current direction' ee graphic/.style = {
    current direction ee graphic,
    rotate=180
  }
}%
```

然后定义 5 个 annotations：

```
\tikzset{
  circuit declare annotation={direction info}{.5\tikzcircuitssizeunit}
  {
    (-1.25\tikzcircuitssizeunit,.3333\tikzcircuitssizeunit) edge[line to] (1.25
    ↪  \tikzcircuitssizeunit,.3333\tikzcircuitssizeunit)
  },
  circuit declare annotation={light emitting}{1.75\tikzcircuitssizeunit}
  {
    (-.2\tikzcircuitssizeunit,.65\tikzcircuitssizeunit) edge[line to] ++(45:1.25
    ↪  \tikzcircuitssizeunit)
    (.2\tikzcircuitssizeunit,.25\tikzcircuitssizeunit) edge[line to] ++(45:1.25
    ↪  \tikzcircuitssizeunit)
  },
  circuit declare annotation={light dependent}{1.75\tikzcircuitssizeunit}
  {
    [shift=(135:1.25\tikzcircuitssizeunit)]
    (.2\tikzcircuitssizeunit,.65\tikzcircuitssizeunit) edge[line to] ++(-45:1.25
    ↪  \tikzcircuitssizeunit)
    (-.2\tikzcircuitssizeunit,.25\tikzcircuitssizeunit) edge[line to] ++(-45:1.25
    ↪  \tikzcircuitssizeunit)
  },
  circuit declare annotation={adjustable}{1.5\tikzcircuitssizeunit}
  {
    [shift=(\tikzlastnode.center)]
    (-1.5\tikzcircuitssizeunit,-1.5\tikzcircuitssizeunit) edge[line to] (1.5
    ↪  \tikzcircuitssizeunit,1.5\tikzcircuitssizeunit)
  },
  circuit declare annotation={adjustable'}{1.5\tikzcircuitssizeunit}
  {
    [shift=(\tikzlastnode.center)]
    (-1.5\tikzcircuitssizeunit,1.5\tikzcircuitssizeunit) edge[line to] (1.5
    ↪  \tikzcircuitssizeunit,-1.5\tikzcircuitssizeunit)
  }
}%
```

下面再看程序库 `circuits.ee.IEC`.

TikZ Library `circuits.ee.IEC`

这个库提供选项:

`/tikz/circuit ee IEC` (no value)

这个样式调用 `circuit ee`, 并且把 IEC 风格的 ee 符号引入当前环境（分组）中。

这个库先调用其他的库:

```
\usetikzlibrary{arrows}%
\usetikzlibrary{circuits.ee}%

\usepgflibrary{shapes.gates.ee.IEC}%
```

然后定义 `circuit ee IEC`, 为 `set` ⟨*symbol name*⟩ `graphic` 赋值:

```
\tikzset{
  circuit ee IEC/.style=
  {
    circuit ee,
    set resistor graphic                = resistor IEC graphic,%% 有变体
    set inductor graphic                = inductor IEC graphic,%% 有变体
    set capacitor graphic               = capacitor IEC graphic,% 无变体
    set contact graphic                 = contact IEC graphic,% 无变体
    set ground graphic                  = ground IEC graphic,% 无变体
    set battery graphic                 = battery IEC graphic,% 无变体
    set diode graphic                   = diode IEC graphic,%% 有变体
    set Zener diode graphic             = Zener diode IEC graphic,%% 有变体
    set tunnel diode graphic            = tunnel diode IEC graphic,%% 有变体
    set backward diode graphic          = backward diode IEC graphic,%% 有变体
    set Schottky diode graphic          = Schottky diode IEC graphic,%% 有变体
    set breakdown diode graphic         = breakdown diode IEC graphic,%% 有变体
    set bulb graphic                    = bulb IEC graphic,% 无变体
    set voltage source graphic          = voltage source IEC graphic,% 无变体
    set current source graphic          = current source IEC graphic,% 无变体
    set make contact graphic            = make contact IEC graphic,%% 有变体
    set break contact graphic           = break contact IEC graphic,% 无变体
    set amperemeter graphic             = amperemeter graphic,
    set voltmeter graphic               = voltmeter graphic,
    set ohmmeter graphic                = ohmmeter graphic,
    set ac source graphic               = ac source graphic,
    set dc source graphic               = dc source graphic,
  },
}%
```

之后就具体定义各个逻辑门符号图形 ⟨*symbol name*⟩ `IEC graphic`, 以及其变体 `var` ⟨*symbol name*⟩ `IEC graphic`, 例如:

```
\tikzset{
  %
```

```
  % normal diode
  %
  diode IEC graphic/.style={
    circuit symbol open,
    circuit symbol size=width 1.25 height 1.25,
    transform shape,
    shape=generic diode IEC,
    /pgf/generic diode IEC/before background={
      \pgfpathmoveto{\pgfqpoint{0pt}{-1pt}}
      \pgfpathlineto{\pgfqpoint{0pt}{1pt}}
      \pgfusepathqstroke
    },
  },
  var diode IEC graphic/.style={
    diode IEC graphic,
    circuit symbol filled,
  },
}%
```

注意有的逻辑门符号图形有“变体”，有的没有。

下面以 `/tikz/resistor` 为例说明 ee-symbol 的用法。

`/tikz/resistor`=⟨*options*⟩ (no default)

这个选项用作 `node` 操作的选项，或者 `to` 操作的选项。

用作 `node` 操作的选项 本选项使得 node 成为一个 resistor node:

```
\tikz [circuit ee IEC]
  \node [resistor] {};
```

与通常的 node 不同，resistor node 不能带有文字内容。可以使用选项 `label`，`info`，`ohm` 给它加标签。可以这样加标签:

5Ω

```
\tikz [circuit ee IEC]
  \node [resistor,ohm=5] {};
```

也可以这样加标签:

5Ω

```
\tikz [circuit ee IEC]
  \node [resistor={ohm=5}] {};
```

用选项 `point up`，`point down`，`point left`，`point right`，`rotate` 能让 resistor node 旋转:

5Ω

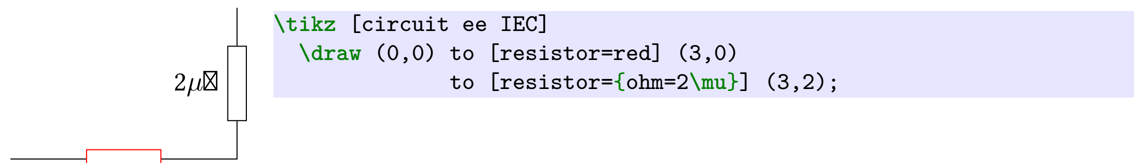
10kΩ

```
\tikz [circuit ee IEC] {
  \node (R1) [resistor,point up,ohm=5] at (3,1) {};
  \node (R2) [resistor,ohm=10k] at (0,0) {};
  \draw (R2) -| (R1);
}
```

用作 `to` 操作的选项 在 `circuit ee IEC` 的有效范围内，当 `resistor` 用作 `to` 操作的选项时，内部程序会调用 `circuit handle symbol`, 效果如下:

1. 当前路径会被裁截出一个缺口以容纳一个 node.
2. 选项 `resistor` 会创建一个 resistor node, 默认它会被旋转以指向路径的切线方向（除非使用 `shift only` 或其它设置）。
3. ⟨*options*⟩ 被传递给 resistor node.
4. 如果 ⟨*options*⟩ 中没有 `pos`，`at start`，`midway` 等选项来指定 resistor node 的位置，就使用 `pos=0.5` 决定的位置。

在 ⟨*options*⟩ 中可以使用标签选项:




```
\tikz [circuit ee IEC]
  \draw (0,0) to [resistor=red] (3,0)
            to [resistor={ohm=2\mu}] (3,2);
```

可以给一个 resistor node 加多个标签，可以在一段路径上放置多个 resistor node.

Inputs, Outputs, and Anchors 每个 ee-symbol 都有一个 `input` anchor 和一个 `output` anchor, 以及各种罗盘 anchor；个别 ee-symbol 会有更多的 anchor.

改变 resistor node 的外观 参考 49.2.6，也可以在 ⟨*options*⟩ 中用选项改变 resistor node 的外观。

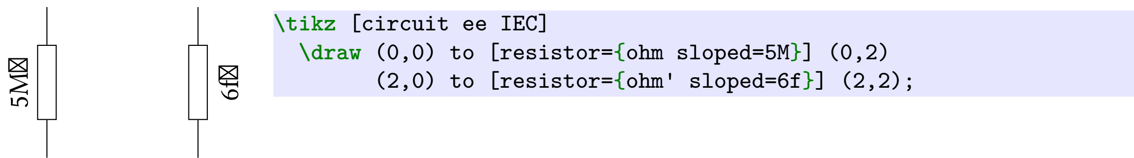
下面以 `/tikz/ohm` 为例说明物理单位的用法。

`/tikz/ohm` (no default)

这个选项给 node 加标签，标签外观是 `$\mathrm{`⟨*value*⟩`}\Omega}$`, 注意 ⟨*value*⟩ 是 `\mathrm` 的参数。


```
\tikz [circuit ee IEC] \draw (0,0) to [resistor={ohm=5M}] (0,2);
```

与 ohm 相关的选项有 `ohm'`，`ohm sloped`，`ohm' sloped`，`every ohm`:



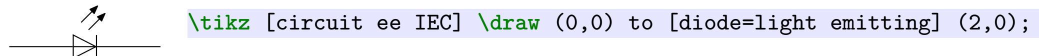

```
\tikz [circuit ee IEC]
  \draw (0,0) to [resistor={ohm sloped=5M}] (0,2)
        (2,0) to [resistor={ohm' sloped=6f}] (2,2);
```

样式选项 `every ohm` 针对每个 ohm 标签。

下面以 `/tikz/light emitting` 为例说明 annotation 选项的用法。

`/tikz/light emitting`=⟨*options*⟩ (no default)

这个 key 用作 ndoe 的选项，它会在 node 附近画一对平行箭头，作为“发光体”标识（annotation)。



```
\tikz [circuit ee IEC] \draw (0,0) to [diode=light emitting] (2,0);
```

在 ⟨*options*⟩ 中的选项可以如下：

1. 使用选项 `red` 等局部地改变 annotation 的外观。
2. 使用选项 `<-` 或 `-latex` 等改变 annotation 的箭头。
3. 使用选项 `ohm`，`info` 等给 annotation 加标签。

not good

better

also good

```
\tikz [circuit ee IEC]
{
  \draw (0,2) to [diode={light emitting,info=not good}] (2,2);
  \draw (0,0) to [diode={light emitting={info=better},
                                      info'=also good}] (2,0);
}
```

与本选项类似的选项是 `light emitting'`;与本选项相关的样式选项有 `every circuit annotation`, `every light emitting`. 可以用下面的样式选项设置 annotation 的箭头：

`/tikz/annotation arrow` (style, no value)

这个样式设置 annotation 的箭头。

49.4.2 指示电流方向的符号

在程序库 `circuits.ee` 中声明了指示电流方向的符号 `current direction`，`current direction'`，这两个符号的形状是 `shape=direction ee`.

参考文件《pgflibraryshapes.gates.ee.code.tex》的 `\pgfarrowsdeclare{direction ee}...`

命令 `\pgfarrowsdeclare` 见文件《pgfcorearrows.code.tex》.

```
\tikz [circuit ee IEC] \draw (0,2) to [current direction={pos=0.8
↪ },current direction'={pos=0.2}] (2,2);
```

49.4.3 Symbols: Basic Elements

49.4.4 Symbols: Diodes

49.4.5 Symbols: Contacts

49.4.6 Symbols: Measurement devices

49.4.7 Units

49.4.8 Annotations

49.4.9 EE-Symbols 的形状

EE-Symbols 的形状由库 `shapes.gates.ee` 和 `shapes.gates.ee.IEC` 定义。

TikZ Library shapes.gates.ee

```
\usepgflibrary{shapes.gates.ee} % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes.gates.ee] % ConTeXt and pure pgf
\usetikzlibrary{shapes.gates.ee} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.ee] % ConTeXt when using TikZ
```

这个库的源文件是《pgflibraryshapes.gates.ee.code.tex》，其中定义了 3 个形状：

- rectangle ee
- circle ee
- direction ee

形状 direction ee 就是符号 current direction，current direction' 的形状：

```
\tikz [circuit ee IEC] \draw (0,0) to[current direction] (1,0);
```

在程序库的默认定义下，形状 direction ee 的路径就是箭头 triangle 45 的路径，箭头 triangle 45 的定义见文件《pgflibraryarrows.code.tex》。

与通常的箭头不同，direction ee 的尺寸并不依赖当前的线宽选项 line width 的值。可以用命令 \pgfsetarrowoptions{direction ee}{5pt} 重设 direction ee 的长度。

Shape rectangle ee

这个形状与通常的 rectangle 相同；它的 anchor 位置只有罗盘位置、各 base 位置、各 mid 位置；其 input anchor 与 west anchor 相同；其 output anchor 与 east anchor 相同。

形状 rectangle ee 的定义是：

```
\pgfdeclareshape{rectangle ee}
{%
  \inheritsavedanchors[from=rectangle]%
  \inheritanchorborder[from=rectangle]%
  \inheritanchor[from=rectangle]{north}%
  \inheritanchor[from=rectangle]{north west}%
  \inheritanchor[from=rectangle]{north east}%
  \inheritanchor[from=rectangle]{center}%
  \inheritanchor[from=rectangle]{west}%
  \inheritanchor[from=rectangle]{east}%
  \inheritanchor[from=rectangle]{mid}%
  \inheritanchor[from=rectangle]{mid west}%
  \inheritanchor[from=rectangle]{mid east}%
  \inheritanchor[from=rectangle]{base}%
  \inheritanchor[from=rectangle]{base west}%
  \inheritanchor[from=rectangle]{base east}%
  \inheritanchor[from=rectangle]{south}%
  \inheritanchor[from=rectangle]{south west}%
  \inheritanchor[from=rectangle]{south east}%
  \inheritbackgroundpath[from=rectangle]%
  % New:
  \anchor{input}{
    \pgf@process{\northeast}%
    \pgf@ya=.5\pgf@y%
```

```
    \pgf@process{\southwest}%
    \pgf@y=.5\pgf@y%
    \advance\pgf@y by \pgf@ya%
  }%
  \anchor{output}{%
    \pgf@process{\southwest}%
    \pgf@ya=.5\pgf@y%
    \pgf@process{\northeast}%
    \pgf@y=.5\pgf@y%
    \advance\pgf@y by \pgf@ya%
  }%
}%
```

Shape `circle ee`

类似 `rectangle ee`.

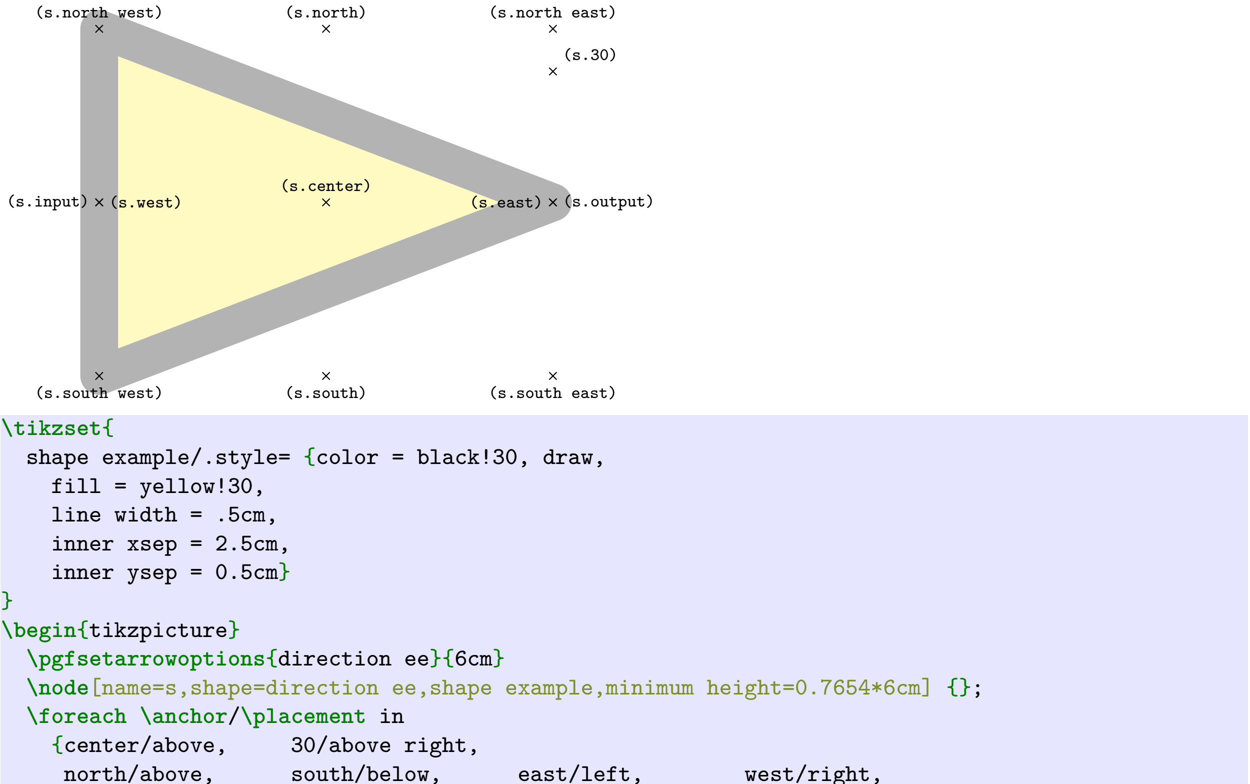
Shape `direction ee`

这是个特殊的形状。它的各个 anchor 都是罗盘位置，都位于矩形上。它的形态在默认下与箭头 `triangle 45` 相同。它的高度值是选项 `/pgf/minimum height` 的值。它的长度值（从 back end 到 tip end 的长度）由命令 `\pgfsetarrowoptions{direction ee}{`⟨*dimension*⟩`}` 设置。

如果要更换 `direction ee` 的形状路径（即换一种箭头样子），可以使用下面的选项:

`/pgf/direction ee arrow=`⟨*right arrow tip name*⟩ (no default)

这个选项更换 `direction ee` 的形状路径（即换一种箭头样子），但不能改变 `direction ee` 的 anchor 位置。




```
\tikzset{
  shape example/.style= {color = black!30, draw,
    fill = yellow!30,
    line width = .5cm,
    inner xsep = 2.5cm,
    inner ysep = 0.5cm}
}
\begin{tikzpicture}
  \pgfsetarrowoptions{direction ee}{6cm}
  \node[name=s,shape=direction ee,shape example,minimum height=0.7654*6cm] {};
  \foreach \anchor/\placement in
    {center/above,     30/above right,
     north/above,      south/below,      east/left,        west/right,
     north east/above, south east/below, south west/below, north west/above,
```

```
    input/left,       output/right}
   \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
     node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

(s.north)

(s.input) (s.output)

(s.south)

```
\tikzset{
  shape example/.style= {color = black!30, draw,
   fill = yellow!30,
   line width = .2cm,
   inner xsep = 2.5cm,
   inner ysep = 0.5cm}
}
\begin{tikzpicture}[direction ee arrow={Stealth[inset=2pt]}]
  \node[name=s,shape=direction ee,shape example,minimum height=1.75cm] {};
  \foreach \anchor/\placement in {north/above, south/below,
                                  output/right, input/left}
   \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
     node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.4.10 IEC 风格的 EE-Symbols 形状

TikZ Library shapes.gates.ee.IEC

```
\usepgflibrary{shapes.gates.ee.IEC} % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes.gates.ee.IEC] % ConTeXt and pure pgf
\usetikzlibrary{shapes.gates.ee.IEC} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.ee.IEC] % ConTeXt when using TikZ
```

这个库定义 IEC 风格的 EE-Symbols 形状。

Shape generic circle IEC

这个形状是圆圈，有 input anchor 和 output anchor. 可以用下面的选项给这个形状添加额外的路径:

`/pgf/generic circle IEC/before background=`⟨*code*⟩ (no default)

当形状为 `generic circle IEC` 的 node 被创建后,本选项的 ⟨*code*⟩ 会被作为“before background path” 画出来，也就是说，在画出圆圈形状路径后，再画出 ⟨*code*⟩ 中的路径，因此圆圈形状路径可能会被 ⟨*code*⟩ 中的路径遮挡。

在执行 ⟨*code*⟩ 时，坐标系会事先被变换为这样的状态：点 `(1pt, 0pt)` 是圆圈上的右侧点；点 `(0pt, 1pt)` 是圆圈上的上部点。也就是说，坐标系的原点位于圆圈的圆心，而圆圈的半径是 `1pt`，x 轴方向向右，y 轴方向向上；这相当于修改了长度单位 `pt` 所代表的实际长度。注意 ⟨*code*⟩ 中的坐标数据要带上长度单位，例如 `(1pt,0.05cm)`, 不要用 `(0.01,0.05)`.

```
\tikz \node [generic circle IEC,
    /pgf/generic circle IEC/before background={
      \draw [red, line
      ↪   width=3mm](0pt,0pt)--+(1pt,0pt)--+(0pt,1pt)--(-1pt,-0.05cm);
},
draw] {Hello world};
```

Shape `generic diode IEC`

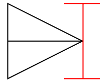
这个形状标识二极管，形状的尺寸决定于当前的 `minimum width`，`minimum height` 选项的值. 它的各个 anchor 如下面的图形所示。

可以用下面的选项在这个形状路径上添加额外的路径：

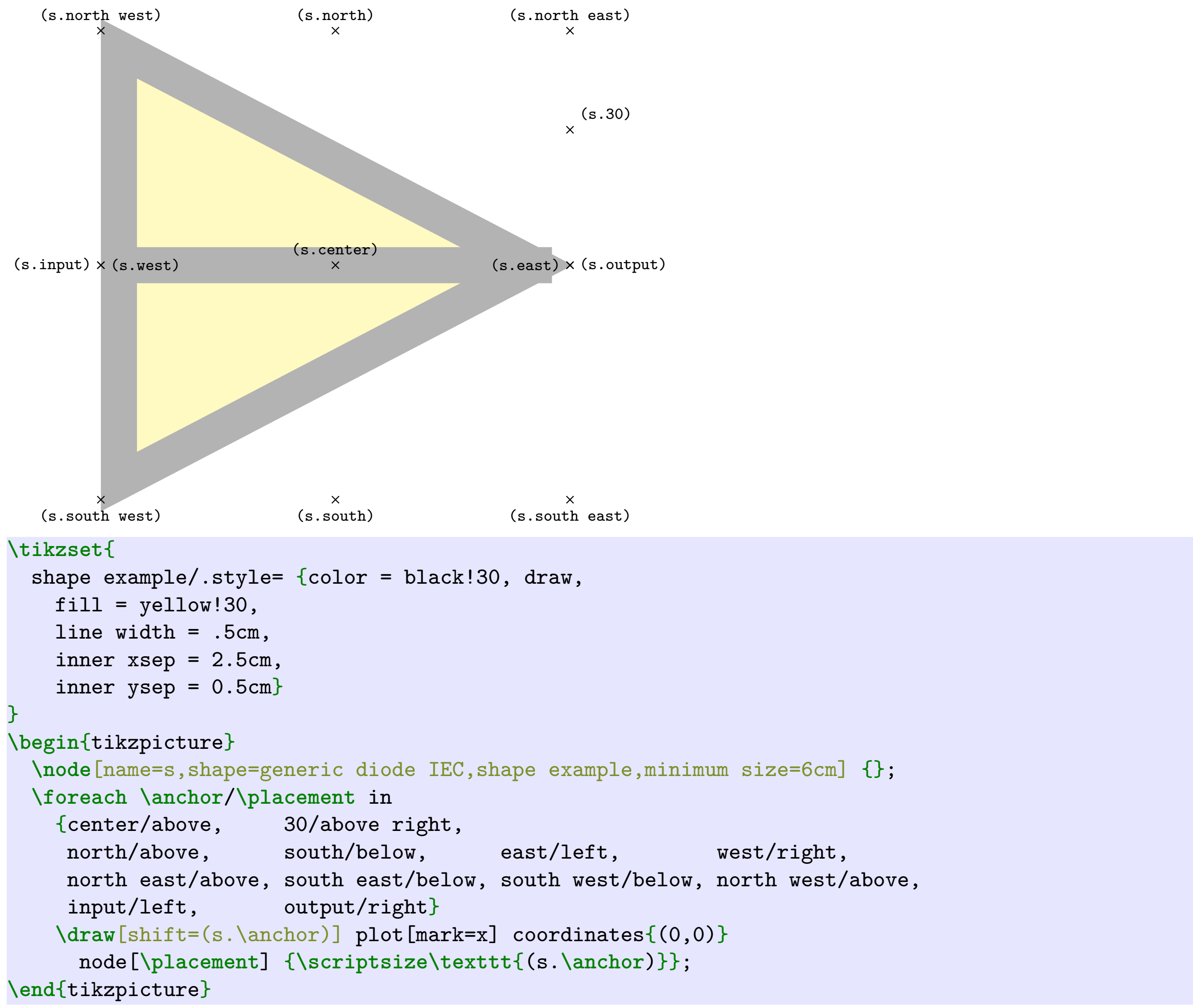
`/pgf/generic diode IEC/before background`=⟨*code*⟩ (no default)

当形状为 `generic diode IEC` 的 node 被创建后，本选项的 ⟨*code*⟩ 会被作为 "before background path" 画出来，也就是说，在画出二极管形状路径后，再画出 ⟨*code*⟩ 中的路径，因此二极管形状路径可能会被 ⟨*code*⟩ 中的路径遮挡。

在执行 ⟨*code*⟩ 时，坐标系会事先被变换为这样的状态：坐标系的原点是二极管三角形的右侧顶点；点 `(0pt，1pt)` 是二极管三角形的高度的一半，方向向上；点 `(1pt，0pt)` 是二极管三角形的高度的一半，方向向右；这相当于修改了长度单位 `pt` 所代表的实际长度。注意 ⟨*code*⟩ 中的坐标数据要带上长度单位，例如 `(1pt,0.05cm)`，不要用 `(0.01,0.05)`.



```
\tikz \node [minimum size=1cm,generic diode IEC,
    /pgf/generic diode IEC/before background={
      \pgfpathmoveto{\pgfqpoint{-.5pt}{-1pt}}
      \pgfpathlineto{\pgfqpoint{.5pt}{-1pt}}
      \pgfpathmoveto{\pgfqpoint{0pt}{-1pt}}
      \pgfpathlineto{\pgfqpoint{0pt}{1pt}}
      \pgfpathmoveto{\pgfqpoint{-.5pt}{1pt}}
      \pgfpathlineto{\pgfqpoint{.5pt}{1pt}}
      \pgfsetstrokecolor{red}
      \pgfusepathqstroke
    },
    draw] {};
```




```
\tikzset{
  shape example/.style= {color = black!30, draw,
    fill = yellow!30,
    line width = .5cm,
    inner xsep = 2.5cm,
    inner ysep = 0.5cm}
}
\begin{tikzpicture}
  \node[name=s,shape=generic diode IEC,shape example,minimum size=6cm] {};
  \foreach \anchor/\placement in
    {center/above,     30/above right,
     north/above,      south/below,      east/left,        west/right,
     north east/above, south east/below, south west/below, north west/above,
     input/left,       output/right}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.5 遇到的问题

使用 xelatex -shell-escape -8bit 编译时，选项 /tikz/ohm→P. 347 默认的物理单位 Ω 丢失。

10⊠

```
\tikz[circuit ee IEC] \node [resistor, ohm=10] {};
```

55 定点算术程序库

TikZ Library fixedpointarithmetic

```
\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, 然后再载入定点算术程序库 `fixedpointarithmetic`, 否则导致错误。

55.1 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 TikZ 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。

55.2 在 PGF 和 TikZ 中使用定点算术

通过使用以下 key，可以在 PGF 和 TikZ 中使用宏包 fp.

`/pgf/fixed point arithmetic=`⟨*options*⟩ (no default)

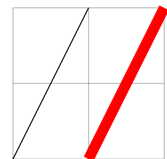
`/tikz/fixed point arithmetic=`⟨*options*⟩ (no default)

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。

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)

先看一个例子：



```
\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 PGF 数学引擎能处理的“小数值”。如果一个数值前面不带星号“*”，就不会使用宏包 fp 来处理该数值。

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乘上比例因子 ⟨*factor*⟩. 本选项不依赖选项 `scale results`.

`/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*⟩.

`/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*⟩.

56 浮点单元程序库

TikZ Library fpu

```
\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。

56.1 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 $-1 \cdot 10^{324}$ 到 $1 \cdot 10^{324}$. 大于 0 的最小数值是 $1 \cdot 10^{-324}$. fpu 的相对精度至少是 $1 \cdot 10^{-4}$，而且对于像加法那样的运算，相对精度是 $1 \cdot 10^{-6}$.

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 $\pm 0 \cdot 10^0$；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。

56.2 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。如果启用，那么 PGF 的那些标准的数学解析过程会被换成 fpu 版本，例如，将 `\pgfmathadd` 换成 `\pgfmathfloatadd`，所有的数值都会用 `\pgfmathfloatparsenumber` 来解析，解析结果仍然保存在 `\pgfmathresult` 中（通常是不带单位的浮点数格式）。

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。

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 分组的限制，故若只是在组内使用 fpu，无需使用 `fpu=false`.

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。

1Y2.17765411e23]

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=float}
\pgfmathparse{exp(50)*42} \pgfmathresult
```

2.17765411e23

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=sci}
\pgfmathparse{exp(50)*42} \pgfmathresult
```

217765411000000000000000.0

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=fixed}
\pgfmathparse{exp(50)*42} \pgfmathresult
```

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。利用命令 `\tikzmath{...}` 做个变通可以消除这个错误：

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=fixed}
\tikzmath{
  \xunit=1/10000;
  \yunit=1/10000;
}
\begin{tikzpicture}[x=\xunit cm, y=\yunit cm]
  \draw [red, ultra thick] (0,0)--(10000,20000);
\end{tikzpicture}
```

下面的代码:

```
\tikzmath{
  \xunit=10^(-5);
  \yunit=10^(-5);
}
\begin{tikzpicture}[x=\xunit cm, y=\yunit cm]
  \draw [red, ultra thick] (0,0)--(10000,20000);
\end{tikzpicture}
```

不会导致错误，但也不会输出所要画的直线段，调用 fpu 程序库可以画出代码中的直线段:

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=fixed}
\tikzmath{
  \xunit=10^(-5);
  \yunit=10^(-5);
}
\begin{tikzpicture}[x=\xunit cm, y=\yunit cm]
  \draw [red, ultra thick] (0,0)--(100000,200000);
\end{tikzpicture}
```

上面的例子说明，调用 fpu 程序库可以提高计算的精度。但是下面的代码:

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=fixed}
\tikzmath{
  \xunit=10^(-6);
  \yunit=10^(-6);
}
\begin{tikzpicture}[x=\xunit cm, y=\yunit cm]
  \draw [red, ultra thick] (0,0)--(1000000,2000000);
\end{tikzpicture}
```

不会导致错误，但也不会输出所要画的直线段。而下面的代码:

```
\pgfkeys{/pgf/fpu, /pgf/fpu/output format=fixed}
\tikzmath{
  \xunit=10^(-10);
  \yunit=10^(-10);
}
\begin{tikzpicture}[x=\xunit cm, y=\yunit cm]
  \draw [red, ultra thick] (0,0)--(10000000000,0);
\end{tikzpicture}
```

导致错误：! Number too big. 这表明，仅仅调用 fpu 程序库并不能使得 Tikz 命令具有 fpu 的计算能力。

56.3 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。

56.4 命令与编程参考

56.4.1 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。由于本命令以纯文本形式读取、处理 ⟨x⟩, 所以参数 ⟨x⟩ 可以是任意数量级和任意精度的数值, 本命令把处理结果以纯文本形式保存在命令 `\pgfmathresult` 中。前面提到，默认以 `float` 格式保存数值。

当手工输入参数 ⟨x⟩ 时，⟨x⟩ 的格式可以是定点数格式，也可以是使用 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。

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。

`/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⟩}`

类似 `\pgfmathfloatparsenumber`, 只是不会执行详细的校验，因此速度快一些。

`\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读取、处理 ⟨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;
```

\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;
```

\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。

\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。

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
```

\pgfmathfloatifflags{⟨*floating point number*⟩}{⟨*flag*⟩}{⟨*true-code*⟩}{⟨*false-code*⟩}

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!}
```

\pgfmathfloattomacro{⟨*x*⟩}{⟨*flags macro*⟩}{⟨*mantissa macro*⟩}{⟨*exponent macro*⟩}

参数 ⟨*x*⟩ 是浮点数格式的，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。

\pgfmathfloattoregisters{⟨*x*⟩}{⟨*flags count*⟩}{⟨*mantissa dimen*⟩}{⟨*exponent count*⟩}

参数 ⟨*x*⟩ 是浮点数格式的，⟨*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。

\pgfmathfloattoregisterstok{⟨*x*⟩}{⟨*flags count*⟩}{⟨*mantissa toks*⟩}{⟨*exponent count*⟩}

参数 ⟨*x*⟩ 是浮点数格式的，⟨*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。

\pgfmathfloatgetflags{⟨*x*⟩}{⟨*flags count*⟩}

参数 ⟨*x*⟩ 是浮点数格式的，⟨*flags count*⟩ 是已定义的整数寄存器，⟨*x*⟩ 的标记会被保存在 ⟨*flags count*⟩ 中。

\pgfmathfloatgetflagstomacro{⟨*x*⟩}{⟨*macro*⟩}

参数 ⟨*x*⟩ 是浮点数格式的，本命令定义宏 ⟨*macro*⟩, 并将 ⟨*x*⟩ 的标记保存在宏 ⟨*macro*⟩ 中。

\pgfmathfloatgetmantissa{⟨*x*⟩}{⟨*mantissa dimen*⟩}

参数 ⟨*x*⟩ 是浮点数格式的, ⟨*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*⟩ 是浮点数格式的, ⟨*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*⟩ 是浮点数格式的,⟨*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。

56.4.2 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⟩}`

按选项 `/pgf/number format/precision`→P. 620 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，保存结果的格式（按有关选项的设置）是定点数格式或科学计数法格式。

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
```

`\pgfmathroundtozerofill{⟨x⟩}`

按选项 `/pgf/number format/precision`→P. 620 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
```

56.4.3 数学运算命令

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。

2Y3.056789e2]

```
\pgfmathfloatadd{1Y2.0e1]}{2Y3.256789e2]}
\pgfmathresult
```

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：

15241.374000000000

```
\pgfmathfloatparsenumber{0.123456}
\let\bili=\pgfmathresult
\pgfmathfloatparsenumber{123456}
\pgfmathfloatmultiply{\pgfmathresult}{\bili}
\pgfmathfloattofixed{\pgfmathresult}
\pgfmathresult
```

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*⟩}

\pgfmathfloatsetextprecision{⟨*shift*⟩}

\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 $\langle x\rangle < \langle y\rangle$, 则本命令设置 \pgfmathresult 的值是 1.0, 并且设置 \global\pgfmathfloatcomparisontrue; 否则设置 \pgfmathresult 的值是 0.0 以及 \global\pgfmathfloatcomparisonfalse.

0.0

```
\pgfmathfloatlessthan{1Y2.1e6]}{1Y2.1e6]}
\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。

\pgfmathfloatifapproxequalrel{⟨*a*⟩}{⟨*b*⟩}{⟨*true-code*⟩}{⟨*false-code*⟩}

本命令会调用命令 \pgfmathfloatparsenumber 来读取 ⟨*a*⟩, ⟨*b*⟩. 本命令会参考选项 /pgf/fpu/rel thresh 的设置来判断 ⟨*a*⟩ 与 ⟨*b*⟩ 的近似程度。本命令计算 ⟨*a*⟩ 与 ⟨*b*⟩ 的相对误差 $\frac{|a-b|}{|b|}$(假设 $\langle b\rangle \neq 0$, 但实际上可以写成 0). 如果相对误差小于选项 rel thresh 设置的值，则执行 ⟨*true-code*⟩, 否则执行 ⟨*false-code*⟩.

/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。

\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
```

56.4.4 用于编程的原始数学程序

当载入 fpu 库后，形式为 \pgfmath@basic@⟨*name*⟩@ 的数学命令可用。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。

56.4.5 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)
```

下面是另一个例子。

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'..

56.5 遇到的问题

按手册中描述,选项 `/pgf/fpu/scale results`→P. 357 的用法与选项 `/pgf/fixed point/scale results`→P. 354 的用法类似。但测试下面代码

```
\begin{tikzpicture}[/pgf/fpu, /pgf/fpu/scale results=1/10000]
  \draw [red] (0,0)--(*10000,*20000);
\end{tikzpicture}
```

会导致错误：! Illegal unit of measure (pt inserted).

57 Lindenmayer System 分形图

一个简单的 L-S

分形图有很多种, Lindenmayer System 是绘制某一类型分形图的方法。程序库 lindenmayersystems 提供了实现简单 Lindenmayer System 的命令。绘制 Cantor 集合示意图的过程可以看作是一个 Lindenmayer System 的例子:

给定一个线段 A_0B_0，对线段 A_0B_0 做以下操作：第一步，将 A_0B_0 做 3 等分，去掉中间的那一段，留下两边的线段，得到 $A_{11}B_{11}$ 和 $A_{12}B_{12}$；第二步，将 $A_{1i}B_{1i}$，$i=1,2$ 进行 3 等分，去掉中间的一段，留下的线段记为 $A_{2j}B_{2j}$，$j=1,2,3,4$；第三步，将 $A_{2j}B_{2j}$，$j=1,2,3,4$ 进行 3 等分，去掉中间的一段，留下的线段记为 $A_{3k}B_{3k}$，$k=1,2,3,4,5,6,7,8$……像这样，每一次操作都是针对上一步的结果，操作内容都是将各个连续线段分别进行 3 等分，去掉中间的一段，留下两边的线段；原来线段 A_0B_0 上的很多点被去掉了，但是还有很多点保留了下来，令操作步骤达到无穷，在极限状态之下，那些保留下来的点构成的集合叫作 Cantor 集合。下面的图形表示了从初始线段到第三步操作的结果。

初始线段 A_0 B_0

第一步的结果 A_{11} B_{11} A_{12} B_{12}

第二步的结果 A_{21} B_{21} A_{22} B_{22} A_{23} B_{23} A_{24} B_{24}

第三步的结果

假设 A_0B_0 的长度是 1，那么初始线段可以这样得到：设想拿一个画笔 p，令 p 右移 1，移动时画线，这就画出了 A_0B_0. 如果令字母 F 表示“右移 1 且在移动时画线”，那么初始线段就可以表示为：`F`.

第一步的结果也可以这样得到：先令画笔 p 右移 $\frac{1}{3}$，移动时画线；再令 p 右移 $\frac{1}{3}$，移动时不画线；再令 p 右移 $\frac{1}{3}$，移动时画线，这就画出了第一步的结果。如果令字母 F 表示“右移 $\frac{1}{3}$ 且在移动时画线”，令字母 f 表示“右移 $\frac{1}{3}$ 且在移动时不画线”，那么第一步的结果就可以表示为：`FfF`.

类似地，如果令字母 F 表示“右移 $\frac{1}{9}$ 且在移动时画线”，令字母 f 表示“右移 $\frac{1}{9}$ 且在移动时不画线”，那么第二步的结果就可以表示为：`FfFfffFfF`.

如果令字母 F 表示“右移 $\frac{1}{27}$ 且在移动时画线”，令字母 f 表示“右移 $\frac{1}{27}$ 且在移动时不画线”，那么第三步的结果就可以表示为：`FfFfffFfFfffffffffFfFfffFfF`.

暂时忽略画笔移动的距离，看看代表各个步骤结果的字母符号之间有什么关系。事实上，表示各个步骤结果的字母符号可以通过一些列的符号替换操作得到，这里使用的替换规则只有两条：

rule1 `F` $\to$ `FfF`，即将 `F` 替换为“`FfF`”；

rule2 `f` $\to$ `fff`，即将 `f` 替换为“`fff`”.

这里使用“F → FfF”这样的格式表达替换规则，箭头前的 F 叫作“前任”，箭头后的 FfF 叫作“继任”。

给定初始线段对应的字母 F，按照替换规则做替换，前 3 个替换步骤的结果是:

步骤	替换结果	使用的规则
初始	`F`	给定的初始符号
第一步	`FfF`	`rule1`
第二步	`FfFfffFfF`	`rule1，rule2`
第三步	`FfFfffFfFfffffffffFfFfffFfF`	`rule1，rule2`

以 n 代表各个步骤，$n=0$ 代表初始状态。在第 n 步的结果中，令 F 表示“向右移动 $\frac{1}{3^n}$ 且在移动时画线”，令 f 表示“向右移动 $\frac{1}{3^n}$ 且在移动时不画线”，那么第 n 步的字母符号串恰好表达了第 n 步的图形。

将 F，f，以及替换规则 rule1，rule2 看作一个体系，这就是一个简单的 Lindenmayer System. 简单地说，给定一组符号、符号替换规则，就给出了一个 Lindenmayer System，简称为“L-S”.

L-S 的分类

一、确定与随机

L-S 有多种类型。在某个 L-S 中做符号替换时，如果有随机化的处理，例如，随机地选择符号替换规则，随机地选择画笔移动距离，随机地选择画笔的旋转角度，那么这个 L-S 就是“随机的”L-S，否则就是“确定的”L-S.

二、是否含有 X，Y

L-S 也可以分为“不含 X，Y 的”与“含 X，Y 的”。这里 X，Y 是体系使用的符号。在替换符号时会用到 X 和 Y，但是 X 和 Y 不对应任何画图动作，即它们对“画笔”的动作无影响。显然，如果一个 L-S 含有 X，Y，那么它具有更高的灵活性和构图能力。

下面的例子是一个含 X，Y 的体系。用下面的条目规定一个 L-S：

- 变量：X，Y；
- 常量：F，$+$，$-$；
- 初始符号：X；
- 规则：$X \to X - YF$，$Y \to FX + Y$；
- 角度：90°.

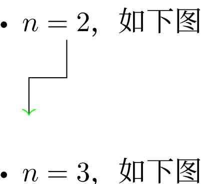
先解释一下上面的各个条目。在初始之下，画笔位于原点，以水平向右的方向为画笔的“前方”。F 表示“画笔向前移动并画线”。$+$ 和 $-$ 表示旋转画笔，旋转的幅度由条目 90° 来规定。$+$ 表示将画笔的“前方”方向逆时针旋转 90°. $-$ 表示将画笔的“前方”方向顺时针旋转 90°. 符号替换规则只有两条，前三次替换是：

$$
\begin{aligned}
&n=0 \quad X\\
&n=1 \quad X-YF\\
&n=2 \quad X-YF-FX+YF\\
&n=3 \quad X-YF-FX+YF-FX-YF+FX+YF
\end{aligned}
$$

令第 n 个步骤中的 F 表示“向前移动 $\frac{1}{2^n}$ 并画线”，上面各个步骤的符号对应的图形分别是：

- $n=0$，符号 X 不引起画笔动作。
- $n=1$，符号 X，Y 不引起画笔动作，只有 $-$ 和 F 引起画笔动作

- $n = 2$，如下图



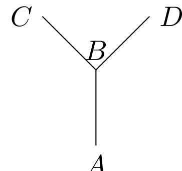
- $n = 3$，如下图

上面图形中的箭头只是为了表明画图时画笔的行动方向，不属于图形本身。

三、是否含有方括号

有的 L-S 含有方括号，即左方括号“`[`”与右方括号“`]`”. 左方括号“`[`”会使得程序将当前的画笔状态（包括画笔的位置、画笔的前方方向）暂时封存在某个地方，继续执行“`[`”后的画图动作，直到遇到右方括号“`]`”，再将画笔恢复到左方括号“`[`”所“封存”的状态。

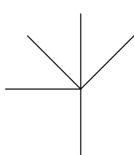
举例来说，令 `F` 表示画笔向前移动 1 单位长度并画线，`F` 的前方方向初始为“向右”；“$+$”表示左传 $45°$；“$-$”表示右转 $45°$，那么符号串 `++F[+F]-F` 画出的图形就是：



上面图形的绘制过程是：

1. 画笔一开始处于 A 点，开头的两个“+”使得画笔的前方转到“向上”，然后画出线段 AB；
2. 然后读取“`[`”，将画笔的状态，即位置 B 和前方“向上”，暂时封存；
3. 然后执行“`+F`”，画笔左传 $45°$，画出线段 BC；
4. 然后读取“`]`”，调出保存的画笔状态，即令画笔状态恢复到——位置 B 和前方“向上”；
5. 然后执行“`-F`”，画笔右传 $45°$，画出线段 BD.

方括号可以套嵌使用，例如，符号 `F`，`+`，`-` 的意义如前一个例子，那么符号串 `++F[[[++F]+F]F]-F` 画出的图形就是



四、其它类型

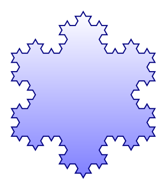
L-S 还有其它类型，不过本程序库只涉及以上几种。

57.1 Overview

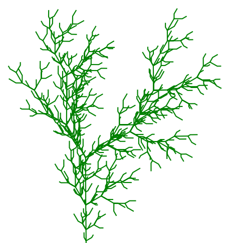
TikZ Library lindenmayersystems

```
\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。



```
\begin{tikzpicture}
\pgfdeclarelindenmayersystem{Koch curve}{
  \rule{F -> F-F++F-F}
}
\shadedraw [top color=white, bottom color=blue!50, draw=blue!50!black]
    [l-system={Koch curve, step=2pt, angle=60, axiom=F++F++F, order=3}]
  lindenmayer system -- cycle;
\end{tikzpicture}
```



```
\begin{tikzpicture}
\draw [green!50!black, rotate=90]
    [l-system={rule set={F -> FF-[-F+F]+[+F-F]}, axiom=F, order=4,
    ↪  step=2pt,
     randomize step percent=25, angle=30, randomize angle percent=5}]
  lindenmayer system;
\end{tikzpicture}
```

57.1.1 声明一个 L-S

下面的命令声明（定义）一个 L-S：

`\pgfdeclarelindenmayersystem{⟨name⟩}{⟨specification⟩}`

⟨*name*⟩ 是所定义（声明）的 L-S 的名称，⟨*specification*⟩ 是 L-S 的定义，其中通常使用命令 `\symbol` 和 `\rule` 来定义，也就是说，定义的 L-S 一般只包括“画图符号”和“符号替换规则”。
本命令的声明全局有效。

`\symbol{⟨name⟩}{⟨code⟩}`

⟨*name*⟩ 是一个或一串由字母或者数字构成的符号，⟨*code*⟩ 是对该 ⟨*name*⟩ 的定义，使得 ⟨*name*⟩ 能执行某种画图动作。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

在程序库中，符号 `F`，`f`，`+`，`-`，`[`，`]` 是具有某种作用的预定义符号，它们的作用分别是：

- `F`, 执行命令 `\pgflsystemdrawforward`，令画笔从当前点向前移动且在移动时画线。
- `f`, 执行命令 `\pgflsystemmoveforward`，令画笔从当前点向前移动且在移动时不画线。
- `+`, 执行命令 `\pgflsystemturnleft`，将画笔的“前方”向左旋，即逆时针转动。

- -, 执行命令 \pgflsystemturnright，将画笔的“前方”向右旋，即顺时针转动。
- [, 执行命令 \pgflsystemsavestate，保存画笔的当前状态。
-], 执行命令 \pgflsystemrestorestate，调出“[”所保存的画笔状态。

下面的选项用来设置各个符号代表的 ⟨*code*⟩.

/pgf/lindenmayer system/step=⟨*length*⟩ (no default, initially 5pt)

这个选项设置画笔向前移动的距离。⟨*length*⟩ 会被保存在 TEX 尺寸宏 \pgflsystemstep 中。

/pgf/lindenmayer system/randomize step percent=⟨*percentage*⟩ (no default, initially 0)

⟨*percentage*⟩ 是百分比下的数值,从 0 到 100,该值会被保存在 TEX 宏 \pgflsystemrandomizesteppercent 中。使用该选项会引起画笔移动长度的随机化，程序中的命令 \pgflsystemrandomizestep 会利用本选项设置的值。

/pgf/lindenmayer system/left angle=⟨*angle*⟩ (no default, initially 90)

⟨*angle*⟩ 会被保存在 TEX 宏 \pgflsystemrleftangle 中。这个选项设置逆时针旋转画笔“前方”方向的角度。

/pgf/lindenmayer system/right angle=⟨*angle*⟩ (no default, initially 90)

⟨*angle*⟩ 会被保存在 TEX 宏 \pgflsystemrrightangle 中。这个选项设置顺时针旋转画笔“前方”方向的角度。

/pgf/lindenmayer system/angle=⟨*angle*⟩ (no default)

同时设置选项 left angle=⟨*angle*⟩ 和 right angle=⟨*angle*⟩.

/pgf/lindenmayer system/randomize angle percent=⟨*percentage*⟩ (no default, initially 0)

⟨*percentage*⟩ 是百分比下的数值，从 0 到 100，该值会被保存在 TEX 宏 \pgflsystemrandomizeanglepercent 中。
使用该选项会引起画笔的“前方”旋转角度的随机化，在程序处理过程中，程序中的命令 \pgflsystemrandomizeleftangle 或 \pgflsystemrandomizerightangle 会利用本选项设置的值。

当符号执行 ⟨*code*⟩ 时，下面的命令可用。

\pgflsystemcurrentstep

这个宏的值是画笔在当前点移动的长度,在初始之下,这个宏的值就是 TEX 尺寸宏 \pgflsystemstep 的值，即由选项 step=⟨*length*⟩ 设定的尺寸，该尺寸的初始值是 5pt.
如果使用了命令 \pgflsystemrandomizestep，那么画笔移动的长度会被随机化。

\pgflsystemcurrentleftangle

这个宏的值是画笔的“前方”方向在当前点逆时针旋转的角度，在初始之下，这个宏的值就是 TEX 宏 \pgflsystemrleftangle 的值，即由选项 left angle=<angle> 设定的角度，该角度的初始值是 90.
如果使用了命令 \pgflsystemrandomizeleftangle，那么画笔的“前方”方向逆时针旋转的角度会被随机化。

`\pgflsystemcurrentrightangle`

这个宏的值是画笔的“前方”方向在当前点顺时针旋转的角度，在初始之下，这个宏的值就是 TEX 宏 `\pgflsystemrrightangle` 的值，即由选项 `right angle=`⟨*angle*⟩ 设定的角度，该角度的初始值是 90.

如果使用了命令 `\pgflsystemrandomizerightangle`，那么画笔的“前方”方向顺时针旋转的角度会被随机化。

当自定义符号时，下面的命令可能会用得上。

`\pgflsystemrandomizestep`

本命令会参考选项 `randomize step percent=`⟨*percentage*⟩ 设置的数值，将保存在 `\pgflsystemcurrentstep` 中的值随机化。

`\pgflsystemrandomizeleftangle`

本命令会参考选项 `randomize angle percent=`⟨*percentage*⟩ 设置的数值，将保存在 `\pgflsystemcurrentleftangle` 中的值随机化。

`\pgflsystemrandomizerightangle`

本命令会参考选项 `randomize angle percent=`⟨*percentage*⟩ 设置的数值，将保存在 `\pgflsystemcurrentrightangle` 中的值随机化。

`\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，本命令会调用 `\pgflsystemrandomizestep`.

`\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，本命令会调用 `\pgflsystemrandomizestep`.

`\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，本命令会调用 `\pgflsystemrandomizeleftangle`.

`\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，本命令会调用 `\pgflsystemrandomizerightangle`.

`\pgflsystemsavestate`

将画笔的当前状态暂时封存到另一个地方。

`\pgflsystemrestorestate`

调出封存的画笔状态。

\rule{⟨*head*⟩->⟨*body*⟩}

这个命令声明符号替换规则。下面的例子中，命令 \rule 的声明中使用的符号 A，B 是未经命令 \symbol 定义的，它们不影响画笔动作，只用于符号替换，故所定义的 L-S 属于“含 X，Y”的类型。

```
\pgfdeclarelindenmayersystem{Hilbert curve}{
  \symbol{X}{\pgflsystemdrawforward}  % 画线符号
  \symbol{+}{\pgflsystemturnright}  % 顺时针转
  \symbol{-}{\pgflsystemturnleft}  % 逆时针转
  \rule{A -> +BX-AXA-XB+}  % 替换规则, A 和 B 不影响画笔
  \rule{B -> -AX+BXB+XA-}  % 替换规则
}
\tikz\draw
↪  [lindenmayer system={Hilbert curve, axiom=A, order=4, angle=90}]
↪  % 初始符号是 A
  lindenmayer system;
```

57.2 使用 L-S

57.2.1 在 PGF 中使用 L-S

\pgflindenmayersystem{⟨*name*⟩}{⟨*axiom*⟩}{⟨*order*⟩}

⟨*name*⟩ 是已经定义（声明）的 L-S，前面的命令 \pgfdeclarelindenmayersystem 可以定义（声明）一个 L-S. 选项 ⟨*axiom*⟩ 规定初始符号。选项 ⟨*order*⟩ 规定符号替换的次数。

本命令会按照 ⟨*name*⟩ 的定义，计算第 ⟨*order*⟩ 次符号替换后的结果，得到一串符号，并将这串符号变成图形。

```
\begin{tikzpicture}
  \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}
```

57.2.2 在 TikZ 中使用 L-S

在 TikZ 中使用 L-S 的句法比较灵活，可以使用路径命令画出先前定义的 L-S，也可以在路径命令中用选项临时定义一个 L-S 并画出其图形。

\path ... lindenmayer system[⟨*keys*⟩] ...;

这个路径句法中，操作 lindenmayer system 会调用本程序库。其中 ⟨*keys*⟩ 可以是本程序库提供的选项，也可以是 TikZ 的选项（如 draw，color，line width）。这个句法与下面几个句法等效：

```
\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。

本程序库还提供了下面的选项, 它们只能用在 TikZ 中。它们的路径都是 `/pgf/lindenmayer system`.

`/pgf/lindenmayer system={⟨keys⟩}` (style, no default)

`/tikz/lindenmayer system={⟨keys⟩}` (style, no default)

这个选项用于临时定义一个 L-S，定义内容由 ⟨*keys*⟩ 指定，⟨*keys*⟩ 中使用的选项 (key) 应该具有路径前缀 `/pgf/lindenmayer system`, 例如前文和下文介绍的选项。

`/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*⟩ 是先前已经定义的 L-S 的名称，用 ⟨*name*⟩ 调用这个 L-S 的定义，画出其图形。

`/pgf/lindenmayer system/axiom={⟨string⟩}` (no default)

设置 L-S 的初始符号串 ⟨*string*⟩.

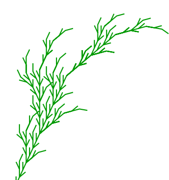
`/pgf/lindenmayer system/order={⟨integer⟩}` (no default)

指定符号替换的次数，获得第 ⟨*integer*⟩ 次符号替换后的结果。

`/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。

下面是一个用选项定义 L-S 并画出其图形的例子。



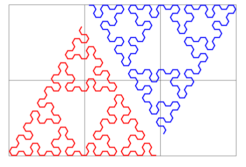
```
\tikz[rotate=65]\draw [green!60!black] l-system
  [l-system={rule set={F -> F[+F]F[-F]}, axiom=F, order=4,
    angle=25,step=3pt}];
```

`/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，当前点就成为这个 node 的锚定点，程序会把该 node 的 ⟨*anchor*⟩ 位置放在这个点上。

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}
```

58 数学程序库

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。

58.1 Overview

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。TikZ 的 calc 程序库也能执行计算，但一般情况下 calc 只用于 TikZ 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。

`\tikzmath{`⟨*statements*⟩`}`

⟨*statements*⟩ 是程序库提供的各种语句，注意

- 每个语句都要以分号“;”结束，否则报错。
- ⟨*statements*⟩ 中不能有空行，否则报错。

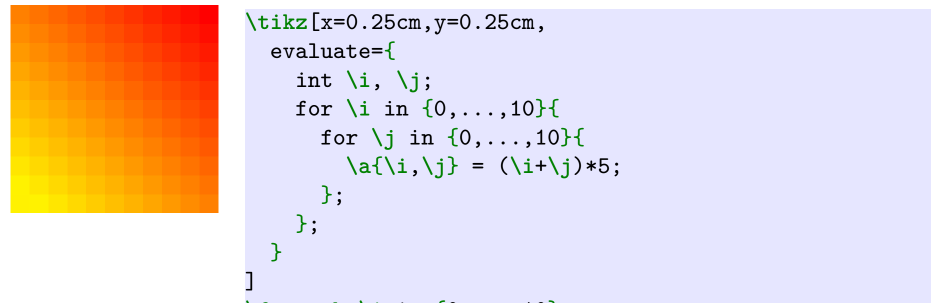
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765,

```
\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
}
```

`/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);
```

程序库 math 提供的语句中有一些关键词（keyword），如上面例子中的 `function`, `if`, `return`, `int`, 注意用空格把关键词分隔起来，否则 Tikz 不能识别关键词。

58.2 赋值语句

本程序库提供两种赋值方式。第一种是使用等号“=”赋值:

`\`⟨*macro*⟩ `=` ⟨*expression*⟩`;`

数学引擎会解析 ⟨*expression*⟩ 的值，得到最终的计算结果，并将结果赋予宏 \⟨*macro*⟩，例如

26.0, 2.0, 11, 225.0pt

```
\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
```

在数学表达式中经常有 $x_1 = 0$，$x_2 = 1$，这种“字母”缀“标号”的写法，本程序库也提供类似的赋值格式，可以给 TeX 宏（不是 TeX 寄存器）加后缀，例如加数字后缀:

7.0, 70.0, 700.0

```
\tikzmath{
  \x1 = 3+4; \x2 = 30+40; \x3 = 300+400;
}
\x1, \x2, \x3
```

也可以加文字后缀，不过需要把后缀文字用花括号括起来：

340, 1435, 6100

```
\tikzmath{
  \c{air} = 340; \c{water} = 1435; \c{steel} = 6100;
}
\c{air}, \c{water}, \c{steel}
```

给变量宏加后缀时，PGF 会有一些特殊的操作，看下面的例子：

7.0, 70.0,

```
\tikzmath{
 \x = 5; \x1 = 3+4; \x2 = 30+40;
}
\x, \x1, \x2,
```

在这个例子中，宏 \x 并没有值，这是因为带后缀的宏 \x1 和 \x2 与前面不带后缀的宏 \x 的名称的“主要部分”相同，后面的宏取消了 \x 的原来的定义，使之没有任何作用。对比下面的例子：

5, 7.0, 70.0,

```
\tikzmath{
 \y=5; \x1 = 3+4; \x2 = 30+40;
}
\y, \x1, \x2,
```

反过来也有类似的效果：

51, 52, 5,

```
\tikzmath{
 \x1 = 3+4; \x2 = 30+40; \x=5;
}
\x1, \x2, \x,
```

显然并没有输出所需要的 \x1 和 \x2 的值，输出的只是在（宏 \x 代表的）数字“5”后面缀上数字“1”或“2”. 这是因为带后缀的宏 \x1 和 \x2 与后面不带后缀的宏 \x 的名称的“主要部分”相同，后面的宏取消了前面宏的原来定义，使之没有任何作用。

另一种赋值方式是使用关键词 let：

let ⟨*variable*⟩ = ⟨*expression*⟩;

⟨*expression*⟩ 是 TEX 宏,在赋值时不执行 ⟨*expression*⟩, 当执行运算时 ⟨*expression*⟩ 会被用 \edef 展开。⟨*expression*⟩ 前面的空格会被忽略, 后面(直到分号之前)的空格会被认为是 ⟨*expression*⟩ 的一部分, 一并赋予 ⟨*variable*⟩.

(5*4)+1-2, “blue”

```
\tikzmath{
  let \x = (5*4)+1-2;
  let \c1 = blue;
}
\x, “\c1”
```

从上面的例子看出，所做的赋值在命令 \tikzmath 之外也是有效的。

58.3 声明变量类型

在默认下，数学引擎解析表达式，解析结果会带有小数点，而且小数部分至少有一个数字，然后将结果赋予变量。本程序库支持 3 种变量类型：整数、实数、坐标。

`integer` ⟨*variable*⟩;
`integer` ⟨*variable1, variable2, ...*⟩;

这里 ⟨*variable*⟩, ⟨*variable1*⟩, ⟨*variable2*⟩……都是 TeX 宏，将它们声明为整数变量，当把某个数值（或者表达式）赋予它们时，把最终的解析结果的小数部分（包括小数点）直接去掉（不做舍入），只保留数值的整数部分。

$x = 26$, $y = 2$, $z = 9$

```
\tikzmath{
  integer \x, \y, \z;
    \x = 4*5+6;
    \y = sin(30)*4;
    \z = log10(512) / log10(2);
  print {$x=\x$, $y=\y$, $z=\z$};
}
```

可以给整数变量宏加后缀来表示不同的整数变量，采用如下办法：

7, 70, 700

```
\tikzmath{
  integer \x;
  \x1 = 3+4; \x2 = 30+40; \x3 = 300+400;
}
\x1, \x2, \x3
```

即先声明一个宏，然后这个宏带上不同的后缀，表示不同的整数变量。如果后缀序号超过“9”，则后缀要用花括号括起来，例如 `\a{10}`, `\a{213}`，等等。后缀也可以使用字母，只需把后缀字母用花括号括起来，例如 `\a{bc}`，这种加后缀的办法与“let ... in”句法是一致的，参考 §14.15.

`int` ⟨*variable*⟩;
`int` ⟨*variable1, variable2, ...*⟩;

等效于关键词 `integer`.

`real` ⟨*variable*⟩;
`real` ⟨*variable1, variable2, ...*⟩;

这里 ⟨*variable*⟩, ⟨*variable1*⟩, ⟨*variable2*⟩……都是 TeX 宏，将它们声明为实数变量，它们保存的值都带有小数点，小数部分至少有一个数字。

给实数变量宏加后缀的办法如同整数变量宏。

`coordinate` ⟨*variable*⟩;
`coordinate` ⟨*variable1, variable2, ...*⟩;

这里 ⟨*variable*⟩, ⟨*variable1*⟩, ⟨*variable2*⟩……都是 TeX 宏，将它们声明为坐标变量。可以将数值构成的坐标，如 `(2cm,3pt)`, `(25:3mm)`, 或 node 坐标系统的各种位置，如 `(my node.60)`, `(my node.east)`, 等等，赋予 ⟨*variable*⟩.

当使用坐标变量 ⟨*variable*⟩ 绘图时，注意给 ⟨*variable*⟩ 加上圆括号，因为宏 ⟨*variable*⟩ 保存的结果只是“尺寸—逗号—尺寸”，是没有圆括号的。

如果需要把某个坐标保存在一个宏中，应当先使用关键词 `coordinate` 声明坐标变量，否则 TikZ 不能识别坐标赋值，观察下面的例子：

00

```
\tikzmath{
 \x=(0,0);
}
\x
```

A

```
\begin{tikzpicture}
  \node(a)[below]{\LARGE $A$};
  \tikzmath{
    coordinate \c, \d;
    \c = (45:20mm);
    \d = (a.150);
  }
  \draw (\d)|-(\c);
\end{tikzpicture}
```

如果还调用了程序库 calc，那么在为坐标变量赋值时，还能使用 §13.5 中介绍的各种坐标计算句法，有的还可以省略符号 $.

d e f c

```
\tikzmath{
  coordinate \c, \d, \e, \f;
  \c = (-1,2)+(1,-1);
  \d = (\c)-(2,-1);
  \e = (\c -| \d);
  \f = ($(\c)!0.5!(\e)$);
}
\tikz \draw (\c) node {c} -- (\d) node {d} -- (\e) node {e} -- (\f)
↪  node {f};
```

可以用前面已声明的坐标点来定义新的坐标点：

```
\tikz{
  \tikzmath{
% 红色控制曲线的 7 个点
    coordinate \first;
    \first1=(0,0);
    \first2=(1.5,0.2);
    \first3=(1.8,0.8);
    \first4=(2,2);
    \first5=($(\first3)!2!(\first4)$);
    \first6=($(\first2)!2!(\first4)$);
    \first7=($(\first1)!2!(\first4)$);
  }
  \draw [->] (-0.3,0)--(4,0);
  \draw [->] (0,-0.3)--(0,4);
  \draw [red] (\first1)..controls(\first2)and(\first3)..(\first4)..controls(\first5)and(
  ↪  \first6)..(\first7);
}
```

当使用关键词 coordinate 声明坐标变量时，例如 coordinate \c;, TikZ 不仅会定义 TeX 宏 \c, 而且还会定义另外两个宏：\cx 和 \cy，前者保存 \c 的 x 分量，后者保存 \c 的 y 分量。

-56.90549pt, 28.45273,

```
\tikzmath{
  coordinate \c, \d, \e, \f;
  \c = (-1,2)+(1,-1);
  \d = (\c)-(2,-1);
  \e = (\c -| \d);
  \f = ($(\c)!0.5!(\e)$);
}
\ex,  % 输出坐标 \e 的 x 分量，带单位
\pgfmathparse{scalar(\fy)} \pgfmathresult,  % 输出坐标 \f 的 y 分量，不带单位
```

给坐标变量宏加后缀的办法如同整数变量宏，看下面的例子：

```
\tikzmath{
coordinate \c;
\c1 = (30:20pt);
\c2 = (210:20pt);
}
\tikz\draw (\cx1,\cy1) -- (\cx2,\cy1) -- (\cx2,\cy2) -- (\cx1,\cy2);
```

在这个例子中注意，坐标 \c1 的 x 分量保存在宏 \cx1 中，而不是 \c1x（没有这个宏）。可以这样比方：\c 是个坐标点集合，带后缀的 \c1 是用数字“1”来索引集合 \c；各个坐标点的 x 分量组成一个集合 \cx，带后缀的 \cx1 是用数字“1”来索引集合 \cx.

3.46411pt,2.0pt

```
\pgfkeys{/aa/bb/.initial={(30:4pt)}}
\tikzmath{
  coordinate \c;
  \c{test} = \pgfkeysvalueof{/aa/bb};
}
\c{test} \par
\tikz\draw [line width=\cy{test}](0,0) -- (1,1);
```

58.4 循环语句

for ⟨*variable*⟩ in {⟨*list*⟩}{⟨*expressions*⟩};

这个语句是 \foreach 命令的删减版。

关于这个语句需要注意：

- ⟨*list*⟩ 是用逗号分隔的列表，其中的条目会被数学引擎解析。如果其中某个条目本身含有逗号，应当用花括号把该条目括起来，例如，{mod(5,2)}.

  $x = 6, v = 64$

```
\tikzmath{
  int \x, \v;
  \v=1;
  for \x in {1,...,{random(3,10)}}{
    \v=\v*2;
  };
  print {$x=\x, v=\v$};
}
```

- 因为 ⟨*list*⟩ 中的每个条目都会被执行，故条目中不能使用 TikZ 的坐标。
- 目前关键词 for 之后只能使用一个变量。
- 可以在 ⟨*list*⟩ 中使用省略号“...”，但注意本程序库对省略号的处理没有 PGF 中的 \foreach 命令那么智能。

- 在循环体内所做的赋值的有效性超出循环体，故可在循环体外利用其中的赋值做计算。

$x_1 = 5, x_2 = 50, y = 2250$

```
\tikzmath{
  int \x, \y;
  \y = 0;
  for \x1 in {1,...,5}{
    for \x2 in {10,20,...,50}{
      \y = \y+\x1*\x2;
    };
  };
}
$x_1=\x1, x_2=\x2, y=\y$
```

58.5 条件语句

if ⟨*condition*⟩ then {⟨*if-non-zero-statements*⟩};

if ⟨*condition*⟩ then {⟨*if-non-zero-statements*⟩} else {⟨*if-zero-statements*⟩};

若 ⟨*condition*⟩ 的执行结果非 0 时，则执行 ⟨*if-non-zero-statements*⟩, 否则执行 ⟨*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}
```

58.6 声明函数

function ⟨*name*⟩ (⟨*arguments*⟩) {⟨*definition*⟩};

这个关键词的用法与 PGF 数学引擎中的选项 `declare function` 类似。

⟨*name*⟩ 是当前环境中没有用过的函数名称。

⟨*arguments*⟩ 是函数的变量参数列表，各个参数采用 TEX 宏的形式，之间用逗号分隔，目前不接受省略号（可变个数的参数形式）。如果函数没有变量参数，可以省去圆括号。目前，PGF 数学引擎中的“数组”不能作为这里的函数的参数。

⟨*definition*⟩ 应当是 `\tikzmath` 能够处理的语句，其中的变量不能超出 ⟨*arguments*⟩ 所列举的变量。

⟨*definition*⟩ 中可以使用关键词 `return` （见下文）。在 ⟨*definition*⟩ 中尽量不要在一个函数定义中定义新的函数，即不要在一个 `function` 的辖域内再使用 `function`.

$7 \times 30 = 210$

```
\tikzmath{
  function product(\x,\y) {
    return \x*\y;
  };
  int \i, \i, \k;
  \i = random(1,10);
  \j = random(20, 40);
  \k = product(\i, \j);
  print { $\i\times \j = \k$ };
}
```

`return` ⟨*expression*⟩;

这个关键词用于函数声明中的定义中，⟨*expression*⟩ 是函数最终应该返回的结果。

58.7 在命令 \tikzmath 的辖域内执行代码

要想在命令 `\tikzmath` 的辖域内执行某些代码，例如，输出代码，绘图代码等，需要相应的措施。如果是要输出代码，则使用关键词 `print`：

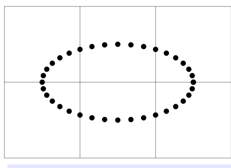
`print {`⟨*code*⟩`};`

⟨*code*⟩ 会被放入一个 TEX 分组内来处理，⟨*code*⟩ 可以是任何 TEX 代码，其中可以使用本程序库的赋值结果、运算结果。

$5^0 = 1$, $5^1 = 5$, $5^2 = 25$, $5^3 = 125$, $5^4 = 625$, $5^5 = 3125$, $5^6 = 15625$,

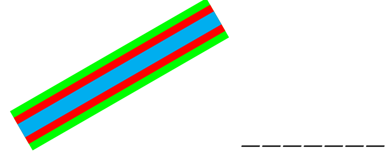
```
\tikzmath{
  int \x, \y, \z;
  \x = random(2, 5);
  for \y in {0,...,6}{
    \z = \x^\y;
    print {$\x^\y=\z$, };
  };
}
```

如果要执行某些绘图命令，则需要将执行命令用花括号括起来，且闭花括号之后要加上分号。花括号里的内容会被放入一个 TEX 分组内来处理。



```
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\tikzmath{
  coordinate \c;
  for \x in {0,10,...,360}{
    \c = (1.5cm, 1cm) + (\x:1cm and 0.5cm);
    { \fill (\c) circle [radius=1pt]; };
  };
}
\end{tikzpicture}
```

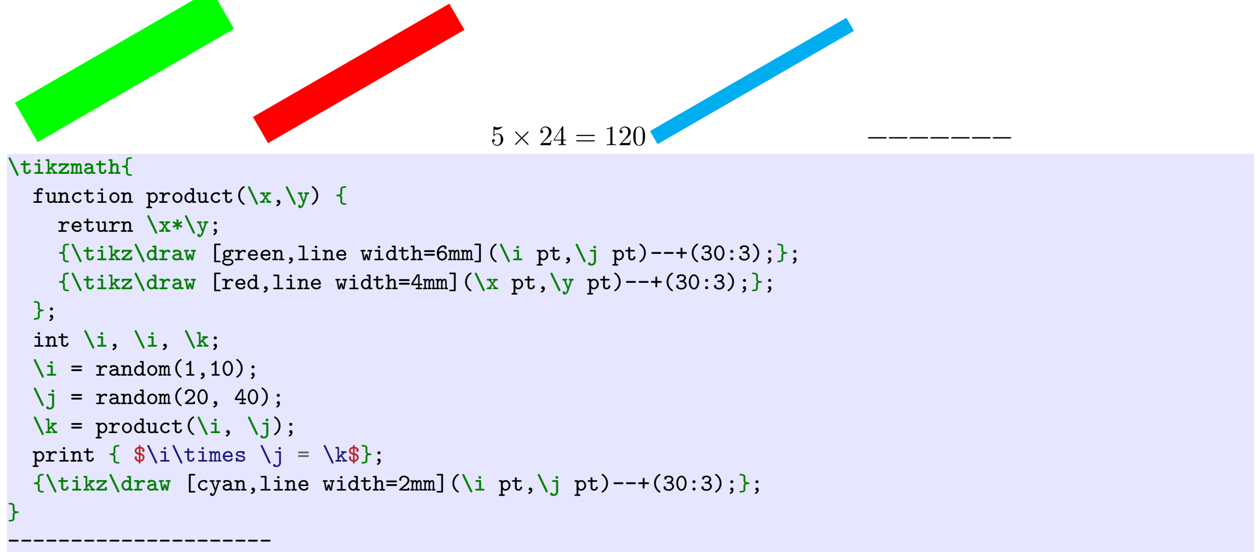
用这种方式，几乎可以在 `\tikzmath` 的辖域内的任何位置插入一个命令：



$2 \times 30 = 60$

```
\begin{tikzpicture}
\tikzmath{
  function product(\x,\y) {
    return \x*\y;
    {\draw [green,line width=6mm](\i pt,\j pt)--+(30:3);};
    {\draw [red,line width=4mm](\x pt,\y pt)--+(30:3);};
  };
  int \i, \i, \k;
  \i = random(1,10);
  \j = random(20, 40);
  \k = product(\i, \j);
  print { $\i\times \j = \k$};
  {\draw [cyan,line width=2mm](\i pt,\j pt)--+(30:3);};
}
\end{tikzpicture}---------------------\\
--------------------------------------\\
--------------------------------------
```

注意在上面的例子中，插入在 `\tikzmath` 辖域内的绘图命令使用同一个坐标系。其中关键词 `print` 输出一串数学模式下的符号，但是这一串符号位于整个图形的基线以下，图形基线默认为图形中的直线 $x = 0$. 对比下面的例子：



$5 \times 24 = 120$

```
\tikzmath{
  function product(\x,\y) {
    return \x*\y;
    {\tikz\draw [green,line width=6mm](\i pt,\j pt)--+(30:3);};
    {\tikz\draw [red,line width=4mm](\x pt,\y pt)--+(30:3);};
  };
  int \i, \i, \k;
  \i = random(1,10);
  \j = random(20, 40);
  \k = product(\i, \j);
  print { $\i\times \j = \k$};
  {\tikz\draw [cyan,line width=2mm](\i pt,\j pt)--+(30:3);};
}
---------------------
```

69 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. 67, `/tikz/shade` →P. 75, `/tikz/shading` →P. 75.

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);
```

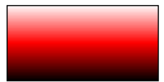
`/tikz/bottom color`=⟨*color*⟩ (no default)

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);
```

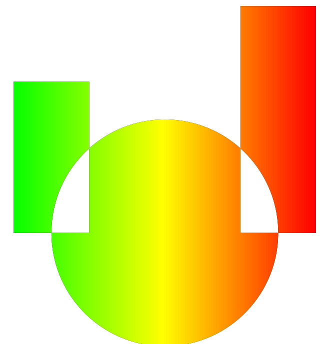
`/tikz/left color=`⟨*color*⟩ (no default)

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);
```

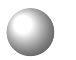
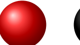
```
\tikz \fill [left color=green,right color=red,
    middle color=yellow]
  (0,0)rectangle(1,2)
  (3,0)rectangle(4,3);
```

Shading ball

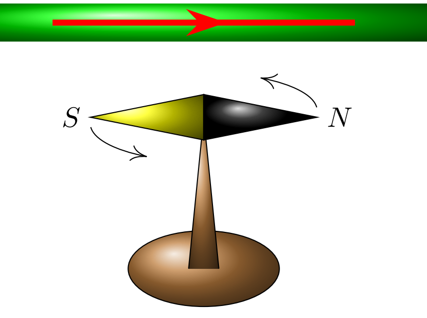
这个颜色渐变模式实现立体光影效果，渐变颜色默认用蓝色。

`/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)

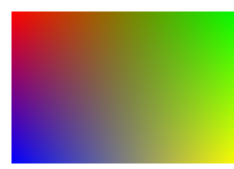
类似上一选项。

`/tikz/lower right`=⟨*color*⟩ (no default)

类似上一选项。

`/tikz/upper right`=⟨*color*⟩ (no default)

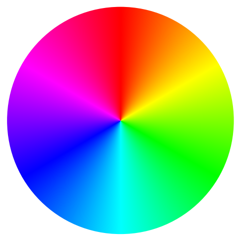
类似上一选项。



```
\tikz\shade[upper left=red,upper right=green,
    lower left=blue,lower right=yellow]
  (0,0) rectangle (3,2);
```

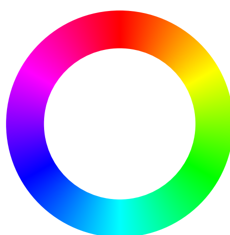
Shading `color wheel`

本模式生成一个色轮。



```
\tikz \shade[shading=color wheel] (0,0) circle (1.5);
```

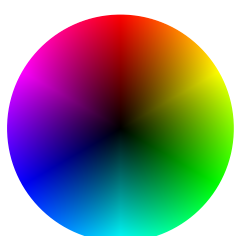
下面的例子中使用了 even odd rule 规则来填充颜色：



```
\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);
```

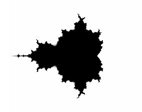
Shading `color wheel white center`

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。



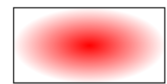
```
\tikz \shade[shading=Mandelbrot set] (0,0) rectangle (2,2);
```

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);
```

70 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。

70.1 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. 80 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。

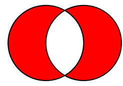
注意只能针对“路径”加阴影，不能针对整个 scope 图形画阴影。

70.2 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。

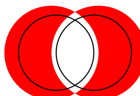


```
\tikz [even odd rule]
  \draw [general shadow={fill=red}]
    (0,0) circle (.5) (0.5,0) circle (.5);
```

还有下面的选项可以使用。

`/tikz/shadow scale`=⟨*factor*⟩ (no default, initially 1)

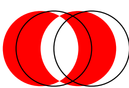
本选项设置阴影相对于原路径的尺寸比例，即使用放缩变换，放缩相对于阴影路径的边界盒子的中心。注意 PGF 用画布变换实现这个选项的作用。



```
\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。

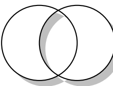
70.3 预定义的阴影

70.3.1 Drop Shadows

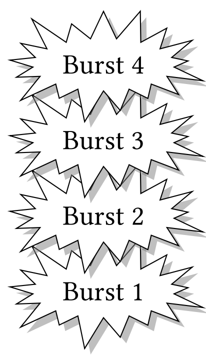
`/tikz/drop shadow`=⟨*shadow options*⟩ (default empty)

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：

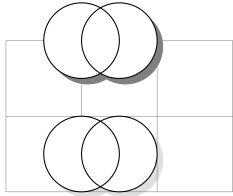
```
shadow scale=1, shadow xshift=.5ex, shadow yshift=-.5ex,
opacity=.5, fill=black!50, every shadow
```



```
\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。

70.3.2 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`.

Hello World!
Hello World!
Hello World!
Hello World!

```
\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}
```

70.4 针对圆形的阴影

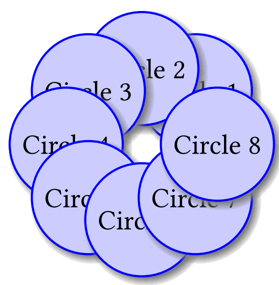
下面的阴影对圆形路径或圆形 node 的效果较好，如果用于其它形状的路径则可能会显得很奇怪。

`/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. 192 设置了填充色的透明度渐变。

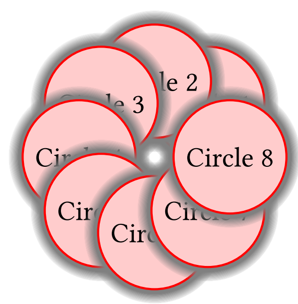



```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
    \node[circle,circular drop shadow,draw=blue,
        fill=blue!20,thick]
      at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

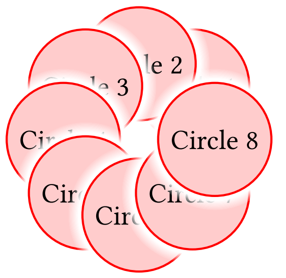
`/tikz/circular glow`=⟨*shadow options*⟩ (no default)

⟨*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：

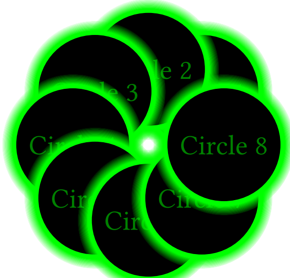
```
shadow scale=1.25, shadow xshift=0pt, shadow yshift=0pt,
fill=black, path fading={circle with fuzzy edge 15 percent},
every shadow,
```



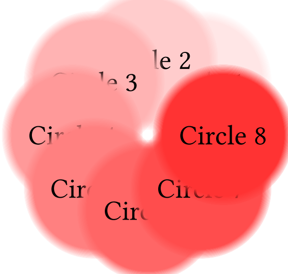

```
\begin{tikzpicture}
  \foreach \i in {1,...,8}
    \node[circle,circular glow,
        fill=red!20,draw=red,thick]
      at (\i*45:1) {Circle \i};
\end{tikzpicture}
```




```
\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}
```

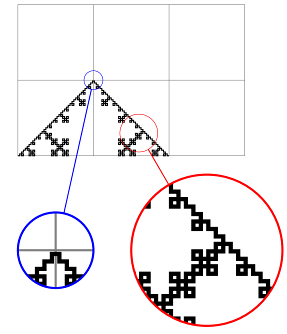
72 Spy 程序库：将图形的局部放大

TikZ Library spy

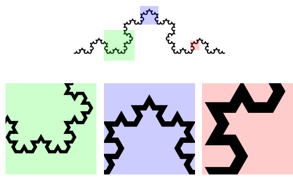
```
\usetikzlibrary{spy} % LaTeX and plain TeX
\usetikzlibrary[spy] % ConTeXt
```

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。

72.1 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}
```



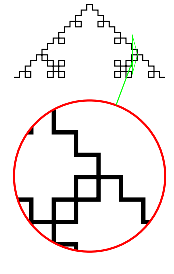
```
\begin{tikzpicture}[spy using overlays={size=12mm}]
  \draw [decoration=Koch snowflake]
    decorate {decorate{decorate{decorate{(0,0) -- (2,0)}}}};
  \spy [green,magnification=3] on (0.6,0.1) in node at (-0.3,-1);
  \spy [blue,magnification=5] on (1,0.5) in node at (1,-1);
  \spy [red,magnification=10] 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。

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。

72.2 spy scopes

/tikz/spy scope=⟨*options*⟩ (default empty)

这个选项用作绘图环境的选项，它会创建一个 spy 域。它的作用是：

- 出于技术上的考虑，重设某些图形状态参数。
- 告诉 TikZ，将 spy 域中的内容保存在某个特殊的内部盒子中。
- 允许在 spy 域中使用命令 \spy.
- 在 spy 域结尾处，创建由命令 \spy 规定的两个 node.
- 选项 ⟨*options*⟩ 会被保存到某个内部样式中，⟨*options*⟩ 会被传递给每个 \spy 命令。
- 在 ⟨*options*⟩ 可以使用以下选项：

/tikz/size=⟨*dimension*⟩ (no default)

这是 /pgf/minimum size→P. 708 的简写，针对 spy-in node.

/tikz/height=⟨*dimension*⟩ (no default)

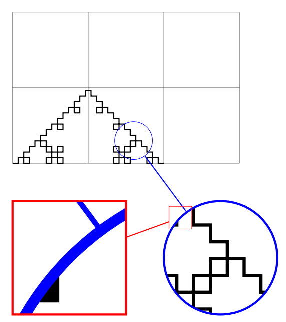
这是 /pgf/minimum height→P. 708 的简写，针对 spy-in node.

/tikz/width=⟨*dimension*⟩ (no default)

这是 /pgf/minimum width→P. 707 的简写，针对 spy-in node.

命令 \spy 创建 spy-in node 和 the spy-on node，这两个 node 内部盛放内容的盒子的尺寸被设置为 0pt，所以需要设置它们的最小尺寸来保持其轮廓外形。这两个 node 的 inner sep 和 outer sep 都被设置为 0pt.

可以把 spy 域套嵌使用，即给套嵌的绘图环境分别加上选项 spy scope. 对于套嵌的 spy 域，先完成内层的 spy 域，后完成外层的 spy 域。利用套嵌的 spy 域可以制作“放大图”的“放大图”。



```
\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);
  \end{scope}
  \spy on (zoom.north west) in node [right] at (0,-1.25);
\end{tikzpicture}
```

72.3 spy scopes

\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。注意这个命令不是 TikZ 命令 \path 的特殊情况，本命令有自己的解析器。注意本命令的句法格式。

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.

创建 spy-in node 和 spy-on node 的过程如下：

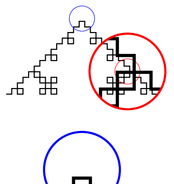
1. 先开启一个域，在这个域中创建这两个 node. 首先环境选项 `spy scope={`⟨*options 1*⟩`}` 指定的 ⟨*options 1*⟩ 被调入，然后命令选项 `\spy[`⟨*options 2*⟩`]` 指定的 ⟨*options 2*⟩ 被调入，后者可能改写前者。
2. 先创建 spy-on node，故对 spy-on node 的设置可以在环境选项 `spy scope={`⟨*options 1*⟩`}` 和命令选项 `\spy[`⟨*options 2*⟩`]` 中作出。
3. 再创建 spy-in node，故 spy-in node 可能会遮挡 spy-on node(如果二者重叠的话)。创建 spy-in node 时会检查并引入下面的样式设置：

`/tikz/every spy in node` (style, no value)

这个样式用于每个 spy-in node.

然后调入 ⟨*node options*⟩ 中的设置。

在默认下，spy-in node 的中心与 spy-on node 的中心重合，即都位于 ⟨*coordinate*⟩ 处。若要修改 spy-in node 的位置可以使用操作 `at(`⟨*at coordinate*⟩`)` 或选项 `at={(`⟨*at coordinate*⟩`)}`.



```
\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}
```

spy-in node 的内容是用画布变换以及剪切作用得到的局部放大图，画布变换用下面的选项来设置：

`/tikz/lens=`⟨*options*⟩ (no default)

⟨*options*⟩ 中应该是变换选项，例如 `scale`，`rotate`，这些变换选项被作为画布变换，其效果显现在 spy-on node 内的图形中。

`/tikz/magnification=`⟨*number*⟩ (no default)

这个选项用来设置放大倍数，等效于 `lens={scale=`⟨*number*⟩`}`.

在默认下，spy-in node 的名称是 `tikzspyinnode`，也可以为 spy-in node 另行命名。可以用 spy-in node 的名称来引用它。注意，在 spy scope 之后（即创建 spy-in node 之后）才可以引用 spy-in node 的名称。

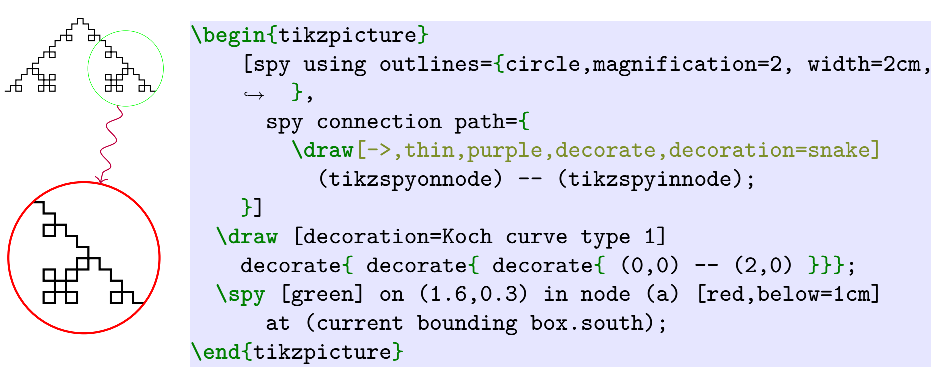
```
\begin{tikzpicture}
  \begin{scope}[spy using outlines={circle,
         magnification=3, size=2cm, connect spies}]
    \draw [decoration=Koch curve type 1]
      decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};
    \spy [red] on (1.6,0.3) in node (a) [left] at (3.5,-1.25);
    \spy [blue, size=1cm]
      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}
```

4. 当 spy-in node 和 spy-on node 被创建后，再执行下面的选项来连接它们：

`/tikz/spy connection path=`⟨*code*⟩ (no default, initially empty)

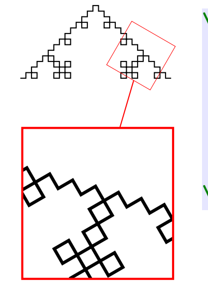
这里 ⟨*code*⟩ 是绘图命令或者其它代码。由于在 spy-in node 和 spy-on node 被创建后再执行这个选项，所以在 ⟨*code*⟩ 中可以使用 spy-in node 和 spy-on node 的名称。例如

```
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}
```

注意在选项 `lens={`⟨*options*⟩`}` 中引入的变换是针对原图的，不是针对 spy-in node 和 spy-on node 的，但如果 ⟨*options*⟩ 中含有旋转变换，例如 `rotate=30`，为了更清楚地标示旋转作用，默认会对 spy-on node 使用一个旋转变换 `rotate=-30`，这样，如果 spy-in node 和 spy-on node 都是矩形并且没有其它的旋转，那么二者的轮廓线条就会出现 30° 的旋转差别。观察下图：



```
\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}
```

`/tikz/every spy on node` (style, no value)

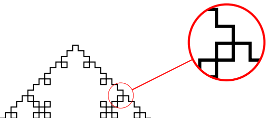
这个样式用于每个 spy-on node. spy-on node 的默认名称是 `tikzspyonnode`，如果需要给 spy-on node 另行命名，可以在 `every spy on node` 中使用 `name` 选项。注意，在 spy scope 之后（即创建 spy-on node 之后）才可以引用 spy-on node 的名称。

72.4 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`.

`/tikz/spy using outlines=`⟨*options*⟩ (default empty)

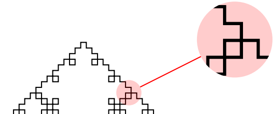
这个选项使得 spy-in node 用 thick 线宽画出，但不填充；使得 spy-on node 用 very thin 线宽画出，但不填充。



```
\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)

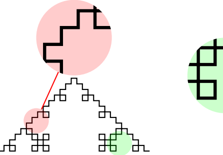
这个选项使得 spy-in node 和 spy-on node 被填充，填充色的不透明度是 20%，但不画出边界线条。



```
\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)

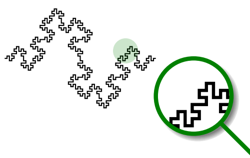
这个选项会在 spy-in node 和 spy-on node 之间画线。



```
\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}
```

72.5 例子

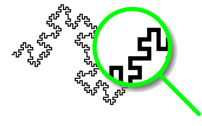
下面的例子中把 spy-in node 的形状设为 `magnifying glass`,这个 node 形状需要调用 shapes.symbols 程序库。



```
\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}
```

79 数据可视化简介

数据可视化（data visualization）是关于数据点（data points）的处理的。PGF 中的数据可视化系统十分强大，但它的底层不易用。

79.1 数据点

例如做一个关于汽车的试验，考虑的物理量有时间、速度大小、运动方向（一个角度值）、加速度这 4 个，那么试验数据就需要用到 4 维空间中的点来描述，一个数据点有 4 个分量 (数据)。

渲染管线（rendering pipeline）会从数据点中抽取一个或数个需要的数据用于某个可视化过程，不同的可视化过程可能会从同一个数据点中抽取不同的数据。

每当使用宏 `\pgfdatapoint` 后，就检查当前的、以 `/data point/` 为前缀的各个 key 的值，用这些值构成一个数据点。

79.2 可视化管线（visualization Pipeline）

所有的数据点会作为一个很长的数据点流（a long stream of complex data points）提供给可视化管线，可视化管线是对数据的一些列处理，处理过程中会多次传递数据。第一次传递叫作“survey phase”，会确定数据的最大值和最小值，从而确定绘图区域。最主要的传递过程叫作“visualization phase”，会把数据转换成可见的线条或点。

80 数据可视化的基本概念

80.1 Overview

本节介绍如何在 Tikz 中实现数据可视化。首先载入 datavisualization 库。

TikZ Library datavisualization

```
\usetikzlibrary{datavisualization} % LaTeX and plain TeX
\usetikzlibrary[datavisualization] % ConTeXt
```

这个库提供的命令 \datavisualization 创建数据可视化图形。这个程序库提供了一些基本的可视化样式，还有其它程序库也提供了一些特别样式。这个库的文件《pgfmoduledatavisualization.code.tex》调用了 oo，shapes 模块以及 fpu 库：

```
\usepgfmodule{oo,shapes}
\usepgflibrary{fpu}
```

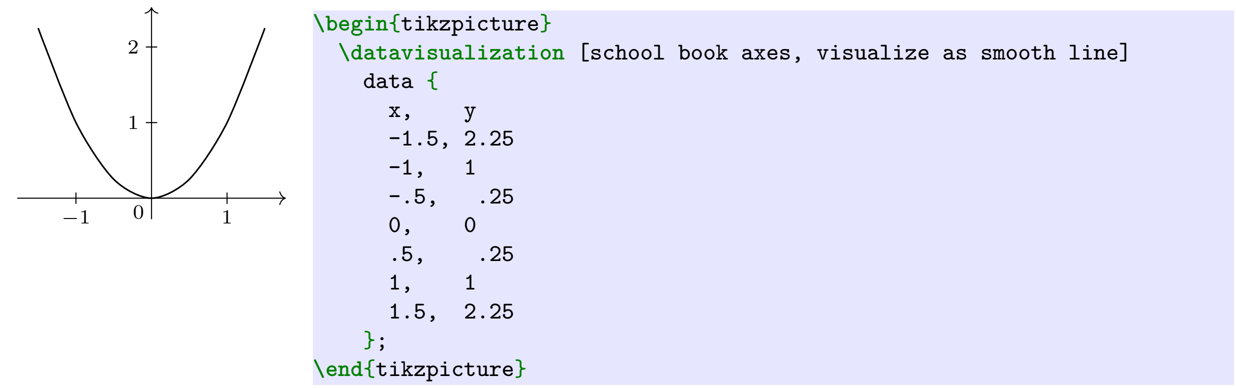
在创建数据可视化图形时至少要做到以下 3 方面：

1. 可视化图形有不同的“风格、类型”，例如“school book plot”，“scientific 2d plot”，“scientific spherical plot”etc, 可视化图形的类型是必须要指定的，通过给命令 \datavisualization 带上选项来确定可视化的类型。
2. 用 data 指令提供数据点。
3. 用其它选项修改图形外观细节，例如坐标轴的刻度线，网格，标签，图例，颜色，字体等。对于这些细节，在初始之下可视化引擎使用 TikZ 风格。

80.2 数据点与数据格式

数据点作为 data 指令的参数被提供出来，数据点可以是内置的（inline, 即在文档中手工编辑数据），也可以放在一个外部文件中，将文件名提供给 data 指令。

数据点按照一定的格式编辑出来，格式有数种，并且可以自己定义新的格式。常用的格式是逗号分隔式（comma separated values format），一个数据点单独占一行，属于一个数据点的各个分量之间用逗号分隔，例如：

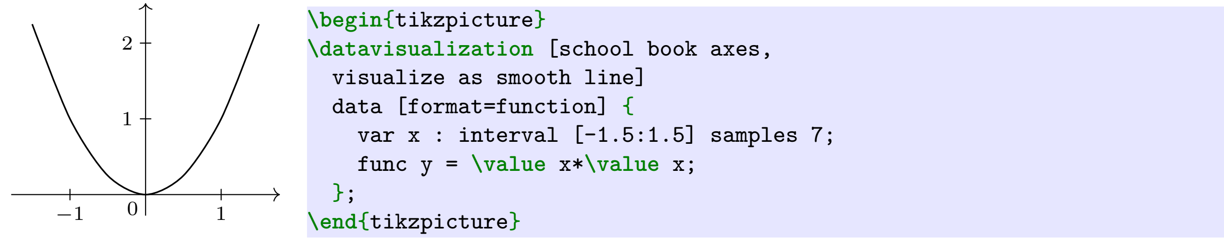



```
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as smooth line]
    data {
      x,   y
      -1.5, 2.25
      -1,   1
      -.5,  .25
      0,    0
      .5,   .25
      1,    1
      1.5,  2.25
    };
\end{tikzpicture}
```

必须注意，在逗号分隔式的第一行中，即声明变量符号“x，y”的行中，逗号之前不能有空格。手册中用单词 attribute 来指称“变量符号”。

再一种常用数据格式是键值式（key-value format），用于函数式绘图。

首先调用程序库 `datavisualization.formats.functions`，然后使用这个程序库提供的 `function` 功能，例如：



```
\begin{tikzpicture}
\datavisualization [school book axes,
  visualize as smooth line]
  data [format=function] {
    var x : interval [-1.5:1.5] samples 7;
    func y = \value x*\value x;
  };
\end{tikzpicture}
```

TEX 提供的数值是最多包含 13 bits 的整数和 16 bits 的小数，数据可视化所需的计算精度和数值范围要超出 TEX 的范围，因此数据可视化引擎在处理数值时并不使用标准的 PGF 的数值表示方式、TEX 尺寸，也不使用标准的解析器，而是使用 fpu 程序库。

限于目前 fpu 程序库的能力，在编辑数据点时要注意以下问题：

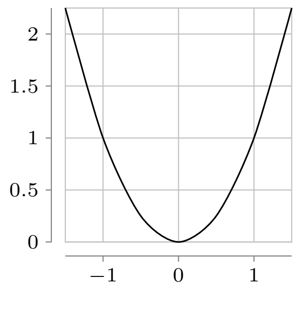
1. 在数据点中可以写下像 100000000000000 或 0.00000000001 这样的数值。
2. 目前 fpu 程序库不支持高级解析，不能把算式 `2+3` 作为数据点的数据，否则导致错误。
3. 如果数据点中的某个数据放在圆括号里，则该数据会用通常的解析器来解析。
   - 100000 是可接受的。
   - `2+3` 导致错误。
   - `(2+3)` 产生数值 `5`.
   - `(100000)` 导致错误，因为 `(100000)` 超出通常的解析器的解析范围。

所以主要的原则就是：算式放入圆括号中、大数值（或小数部分很长的数值）不放入圆括号中。将来这一点可能会改进。

80.3 轴，刻度线，网格

多数图形有 2 个或 3 个坐标轴，可视化引擎还会使用极坐标系。利用可视化引擎可以细致地调整刻度线、网格线的外观，它们有预定义的处理方式，除非必要，不必手动调整。水平的轴叫做 x-axis, 竖直的轴叫做 y-axis, 有时还有倾斜的 z-axis. 通常情况下，“轴”表现为一个带有刻度线的线段，但这只是“轴”的外在表现，“轴”实际上是一套数据处理过程。

```
\begin{tikzpicture}
  \datavisualization [
    scientific axes,
    x axis={length=3cm, ticks=few},
    visualize as smooth line
  ]
  data [format=function] {
    var x : interval [-1.5:1.5] samples 7;
    func y = \value x*\value x;
  };
\end{tikzpicture}
```

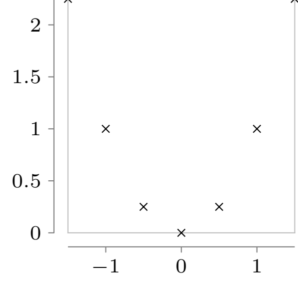



```
\begin{tikzpicture}
  \datavisualization [
    scientific axes=clean,
    x axis={length=3cm, ticks=few},
    all axes={grid},
    visualize as smooth line
  ]
  data [format=function] {
    var x : interval [-1.5:1.5] samples 7;
    func y = \value x*\value x;
  };
\end{tikzpicture}
```

80.4 显像器（visualizer）

最常用的显像器是 `line visualizer`, 会把数据转为线条图形。`scatter plot visualizer` 会把数据变成散点图。还有其它的显像器，例如用于绘制统计图形的显像器。

```
\begin{tikzpicture}
  \datavisualization [
    scientific axes=clean,
    x axis={length=3cm, ticks=few},
    visualize as smooth line
  ]
  data [format=function] {
    var x : interval [-1.5:1.5] samples 7;
    func y = \value x*\value x;
  };
\end{tikzpicture}
```



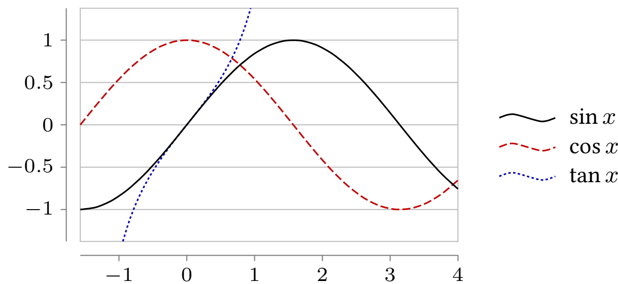

```
\begin{tikzpicture}
  \datavisualization [
    scientific axes=clean,
    x axis={length=3cm, ticks=few},
    visualize as scatter
  ]
  data [format=function] {
    var x : interval [-1.5:1.5] samples 7;
    func y = \value x*\value x;
  };
\end{tikzpicture}
```

80.5 样式表和图例

在一个可视化图形中可能有多个线条，不同线条由不同显像器生成，每个线条（显像器）都有自己特殊的样式以相互区别。当然可以手工为每个线条设置样式，但如果线条太多就显得比较麻烦而且容易混淆。为此，可视化系统提供了“样式表”（style sheets）的概念。

样式表是由数个样式组成的列表，第 i 个线条（显像器）使用样式表中的第 i 个样式。如果线条太多，超出样式表中样式的个数，则过多的线条样式使用默认样式。

还可以添加图例或标签来为图形做注解。




```
\begin{tikzpicture}[baseline]
  \datavisualization [ scientific axes=clean,
                       y axis=grid,
                       visualize as smooth line/.list={sin,cos,tan},
                       style sheet=strong colors,
                       style sheet=vary dashing,
                       sin={label in legend={text=$\sin x$}},
                       cos={label in legend={text=$\cos x$}},
                       tan={label in legend={text=$\tan x$}},
                       data/format=function ]
    data [set=sin] {
      var x : interval [-0.5*pi:4];
      func y = sin(\value x r);
    }
    data [set=cos] {
      var x : interval [-0.5*pi:4];
      func y = cos(\value x r);
    }
    data [set=tan] {
      var x : interval [-0.3*pi:.3*pi];
      func y = tan(\value x r);
    };
\end{tikzpicture}
```

80.6 用法

在 {tikzpicture} 环境或 {scope} 环境中可以使用命令 \datavisualization 绘制数据可视化图形，把绘图环境套嵌起来，可以在一个 {tikzpicture} 环境中绘制多个可视化图形。

\datavisualization[⟨*data visualization options*⟩]⟨*data specification*⟩;

这个命令只能用在 {tikzpicture} 环境或 {scope} 环境中。

⟨*data visualization options*⟩ 用于设置图形的外观，其中的选项会被加上 key 路径前缀 /tikz/data visualization/ 来处理，这意味着一般的 TikZ 选项，例如 draw，red，thin 等不能用在这里。你至少要写出两个选项，第一，指定坐标轴样式，例如 school book axes 或 scientific axes; 第二，指定显像器，例如选项 visualize as line 画出直线形。

与命令 \path 类似，⟨*data visualization options*⟩ 不必紧跟在命令之后，可以放在语句的其它位置。

⟨*data specification*⟩ 提供绘图数据。

命令以分号结束。

\datavisualization ... data[⟨*options*⟩]{⟨*inline data*⟩}...;

其中的 data 指令提供绘图数据，在一个 \datavisualization 内可以多次使用 data 指令以提供多组数据。

如果没有 ⟨*inline data*⟩, 那么就应该在 ⟨*options*⟩ 中提供包含绘图数据的外部文件名称，此时 ⟨*inline data*⟩ 外围的花括号也必须去掉。

如果有 ⟨*inline data*⟩, 须用花括号将其括起来，此时程序不会读取外部文件，即使在 ⟨*options*⟩ 中提供了外部文件名称。

⟨*options*⟩ 中的选项会被冠以前缀 `/pgf/data/` 来执行，以下选项可用：

`/pgf/data/read from file=`⟨*file*⟩ (no default, initially empty)

这个选项用来调用外部的数据文件。注意如果调用外部文件就不要使用 ⟨*inline data*⟩ 及其外围的花括号。

```
\datavisualization ...
  data [read from file=file1.csv]
  data [read from file=file2.csv];
```

如果 `read from file` 的值留空，则应提供 ⟨*inline data*⟩ 并用花括号将其括起来。

```
\datavisualization ...
  data {
    x, y
    1, 2
    2, 3
  };
```

`/pgf/data/format=`⟨*format*⟩ (no default, initially `table`)

这个选项决定按照什么格式规则来读取数据，读取规则应当与数据格式一致。默认的格式是 `table`，即“逗号分隔式”（comma-separated values）。数据格式的第一行是变量名（attribute），不同变量名用逗号分隔；后续各行的每一行都是一个数据点，一个数据点内的各个数据用逗号分隔；每个数据都与第一行相应的变量名对应，可以看作是是该变量的一个值。

为多个 `data` 指令统一设置选项 当 `data` 指令有数个时，为它们统一设置某些选项可能会比较合适。

`/tikz/every data` (style, no value)

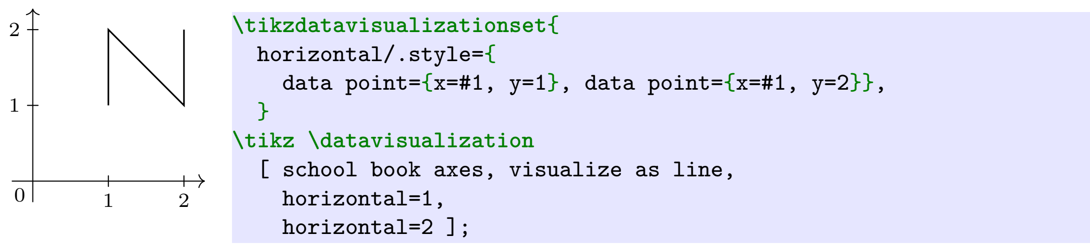
这个选项设置的样式会赋予每个 `data` 指令。

另外，每当前缀路径为 `/tikz/data visualization/data` 的选项被使用时，该选项的前缀路径也会被映射为 `/pgf/data`，这意味着，可以把以 `/pgf/data` 为前缀路径的选项，例如 `data/format=table` 用在命令 `\datavisualization` 的选项中，通过该命令，这个选项会传递给命令之内的各个 `data` 指令。

关于数据格式中的换行 数据格式中，每个数据点占一行，这当然用到“回车”符号来换行，但是 TeX 会取消回车的换行作用，仅把回车当作空格。因此在处理 inline 数据点时，可视化系统暂时（局部地）重定义了回车符号，使之具有换行作用。但是如果在可视化系统处理 inline 数据点之前 TeX 就处理了 inline 数据点，那么就会取消回车的换行作用，可视化系统无法正确读取数据点。因此，如果某个宏不能处理“脆弱”（fragile）代码，就不能把 `\datavisualization` 命令语句作为这个宏的参数，例如，当可视化命令用在 beamer 文类的帧（frame）内时，需要添加 `fragile` 选项。如果可视化命令调用外部数据文件，就不会出现这个问题。

`/tikz/data visualization/data point=`⟨*options*⟩ (no default)

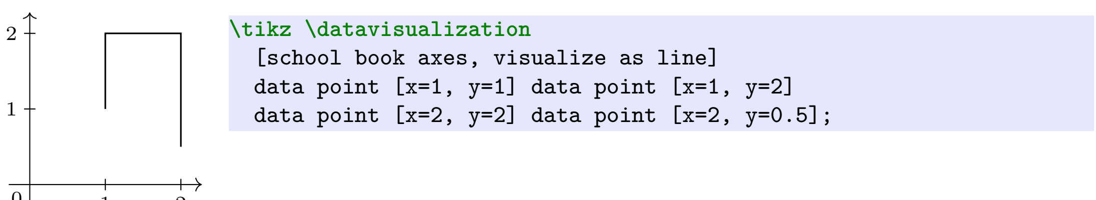
这个选项的值使用 key-value 的形式，例如 `data point={x=1, y=1}`, 这样定义一个（只定义一个）数据点，它可以用在样式（style）中。




```
\tikzdatavisualizationset{
  horizontal/.style={
    data point={x=#1, y=1}, data point={x=#1, y=2}},
  }
\tikz \datavisualization
  [ school book axes, visualize as line,
    horizontal=1,
    horizontal=2 ];
```

`\datavisualization ... data point[`⟨*options*⟩`]...;`

这个命令与作为选项（key）的 `data point={}` 的作用一样，用于定义一个数据点。⟨*options*⟩ 中的选项会被冠以前缀路径 `/data point` 来执行，因此其中的选项必须使用 key-value 的赋值形式，并且只能定义一个数据点。




```
\tikz \datavisualization
  [school book axes, visualize as line]
  data point [x=1, y=1] data point [x=1, y=2]
  data point [x=2, y=2] data point [x=2, y=0.5];
```

`\datavisualization ... data group[`⟨*options*⟩`]{`⟨*name*⟩`}={`⟨*data specifications*⟩`}...;`

⟨*data specifications*⟩ 是用 `data` 指令或 `data point` 指令定义的一组数据点，指令 `data group` 将这组数据点看作一个整体，将其命名为 ⟨*name*⟩, 可以用 ⟨*name*⟩ 引用这组数据点，并且 ⟨*name*⟩ 是全局（全文档）有效的，可以在命令之外引用。如果先前已经存在一个名称同样为 ⟨*name*⟩ 的数据点组，那么旧的数据点组会被“忘记”，⟨*name*⟩ 指向新的数据点组。数据点组并不传递给渲染管线，而是被立即解析，这里的 ⟨*options*⟩ 会与解析出来的 ⟨*data specifications*⟩ 一起保存起来。

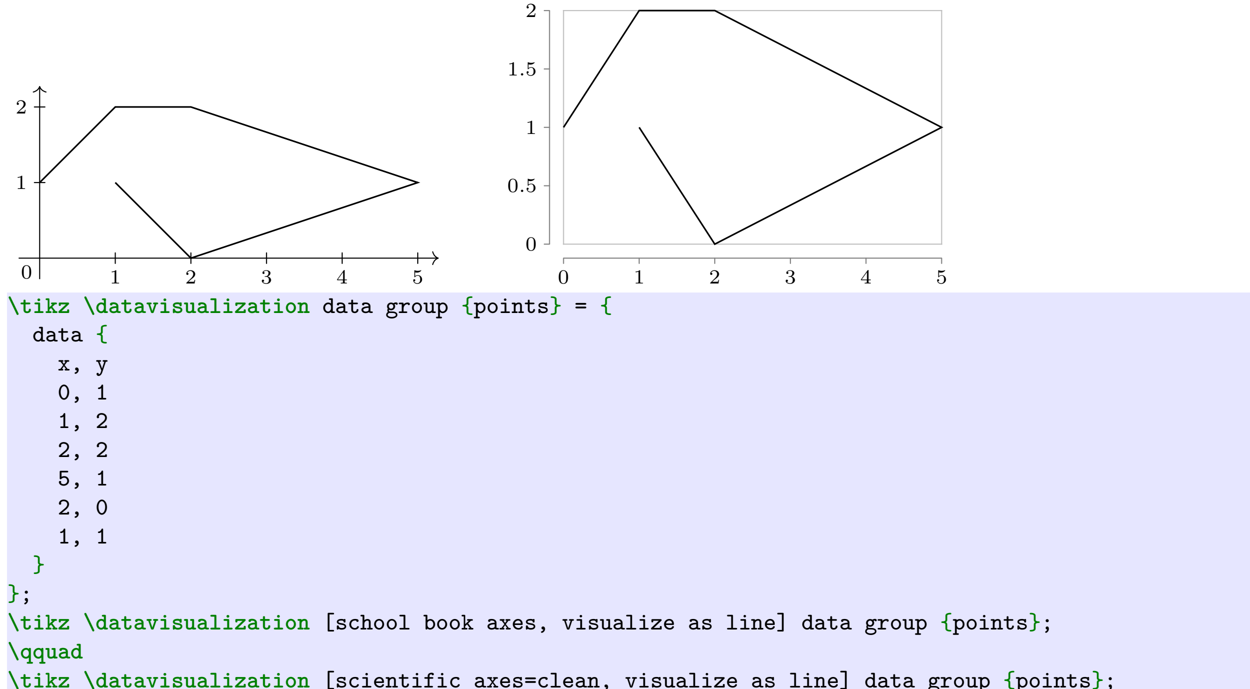
`\datavisualization ... data group[`⟨*options*⟩`]{`⟨*name*⟩`}+={`⟨*data specifications*⟩`}...;`

这里 ⟨*name*⟩ 是已有的数据点组名称，“+”的意思是把 ⟨*data specifications*⟩ 添加到原来的 ⟨*name*⟩ 中，从而扩充之。

`\datavisualization ... data group[`⟨*options*⟩`]{`⟨*name*⟩`}...;`

这里 ⟨*name*⟩ 之后没有“+”或“=”, ⟨*name*⟩ 是已有的数据点组名称，命令会在当前位置插入名称为 ⟨*name*⟩ 的数据点组，与 ⟨*name*⟩ 相关联的 ⟨*options*⟩ 一同被执行。

下面是个例子，先创建一个数据点组，然后引用之：




```
\tikz \datavisualization data group {points} = {
  data {
    x, y
    0, 1
    1, 2
    2, 2
    5, 1
    2, 0
    1, 1
  }
};
\tikz \datavisualization [school book axes, visualize as line] data group {points};
\qquad
\tikz \datavisualization [scientific axes=clean, visualize as line] data group {points};
```

`\datavisualization ... scope[⟨options⟩]{⟨data specification⟩}...;`

本命令中的 scope 指令是可以套嵌使用的。⟨*data specification*⟩ 是使用 data 或 scope 指令创建的数据点组；⟨*options*⟩ 中的选项会被冠以前缀 /pgf/data/ 来执行，并且只对 ⟨*data specification*⟩ 中的数据点组有效。

`\datavisualization ... info[⟨options⟩]{⟨code⟩}...;`

指令 info 所辖的 ⟨*code*⟩ 是通常的 TikZ 绘图代码，在数据可视化过程结束（但命令未结束）时执行 ⟨*code*⟩, 而这里 ⟨*options*⟩ 中的选项会被冠以 /tikz/ 来执行，也就是说，这些选项是通常的 TikZ 选项。这个句法的作用是把 TikZ 绘图代码引入到可视化命令之内，在绘图代码中可以使用可视化图形中的坐标。可视化图形有自己的坐标系，该坐标系中的点只能在可视化命令之内引用，在可视化命令之外不能引用。

Coordinate system visualization

坐标系统 visualization 中的点用如下句法指定：

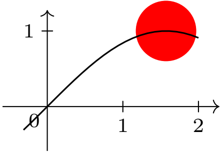
(visualization cs:⟨*list of attribute-value pairs*⟩)

对于 ⟨*list of attribute-value pairs*⟩ 中的每个“变量名 = 值”（⟨*attribute*⟩=⟨*value*⟩），都会被转换成 /data point/⟨*attribute*⟩=⟨*value*⟩. 注意，一定要写出正确的 ⟨*attribute*⟩, 否则该点可能会被忽略为 (0,0).

```
\begin{tikzpicture}[baseline]
\datavisualization [ school book axes, visualize as line ]
  data [format=function] {
    var x : interval [-0.1*pi:2];
    func y = sin(\value x r);
  }
  info {
    \draw [red] (visualization cs: x={(.5*pi)}, y=1)
      circle [radius=1pt]
      node [above,font=\footnotesize] {extremal point};
  };
\end{tikzpicture}
```

`\datavisualization ... info'[⟨options⟩]{⟨code⟩}...;`

这个选项与 `info` 类似，只是在可视化过程开始时执行 ⟨*code*⟩, 所画出的图形会被可视化图形遮挡，所以可以用这个选项画出可视化图形的背景。




```
\begin{tikzpicture}[baseline]
\datavisualization [ school book axes, visualize as line ]
  data [format=function] {
    var x : interval [-0.1*pi:2];
    func y = sin(\value x r);
  }
  info' {
    \fill [red] (visualization cs: x={(.5*pi)}, y=1)
      circle [radius=4mm];
  };
\end{tikzpicture}
```

Predefined node `data visualization bounding box`

这个预定义的矩形 node 保存的是“当前”可视化图形（包括坐标轴、图例、标签等）的边界盒子（bounding box），在 `info` 所辖的 ⟨*code*⟩ 中可能会比较有用。

Predefined node `data bounding box`

这个预定义的矩形 node 保存的是恰好能够包含当前可视化图形中的“数据点”的边界盒子，这当然就不包括坐标轴、刻度线、图例、网格、标签等等。

80.7 在数据可视化过程中执行用户自定义的代码

下面的选项用在命令 `\datavisualization` 中，可以在可视化过程的不同阶段执行相应的自定义代码。

`/tikz/data visualization/before survey=⟨code⟩` (no default)
`/tikz/data visualization/at start survey=⟨code⟩` (no default)
`/tikz/data visualization/at end survey=⟨code⟩` (no default)
`/tikz/data visualization/after survey=⟨code⟩` (no default)

`/tikz/data visualization/before visualization=`⟨*code*⟩ (no default)

`/tikz/data visualization/at start visualization=`⟨*code*⟩ (no default)

`/tikz/data visualization/at end visualization=`⟨*code*⟩ (no default)

`/tikz/data visualization/after visualization=`⟨*code*⟩ (no default)

80.8 创建新对象

81 用于数据可视化的数据格式

81.1 Overview

81.2 简介

编辑数据点时要按照一定的格式，即数据格式（data format），数据按照格式编辑成一个矩阵，或者说一个数据表格。常用的是逗号分隔式（csv format，即 comma separated values），key–value format, 还有比较复杂的 pdb-format. 你可以指定某个格式解析器（用选项 /pgf/data/format），否则使用默认格式解析器（默认格式是 table，即逗号分隔式）。

举例来说，假设给出如下数据点：

```
x, y, z
0, 0, 0
1, 1, 0
1, 1, 0.5
0, 1, 0.5
```

这是逗号分隔式数据点，对应的格式解析器是 `table`. 数据表的第一行声明标识符（attribute, 相当于变量名），标识符用逗号分隔。一个数据点占用一行，一个点内的各个数据用逗号分隔。数据用于给标识符赋值，例如，对于最后一行，数据 `0` 会被转换成 `/data point/x=0`, 数据 `1` 会被转换成 `/data point/y=1`, 数据 `0.5` 会被转换成 `/data point/z=0.5`.

必须注意，在声明标识符的第一行中，变量名与逗号之间不能有空格。否则会导致错误，错误信息是：

! Package PGF Math Error: Sorry, an internal routine of the floating point unit got an ill-formatted floating point number ‘’. The unreadable part was near ”

81.3 内置格式

系统预设了数种数据格式。

如果不指定一种数据格式，系统就默认数据格式为 `table`.

Format `table`

即逗号分隔式。第一个非空行（头行，headline）用于声明标识符（attribute），标识符用来储存数据。这在前文已经解释过。再看一个例子：

```
angle, radius
0, 1
45, 2
90, 3
135, 4
```

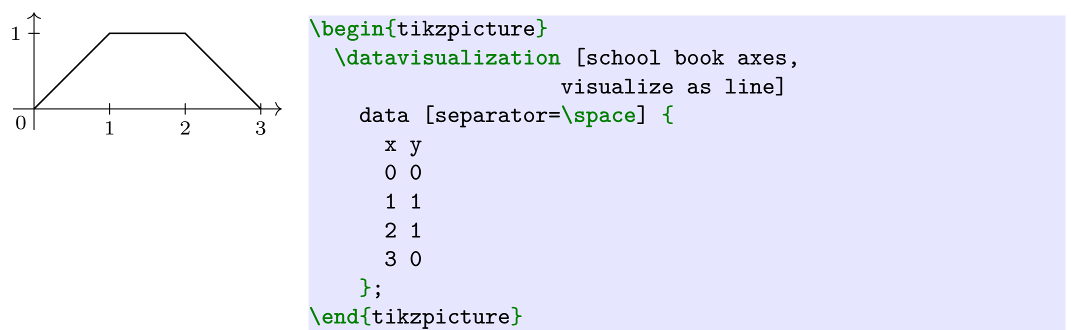
这个表格的首行设置两个标识符（即 key）`/data point/angle` 和 `/data point/radius`, 每行的第一个数据赋予 `angle`, 第二个赋予 `radius`. 如果一行中的数据个数少于 headline 中声明的标识符（attribute）的个数，那么最后一个标识符（attribute, 变量）的值就是空的。如果一行中的数据个数多于 headline 中声明的标识符（attribute）的个数，那么多余的数据或被保存到 `/data point/attribute` ⟨*column number*⟩ 中，⟨*column number*⟩ 是数据所在列的列号，数据表的第一列的列号是 1, 其余列的列号类推。

注意，如果不在 headline 中声明标识符，那么默认的标识符使用 `x`，`y`，`z` 这三个符号。如果使用其它符号，则需要对“轴”做相应设置，否则不能顺利绘图。

对于 `table` 格式，可以自定义分隔数据的符号（分列符）:

`/pgf/data/separator`=⟨*character*⟩ (no default, initially `,`)

这个选项初始值是逗号，所设置的符号用于分隔一行内的不同数据。如果要把分列符设为空格, 可以把该选项的值留空或者写下 `separator=\space`. 注意, 如果先将分列符设为 (例如) 分号 `separator=;`, 之后又想改回逗号, 则必须把逗号用花括号括起来 `separator={,}`.



```
\begin{tikzpicture}
  \datavisualization [school book axes,
                      visualize as line]
    data [separator=\space] {
      x y
      0 0
      1 1
      2 1
      3 0
    };
\end{tikzpicture}
```

注意当使用这个选项时，就默认为 `table` 格式。

`/pgf/data/headline`=⟨*headline*⟩ (no default)

这个选项设置数据格式的首行（headline），然后在数据表格中就可以省去设置标识符的首行。

```
\begin{tikzpicture}
  \datavisualization [school book axes,
                      visualize as line]
    data [headline={x, y}] {
      0, 0
      1, 1
      2, 1
      3, 0
    };
\end{tikzpicture}
```

注意当使用这个选项时，就默认认为 table 格式。

Format named

这个格式中，一个数据点仍然占一行，一行内的数据仍然用逗号分隔，只不过每个数据使用 ⟨*attribute*⟩=⟨*value*⟩ 这种赋值形式写出来，例如 x=5， lo=500，这会设置 /data point/x=5，/data point/lo=500。

```
\begin{tikzpicture}
\datavisualization [school book axes,
                    visualize as line]
    data [format=named] {
      x=0, y=0
      x=1, y=1
      x=2, y=1
      x=3, y=0
    };
\end{tikzpicture}
```

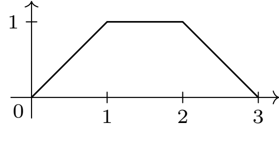
在给标识符赋值时，不仅可以赋予单个数值，还可以赋予一个数组，而且数组中可以使用省略号构建等差数组，如 x={1，2，3}，x={1,...，5} 这种形式。观察下图：

```
\tikz \datavisualization [scientific axes=clean, visualize as scatter/.list={a,b,c},
                          style sheet=cross marks]
  data [format=named] {
    x=0, y={1,2,3}, set=a
    x={2,3,4}, y={3,4,5,7}, set=b
    x=6, y={5,7,...,15}, set=c
  };
```

数据格式的第 2 行实际构造了一个 4×3 矩阵。

Format TeX code

这个格式直接用 TEX 代码指定数据点。TEX 代码的内容主要是对 /data point/⟨*attribute*⟩ 赋值并使用宏 \pgfdatapoint 创建一个数据点。注意，每当一行结束时，控制权会交还给格式手柄（format handler），因此一个命令要在一行内写完，不要跨行。当然也并非每一行都必须要创建一个数据点。




```
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as line]
    data [format=TeX code] {
      \pgfkeys{/data point/.cd,x=0, y=0} \pgfdatapoint
      \pgfkeys{/data point/.cd,x=1, y=1} \pgfdatapoint
      \pgfkeys{/data point/x=2}           \pgfdatapoint
      \pgfkeyssetvalue{/data point/x}{3}
      \pgfkeyssetvalue{/data point/y}{0} \pgfdatapoint
    };
\end{tikzpicture}
```

81.4 函数格式

TikZ Library `datavisualization.formats.functions`

```
\usetikzlibrary{datavisualization.formats.functions} % LaTeX and plain TeX
\usetikzlibrary[datavisualization.formats.functions] % ConTeXt
```

这个程序库允许用户利用函数表达式提供数据点来绘制函数图形。

Format `function`

在这个格式下，把变量和函数式提供给 `data` 指令，Tikz 根据函数式自动创建样本点并绘制函数图形。

在声明变量 (variable declaration) 和函数 (function declaration) 的句法中用到 `var`, ⟨*variable*⟩, `func`, ⟨*attribute*⟩, 这些都是起到标示作用的符号，因此它们与其它符号之间必须有空格分隔。

变量声明 提供变量用到“变量声明”(variable declaration)，变量声明的句法如下：

1. `var` ⟨*variable*⟩ `:` `interval[`⟨*low*⟩`:`⟨*high*⟩`]` `samples` ⟨*number*⟩`;`

   注意 `var` 与 ⟨*variable*⟩ 之间有空格，⟨*variable*⟩ 与冒号“:”之间有空格。如果没有上面句法中的 `samples`, 那么样本点的数量由下一选项的值决定：

   `/pgf/data/samples=`⟨*number*⟩ (no default, initially 25)

   如果没有上面句法中的 `samples`, 本选项就决定样本点的数量。

   ⟨*variable*⟩ 是变量名，即标识符 attribute，它的 key 路径是 `/data point/`⟨*variable*⟩.

   上面句法中的 `interval` 设置变量的变化区间。

2. `var` ⟨*variable*⟩ `:` `interval[`⟨*low*⟩`:`⟨*high*⟩`]` `step` ⟨*step*⟩`;`

   注意 `var` 与 ⟨*variable*⟩ 之间有空格，⟨*variable*⟩ 与冒号“:”之间有空格。当计算样本点时，`step` 设置变量的变化步长，即 ⟨*variable*⟩ 的值从 ⟨*low*⟩ 开始，以 ⟨*step*⟩ 为公差增长。

3. `var` ⟨*variable*⟩ `:` `{`⟨*values*⟩`};`

注意 var 与 ⟨*variable*⟩ 之间有空格，⟨*variable*⟩ 与冒号“:”之间有空格。⟨*values*⟩ 是个数值列表，它直接规定变量 ⟨*variable*⟩ 的取值，该列表可以使用 \foreach 的省略号来构造等差数列。

4. 预定义的特殊变量set.

在一个 data 指令中，可以定义多个变量，各个变量独立变化，例如写下

```
var x : interval [0:1]
var y : interval [0:1]
```

对于每个变量来说，其样本点数量默认为 25，故由 x，y 构成的点 (x，y) 的个数就是 625.

函数声明 声明函数的句法是：

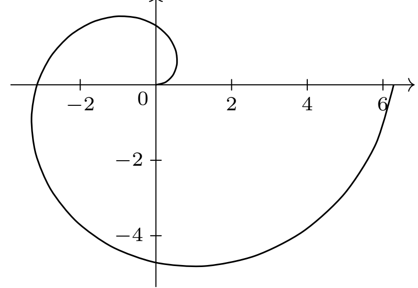
func ⟨*attribute*⟩ = ⟨*expression*⟩;

注意 func 与 ⟨*attribute*⟩ 之间有空格，⟨*attribute*⟩ 与等号“=”之间有空格。⟨*expression*⟩ 是函数表达式，会用标准的 TikZ 数学解析器来解析之。应该用变量声明中所声明的变量来构造 ⟨*expression*⟩，程序根据自变量的样本点值来计算函数值，函数值储存在 /data point/⟨*attribute*⟩ 中。
⟨*expression*⟩ 中的变量必须使用这种格式：

\value{⟨*variable*⟩}

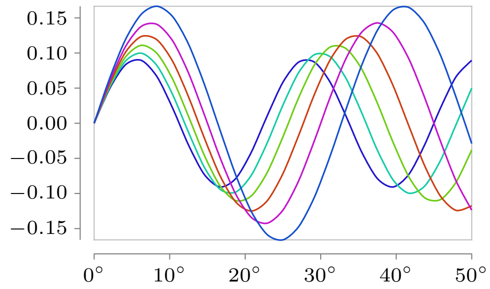
这个宏会被展开为 /data point/⟨*variable*⟩ 的值来计算。

在一个 data 指令中可以声明多个函数。




```
\tikz
  \datavisualization
    [school book axes, visualize as smooth line]
    data [format=function] {
      var x : interval [-1.5:1.5];
      func y = \value x * \value x;
    };
```

```
\tikz \datavisualization [
    school book axes,
    all axes={unit length=5mm, ticks={step=2}},
    visualize as smooth line]
  data [format=function] {
    var t : interval [0:2*pi];
    func x = \value t * cos(\value t r);
    func y = \value t * sin(\value t r);
  };
```




```
\tikz \datavisualization [
    scientific axes=clean,
    y axis={ticks={style={
      /pgf/number format/fixed,
      /pgf/number format/fixed zerofill,
      /pgf/number format/precision=2}}},
    x axis={ticks={tick suffix=${}^\circ$}},
    visualize as smooth line/.list={1,2,3,4,5,6},
    style sheet=vary hue]
  data [format=function] {
    var set : {1,...,6};
    var x : interval [0:50];
    func y = sin(\value x * (\value{set}+10))*
      (\value{set}+5)^(-1);
  };
```

81.5 数据处理过程

当数据传递给可视化系统后，都会经由命令 \pgfdata 处理，这个命令是 datavisualization 模块定义的，此命令可以解析外部数据，也可以解析 inline 数据。因为 TEX 总是从左到右、逐行读取文件，所以提供给数据格式解析器的数据也要按照从左到右、逐行编辑的方式编写。

\pgfdata[⟨*options*⟩]{⟨*inline data*⟩}

本命令处理数据，然后传递给渲染管线 (visualization pipeline)，仅当适当设置“数据可视化对象”后此命令才可用，“数据可视化对象”(data visualization object) 指的是保存在 /pgf/data visualization/obj 中的对象。

基本选项 在 ⟨*options*⟩ 中的选项会被冠以 /pgf/data/ 来执行，可以使用以下选项：

/pgf/data/read from file=⟨*filename*⟩ (no default, initially empty)

如果写出 ⟨*filename*⟩，就从文件 ⟨*filename*⟩ 中读取数据，此时不需要 {⟨*inline data*⟩} 这一部分。如果不写出 ⟨*filename*⟩，则需要给出 {⟨*inline data*⟩} 这一部分。

```
\pgfdata[format=table, read from file=file1.csv]
\pgfdata[format=table, read from file=file2.csv]
```

```
\pgfdata[format=table]
{
  x, y
  1, 2
  2, 3
}
```

/pgf/data/inline (no value)

这是 read from file={} 的简写，此时需要提供 {⟨*inline data*⟩} 这一部分。

/pgf/data/format=⟨*format*⟩ (no default, initially table)

本选项指定数据表的格式。⟨*format*⟩ 是用命令 \pgfdeclaredataformat →P. 414 声明的数据格式。

`/pgf/every data` (style, no value)

在 `\pgfdata` 解析 ⟨*options*⟩ 之前，会先执行这个样式。注意这个 key 的路径是 `/pgf/`, 不是 `/pgf/data/`. Tikz 会利用本选项,使得样式 `/tikz/every data` 和 `/tikz/datavisualization/every data` 被执行，所以使用 Tikz 时，最好使用样式 `/tikz/every data`.

收集数据 当数据格式和数据来源确定后，数据就会被收集。此时的数据并不会被“细致地”解析，只是收集起来以便稍后在可视化过程中解析。

- 在指定了外部数据来源的情况下，通过 `add data` 这个 method 告知数据可视化对象，稍后 (在可视化过程中) 从指定的文件中读取数据。
- 在提供 inline 数据的情况下，针对所采用的数据格式，某些符号的类别 (catcode) 会被修改，另外也会把换行符号 (return) 改为活动符 (active character)，以保证 TEX 能正确地读取数据。然后 inline 数据会被作为通常的参数读取。数据可视化对象会保存所有数据，且在稍后采用格式解析器来解析数据。

解析数据 在数据可视化过程中，通过 `add data` 这个 method 添加到数据可视化对象中的代码会被执行数次，代码使用命令 `\pgfdatapoint` 创建数据点。当 `\pgfdata` 调用 `add data` 时，传递给数据可视化对象的代码会调用一些与 `add data` 相关联的内部宏，这些内部宏做以下事情:

1. 针对数据格式修改符号的类代码。
2. 执行起始代码 (startup code)，将解析过程中的参数初始化。
3. 用对应数据格式的“手柄”(format-specific code handler) 逐行处理数据，此手柄把当前行的数据作为“输入”，调用命令 `\pgfdatapoint` 创建一个数据点。空行由特殊的代码处理。
4. 末了，执行 format-specific end code.

数据组 下面 3 个选项用于创建数据组 (data sets)。一个数据组实际上就是一个宏，其中保存了被解析的数据，这个宏可以多次使用。

`/pgf/data/new set=`⟨*name*⟩ (no default)

创建一个名称为 ⟨*name*⟩ 的空数据组。如果已经存在名称为 ⟨*name*⟩ 的数据组，则旧数据组会被覆盖。数据组是全局有效的。本选项不能用作命令 `\pgfdata` 的选项。

`/pgf/data/store in set=`⟨*name*⟩ (no default)

如果名称为 ⟨*name*⟩ 的数据组已经存在，那么当前的 `\pgfdata` 命令所解析的数据不会直接传递给渲染管线，而是会被添加（附加，appended）到数据组 ⟨*name*⟩ 中，命令 `\pgfdata` 的选项也会一并被添加（附加）。本选项不能用作命令 `\pgfdata` 的选项。

`/pgf/data/use set=`⟨*name*⟩ (no default)

当本选项用作命令 `\pgfdata` 的选项后，所处理的是数据组 ⟨*name*⟩ 中的数据。

81.6 定义新数据格式

`datavisualization` 模块提供下面的命令用来定义新数据格式。

`\pgfdeclaredataformat`{⟨*format name*⟩}{⟨*catcode code*⟩}{⟨*startup code*⟩}{⟨*line arguments*⟩}{⟨*line code*⟩}{⟨*empty line code*⟩}{⟨*end code*⟩}

本命令定义名称为 ⟨*format name*⟩ 的数据格式。使用 ⟨*format name*⟩ 后，数据解析过程会参照 ⟨*format name*⟩ 的定义如下进行:

1. 执行 ⟨*catcode code*⟩, 即改变某些符号的类别。
2. 执行起始代码 ⟨*startup code*⟩.
3. ⟨*line arguments*⟩ 是某个内部宏的参数列表，这个宏的定义内容 (body) 则是 ⟨*line code*⟩. 数据表中的每个非空行会被作为参数传递给这个内部宏，并用 ⟨*line code*⟩ 做处理。
4. 数据表中的空行由 ⟨*empty line code*⟩ 处理。通常只需要忽略空行，将 ⟨*empty line code*⟩ 空置即可。
5. 在数据表的结束处执行 ⟨*end code*⟩.

假设某个数据表中的各行都是 `(1.2,3.2)` 这种形式的数据点:

```
# This is some data formatted according to the "coordinates" format
(0,0)
(0.5,0.25)
(1,1)
(1.5,2.25)
(2,4)
```

下面定义一种数据格式来解析这种形式是数据表，并把 # 作为注释符号:

```
\pgfdeclaredataformat{coordinates}
% 把 # 用作注释符号
{\catcode `\#=14\relax}
% 起始代码为空
{}
% 规定参数列表
{(#1,#2)}
% 规定处理参数的代码
{
  \pgfkeyssetvalue{/data point/x}{#1}
  \pgfkeyssetvalue{/data point/y}{#2}
  \pgfdatapoint
}
% 忽略空行
{}
% 结束代码为空
{}
```

这样定义后，就可以使用 `format=coordinates` 了:

```
\begin{tikzpicture}
\datavisualization[school book axes, visualize as smooth line]
  data [format=coordinates] {
    # This is some data formatted according
    # to the "coordinates" format
    (0,0)
    (0.5,0.25)
    (1,1)
    (1.5,2.25)
```

```
    (2,4)
  };
\end{tikzpicture}
```

82 坐标轴

82.1 Overview

对于平面绘图来说，一般需要 2 个维度来描述数据，笛卡尔系是横坐标和纵坐标，极坐标系是角度和极径，在可视化引擎看来，两种坐标系都涉及 2 个“轴”。在可视化图形中画出坐标轴时可能需要考虑：

- 画出的坐标轴的实际长度。假如变量 x 的数据变化范围是 $[10^{-10}, 10^{10}]$, 而在页面上画出的 x 坐标轴长度是 10cm, 这时就需要一个映射，把 $[10^{-10}, 10^{10}]$ 映射到 [0cm, 10cm]. 有时候还需要对数化的坐标轴，这需要相应的设置。
- 坐标轴的刻度线、刻度线的标签。
- 坐标轴的网格线。
- 坐标轴自己的标签。

一般情况下，数据可视化系统中不能直接使用 TikZ 的绘图命令或选项，如命令 `\draw`, 关于线型、颜色的选项，关于 node 的各种选项等等。但是对于绘图来说，TikZ 的命令或选项十分有力，因此可视化系统定义了能够沟通数据可视化与 TikZ 的命令和选项。例如命令 `\datavisualization ... info`→P. 405, `\datavisualization ... info'`→P. 406, 选项 `/tikz/data visualization/style`→P. 436, `/tikz/data visualization/r` `style`→P. 437, 这两个选项中的参数（也是选项）都是以 `/tikz/` 开头的 key，但是在这两个选项中，并非所有的 TikZ 选项都有效。这两个选项可以用在很多绘图要素中，例如，刻度 ticks，网格 grid，可视化轴 visualize axis 等等。

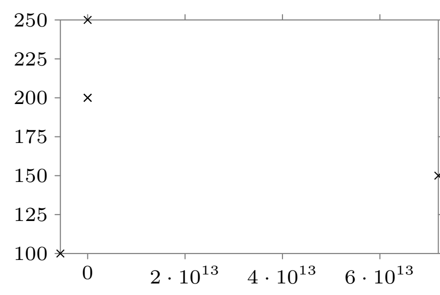
82.2 轴的基本设置

在数据可视化图形中可能会画出坐标轴，画出来的坐标轴通常表现为一个线段，线段上还可能会有刻度线、网格线。但是画出来的坐标轴并不是这里所说的“轴”，这里说的“轴”其实是一个数据处理模式、计算过程，它将变量值对应到页面上的某个点。定义“轴”后可以用线段来表现“轴”。举例来说，写出数据点

```
x, y
-5600000000000, -1
0, 0
7200000000000, 1
```

其中给出两个变量名（attribute）：`x`, `y`，它们的 key 路径是 `/data point/x` 和 `/data point/y`. 注意 `x`, `y` 不是“轴”的名称，只是变量名，每个变量都应该对应一个“轴”，例如对于预定义的坐标轴系统 `/tikz/data visualization/scientific axes`→P. 425 和 `/tikz/data visualization/school book axes`→P. 427, 默认 `x` 与轴 `x axis` 对应，`y` 与轴 `y axis` 对应，也就是说，轴也有自己的名称，有名称就便于加以引用、设置。变量 `x` 的值集是区间 $[-5.6 \cdot 10^{12}, 7.2 \cdot 10^{12}]$ 的子集，这个区间跨度很大，如果不适当处理一下就不便于在页面上显示。为了能在页面上适当显示数据点，需要对变量的数据做处理，例如，

对区间 $[-5.6 \cdot 10^{12}, 7.2 \cdot 10^{12}]$ 做变换，使之对应一个 5cm 长的线段，那么变量 x 的各个值就可以对应此线段上的点——这种对应是“轴”的功能之一。对于预定义的坐标轴系统来说，轴会自动检查自变量值的范围，并把这个范围转变成一个长度比较“合理的”区间以便于绘图。如果需要对数坐标轴，就需要设置相应的选项了。




```
\tikz \datavisualization [scientific axes,
    visualize as scatter]
  data {
    x, y
    -5600000000000, 100
    1910, 200
    1950, 200
    1960, 250
    7200000000000, 150
  };
```

82.2.1 用法

用下面的选项创建一个“轴”并为其命名:

`/tikz/data visualization/new axis base`=⟨*axis name*⟩ (no default)

创建一个轴，⟨*axis name*⟩ 是轴名称。⟨*axis name*⟩ 本身有大量的参数，这些参数可以对轴的各方面特性做出详细设置，不过在初始之下这些参数的值多数都是空的，需要你自己指定这些参数的值。后文介绍的很多选项都是用于设置这些参数值的。

本选项的效果之一是自动把 ⟨*axis name*⟩ 做成一个 key:

`/tikz/data visualization/`⟨*axis name*⟩=⟨*options*⟩ (no default)

用选项 `new axis base=`⟨*axis name*⟩ 创建一个名称为 ⟨*axis name*⟩ 的轴后，这个 key 会被自动创建，利用这个 key 可以对轴 ⟨*axis name*⟩ 做详细设置。⟨*options*⟩ 中的选项会被冠以路径前缀 `/tikz/data visualization/axis options` 来执行。

`/tikz/data visualization/all axes`=⟨*options*⟩ (no default)

⟨*options*⟩ 对当前环境内的所有的轴有效。

以上两个选项中的 ⟨*options*⟩ 之内可以用的选项将在下文介绍。

注意选项 `new axis base` 只是声明一个轴，它并不把轴与某个变量 (attribute) 对应起来，也不创建 transformation object, 即不会把变量值 (数据点) 映射到页面上。例如写出

```
[new axis base=x axis,
 new axis base=y axis,
 x axis={attribute=x},
 y axis={attribute=y}]
```

即声明轴 `x axis` 并且将变量 x 与之对应，声明轴 `y axis` 并且将变量 y 与之对应，但这样并不能规定轴 `x axis` 就是页面上的横轴，也不能规定轴 `y axis` 就是页面上的纵轴，也就不能把数据点 (x, y) 与页面上的点对应起来。而某些特殊的选项，如 `/tikz/data visualization/new Cartesian axis`→ P. 423, 先用 `new axis base` 创建轴，然后创建一个内部的“对象”(object) 把变量值映射到页面上。

82.2.2 与轴对应的变量

每个轴都对应一个变量，对于预定义的轴系统，默认 `x axis`, `y axis`, `z axis` 这 3 个轴对应的变量名称应该分别是 x, y, z, 变量名应该在数据点列表的首行给出。如果数据点列表的首行给出的变量名是其它符号，就需要用下面的选项把轴名称跟变量名对应起来。

`/tikz/data visualization/axis options/attribute`=⟨*attribute*⟩ (no default)

这个选项用在轴名称选项 ⟨*axis name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，将这个轴名称与变量名 ⟨*attribute*⟩ 对应起来，这个轴会监测该变量的取值，将变量值映射到一个合理的区间中。

```
\tikz \datavisualization [scientific axes,
    x axis={attribute=people, length=2.5cm, ticks=few},
    y axis={attribute=year},
    visualize as scatter]
  data {
    year, people
    1900, 100
    1910, 200
    1950, 200
    1960, 250
    2000, 150
  };
```

82.2.3 变量值的范围

变量的当前值储存在 `/data point/`⟨*attribute*⟩ 中，可视化引擎会检查变量的最大值与最小值从而确定变量的变化范围，称之为“变量值集”、“变量值区间”（attribute range interval），变量值集一般由数据点列表决定。可以用下面的选项向变量值集中添加数据，从而改变视化图形中坐标轴的显示范围。

`/tikz/data visualization/axis options/include value`=⟨*list of value*⟩ (no default)

这个选项用在轴名称选项 ⟨*axis name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，使得该轴包含 ⟨*list of value*⟩ 列出的数据，而不仅仅包含该轴对应的变量值集。

```
\tikz \datavisualization [scientific axes,
    all axes={length=3cm},
    visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

```
\tikz \datavisualization [scientific axes,
    all axes={length=3cm},
    visualize as line,
    x axis={include value={12,30}},
    y axis={include value=0}]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

`/tikz/data visualization/axis options/min value`=⟨*value*⟩ (no default)

这个选项用在轴名称选项 ⟨*axis name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，指定该轴对应的变量值集的最小值，如果 ⟨*value*⟩ 小于原来变量值集的最小值, 则相当于向该变量值集合中添加数据 ⟨*value*⟩. 如果 ⟨*value*⟩ 大于原来变量值集的最小值，那么变量值集中小于 ⟨*value*⟩ 的数据将被排除在可视化范围之外。

`/tikz/data visualization/axis options/max value`=⟨*value*⟩ (no default)

这个选项用在轴名称选项 ⟨*axis name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，指定该轴对应的变量值集的最大值，作用类似 `min value`.

82.2.4 轴对数据的变换

对变量值的线性变换 假设变量值属于数轴 $\mathscr{L}$(没有长度单位), $s_1, s_2, t_1, t_2 \in \mathscr{L}$, 规定一个线性变换:

$$L_{\mathscr{L}}\colon \mathscr{L} \to \mathscr{L}, \quad L_{\mathscr{L}}(s) = t = t_1 + (s - s_1)\frac{t_2 - t_1}{s_2 - s_1}.$$

这个变换把区间 $[s_1, s_2]$ 变成 $[t_1, t_2]$. 实现这个线性变换 $L_{\mathscr{L}}$ 的是下面的选项:

`/tikz/data visualization/axis options/scaling`=⟨s_1⟩ `at` ⟨t_1⟩ `and` ⟨s_2⟩ `at` ⟨t_2⟩ (no default)

这个选项用在轴名称选项 ⟨*axis name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，其中 ⟨s_1⟩, ⟨s_2⟩ 是变换前的数值，由 fpu 库的命令 `\pgfmathfloatparsenumber` 处理，所以应当是纯数值；⟨t_1⟩, ⟨t_2⟩ 是变换后的值，由 PGF 命令 `\pgfmathparse` 处理，所以 ⟨t_1⟩, ⟨t_2⟩ 可以是纯数值，也可以是带有长度单位的尺寸。

命令 `\pgfmathparse{`⟨t_1⟩`}` 的特点是：若 ⟨t_1⟩ 是带单位的尺寸，则给把单位转换为 `pt` 再由 TeX 做进一步的处理；最后将处理结果中的单位 `pt` 去掉，只把结果的数值部分保存到 `\pgfmathresult` 中；若 ⟨t_1⟩ 是纯数值，则处理结果与带上单位 `pt` 的情况一样。按 TeX 本身的限制，`\pgfmathparse` 不能处理超出 ±16384`pt` 的值。

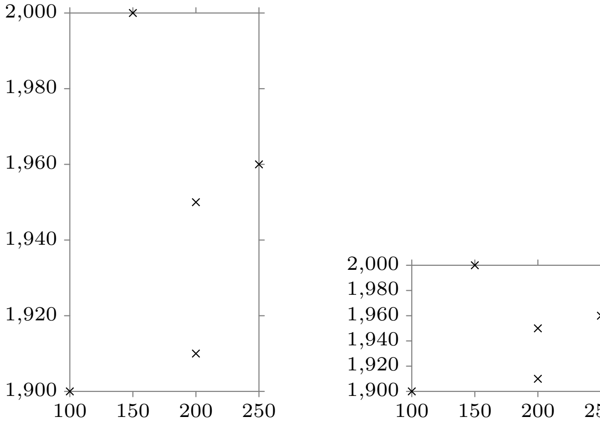
例如，由于 `5cm=142.2637pt`, 所以

```
scaling=1000 at 0cm and 1000000 at 5cm
```

等效于

```
scaling=1000 at 0 and 1000000 at 142.2637
```

下面例子中两个图形对于 `y axis` 的设置不同，导致纵轴的高度不同。



```
\tikz \datavisualization
    [scientific axes,
     x axis={attribute=people, length=2.5cm, ticks=few},
     y axis={attribute=year, scaling=1900 at 0cm and 2000 at 5cm},
     visualize as scatter]
  data {
    year, people
    1900, 100
    1910, 200
    1950, 200
    1960, 250
    2000, 150
  };
\qquad
\tikz \datavisualization
    [scientific axes,
     x axis={attribute=people, length=2.5cm, ticks=few},
     y axis={attribute=year, scaling=1800 at 0cm and 2100 at 5cm},
     visualize as scatter]
  data {
    year, people
    1900, 100
    1910, 200
    1950, 200
    1960, 250
    2000, 150
  };
```

完成上述线性变换后,给变换后的值 t 带上长度单位 pt, 就把区间 $[s_1, s_2]$ 线性地映射到线段 $[t_1\text{pt}, t_2\text{pt}]$, 此线段的长度是 $|t_2 - t_1|$pt, 这应该就是用户所需要的、用来代表变量值集范围的、“长度合理的”画图区间。所以 `scaling` 的作用有两方面：(1) 对变量值做线性变换；(2) 确定“长度合理的”画图区间。

在上述线性变换下，$s \to t$, 变换前的变量值 s 仍然被储存在 `/data point/`⟨*attribute*⟩ 中，而变换后的 t 会被储存在 `/data point/`⟨*attribute*⟩`/scaled` 中。假如设置:

```
[x axis = {attribute = x, scaling=1000 at 20 and 2000 at 30}]
```

那么与 `/data point/x=1200` 对应的是 `/data point/x/scaled=22`.

数值的格式以及关键词 `min` 和 `max` s_1, s_2 可以是 `3.14` 或 `10000000000` 这样的数值，也可以是 `(pi/2)` 这样用圆括号括起来的表达式。关键词 `min` 和 `max` 分别代表变量值集中的最小值和最大值。例如在前面的例子中，对于变量 `year` 来说，关键词 `min` 的值是 1900, 关键词 `max` 的值是 2000. 注意关键词 `min` 和 `max` 只对 s_1 和 s_2 有意义。这两个关键词的典型用法是

```
scaling = min at 0cm and max at 5cm
```

注意区间 $[s_1, s_2]$ 与区间 [`min`, `max`] 可以有公共元素，也可以没有公共元素。

非线性变换 选项 `scaling=`⟨s_1⟩ `at` ⟨t_1⟩ `and` ⟨s_2⟩ `at` ⟨t_2⟩ 规定变量值与线段 $[t_1\text{pt}, t_2\text{pt}]$ 上的点成线性对应。如果要让变量值与坐标轴上的点成非线性对应，那么可以：(1) 先对变量值做一个非线性变换；(2) 再把变换后的值与坐标轴上的点线性地对应起来。对变量值的非线性变换用到下面的选项:

`/tikz/data visualization/axis options/function=`⟨*code*⟩ (no default)

这个选项用在轴名称选项 ⟨*axis name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，其中的 ⟨*code*⟩ 定义一个函数 f:

$$f(s) = t = t_1 + (f(s) - f(s_1))\frac{t_2 - t_1}{f(s_2) - f(s_1)}.$$

例如对变量值取自然对数，把坐标轴做成自然对数轴:

```
⟨axis name⟩={function=\pgfdvmathln{\pgfvalue}{\pgfvalue}}
```

其中第二个 `\pgfvalue` 代表映射前的变量值，第一个 `\pgfvalue` 代表映射后的值。
在文件《pgfmoduledatavisualization.code.tex》中对 `\pgfdvmathln` 的定义是:

```
\def\pgfdvmathln#1#2{%
  \pgfmathfloatln{#2}%
  \let#1=\pgfmathresult%
}
```

即先对 `#2` 取自然对数，然后把结果保存在 `#1` 中，`#2` 和 `#1` 都是浮点数格式的。

```
\tikz \datavisualization
    [scientific axes,
     x axis={ticks={major={at={1,10,100,1000}}},
       scaling=1 at 0cm and 1000 at 3cm,
       function=\pgfdvmathln{\pgfvalue}{\pgfvalue}},
     visualize as scatter]
  data [format=named] {
    x={1,100,...,1000}, y={1,2,3}
  };
```

对变量值做开平方变换:

```
⟨axis name⟩={function=\pgfdvmathunaryop{\pgfvalue}{sqrt}{\pgfvalue}}
```

在文件《pgfmoduledatavisualization.code.tex》中对 `\pgfdvmathunaryop` 的定义是:

```
\def\pgfdvmathunaryop#1#2#3{%
  \csname pgfmathfloat#2\endcsname{#3}%
  \let#1=\pgfmathresult%
}
```

```
\tikz \datavisualization
    [scientific axes,
     x axis={ticks=few,
       scaling=1 at 0cm and 1000 at 3cm,
       function=\pgfdvmathunaryop{\pgfvalue}{sqrt}{\pgfvalue}},
     visualize as scatter]
  data [format=named] {
    x={1,100,...,1000}, y={1,2,3}
  };
```

设置默认的 scaling 下面的选项设置 `scaling` 的默认值:

`/tikz/data visualization/axis options/scaling/default=`⟨*text*⟩ (initially `0 at 0 and 1 at 1`)

⟨*text*⟩ 就是 ⟨s_1⟩ `at` ⟨t_1⟩ `and` ⟨s_2⟩ `at` ⟨t_2⟩ 这种形式的，其初始值是 `0 at 0 and 1 at 1`. 如果不明显地指定 `scaling` 的值，则默认其值为 ⟨*text*⟩.

82.2.5 对数轴

`/tikz/data visualization/axis options/logarithmic` (no value)

这个选项用在轴名称选项 ⟨*axis name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，将轴 ⟨*axis name*⟩ 设为对数轴，本选项还会对坐标轴的刻度线做相应的设置。本选项的作用是：

- 利用选项 `function` 对变量值做对数变换。
- 创建刻度线、网格线的策略使用 `exponential strategy`.
- 设置 `scaling/default=1 at 0 and 10 at 1`.

```
\tikz \datavisualization [scientific axes,
    x axis={logarithmic},
    y axis={logarithmic},
    visualize as line]
  data [format=function] {
    var x : interval [0.01:100];
    func y = \value x * \value x;
  };
```

这个选项对极坐标下的轴也有效，下面的例子用了 `new polar axes` 选项，使用该选项需要调用程序库 datavisualization.polar，该选项用于声明极坐标系，在默认下坐标系的角度轴的名称是 `angle axis`, 极径轴的名称是 `radius axis`.

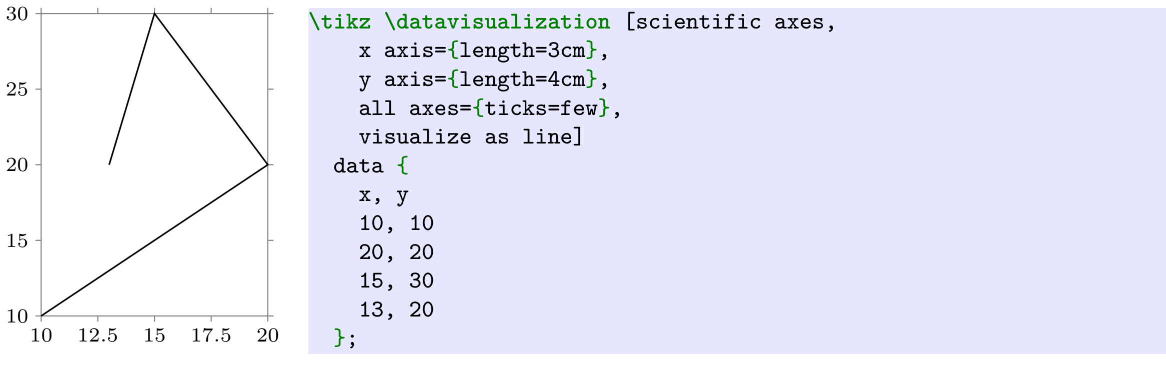
```
\tikz \datavisualization
    [new polar axes,
     angle axis={logarithmic, scaling=1 at 0 and 90 at 90},
     radius axis={scaling=0 at 0cm and 100 at 3cm},
     visualize as scatter]
  data [format=named] {
    angle={1,10,...,90}, radius={1,10,...,100}
  };
```

```
\tikz \datavisualization
    [new polar axes,
     angle axis={degrees},
     radius axis={logarithmic, scaling=1 at 0cm and 100 at 3cm},
     visualize as scatter]
  data [format=named] {
    angle={1,10,...,90}, radius={1,10,...,100}
  };
```

82.2.6 设置坐标轴的长度和单位长度

`/tikz/data visualization/axis options/length`=⟨*dimension*⟩ (no default)

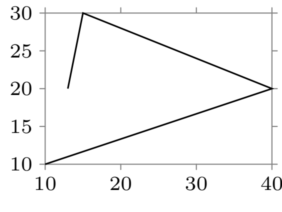
这个选项用在轴名称选项 ⟨*axis name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中, 设置 `scaling=min at 0cm and max at` ⟨*dimension*⟩, 从而规定坐标轴的长度尺寸。



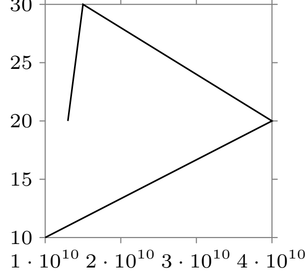

```
\tikz \datavisualization [scientific axes,
    x axis={length=3cm},
    y axis={length=4cm},
    all axes={ticks=few},
    visualize as line]
  data {
    x, y
    10, 10
    20, 20
    15, 30
    13, 20
  };
```

`/tikz/data visualization/axis options/unit length=`⟨*dimension*⟩ `per` ⟨*number*⟩ `units` (no default)

这个选项用在轴名称选项 ⟨*axis name*⟩`=`⟨*options*⟩ 的 ⟨*options*⟩ 中，设置 `scaling=0 at 0cm and 1 at` ⟨*dimension*⟩. 其中的 `per` ⟨*number*⟩ `units` 是可选的,指定 ⟨*number*⟩ 个数量的实际长度为 ⟨*dimension*⟩.



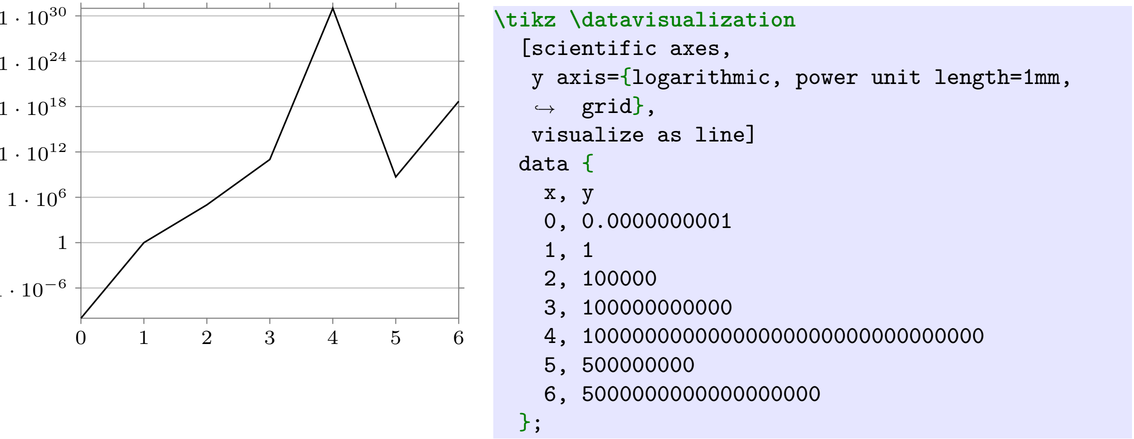

```
\tikz \datavisualization [scientific axes,
    all axes={ticks=few, unit length=1mm},
    visualize as line]
  data {
    x, y
    10, 10
    40, 20
    15, 30
    13, 20
  };
```




```
\tikz \datavisualization
    [scientific axes,
     x axis={unit length=1mm per 1000000000 units,
         ticks=few},
     visualize as line]
  data {
    x, y
    10000000000, 10
    40000000000, 20
    15000000000, 30
    13000000000, 20
  };
```

`/tikz/data visualization/axis options/power unit length=`⟨*dimension*⟩ (no default)

这个选项与 `logarithmic` 一起使用，本选项设置 `scaling=1 at 0cm and 10 at` ⟨*dimension*⟩.




```
\tikz \datavisualization
  [scientific axes,
   y axis={logarithmic, power unit length=1mm,
   ↪  grid},
   visualize as line]
  data {
    x, y
    0, 0.0000000001
    1, 1
    2, 100000
    3, 100000000000
    4, 10000000000000000000000000000000
    5, 500000000
    6, 5000000000000000000
  };
```

82.2.7 坐标轴的标签

`/tikz/data visualization/axis options/label={[⟨options⟩]⟨text⟩}` (默认为数学模式标签)

这个选项用在轴名称选项 ⟨*axis name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，把 ⟨*text*⟩ 设为坐标轴的标签。⟨*options*⟩ 中的选项会被冠以前缀 `/tikz/data visualization/` 来执行，因此可以用选项 `/tikz/data visualization/node style`→P. 437 把标签作为 node 来编辑。

```
\tikz \datavisualization
    [scientific axes,
     x axis = {label, length=2.5cm},
     y axis = {label={[node style={fill=blue!20}]{$x^2$}}},
     visualize as smooth line]
  data [format=function] {
    var x : interval [-3:5];
    func y = \value x * \value x;
  };
```

注意选项 `label` 并不直接创建一个 node 标签，因为它并不知道放置标签的位置。指定标签位置的是 `/tikz/data visualization/axis options/visualize label`→P. 460.

82.2.8 将变量值与页面上的坐标系相关联

与选项 `new axis base` 不同，下面的选项 `new Cartesian axis` 会创建一个 transformation object，将变量值映射到页面坐标系中，即把变量值与页面上的点对应起来。但这个选项不会导致坐标轴被“画出”，能画出坐标轴的选项是 `/tikz/data visualization/axis options/visualize axis`→P. 450.

`/tikz/data visualization/new Cartesian axis=⟨name⟩` (no default)

本选项创建一个名称为 ⟨*name*⟩ 的 Cartesian 轴，其定义在文件《tikzlibrarydatavisualization.code.tex》中：

```
\tikzdatavisualizationset{%
  new Cartesian axis/.style={
    new axis base={#1},
    new object={
      class=linear transformer,
```

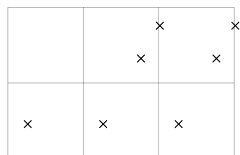
```
      store=/tikz/data visualization/#1/linear transformer,
      arg1/.expanded=\pgfkeysvalueof{/tikz/data visualization/#1/attribute
      ↪  }/scaled,
      arg2 from key=/tikz/data visualization/#1/unit vector,
    },
    #1/scaling/default/.initial=0 at 0 and 1 at 1cm,
    #1/unit vector/.initial=\pgfqpoint{1pt}{0pt},
  }
}%
```

可见它的 `scaling` 初始设置是 `0 at 0 and 1 at 1cm`, 它的 `unit vector` 的初始设置是 `(1pt,0pt)`.

可以用下面的选项指定轴的单位向量:

/tikz/data visualization/axis options/unit vector=⟨*coordinate*⟩ (no default, initially `(1pt,0pt)`)

这个选项的初始值是 `(1pt,0pt)`. 假设页面上的一条直线 l 过原点且方向为 ⟨*coordinate*⟩, 把原点和 ⟨*coordinate*⟩ 看作是 l 的一维坐标系。再设与坐标轴 l 对应的变量是 V_l, 变量的某个值 s 被变换为数值 t, 那么与值 s 对应的点就是 $t \cdot$ ⟨*coordinate*⟩, 这个点位于直线 l 上, 即位于页面上。所以用这种方法创建的是仿射坐标系。



```
\begin{tikzpicture}
\draw [help lines] (0,0) grid (3,2);
\datavisualization
    [new Cartesian axis=x axis,
     x axis={attribute=x},
     new Cartesian axis=y axis,
     y axis={attribute=y},
     x axis={unit vector=(0:1pt)},
     y axis={scaling=0 at 10pt and 10 at 200pt, unit vector=(60:1.5pt)},
     visualize as scatter]
  data {
    x, y
    0, 0
    1, 0
    2, 0
    1, 1
    2, 1
    1, 1.5
    2, 1.5
  };
\end{tikzpicture}
```

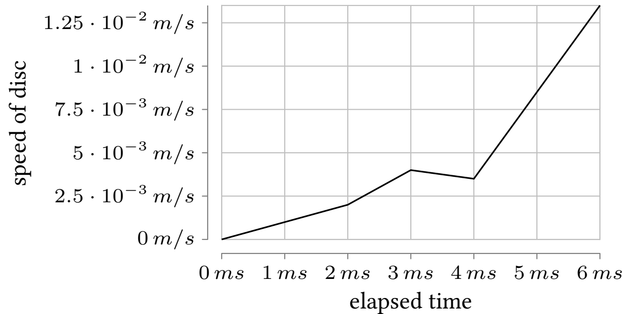
82.3 轴系统

一个轴系统 (axis system) 就是这样一组轴：有两个或数个轴，每个轴都有自己“全面、周到”的设置，例如，长度或者单位长度，对应的变量，变量值的变换函数，也有刻度线、网格、标签设置等等。

82.3.1 用法

有很多选项能对轴做出设置，例如

```
x axis = {attribute = time} 将轴与变量 time 对应起来
all axes={grid} 给坐标轴加网格线
all axes={ticks=few} 设置坐标轴的刻度线
x axis={ticks={tick unit=cm}}, y axis={ticks={tick unit=m/s^2}} 刻度线的间隔单位
x axes={label={time $t$ (ms)}}, y axis={label={distance $d$ (mm)}} 给坐标轴加标签
```




```
\tikz \datavisualization
    [scientific axes=clean,
     x axis={attribute=time, ticks={tick unit=ms}, label={elapsed time}},
     y axis={attribute=v, ticks={tick unit=m/s}, label={speed of disc}},
     all axes=grid,
     visualize as line]
  data {
    time, v
    0, 0
    1, 0.001
    2, 0.002
    3, 0.004
    4, 0.0035
    5, 0.0085
    6, 0.0135
  };
```

82.3.2 Scientific Axis Systems

`/tikz/data visualization/scientific axes`=⟨*options*⟩ (no default)

作为命令的选项,将图形中的坐标系设为 `scientific axes`, ⟨*options*⟩ 中的选项将被冠以前缀 `/tikz/data visualization/scientific axes` 来执行，以此为前缀的 key 都可以用在 ⟨*options*⟩ 中。

在文件《tikzlibrarydatavisualization.code.tex》中用了 162 行代码来定义轴系统 `scientific axes`. 从定义中看，`scientific axes` 是以轴系统 `xy Cartesian` 为基础来定义的，轴系统 `xy Cartesian` 默认两个轴的名称分别是 `x axis`, `y axis`, 这两个轴对应的变量名 (attribute) 分别被默认为 `x`， `y`.

可以用下面的选项对 `scientific axes` 做某些设置。

`/tikz/data visualization/scientific axes/width`=⟨*dimension*⟩ (no default, initially 5cm)

本选项设置横轴的实际长度，初始值 5cm， 横轴的默认名称是 `x axis`.

`/tikz/data visualization/scientific axes/height`=⟨*dimension*⟩ (no default)

本选项设置纵轴的实际长度，横轴的默认名称是 `y axis`. 默认下，纵轴长度与横轴长度之比为黄金分割比例，即 `width` 乘上 0.618 就是 `height`.

轴系统 `scientific axes` 会在数据区域的外围设置一个矩形外框，框线是灰色（black!50）的，当使用选项 `scientific axes/clean` 时就能看到这个灰色框线。

`/tikz/data visualization/every scientific axes` (style, no value)

用这个样式能重设轴系统 `scientific axes` 的默认设置。

`/tikz/data visualization/scientific axes/outer ticks` (no value)

使得 `scientific axes` 的刻度线指向图形外部，这是默认的。

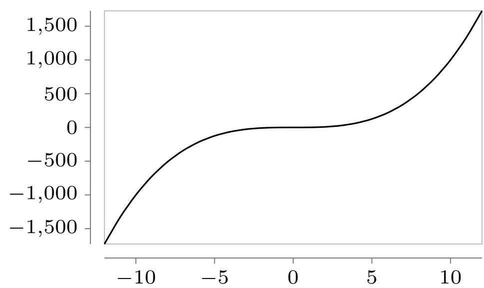
`/tikz/data visualization/scientific axes/inner ticks` (no value)

使得 `scientific axes` 的刻度线指向图形内部。

```
\begin{tikzpicture}
\datavisualization [scientific axes=inner ticks,
    visualize as smooth line]
  data [format=function] {
    var x : interval [-12:12];
    func y = \value x*\value x*\value x;
  };
\end{tikzpicture}
```

`/tikz/data visualization/scientific axes/clean` (no value)

本选项使得坐标轴以及刻度线脱离数据绘图区域，而绘图区域会用灰色框线标示出来，这样轴线、刻度线就不会与数据点重叠，以免干扰读图。




```
\tikz \datavisualization
    [scientific axes=clean,
     visualize as smooth line]
  data [format=function] {
    var x : interval [-12:12];
    func y = \value x*\value x*\value x;
  };
```

`/tikz/data visualization/scientific axes/standard labels` (no value)

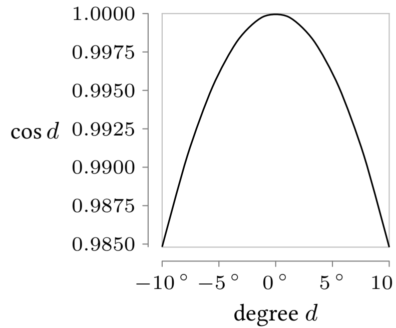
这个选项是默认的。本选项使得横轴的标签位于横轴中间的下部，标签方向是从左向右的；纵轴的标签位于纵轴中间的左侧，并且标签方向是从下向上的。

0.15
0.1
$5 \cdot 10^{-2}$
0
$-5 \cdot 10^{-2}$
-0.1
-0.15
$\sin d$
-10° -5° 0° 5° 10°
degree d

```
\tikz \datavisualization
  [scientific axes={clean, standard labels},
   visualize as smooth line,
   x axis={label=degree $d$,
      ticks={tick unit={}^\circ},
      length=3cm},
   y axis={label=$\sin d$}]
data [format=function] {
  var x : interval [-10:10] samples 10;
  func y = sin(\value x);
};
```

/tikz/data visualization/scientific axes/upright labels (no value)

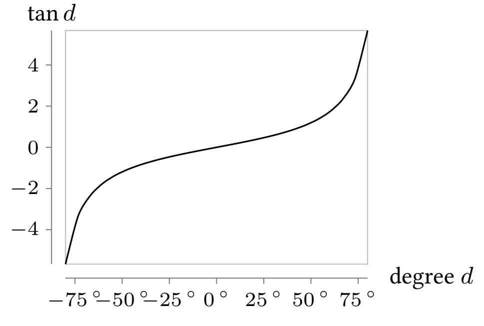
与 `standard labels` 类似，只是纵轴的标签方向是从左向右的。




```
\tikz \datavisualization
  [scientific axes={clean, upright labels},
    visualize as smooth line,
    x axis={label=degree $d$,
    ticks={tick unit={}^\circ},
      length=3cm},
    y axis={label=$\cos d$, include value=1,
    ticks={style={
      /pgf/number format/precision=4,
      /pgf/number format/fixed zerofill}}}]
data [format=function] {
  var x : interval [-10:10] samples 10;
  func y = cos(\value x);
};
```

/tikz/data visualization/scientific axes/end labels (no value)

本选项将坐标轴的标签放在坐标轴的末端中间。




```
\tikz \datavisualization
  [scientific axes={clean, end labels},
   visualize as smooth line,
   x axis={label=degree $d$,
    ticks={tick unit={}^\circ},
    length=4cm},
   y axis={label=$\tan d$}]
data [format=function] {
  var x : interval [-80:80];
  func y = tan(\value x);
};
```

82.3.3 School Book Axis Systems

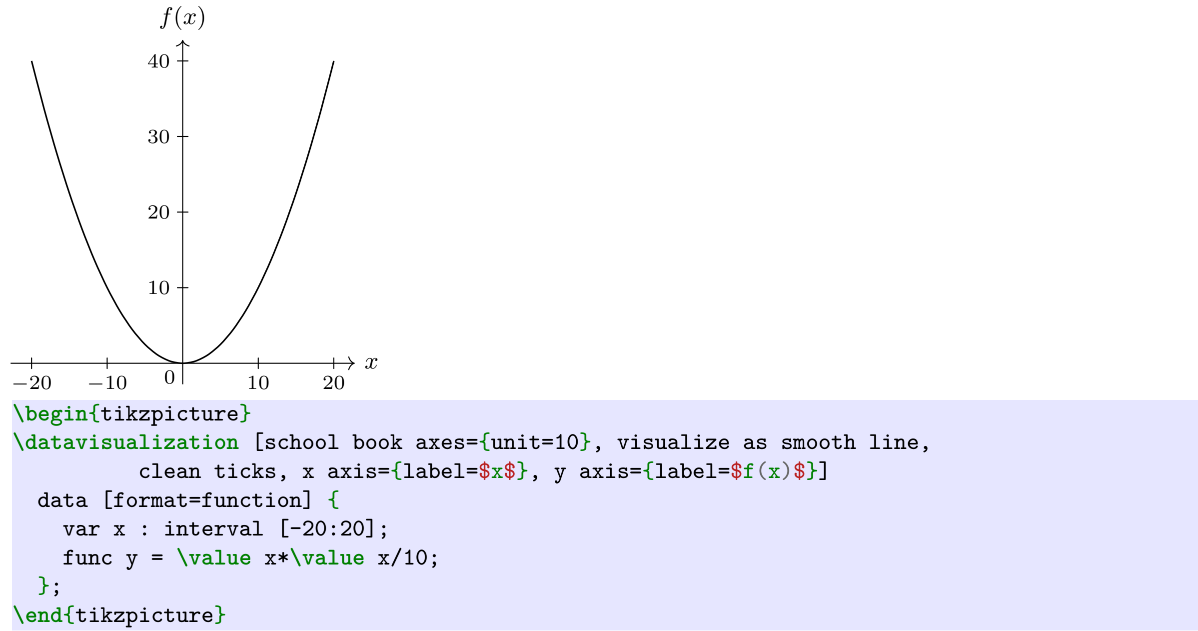
/tikz/data visualization/school book axes=⟨*options*⟩ (no default)

本选项将轴系统设为 `school book axes`. 在默认下，两个坐标轴交于原点，两个坐标轴的单位长度都是 1cm. 这个轴系统也是以轴系统 `xy Cartesian` 为基础来定义的。

⟨*options*⟩ 中可以使用下面的选项。

`/tikz/data visualization/school book axes/unit` (=⟨*value*⟩)

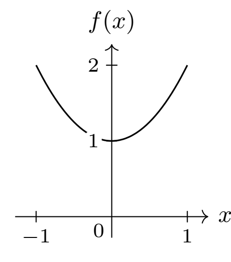
本选项相当于 `scaling=0 at 0cm and` ⟨*value*⟩ `at 1cm`, 刻度值也会随之变化。




```
\begin{tikzpicture}
\datavisualization [school book axes={unit=10}, visualize as smooth line,
          clean ticks, x axis={label=$x$}, y axis={label=$f(x)$}]
  data [format=function] {
    var x : interval [-20:20];
    func y = \value x*\value x/10;
  };
\end{tikzpicture}
```

`/tikz/data visualization/school book axes/standard labels` (no default)

这个选项使得横轴的标签位于横轴右端中间，纵轴标签位于纵轴上端中间。目前，教科书坐标系仅仅支持这种轴标签位置。




```
\begin{tikzpicture}
\datavisualization [school book axes={standard labels},
    visualize as smooth line, clean ticks,
    x axis={label=$x$}, y axis={label=$f(x)$}]
  data [format=function] {
    var x : interval [-1:1];
    func y = \value x*\value x + 1;
  };
\end{tikzpicture}
```

82.3.4 底层的笛卡尔坐标系

在文件《tikzlibrarydatavisualization.code.tex》中有如下定义：

```
\tikzdatavisualizationset{
  xy Cartesian/.style={
    new Cartesian axis=x axis,
    x axis={attribute=x,unit vector={(1pt,0cm)}},
    new Cartesian axis=y axis,
    y axis={attribute=y,unit vector={(0cm,1pt)}}
  },
  Xy axes/.style={x axis={#1},y axis={#1}},
  uv Cartesian/.style={
    new Cartesian axis=u axis,
```

```
    u axis={attribute=u,unit vector={(1pt,0cm)}},
    new Cartesian axis=v axis,
    v axis={attribute=v,unit vector={(0cm,1pt)}}
  },
  uv axes/.style={u axis={#1},v axis={#1}},
}%

\tikzdatavisualizationset{
  xyz Cartesian cabinet/.style={
    xy Cartesian,
    new Cartesian axis=z axis,
    z axis={attribute=z,unit vector={(-0.353553pt,-0.353553pt)}}
  },
  xyz axes/.style={x axis={#1},y axis={#1},z axis={#1}},
  uvw Cartesian cabinet/.style={
    uv Cartesian,
    new Cartesian axis=w axis,
    w axis={attribute=w,unit vector={(-0.353553pt,-0.353553pt)}}
  }
  uvw axes/.style={u axis={#1},v axis={#1},w axis={#1}},
}%
```

上面代码定义了轴系统 `xy Cartesian`，`uv Cartesian`，`xyz Cartesian cabinet`，`uvw Cartesian cabinet`，这些轴系统都用 `new Cartesian axis` 来定义，都默认“不画出坐标轴”。前面例子中使用的轴系统 `scientific axes`，`school book axes` 都是以底层的轴系统 `xy Cartesian` 为基础构造的。

`/tikz/data visualization/xy Cartesian` (no value)

本选项创建两个轴 `x axis`, `y axis`, 都默认 `unit vector={(1pt,0cm)}`.

```
\begin{tikzpicture}
\datavisualization [xy Cartesian, visualize as smooth line]
  data [format=function] {
    var x : interval [-1.25:1.25];
    func y = (\value x)^3;
  };
\end{tikzpicture}
```

`/tikz/data visualization/xy axes=`⟨*options*⟩ (no default)

⟨*options*⟩ 会成为轴 `x axis`, `y axis` 的选项。

`/tikz/data visualization/xyz Cartesian cabinet` (no default)

这个选项与 `xy Cartesian` 类似，创建三个轴 `x axis`, `y axis`, `z axis`, 第 3 个轴指向左下方，其一个长度单位的实际长度是 $\frac{1}{2}\sin 45°$cm，这也称为“斜二侧投影”（cabinet projection ）。

`/tikz/data visualization/xyz axes=`⟨*options*⟩ (no default)

⟨*options*⟩ 会成为轴 `x axis`, `y axis`, `z axis` 的选项。

`/tikz/data visualization/uv Cartesian` (no default)

这个选项与 `xy Cartesian` 类似，创建 2 个轴 `u axis`, `v axis`.

`/tikz/data visualization/uv axes=`⟨*options*⟩ (no default)

⟨*options*⟩ 会成为轴 `u axis`, `v axis` 的选项。

`/tikz/data visualization/uvw Cartesian` (no default)

这个选项与 `xyz Cartesian` 类似，创建 3 个轴 `u axis`, `v axis`, `w axis`, 第 3 个轴指向左下方，其一个长度单位的实际长度是 $\frac{1}{2}\sin 45°\text{cm}$，这也称为“斜二侧投影”（cabinet projection）。

`/tikz/data visualization/uvw axes=`⟨*options*⟩ (no default)

⟨*options*⟩ 会成为轴 `u axis`, `v axis`, `w axis` 的选项。

82.4 坐标轴的刻度和网格

82.4.1 概略

一般情况下网格与坐标轴的刻度线是重合的（如果有的话），有很多选项对刻度和网格是通用的。

刻度包括两个方面：刻度线和刻度值。刻度线分为 3 种：主刻度线（major，线较长），副刻度线（minor，线较短），次副刻度线（subminor，线最短），通常只有主刻度线才有刻度值标签。

网格线也分为 3 种：major，minor，subminor，主网格线最粗，副网格线较细，次副网格线最细。

与单词 major 有关的选项一般是关于主刻度线（网格线）的，与单词 minor 有关的选项一般是关于副刻度线（网格线）的。

有数种自动添加刻度的策略，也可以手工添加刻度。如果需要的刻度较多，则使用自动添加刻度的策略比较简便。但是刻度线的位置应该恰当，例如一般情况下，能用整数就不用小数，如果用小数则 2.5 要比 2.497 更好。自动添加刻度的策略需要做某些计算来确定“好”的刻度线的位置。

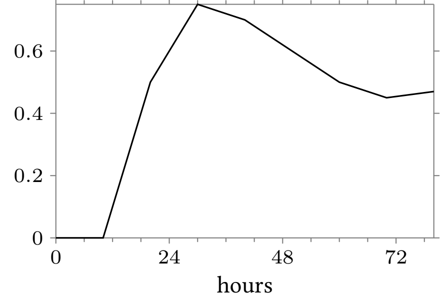
横轴的网格线是一组与横轴垂直的直线，纵轴的网格线是一组与纵轴垂直的直线，所以要想得到网格，需要给两个轴都带上网格选项，或者给可视化命令带上选项 `all axes=grid`.

82.4.2 刻度和网格的主要选项

`/tikz/data visualization/axis options/ticks=`⟨*options*⟩ (default `some`)

用于轴名称的选项中，可用于 ⟨*options*⟩ 的选项在后文介绍，默认为 `some`（设置大约 5 个刻度线）。本选项只是一种挑选刻度线的机制，并不实际画出刻度线，选项 `visualize ticks` 画出刻度线。

⟨*options*⟩ 中的选项将被冠以 `/tikz/data visualization/` 来执行。如果在一个轴中多次使用 `ticks` 选项，则选项作用会累加。




```
\tikz \datavisualization [scientific axes, visualize as line,
        x axis={ticks={step=24, minor steps between steps=3},
        ↪  label=hours}]
  data {
    x, y
    0, 0
    10, 0
    20, 0.5
    30, 0.75
    40, 0.7
    50, 0.6
    60, 0.5
    70, 0.45
    80, 0.47
  };
```

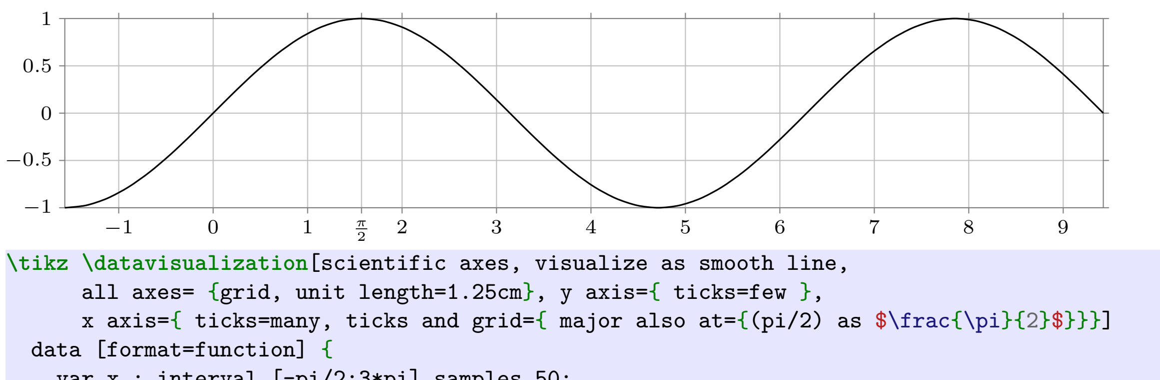
`/tikz/data visualization/axis options/grid=⟨options⟩` (default at default ticks)

与选项 `ticks` 类似，可用于 `ticks` 的选项也能用于本选项，当然本选项只对网格线有效。本选项并不实际画出网格，选项 `visualize grid` 画出网格。

本选项的默认值是 `at default ticks`, 这会使得网格线与刻度线重合。

`/tikz/data visualization/axis options/ticks and grid=⟨options⟩` (no default)

⟨*options*⟩ 会被传递给 `ticks` 和 `grid`.




```
\tikz \datavisualization[scientific axes, visualize as smooth line,
      all axes= {grid, unit length=1.25cm}, y axis={ ticks=few },
      x axis={ ticks=many, ticks and grid={ major also at={(pi/2) as $\frac{\pi}{2}$}}}]
  data [format=function] {
    var x : interval [-pi/2:3*pi] samples 50;
    func y = sin(\value x r);
  };
```

82.4.3 计算刻度线和网格线位置的半自动机制

以下针对刻度所做的介绍，对网格线也是有效的。

预定义的自动添加刻度的策略有两个：linear steps 策略和 exponential steps 策略。linear steps 策略创建的刻度是等差数列，假设某个坐标轴使用 linear steps 策略，并且需要画出的变量值集是区间 $[a,b]$，该策略创建的刻度就是 $i\cdot s+p\in[a,b]$，$i\in\mathbb{Z}$，其中 s 是公差（即步长 `step`），p 是初值（`phase`），i 取不同的值就得到不同的刻度，这些刻度是主刻度；而 exponential steps 策略则适用于对数坐标轴，创建的刻度是以 $10^{i\cdot s+p}$ 为公比的等比数列，这些刻度也是主刻度。

linear steps 策略是默认的策略。

可以手工调整上述两个策略中的 `step`，`phase`, 以及副刻度线，用到下面的选项：

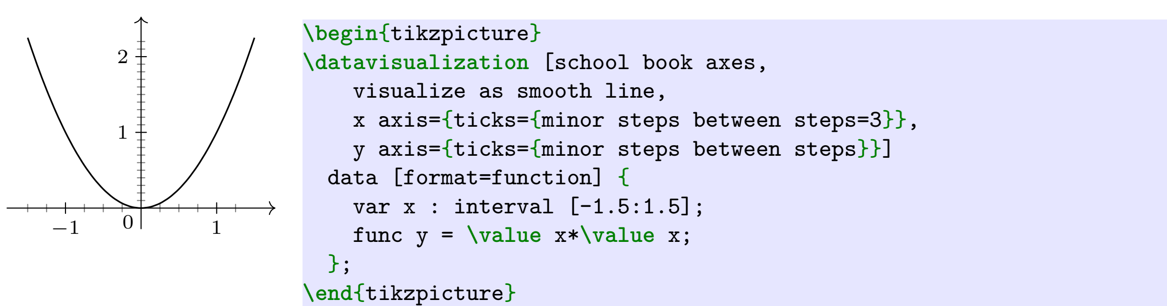
/tikz/data visualization/step=⟨*value*⟩ (no default, initially 1)

设置 linear steps 策略和 exponential steps 策略的步长 step.

```
\tikz \datavisualization [school book axes,
    visualize as smooth line,
    y axis={ticks={step=0.5}}]
  data [format=function] {
    var x : interval [0:2];
    func y = \value x*\value x/2;
  };
```

/tikz/data visualization/minor steps between steps=⟨*number*⟩ (default 9)

设置两个主刻度线之间的副刻度线的数目。本选项的默认值是 9, 即把相邻两个主刻度线之间的线段分成 10 小段。对于 linear steps 策略来说, 副刻度线会等分两个主刻度线之间的区间。对于 exponential steps 策略来说，副刻度线仍然是按等比间隔分布的，是疏密不均的。




```
\begin{tikzpicture}
\datavisualization [school book axes,
    visualize as smooth line,
    x axis={ticks={minor steps between steps=3}},
    y axis={ticks={minor steps between steps}}]
  data [format=function] {
    var x : interval [-1.5:1.5];
    func y = \value x*\value x;
  };
\end{tikzpicture}
```

/tikz/data visualization/phase=⟨*value*⟩ (no default, initially 0)

设置 linear steps 策略和 exponential steps 策略的 phase.

给定变量值集区间 $[a,b]$, 选项 step 和 phase 的值，就把主刻度线决定了，但这些刻度线未必处于“好”的位置上，也就是说，刻度值未必是简洁或者是利于读图的值。

82.4.4 计算刻度线和网格线位置的自动机制

以下针对刻度所做的介绍，对网格线也是有效的。

手工设置刻度的 step 和 phase 的值可能导致“不好”的刻度线位置，为了避免这一点，可以使用自动机制。将下面的选项作为 ticks= 的参数。

/tikz/data visualization/about=⟨*number*⟩ (no default)

这个选项意思是“设置大约 ⟨*number*⟩ 个主刻度线”，实际得到的主刻度线可能少于也可能多于 ⟨*number*⟩ 个，实际的个数决定于计算过程，计算过程依赖于轴所选择的自动计算刻度的策略。

linear steps 策略下的计算 假设需要可视化的区间是 $[a,b]$，首先将该区间按整数 ⟨*number*⟩ 做等分，得到步长 $s=m\cdot 10^k$，$1\leqslant m\leqslant 10$，$k\in\mathbb{Z}$，注意 s 的写法。由于 phase 的初始值是 0，因此确定了步长就确定了主刻度线的个数，但这样得到刻度值未必足够“好”。一般而言 m 决定了步长 s 是否为一个简洁

的数字，如果 m 不够简洁，就需要选择一种机制将 m 换成另外一个比较简洁的数字 m'. 有数种预定义的机制可供使用，默认的机制是“`standard about strategy`”，此机制下 m' 的值只允许是 1, 2, 2.5, 5 这四个值中的某一个。也可以用下一个选项自定义一种机制:

`/tikz/data visualization/about strategy`=⟨*list*⟩ (no default)

⟨*list*⟩ 是个逗号分隔的列表，列表项的形式是 ⟨*threshold*⟩/⟨*value*⟩, 斜线“/”之前的值是“界限值”，斜线之后的值是该界限值对应的转换值。找到与 m 最接近且大于 m 的那个界限值，把 m 换成相应的转换值，从而得到 m' 的值。举例说，给某个轴加上如下选项:

```
about strategy={1.5/1.0,2.3/2.0,4/2.5,7/5,11/10}
```

假如 $m = 3.141$，那么在列表中与 3.141 最接近且大于 3.141 的那个界限值是 4，界限值 4 对应的转换值是 2.5， 于是 $m' = 2.5$；对于 $m = 6.3$， 得到 $m' = 5$.

exponential steps 策略下的计算 假设需要可视化的区间是 $[a, b]$, 将该区间对数化为 $[\lg a, \lg b]$，将对数化后的区间按整数 ⟨*number*⟩ 做等分，得到步长 $s = m \cdot 10^k$， $1 \leqslant m \leqslant 10$， $k \in \mathbb{Z}$，注意 s 的写法。有了这个步长就可以得到区间 $[\lg a, \lg b]$ 内的刻度值，然后再以 10 为底做指数运算，得到原来区间 $[a, b]$ 内的刻度值，因此如果 s 是小数，例如 $s = 2.5$，就有可能得到无理数刻度值 $10^{2.5} = 100 \cdot \sqrt{10}$，显然不够简洁。因此在 exponential steps 策略下没有一种自动计算并转换步长的机制，而是使用一种固定的机制来把 m 换成 m', 而 m' 的值只允许是 1, 3, 6, 10 这四个值中的某一个。因此在这个策略下，选项 `about strategy=`⟨*list*⟩ 会被忽略。

linear steps 策略下预定义的计算刻度线的机制 下面的预定义的计算刻度线的机制是针对 linear steps 策略的。

`/tikz/data visualization/standard about strategy` (no value)

这是默认的机制，m' 的值只允许是 1, 2, 2.5, 5 这四个值中的某一个。

`/tikz/data visualization/euro about strategy` (no value)

字面意思是欧元硬币（Euro coins）面值，m' 的值只允许是 1, 2, 5 这三个值中的某一个。

`/tikz/data visualization/half about strategy` (no value)

m' 的值只允许是 1, 5 这两个值中的某一个。

`/tikz/data visualization/decimal about strategy` (no value)

m' 的值只允许是 1.

`/tikz/data visualization/quarter about strategy` (no value)

m' 的值只允许是 1, 2.5, 5 这三个值中的某一个。

`/tikz/data visualization/int about strategy` (no value)

m' 的值只允许是 1, 2, 3, 4, 5 这五个值中的某一个。

/tikz/data visualization/many (no value)

是 about=10 的简写。

/tikz/data visualization/some (no value)

是 about=5 的简写。

/tikz/data visualization/few (no value)

是 about=3 的简写。

/tikz/data visualization/none (no value)

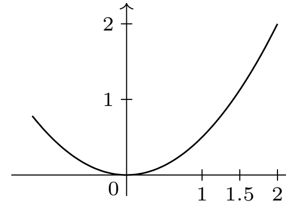
取消步长计算，不会在坐标轴上自动添加刻度（包括刻度线和刻度值），除非另用 step= 显式地做设置。

82.4.5 手工确定刻度线和网格线的位置

刻度和网格有主、副、次副三个等级，可以手工方式来设置它们。

/tikz/data visualization/major=⟨*options*⟩ (no default)

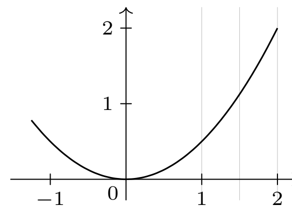
这个选项可用做 ticks 或 grid 的选项，指定主刻度（网格线）的位置，在 ⟨*options*⟩ 中使用选项 at= 或 also at= 来指定主刻度的位置。




```
\tikz \datavisualization [ school book axes,
    visualize as smooth line,
    x axis={ticks={major={at={1, 1.5, 2}}}}]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

/tikz/data visualization/minor=⟨*options*⟩ (no default)

这个选项可用做 ticks 或 grid 的选项，指定副刻度（网格线）的位置。



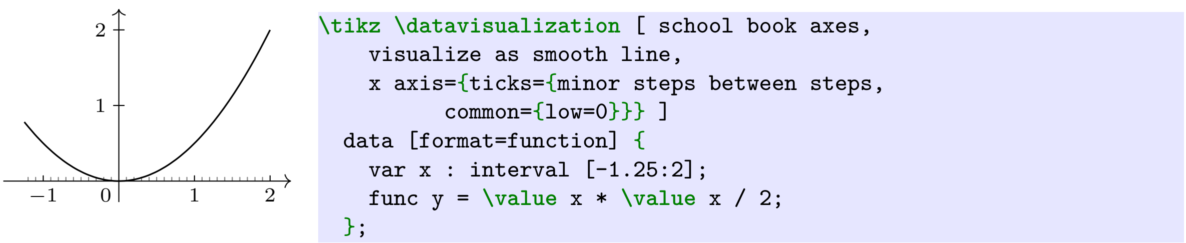

```
\tikz \datavisualization [ school book axes,
    visualize as smooth line,
    x axis={grid={minor={at={1, 1.5, 2}}}}]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

/tikz/data visualization/subminor=⟨*options*⟩ (no default)

这个选项可用做 ticks 或 grid 的选项，指定次副刻度（网格线）的位置。

/tikz/data visualization/common=⟨*options*⟩ (no default)

这个选项可用做 `ticks` 或 `grid` 的选项，其中的 ⟨*options*⟩ 对 `major`，`minor`，`subminor` 这 3 个 key 都有效，因此 ⟨*options*⟩ 中不能使用位置选项 `at=` 或 `also at=`.



```
\tikz \datavisualization [ school book axes,
    visualize as smooth line,
    x axis={ticks={minor steps between steps,
          common={low=0}}} ]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

以上 `major`，`minor`，`subminor`，`common` 选项中可以带有设置外观的选项,例如 `/tikz/data visualization/styl` `/tikz/data visualization/low`→P. 451, `/tikz/data visualization/high`→P. 452 等, 也可带有位置选项，如下。

下面的位置选项 `at=` 或 `also at=` 指定主（副、次副）刻度（网格线）的位置。

`/tikz/data visualization/at=`⟨*list*⟩ (no default)

⟨*list*⟩ 是个列表，将被宏 `\foreach` 处理，因此列表的形式比较灵活，其中可以使用省略号来构造等差项。空列表将被忽略。

```
\tikz \datavisualization [ school book axes,
    visualize as smooth line,
    x axis={ticks={major={at={-1,0.5,(pi/2)}}}}]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

如果用这个选项给出主（副、次副）刻度（网格线）的位置列表，那么此选项之前的以各种（自动或手工）方式确定的主（副、次副）刻度（网格线）的位置将被压制。使用该选项的一般句法是，例如，对于主刻度：

```
⟨axis name⟩= {ticks = {major = {at = {...}}}}
```

列表中的数字可以带有指令 `as` 来决定该数字的外观，例如 `1.2 as{$x$}`，那么刻度值 1.2 将被显示为数学模式下的字母 x，而不是一个数字。注意 `as` 是指令，其后的花括号里是该指令的参数。

```
\tikz \datavisualization [ school book axes,
    visualize as smooth line,
    x axis={ticks={major={at={-1,0.5,
          (pi/2)as{$\frac{\pi}2$}}}}}]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

关于本选项，有以下稍微简捷一些的形式：

`/tikz/data visualization/major at=`⟨*list*⟩ (no default)

这个选项是 `major={at={`⟨*list*⟩`}}` 的简写。

/tikz/data visualization/minor at=⟨*list*⟩ (no default)

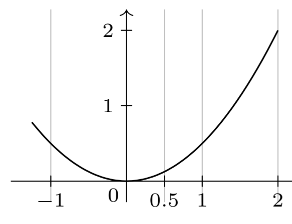
这个选项是 minor={at={⟨*list*⟩}} 的简写。

/tikz/data visualization/subminor at=⟨*list*⟩ (no default)

这个选项是 subminor={at={⟨*list*⟩}} 的简写。

/tikz/data visualization/also at=⟨*list*⟩ (no default)

这个选项“添加”某些主（副、次副）刻度（网格线）位置，并不清除此选项之前的以各种（自动或手工）方式确定的主刻度（网格线）的位置，因此如果多次使用该选项添加位置，那么这些位置会累计并显示在图形中。但是如果在 also at= 之后使用 at=，那么只有 at= 设置的位置会被显示。




```
\tikz \datavisualization [ school book axes,
    visualize as smooth line,
    x axis={grid, ticks and grid={
            major={also at={0.5}}}}]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

关于本选项，也有以下稍微简捷一些的形式：

/tikz/data visualization/major also at=⟨*list*⟩ (no default)

这个选项是 major={also at={⟨*list*⟩}} 的简写。

/tikz/data visualization/minor also at=⟨*list*⟩ (no default)

这个选项是 minor={also at={⟨*list*⟩}} 的简写。

/tikz/data visualization/subminor also at=⟨*list*⟩ (no default)

这个选项是 subminor={also at={⟨*list*⟩}} 的简写。

82.4.6 刻度与网格线的样式：概略

82.4.7 刻度与网格线的样式：style 与 node Style

下面介绍如何设置刻度线、刻度值、网格线的外观样式。

刻度值标签是作为 node 标签来创建的。

数据可视化系统中的 key 都有前缀 /tikz/data visualization, 一般情况下以 /tikz/ 开头的 key 不能直接用在可视化命令中。Tikz 中有许多针对形状、旋转、颜色、透明度等外观做设置的 key ，这些 key 可以通过间接的办法在可视化图形中起作用，例如通过 info, info' 指令，还可以通过下面的 style, node style 等选项。

/tikz/data visualization/style=⟨*TikZ options*⟩ (no default)

⟨*TikZ options*⟩ 中是以 /tikz/ 开头的 key, 这个 style 选项可以多次使用，效果累加。这些 ⟨*TikZ options*⟩ 中选项不会被立即执行，而是在可视化引擎调用 TikZ 后才会被执行，或者用下面的 styling 选项使得它们立即被执行。这个选项可用于设置刻度或网格线的外观，例如：

```
ticks={style=blue} 所有刻度值标签（不包括刻度线）是蓝色
ticks={major={style=blue}} 所有主刻度线、主刻度值都是蓝色
grid={style=red} 所有网格线都是红色
grid={major={style=red}} 主网格线是红色
grid={major={low=0.5, high=2}} 主网格线的起止位置，参考 \S77.5.2.
grid={major={style=red},   major also at={65 as[style=cyan]}} 画一条特殊的主网格线
ticks={major also at={65 as[low=-1.5em,style=cyan]$K$}}} 画一个特殊的主刻度线和主刻度
↪ 值
```

下面例子中，用 style 设置 y 轴的主网格线的颜色以及刻度值标签的颜色：


```
\tikz \datavisualization [scientific axes,
    all axes={ticks={style=blue}, length=3cm},
    y axis={grid, grid={minor steps between steps,
            major={style=red}}},
    visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

/tikz/data visualization/styling (no value)

这个选项只能直接用作命令 \datavisualization 的选项。当 style 选项调用 ⟨*TikZ options*⟩ 时，如果要让这些 ⟨*TikZ options*⟩ 被立即执行，就可以给 \datavisualization 命令带上这个选项，一般情况下不必使用这个选项。

/tikz/data visualization/node style=⟨*TikZ options*⟩ (no default)

这个选项类似 style= 选项，这个选项设置刻度值标签的外观。对于预定义的坐标系统，在默认下，下横轴的刻度值标签位于刻度线的下端点之下，也就是说，作为 node 的刻度值标签的锚定点是刻度线的下端点，标签的 anchor 位置是 north; 上横轴的刻度值标签位于刻度线的上端点之上，即作为 node 的刻度值标签的锚定点是刻度线的上端点，标签的 anchor 位置是 south; 左纵轴的刻度值标签位于刻度线的左端点之左，右纵轴的刻度值标签位于刻度线的右端点之右。标签的 anchor 位置可用本选项修改。


```
\tikz \datavisualization [scientific axes,
    all axes={ticks={node style=red}, length=3cm},
    visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

/tikz/data visualization/node styling (no value)

类似 styling 选项。

82.4.8 网格线的样式

当网格线被可视化时，下面的 key, styles, 或设置会被依次执行：

1. grid layer.
2. every grid.
3. every major grid 或 every minor grid 或 every subminor grid.
4. 针对单条网格线的特别设置。
5. styling.

以上 key, style 的路径都以 /tikz/data visualization/ 为前缀。保存在 grid layer 和 styling 中的选项路径都以 /tikz/ 为前缀。

/tikz/data visualization/grid layer (style, initially on background layer)

这个选项是个样式（style），其中所存储的样式会被冠以前缀 /tikz/ 来执行。这个选项指定网格线所在的“层”，默认是背景层，也就是说，图形会遮挡网格线。通常只需要将这个样式空置或者设置为 on background layer. 在 backgrounds 库中如下定义选项 on background layer:

```
% Layers

\pgfdeclarelayer{background}%
\pgfsetlayers{background,main}%


% Switch command
\tikzset{on background layer/.style={
    execute at begin scope={%
      \pgfonlayer{background}%
      \let\tikz@options=\pgfutil@empty%
      \tikzset{every on background layer/.try,#1}%
      \tikz@options},
    execute at end scope={\endpgfonlayer}
  }
}%
```

可见选项 on background layer 还可以有自己的参数。

文件《tikzlibrarydatavisualization.code.tex》会调用 backgrounds 库。

下面的例子中重设网格线所在的层，但是没有指定是哪个层，就把网格线改到了 main 层：

```
\tikz \datavisualization [scientific axes,
    all axes={length=3cm, grid,
        grid={minor steps between steps}},
    grid layer/.style=, % 这导致网格线遮挡图形
    visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

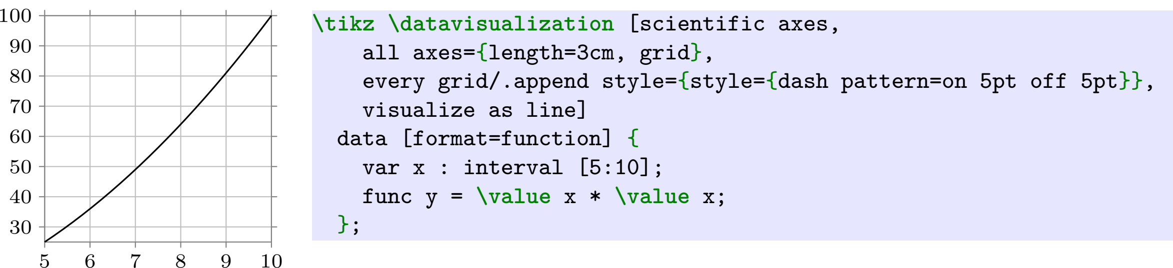
/tikz/data visualization/every grid (style, no value)

这个样式对所有网格线作设置，默认值是：

```
every grid/.style={high=max,low=min},
```

选项 /tikz/data visualization/low →P. 451, /tikz/data visualization/high →P. 452 将在后文介绍。

可以在这个选项样式中使用 style= 选项来调用 TikZ 的 key：




```
\tikz \datavisualization [scientific axes,
    all axes={length=3cm, grid},
    every grid/.append style={style={dash pattern=on 5pt off 5pt}},
    visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

上面例子中的 style={dash pattern=on 5pt off 5pt} 没有起作用，是因为主刻度线有自己的默认设置，见下面的 every major grid, 并且 every major grid 在 every grid 之后被执行。

/tikz/data visualization/every major grid (style, no value)

这个选项对所有的主网格线作设置，对于使用 new axis base 声明的轴来说，它的默认是：

```
style = {solid, help lines, thin, black!25}
```

可以在这个选项样式中使用 style= 选项来调用 TikZ 的 key：

```
\tikz \datavisualization [scientific axes,
    all axes={length=3cm},
    y axis={grid, grid={minor steps between steps=1}},
    every major grid/.style = {style={blue, thin, densely dashed}},
    every minor grid/.style = {style={densely dotted}},
    visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

/tikz/data visualization/every minor grid (style, no value)

这个选项对所有的副网格线作设置，对于使用 new axis base 声明的轴来说，它的默认是：

```
style = {solid, help lines, thin, black!25}
```

可以在这个选项样式中使用 style= 选项来调用 TikZ 的 key.

/tikz/data visualization/every subminor grid (style, no value)

这个选项对所有的次副网格线作设置，对于使用 new axis base 声明的轴来说，它的默认是：

```
style = {solid, help lines, thin, black!10}
```

可以在这个选项样式中使用 style= 选项来调用 TikZ 的 key.

82.4.9 刻度线与刻度值标签的样式

当添加刻度线时，下面的 key, styles 会被依次执行：

1. every ticks.

2. every major ticks 或 every minor ticks 或 every subminor ticks.
3. 针对某个刻度线的特别设置。
4. tick layer.
5. every odd tick 或 every even tick.
6. draw.
7. styling.

当添加刻度值标签时，下面的 key, styles 会被依次执行:

1. every ticks.
2. every major ticks 或 every minor ticks 或 every subminor ticks.
3. 针对某个刻度值标签的特别设置。
4. tick node layer.
5. every odd tick 或 every even tick.
6. styling.
7. node styling.

/tikz/data visualization/every ticks (style, no value)

这个样式对所有的刻度线和刻度值标签作设置，可以在这个样式中使用选项 style=, node style= 来分别设置刻度线和刻度值标签。刻度值标签的默认设置是:

```
node style={
  font=\footnotesize,
  inner sep=1pt,
  outer sep=.1666em,
  rounded corners=1.5pt
}
```

/tikz/data visualization/every major ticks (style, no value)

这个样式对所有的主刻度线和主刻度值标签作设置，可以在这个样式中使用选项 style=, node style= 来分别设置主刻度线和主刻度值标签。主刻度线的默认设置是:

```
style={line cap=round}, tick length=2pt
```

/tikz/data visualization/every minor ticks (style, no value)

这个样式对所有的副刻度线作设置，可以在这个样式中使用选项 style= 设置副刻度线，默认设置是:

```
style={help lines,thin, line cap=round}, tick length=1.4pt
```

/tikz/data visualization/every subminor ticks (style, no value)

这个样式对所有的次副刻度线作设置，可以在这个样式中使用选项 style= 设置次副刻度线，默认设置是:

```
style={help lines, line cap=round}, tick length=0.8pt
```

`/tikz/data visualization/tick layer` (style, initially on `background layer`)

类似 `grid layer` 选项，指定刻度线所在的“层”，这个样式中的选项被冠以 `/tikz/` 来执行。

`/tikz/data visualization/tick node layer` (style, initially empty)

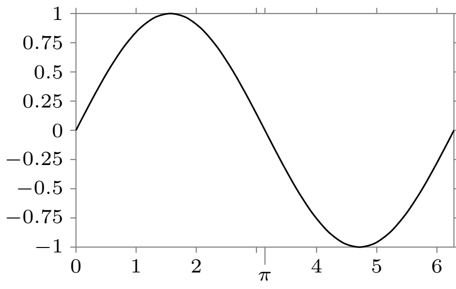
类似 `grid layer` 选项，指定刻度值标签所在的“层”，这个样式中的选项被冠以 `/tikz/` 来执行。默认把 node 刻度标签放在 main 层。

82.4.10 设置个别刻度的样式

有时候图形中的某些点有特殊的意义，这些点对应的刻度值或刻度线需要特别设置，此时可以用下面的选项:

`/tikz/data visualization/options at=`⟨*value*⟩ `as [`⟨*options*⟩`]` (no default)

⟨*value*⟩ 指定坐标轴上相应的刻度位置，`as` 是个指令，将 ⟨*options*⟩ 添加到这些位置上的刻度线或刻度值标签中来执行，改变刻度线或刻度值标签的外观。注意，如果 ⟨*options*⟩ 中有逗号，则需要用花括号把 ⟨*options*⟩ 括起来：⟨*value*⟩ `as [{`⟨*options*⟩`}]`.




```
\tikz \datavisualization [scientific axes,
    visualize as smooth line,
    x axis={ticks={major={
      options at = 3 as [no tick text],
      also at = (pi) as
        [{tick text padding=1ex}] $\pi$}}}]
  data [format=function] {
    var x : interval[0:2*pi];
    func y = sin(\value x r);
  };
```

`/tikz/data visualization/no tick text at=`⟨*value*⟩ (no default)

这是 `options at=`⟨*value*⟩ `as [no tick text]` 的简写。

82.4.11 其它刻度值标签选项

在 `/tikz/data visualization/at`→P. 435=⟨*list*⟩ 或 `/tikz/data visualization/also at`→P. 436=⟨*list*⟩ 的 ⟨*list*⟩ 中可以使用下面的句法:

⟨*value*⟩ ⟨*value*⟩ `as [`⟨*local options*⟩`]` ⟨*value*⟩ `as [`⟨*local options*⟩`]` ⟨*text*⟩

当使用 `at={`⟨*value*⟩`...}`，`also at={`⟨*value*⟩`...}` 指定刻度位置时，在 ⟨*value*⟩ 后加 `as` 指令，可以改变相应的刻度值标签的样式。⟨*local options*⟩ 是针对标签样式的选项。如果 `as` 指令之后有 ⟨*text*⟩, 则标签将显示为 ⟨*text*⟩. 注意，如果 ⟨*local options*⟩ 中有逗号，则需要用花括号把 ⟨*local options*⟩ 括起来：⟨*value*⟩ `as [{`⟨*local options*⟩`}]` ⟨*text*⟩.

⟨*local options*⟩ 中可用选项 `style=`，`node style=` 来分别设置该位置处的刻度线、刻度值标签、网格线的样式，可以用 `low`, `high` 选项。

```
\tikz \datavisualization [scientific axes=clean,
  x axis={length=2.5cm, ticks={major at={
    5,
    6 as [style=red],
    7 as [{style=blue, low=-1em}],
    8 as [style=green] $2^3$,
    10 as ten
    }}},
  visualize as line]
data [format=function] {
  var x : interval [5:10];
  func y = \value x * \value x;
};
```

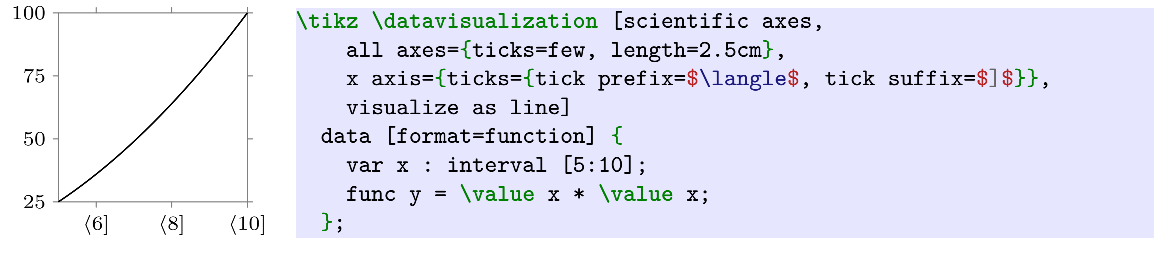
```
\tikz \datavisualization [scientific axes,
  all axes={ticks={major={style=blue}}, length=3cm},
  y axis={grid,
    grid={major={style=red}, major also at={65 as[style=cyan]}},
    ticks={major also at={65 as[{low=-1.5em,style=cyan}]$K$}}},
  visualize as line]
data [format=function] {
  var x : interval [5:10];
  func y = \value x * \value x;
};
```

如果数值 0.0000000015 作为刻度值标签，一般情况下，它会被显示为 $1.5 \cdot 10^{-9}$，也就是说，可视化系统并不是简单地将数值排版为文本标签，而是先将标签数值传递给一个 typesetter，它会利用 TeX 的排版功能来编辑标签，并将标签文字置于数学模式中，除非使用 `as[...]`⟨*text*⟩ 来指定标签文本。

刻度值标签是个 node, 其内容通常有 3 个部分：前缀、标签文本（如某种格式的数值）、后缀（例如物理单位），这三个部分会被依次放入 node 标签的内容中，一起构成刻度值标签。这三个部分分别由下面的选项指定。

`/tikz/data visualization/tick prefix=`⟨*text*⟩ (no default, initially empty)

这个选项设置刻度值标签的前缀。




```
\tikz \datavisualization [scientific axes,
    all axes={ticks=few, length=2.5cm},
    x axis={ticks={tick prefix=$\langle$, tick suffix=$]$}},
    visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

`/tikz/data visualization/tick suffix=`⟨*text*⟩ (no default, initially empty)

这个选项设置刻度值标签的后缀。

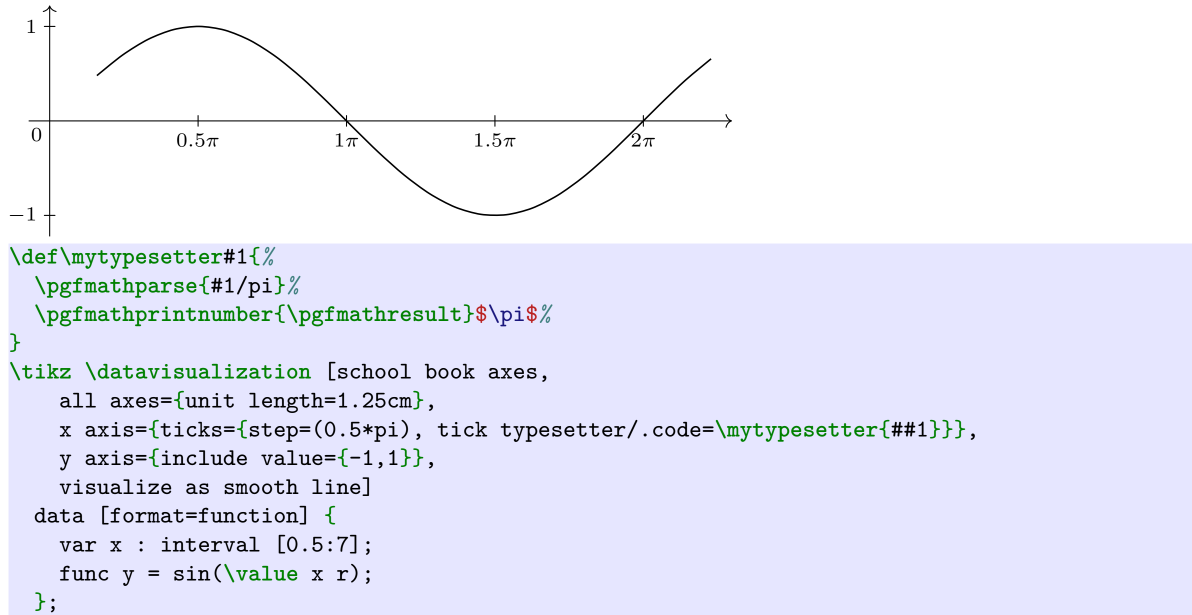
`/tikz/data visualization/tick unit=`⟨*roman math text*⟩ (no default)

这是 `tick suffix={$\rm `⟨*roman math text*⟩`$}` 的简写，⟨*roman math text*⟩ 处于数学模式中。

```
\tikz \datavisualization [scientific axes,
    all axes={length=3cm},
    x axis={ticks={tick unit=s}},
    y axis={ticks={tick unit=m/s^2}},
    visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

/tikz/data visualization/tick typesetter⟨*value*⟩ (no default)

用于定义数值标签编辑器 typesetter，在默认下该选项使用命令 \pgfmathprintnumber 将数值标签显示为科学计数法格式。下面的例子展示了如何自定义 `tick typesetter`.

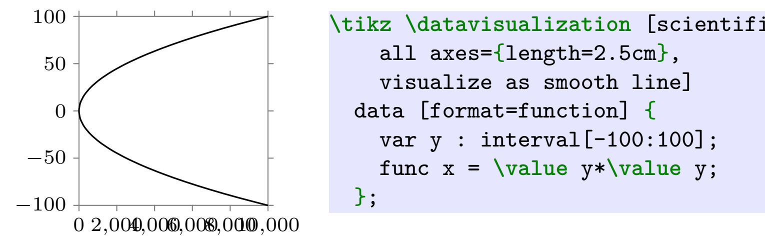



```
\def\mytypesetter#1{%
  \pgfmathparse{#1/pi}%
  \pgfmathprintnumber{\pgfmathresult}$\pi$%
}
\tikz \datavisualization [school book axes,
    all axes={unit length=1.25cm},
    x axis={ticks={step=(0.5*pi), tick typesetter/.code=\mytypesetter{##1}}},
    y axis={include value={-1,1}},
    visualize as smooth line]
  data [format=function] {
    var x : interval [0.5:7];
    func y = sin(\value x r);
  };
```

注意上面例子中，`step=(0.5*pi)` 一定要带有圆括号，因为 `0.5*pi` 要由 \pgfmathparse 处理。

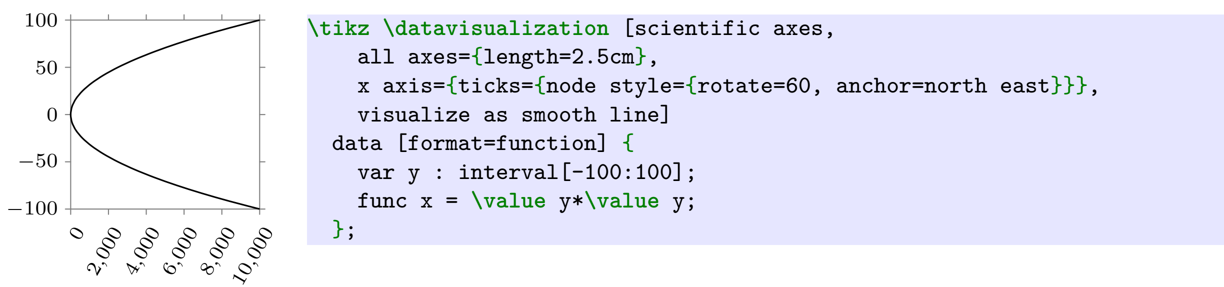
82.4.12 交错叠放刻度值标签

有时候刻度值标签过长以致发生重叠，例如



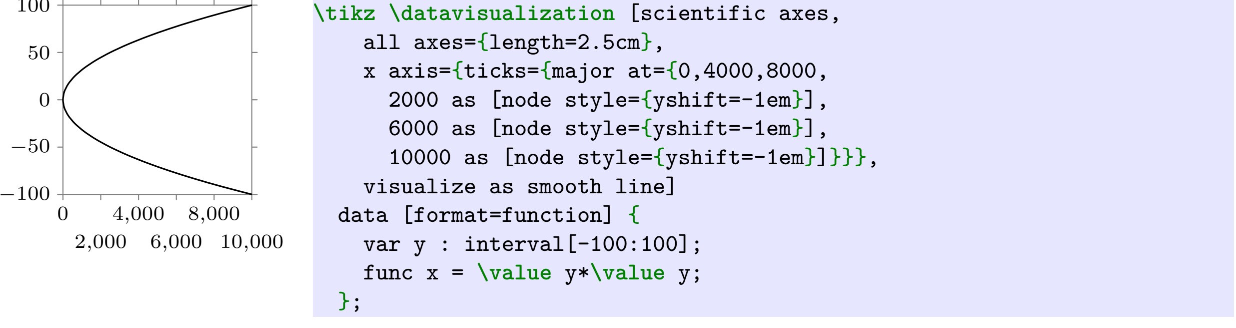

```
\tikz \datavisualization [scientific axes,
    all axes={length=2.5cm},
    visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

解决这种问题的办法有两个，一个是旋转标签：



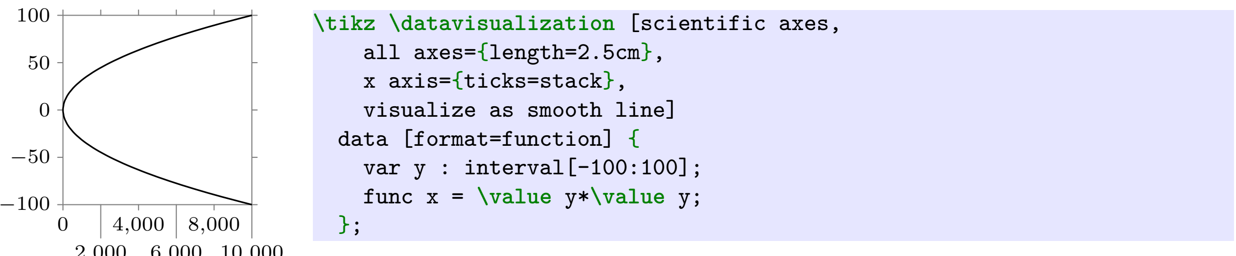
```
\tikz \datavisualization [scientific axes,
    all axes={length=2.5cm},
    x axis={ticks={node style={rotate=60, anchor=north east}}},
    visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

另一个办法是将标签交错叠放：



```
\tikz \datavisualization [scientific axes,
    all axes={length=2.5cm},
    x axis={ticks={major at={0,4000,8000,
      2000 as [node style={yshift=-1em}],
      6000 as [node style={yshift=-1em}],
      10000 as [node style={yshift=-1em}]}}},
    visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

实现这种叠放效果的便捷办法是使用 `stack` 选项：



```
\tikz \datavisualization [scientific axes,
    all axes={length=2.5cm},
    x axis={ticks=stack},
    visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

在交错叠放时，刻度值标签按照奇数、偶数位置交错叠放。程序会考虑所有的主刻度、副刻度、次副刻度，按它们在代码中出现的次序来确定刻度的奇数、偶数位置，而不是按照它们在坐标轴上的左右（上下）位置来确定刻度的奇偶位置。例如，

```
ticks={major at={1,2,3,4}, major also at={0,-1,-2}, minor at={9,8,7}}
```

其中的 $1, 3, 0, -2, 9, 7$ 处于奇数位置。

当刻度值标签偏移后，相应的刻度线也会变长。

在绘图时，在图形的上部和下部都可能出现横轴，上下横轴都可能带有刻度。横轴刻度线的下端点对应单词“`low`”，刻度线的上端点对应单词“`high`”；对于预定义的坐标系统，默认下横轴的刻度值标签位于刻度线的下端点之下，也就是说，作为 node 的刻度值标签的锚定点是刻度线的下端点，标签的 anchor 位置是 `north`; 上横轴的刻度值标签位于刻度线的上端点之上，即作为 node 的刻度值标签的指向点是刻度线的上端点，标签的 anchor 位置是 `south`.

图形中也可能会有左纵轴和右纵轴，情况是类似的。纵轴刻度线的左端点对应单词“`low`”，右端点对应单词“`high`”；对于预定义的坐标系统，默认左纵轴的刻度值标签位于刻度线的左端点之左，标签的 anchor 位置是 `east`; 右纵轴的刻度值标签位于刻度线的右端点之右，标签的 anchor 位置是 `west`.

横轴刻度线的下端点相对于横轴的偏移尺寸由 `low=`⟨*dimension*⟩ 确定；横轴刻度线的上端点相对于横

轴的偏移尺寸由 `high` = ⟨*dimension*⟩ 确定。纵轴刻度线的左端点相对于纵轴的偏移尺寸由 `low=`⟨*dimension*⟩ 确定；纵轴刻度线的右端点相对于纵轴的偏移尺寸由 `high` = ⟨*dimension*⟩ 确定。

`/tikz/data visualization/tick text low even padding`=⟨*dimension*⟩ (no default, initially `0pt`)

这个选项针对下横轴或者左纵轴，因为这两个轴的刻度值标签处于刻度线的“`low`”端点的 low 侧。该选项使得偶数位置刻度线的“`low`”端点相对于轴的偏移尺寸“增加”⟨*dimension*⟩; 这会使得偶数位置的刻度值标签随之偏移，从而与奇偶位置的刻度值标签发生错位。例如对于下横轴来说，若 ⟨*dimension*⟩ 是足够大的带单位的负值尺寸，则该选项使得偶数位刻度值标签处于奇数刻度值标签之下。如果刻度还有 `low=` 选项，则 ⟨*dimension*⟩ 会叠加到 `low=` 的值上。

100
50
0
−50
−100
0 4,000 8,000
2,000 6,000 10,000

```
\tikz \datavisualization [scientific axes,
  all axes={length=2.5cm},
  x axis={ticks={tick text low even padding=-1em}},
  visualize as smooth line]
data [format=function] {
  var y : interval[-100:100];
  func x = \value y*\value y;
};
```

`/tikz/data visualization/tick text low odd padding`=⟨*dimension*⟩ (no default, initially `0pt`)
`/tikz/data visualization/tick text high even padding`=⟨*dimension*⟩ (no default, initially `0pt`)

这个选项针对上横轴或右纵轴，因为这两个轴的刻度值标签处于刻度线的“`high`”端点的 high 侧。该选项指定轴的偶数位置刻度线的“`high`”端点相对于轴的偏移尺寸，同时使得偶数位置的刻度值标签随之偏移，从而与奇偶位置的刻度值标签发生错位。

`/tikz/data visualization/tick text high odd padding`=⟨*dimension*⟩ (no default, initially `0pt`)
`/tikz/data visualization/tick text odd padding`=⟨*dimension*⟩ (no default)

同时设置 `tick text odd low padding` 和 `tick text odd high padding`.

`/tikz/data visualization/tick text even padding`=⟨*dimension*⟩ (no default)

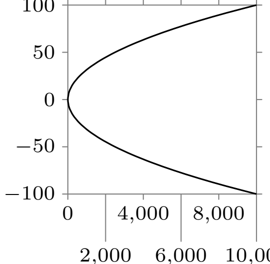
同时设置 `tick text even low padding` 和 `tick text even high padding`.

`/tikz/data visualization/tick text padding`=⟨*dimension*⟩ (no default)

同时设置所有刻度值标签的 text padding, 相当于给（有刻度值标签的）刻度线附加一个长度尺寸，改变刻度线的长短。

`/tikz/data visualization/stack`=⟨*dimension*⟩ (default `1em`)

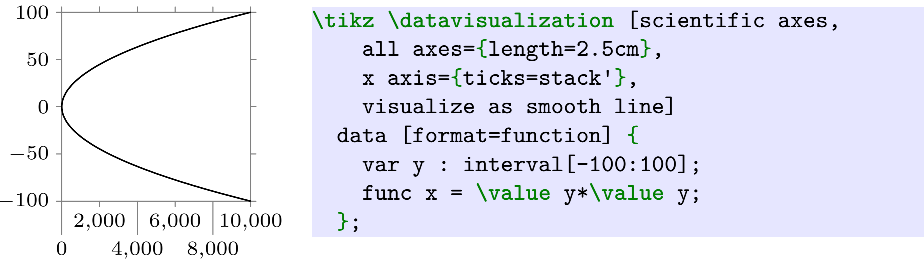
这是 `tick text even padding=`⟨*dimension*⟩ 的简写。




```
\tikz \datavisualization [scientific axes,
    all axes={length=2.5cm},
    x axis={ticks={stack=1.5em}},
    visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

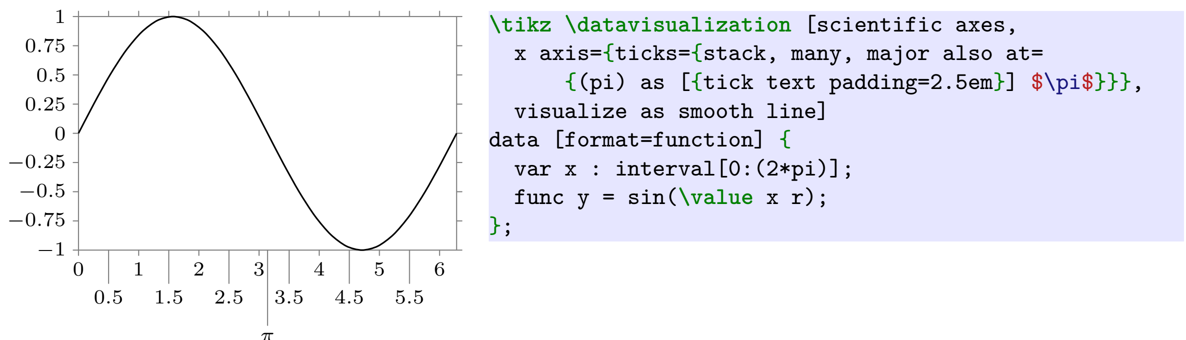
`/tikz/data visualization/stack'`=⟨*dimension*⟩ (no default)

这是 `tick text odd padding=`⟨*dimension*⟩ 的简写。




```
\tikz \datavisualization [scientific axes,
    all axes={length=2.5cm},
    x axis={ticks=stack'},
    visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

用选项 `at`，`also at` 设置的刻度值标签也可以有交错叠放效果：




```
\tikz \datavisualization [scientific axes,
  x axis={ticks={stack, many, major also at=
      {(pi) as [{tick text padding=2.5em}] $\pi$}}},
  visualize as smooth line]
data [format=function] {
  var x : interval[0:(2*pi)];
  func y = sin(\value x r);
};
```

82.4.13 自动添加刻度的策略

有两种预定义的自动添加刻度的策略：linear steps 和 exponential steps.

`/tikz/data visualization/axis options/linear steps` (no value)

这是默认的策略。

`/tikz/data visualization/axis options/exponential steps` (no value)

82.4.14 定义新的添加刻度的策略

82.5 创建新的轴系统

首先注意，一个轴系统并不直接与数据的变化范围相关联，也没有刻度、网格线等可视化内容，数据范围、刻度等内容都是用其它指令或选项添加到轴系统中的。

创建一个新的轴系统并将它可视化，主要包括 4 方面：

1. 坐标轴的可视化。首先区分“轴”与“可视化的轴”。轴是一套处理过程，并不是可见的。以直角坐标系的横轴为例，假设这个轴对应一个变量 x, x 的取值范围 $[s_1, s_2]$ 就是该轴的范围。最初该轴是不可见的，打个比方，有个“显像器”可以决定横轴的哪一部分能够被可视化。在默认下，“显像器”将横轴可视化为一个有一定长度的线段，线段左端点对应变量 x 的最小值 s_1, 线段右端点对应变量 x 的最大值 s_2. 但是可以用某些选项来改变“显像器”的作用范围，仅仅把 $[s_1, s_2]$ 的某个

子区间 $[s'_1, s'_2] \subset [s_1, s_2]$ 可视化为一个线段，尽管 $[s_1, s_2] - [s'_1, s'_2]$ 不画出，但在图形中仍然占据空白位置，因为“显像器”不改变可视化数据区域。

自定义坐标轴的可视化由选项 `visualize axis` 辖制，涉及轴的显示范围、可视化轴的位置、颜色、箭头、线宽等内容。

2. 网格线的可视化。自定义坐标轴的网格线的可视化由选项 `visualize grid` 来辖制，该选项指定默认的网格线的外观，例如网格线的方向、起点、终点、线型、颜色、线宽等。在可视化绘图命令中，要画出网格线只需给轴使用 `grid` 选项，画出的网格线的外观会按选项 `visualize grid` 所指定的样式，也可以临时修改。
3. 刻度的可视化。自定义坐标轴的刻度包括刻度线和刻度值标签两方面，其可视化由选项 `visualize ticks` 来辖制，该选项指定默认的刻度线、标刻度签线的外观，例如刻度的方向、起点、终点、颜色、标签样式等。在可视化绘图命令中，要画出刻度只需给轴使用 `ticks` 选项，画出的刻度的外观会按选项 `visualize ticks` 所指定的样式，也可以临时修改。
4. 坐标轴标签的可视化。自定义坐标轴的标签的可视化由选项 `visualize label` 来辖制，该选项指定默认的轴标签的外观，例如标签的位置、样式等。在可视化绘图命令中，要画出轴标签只需给轴使用 `label` 选项，画出的刻度的外观会按选项 `visualize label` 所指定的样式，也可以临时修改。

在前文多处涉及选项 `low` 和 `high`, 这两个选项在不同的地方有不同的意义和作用，其意义主要有 3 个：

- 对轴的可视化来说，即用作选项 `visualize axis` 的参数时，`low` 对应可视化轴的起点，`high` 对应可视化轴的终点。此时二者的格式是 `low=`⟨*value*⟩, `high=`⟨*value*⟩, 其中 ⟨*value*⟩ 是该轴对应的变量值区间内的值。
- 对于网格线的可视化来说，即用作选项 `visualize grid` 的参数时，`low` 对应网格线的起点，`high` 对应网格线的终点。此时二者的格式是 `low=`⟨*value*⟩, `high=`⟨*value*⟩, 其中 ⟨*value*⟩ 是网格线的“方向轴”所对应的变量值区间内的值。
- 对于刻度线的可视化来说，即用作选项 `visualize ticks` 的参数时，`low` 对应刻度线的起点，`high` 对应刻度线的终点。此时二者的格式是 `low=`⟨*dimension*⟩, `high=`⟨*dimension*⟩, 其中 ⟨*dimension*⟩ 是带单位的正值或负值尺寸。

82.5.1 创建一个轴系统

下面通过一个例子来说明如何创建轴系统。下面例子中，轴系统是笛卡尔式的，名称是 `our system`，其中包括 3 个轴，一个水平轴 `x axis`，一个左侧纵轴 `left axis`，一个右侧纵轴 `right axis`.

一个轴系统是作为 key 被创建的，其前缀是 `/tikz/data visualization`，例如：

```
\tikzset{
  data visualization/our system/.style={
    ...
  }
}
```

这个句子创建一个名称为 `our system` 的 key, 在这个 key 中添加 3 个轴：

```
\tikzset{
  data visualization/our system/.style={
    new Cartesian axis=x axis,
```

```
    new Cartesian axis=left axis,
    new Cartesian axis=right axis,
    x axis={attribute=x},
    left axis={unit vector={(0cm,1pt)}},
    right axis={unit vector={(0cm,1pt)}},
  }
}
```

注意由 /tikz/data visualization/new Cartesian axis→P. 423 声明的轴是具有单位向量的,轴 x axis 的单位向量采用默认值 (1pt,0pt). 轴 left axis 和 right axis 的单位向量是 (0cm,1pt). 轴 x axis 对应的变量名称是 x. 轴 left axis 和 right axis 尚未对应变量。

再规定轴的长度，例如使用科学坐标系的轴长度:

```
x axis ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/width}},
left axis ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}
↪  },
right axis={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}}
```

其中 /tikz/data visualization/scientific axes/width 是科学坐标系的横轴的长度，初始值为 5cm；/tikz/data visualization/scientific axes/height 是科学坐标系的纵轴的高度，与横轴长度成黄金分割比。

通过以上设置就定义了一个轴系统，可以调用这个轴系统来画图了。其中各个轴对应的变量名称, 单位向量，轴长度等项目，都可以在绘图时做临时修改。

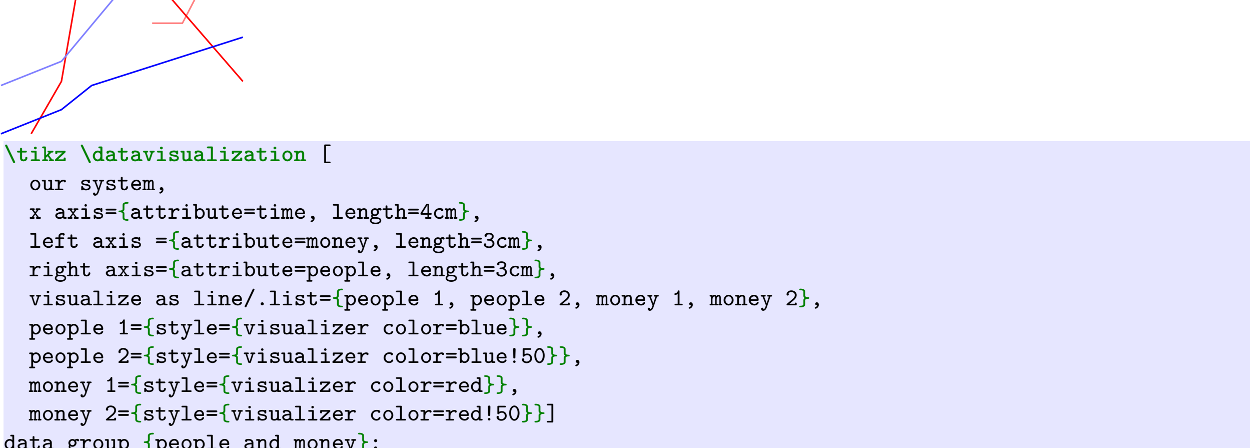
```
\tikzset{
  data visualization/our system/.style={
    new Cartesian axis=x axis,
    new Cartesian axis=left axis,
    new Cartesian axis=right axis,
    x axis={attribute=x, length=\pgfkeysvalueof{/tikz/data visualization/scientific
    ↪  axes/width}},
    left axis={unit vector={(0cm,1pt)}, length=\pgfkeysvalueof{/tikz/data
    ↪  visualization/scientific axes/height}},
    right axis={unit vector={(0cm,1pt)}, length=\pgfkeysvalueof{/tikz/data
    ↪  visualization/scientific axes/height}},
  }
}
```

下面再定义一个数据组:

```
\tikz \datavisualization data group {people and money} = {
  data [set=people 1] {
    time, people
    1900, 1000000000
    1920, 1500000000
    1930, 2000000000
    1980, 3000000000
  }
  data [set=people 2] {
```

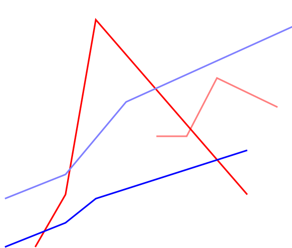
```
    time, people
    1900, 2000000000
    1920, 2500000000
    1940, 4000000000
    2000, 5700000000
  }
  data [set=money 1] {
    time, money
    1910, 1.1
    1920, 2
    1930, 5
    1980, 2
  }
  data [set=money 2] {
    time, money
    1950, 3
    1960, 3
    1970, 4
    1990, 3.5
  }
};
```

用上面代码定义的 `our system` 画出这个 `data group`，并且临时修改轴对应的变量名、轴的长度：



```
\tikz \datavisualization [
  our system,
  x axis={attribute=time, length=4cm},
  left axis ={attribute=money, length=3cm},
  right axis={attribute=people, length=3cm},
  visualize as line/.list={people 1, people 2, money 1, money 2},
  people 1={style={visualizer color=blue}},
  people 2={style={visualizer color=blue!50}},
  money 1={style={visualizer color=red}},
  money 2={style={visualizer color=red!50}}]
data group {people and money};
```

也可以将轴系统的定义放在命令 `\datavisualization` 的选项中：



```
\tikz \datavisualization [
  our system/.style={
    new Cartesian axis=x axis,
    new Cartesian axis=left axis,
    new Cartesian axis=right axis,
    left axis={unit vector={(0cm,1pt)}},
    right axis={unit vector={(0cm,1pt)}},
  },
  our system,
  x axis={attribute=time, length=4cm},
  left axis ={attribute=money, length=3cm},
  right axis={attribute=people, length=3cm},
  visualize as line/.list={people 1, people 2, money 1, money 2},
  people 1={style={visualizer color=blue}},
  people 2={style={visualizer color=blue!50}},
  money 1={style={visualizer color=red}},
  money 2={style={visualizer color=red!50}}]
data group {people and money};
```

82.5.2 坐标轴的可视化

如果要将坐标轴的某个要素可视化，例如坐标轴的位置，可视化区间，坐标轴的颜色、线型、线宽等（但不包括刻度、网格线），就必须将该要素作为选项 `visualize axis` 的参数，也就是说这个选项辖制轴的可视化内容，但是不辖制像 `axis layer`, `every axis` 这种 style 选项。

在一个可视化过程中，如果某个轴的选项中多次使用 `visualize axis` 选项，那么它们的效果会累计。

`/tikz/data visualization/axis options/visualize axis=`⟨*options*⟩ (no default)

这个 key 作为选项传递给轴，直接使得轴可视化。⟨*options*⟩ 中可以使用选项 `style=` 来设置可视化轴的外观，也可用下面的选项来决定轴画在什么位置以及轴的长度。轴的位置是利用两个轴的相对位置来确定的。例如对于前面的定义的轴系统 `our system` 来说，可以规定轴 `left axis` 与轴 `x axis` 的相对位置为：

```
left axis={ visualize axis={ x axis={ goto=min } }
```

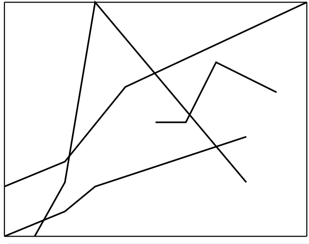
这个句子指定轴 `left axis` 经过轴 `x axis` 上刻度值为 `min` 的点，即轴 `x axis` 对应的变量值集的最小值点。

可视化坐标轴相对的位置，轴的可视化区间 如前述，轴的位置是相互确定的。

`/tikz/data visualization/axis options/goto=`⟨*value*⟩ (no default)

这个选项的作用如前述。⟨*value*⟩ 是个属于变量值集的数字，也可以是下述之一：

- `min`，轴所对应的变量值集的最小值点。
- `max`，轴所对应的变量值集的最大值点。
- `padded min`, 这是在 `min` 的基础上再附加一个偏移尺寸所确定的值，偏移尺寸由选项 `padding min =` ⟨*dimension*⟩ 确定，见后文。
- `padded max`, 类似 `padded min`.

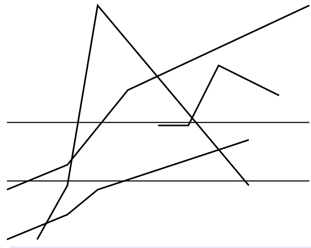


```
\tikzset{
  data visualization/our system/.append style={
    left axis= {visualize axis={x axis= {goto=min}}},
    right axis={visualize axis={x axis= {goto=max}}},
    x axis= {visualize axis={left axis={goto=min}},
        visualize axis={left axis={goto=max}}},
  }
}
\tikz \datavisualization [
  our system,
  x axis={attribute=time, length=4cm},
  left axis ={attribute=money},
  right axis={attribute=people},
  visualize as line/.list={people 1, people 2, money 1, money 2}]
data group {people and money};
```

上面的例子中，横轴 `x axis` 里面两次使用 `visualize axis` 选项，使得横轴画了两次，一次经过纵轴 `left axis` 的上端，一次经过纵轴 `left axis` 的下端。

`/tikz/data visualization/axis options/goto pos`=⟨*fraction*⟩ (no default)

这个选项类似 `goto`，如果设置 `scaling=min at c and max at d`，即"合理区间"是 `[c,d]`，那么本选项决定的位置是 $0+\langle fraction\rangle*(d-c)$，也就是说，⟨*fraction*⟩ 是针对合理区间来计算的，与变量值集无关. ⟨*fraction*⟩ 可以是任意小数，不必限于 0 到 1 之间。

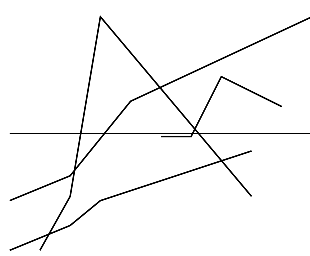


```
\tikzset{
  data visualization/our system/.append style={
    x axis= {visualize axis={left axis={goto pos=0.25}},
        visualize axis={left axis={goto pos=0.5}}},
  }
}
\tikz \datavisualization [
  our system,
  x axis={attribute=time, length=4cm},
  left axis ={attribute=money},
  right axis={attribute=people},
  visualize as line/.list={people 1, people 2, money 1, money 2}]
data group {people and money};
```

`/tikz/data visualization/low`=⟨*value*⟩ (no default)

这个选项用作 visualize axis 的参数，或者用于其它可视化过程中。“轴”与“可视化的轴”的区别在本节开头已经解释过了，在这里单词 low 指的是可视化轴线段的起点，⟨*value*⟩ 是该起点对应的变量值。本选项与 min value=⟨*value*⟩ 不同，本选项不会改变“合理区间”（即不改变可视化数据的范围），只是用来决定变量值区间的哪一部分能被可视化。

其中的 ⟨*value*⟩ 可以是 min, max, padded min, padded max 等, 也可以是某个变量值, 默认为 low=min. 继续沿用前面的例子，但是将 x axis 轴的显示区间由 [1900,2000] 改为 [1930,2000], 位置在纵轴下端之下 (goto pos=-0.15)，颜色为红色，如下所示:



```
\tikzset{
  data visualization/our system/.append style={
    x axis= {visualize axis={left axis={goto pos=-0.15}, style=red, low=1930},
          visualize axis={left axis={goto pos=0.5}}},
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=4cm},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

从上面的例子可见，对于横轴来说，low= 引起横轴的水平方向上可视化区间的左端点变化。类似地，对于纵轴来说，low= 引起纵轴的竖直方向上可视化区间的下端点变化。

选项 low= 还可以用于规定网格线、刻度线的起点。

/tikz/data visualization/high=⟨*value*⟩ (no default)

与 low 类似，只是决定轴的可视化部分（线段）的终点，默认为 high=max.

/tikz/data visualization/padded (no value)

该选项同时设置 low=padded min 和 high=padded max.

轴线 对于仿射坐标系，轴的可视化区间线段是利用 \pgfpathdvmoveto 和 \pgfpathdvlineto 来构造的。对于极坐标系，某一半径的角度轴是用 arc 构造的。

轴的样式 可以用 style= 选项设置轴的颜色、线型、箭头等外观，只需将 style={⟨*TikZ options*⟩} 放在 visualize axis={...} 中。此外还有以下两个样式选项可用:

/tikz/data visualization/axis layer (style, initially on background layer)

与 `grid layer` 类似，这个选项决定可视化的轴线段画在那个“层”上。因为这个选项针对所有的轴，所以只能直接用作命令 `\datavisualization` 的选项或者用在 `\tikzset{...}` 中，不受选项 `visualize axis` 的辖制，例如

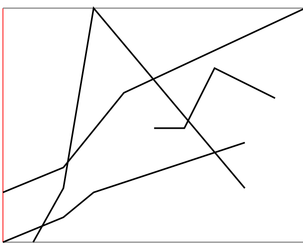
```
\tikzset{
  data visualization/our system/.append style={
    axis layer/.style=,
    ......
```

```
\tikzset{
  data visualization/our system/.append style={
    x axis= {visualize axis={left axis={goto pos=0.25}, style={red,line width=2pt}, low=1930},
      visualize axis={left axis={goto pos=0.5}}},
  }
}
\tikz \datavisualization [
    our system,
    axis layer/.style=, % 画出的坐标轴处于 main 层，遮挡图形
    x axis={attribute=time, length=4cm},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

`/tikz/data visualization/every axis` (style, no value)

用来设置某个轴的初始样式，直接用作命令 `\datavisualization` 的选项或者用在 `\tikzset{...}` 中，不受选项 `visualize axis` 的辖制，例如

```
every axis/.style={style=black!50}
```



```
\tikzset{
  data visualization/our system/.append style={
    every axis/.style={style=black!50},
    left axis= {visualize axis={x axis= {goto=min}, style=red!75}},
    right axis={visualize axis={x axis= {goto=max}, style=blue!75}},
    x axis= {visualize axis={left axis={goto=min}},
    visualize axis={left axis={goto=max}}},
  }
}
```

```
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=4cm},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

在执行 `every axis` 前，先执行 `low=min`, `high=max`.

选项 `styling` 也是可用的。

坐标轴的偏移 对一般的 `scientific axes` 坐标系，坐标轴是相交的；对于 `scientific axes = clean`，坐标轴是相离的，即轴发生偏移。前面的选项 `goto` 用变量值设置两个轴的相对位置，`goto pos` 用一个小数设置两个轴的相对位置，在这两个选项设置的基础上，选项 `padded min`, `padded max`, `padded` 进一步指定轴做偏移。用带单位的尺寸指定轴的偏移量有时要比用变量值更方便，所以这些选项是相互补充、各有优点的。

为了方便，约定一个称呼：如果一个轴的位置是参照另一个轴（用选项 `goto` 等）确定的，这“另一个轴”称为该轴的“参照轴”。

两个坐标轴的相对偏移尺寸用下面的选项设置。

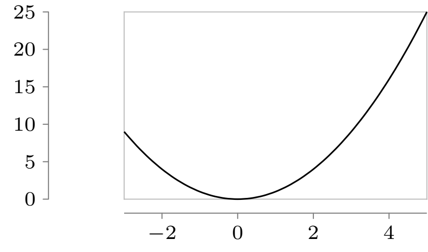
`/tikz/data visualization/axis options/padding min=`⟨*dimension*⟩ (no default)

这个选项指定一个偏离尺寸 ⟨*dimension*⟩, 用于来配合选项 `goto=padded min`. 假设如下设置：

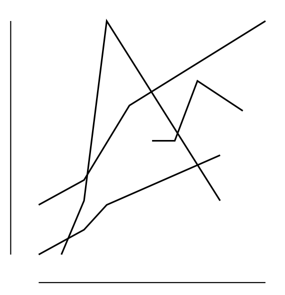
```
x axis={padding min=-1cm},
y axis={visualize axis={x axis={goto=padded min}}
```

再假设画出来的 `y axis` 轴与 `x axis` 轴交于点 P, 那么点 P 与 `x axis` 轴的 min 点的距离就是 `-1cm`. 也就是说，当某个轴带有本选项后，在该轴上就会有一个特殊的点 P, 点 P 与该轴的 min 点的距离就是 ⟨*dimension*⟩; 点 P 与选项 `goto=padded min` 相对应。通常 ⟨*dimension*⟩ 应当为带单位的负值尺寸。

对于标准的预定义坐标系统 `scientific axes=clean`, 有个默认的偏离尺寸，如果再使用这个选项指定一个尺寸 ⟨*dimension*⟩, 则偏离会叠加，而且纵轴与横轴不相交，有灰色框线来标示绘图区域。但是对于自定义坐标系，坐标轴偏移后，如果没有其它设置，纵轴与横轴仍然相交，也没有灰色框线来标示绘图区域。




```
\begin{tikzpicture}
\datavisualization [scientific axes={clean, width=4cm},
    x axis={padding min=-1cm},
    visualize as smooth line]
  data [format=function] {
    var x : interval [-3:5];
    func y = \value x * \value x;
  };
\end{tikzpicture}
```



```
\tikzset{
  data visualization/our system/.append style={
    all axes= {padding=1em},
    left axis= {visualize axis={x axis= {goto=padded min}}},
    right axis={visualize axis={x axis= {goto=padded max}, padded}},
    x axis= {visualize axis={left axis={goto=padded min}},
             visualize axis={left axis={goto=padded max}, padded}},
  }
}
\tikz \datavisualization [
  our system,
  x axis={attribute=time, length=3cm},
  left axis ={attribute=money},
  right axis={attribute=people},
  visualize as line/.list={people 1, people 2, money 1, money 2}]
data group {people and money};
```

`/tikz/data visualization/axis options/padding max=`⟨*dimension*⟩ (no default)

这个选项与 `padding min=` 类似，不过针对轴的 max 点，用于来配合选项 `goto=padded max`，通常 ⟨*dimension*⟩ 应当为带单位的正值尺寸。

`/tikz/data visualization/axis options/padding=`⟨*dimension*⟩ (no default)

这里 ⟨*dimension*⟩ 应当是带单位的正值尺寸,这个选项会同时设置 `padding max=`⟨*dimension*⟩ 和 `padding min=-`⟨*dimension*⟩.

82.5.3 可视化网格线

选项 `visualize grid` 辖制网格线的可视化，例如网格线的位置、颜色、线型等。如果在某个轴的选项中多次使用该选项，则它们的效果会被累计。

`/tikz/data visualization/axis options/visualize grid=`⟨*options*⟩ (no default)

这个 key 作为轴的选项，导致该轴的网格线可视化，⟨*options*⟩ 决定网格线的默认外观，如起止点、颜色等，其中可用 `style=` 选项。该选项完成设置后，在可视化命令中给轴使用 `grid` 选项会按设置画出网格线，也可临时修改网格线的外观。

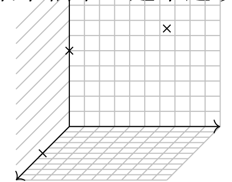
网格线的方向 “网格线”应该是对坐标空间的一种分割，对于不同的坐标系来说，“网格线”有不同的含义。对于平面仿射坐标系，各个轴的网格线就是一组平行线段。对于平面极坐标系，网格线就是一组射线或者一组同心圆弧。对于 3 维直角坐标系，分割空间的是平面，所以一个轴的“网格线”应该是一组与该轴正交的平面，但是在绘图时最好还是用线段来画网格线。对于 3 维球坐标系，分割空间的是纬平面和经平面，但最好用经纬网做网格线。

在一个可视化过程中，如果某个轴的选项中多次使用 `visualize grid` 选项，那么它们的效果会累计。

对于 3 维直角坐标系的 x 轴，其网格线有两个方向，一个方向是 y 轴方向，另一个方向是 z 轴方向，用下面的 key 指定网格线的方向:

/tikz/data visualization/direction axis=⟨*axis name*⟩ (no default)

当某个轴带上这个选项后，这个轴的网格线的方向就与轴 ⟨*axis name*⟩ 的方向一致。



```
\begin{tikzpicture}
\tikz \datavisualization [
  xyz Cartesian cabinet,
  all axes={visualize axis={low=0, style=->}, grid=many},
  x axis={visualize grid={direction axis=z axis},
          visualize grid={direction axis=y axis}},
  y axis={visualize grid={direction axis=x axis},
          visualize grid={direction axis=z axis}},
  z axis={visualize grid={direction axis=x axis}},
  visualize as scatter]
data {
  x, y, z
  0, 0, 1
  0, 1, 0
  2, 2, 2
};
\end{tikzpicture}
```

网格线的外观

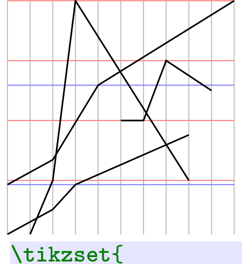
主网格线、副网格线、次副网格线的外观 可以在 /tikz/data visualization/axis options/grid→P. 431 中使用 /tikz/data visualization/style→P. 436 设置网格线的外观。/tikz/data visualization/grid layer→P. 438, /tikz/data visualization/every grid→P. 438, /tikz/data visualization/every major grid→P. 439, /tikz/data visualization/every minor grid→P. 439, /tikz/data visualization/every subminor grid→P. 439 这些样式(style)可用于设置网格线的外观(应该直接用作命令 \datavisualization 的选项或者用在 \tikzset{...} 中)。在默认下，这些样式已经对网格线的初始状态做了比较合理地设置，如无特别需要，可以不做修改。

选项 /tikz/data visualization/major→P. 434, /tikz/data visualization/minor→P. 434, /tikz/data visualization/subminor→P. 434, /tikz/data visualization/common→P. 434 可以用做 grid= 的参数，也可以用做 visualize grid= 的参数。设置网格线的起止点要用到选项 low=⟨*value*⟩ 和 high=⟨*value*⟩, 这里的 ⟨*value*⟩ 是网格线的方向轴（与网格线平行的轴）上的变量值，注意此时的 ⟨*value*⟩ 不能换成带单位的尺寸 ⟨*dimension*⟩，否则无法读取尺寸，导致错误。

轴线可以看作是特殊的网格线，有些用于轴线的选项也能用于网格线。

```
\tikz \datavisualization [
  xy Cartesian,
  all axes={visualize axis={low=0, style=->}, grid={some, minor steps between steps}},
  x axis= {visualize grid={direction axis=y axis, minor={low=0.25, high=1.75, style=red!50}}},
  ↪  % 注意这里对横轴的副网格线的规定
  visualize as scatter]
data {
  x, y
  0, 0
  3, 3
};
```

再用前面定义的 our system 作一个图：



```
\tikzset{
  data visualization/our system/.append style={
    x axis= {visualize grid={direction axis=left axis}},
    left axis= {visualize grid={direction axis=x axis, common={style=red!50}}},
    right axis={visualize grid={direction axis=x axis, common={style=blue!50}}},
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=3cm, grid=many},
    left axis ={attribute=money, grid=some},
    right axis={attribute=people, grid=few},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

82.5.4 刻度线、刻度值标签的可视化

刻度包括刻度线、刻度值标签（node）两方面，需要用到选项 `visualize ticks`，这个选项辖制刻度的可视化，例如刻度的位置、颜色等。如果在某个轴的选项中多次使用该选项，则它们的效果会被累计。

`/tikz/data visualization/axis options/visualize ticks=`⟨*options*⟩ (no default)

与选项 `visualize grid` 类似。可以在 ⟨*options*⟩ 使用选项 `style=` 设置刻度线的外观；使用选项 `node style=` 设置刻度值标签的外观，anchor 位置等。在可视化命令中，给轴使用选项 `ticks` 可以按设置画出刻度，也可以临时修改刻度的外观。

刻度线可以看作是“迷你版”的网格线，所以有些能用于网格线的选项 (如 direction axis) 也能用于刻度线。但与网格线不同的是，刻度线总是直的，而极坐标网格、经纬网中的网格线都有弯曲。

再用前面定义的 our system 作图：

```
\tikzset{
  data visualization/our system/.append style={
    x axis={visualize ticks={direction axis=left axis, left axis={goto=min}},
        visualize ticks={direction axis=left axis, left axis={goto=max}},
    }
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=3cm, ticks=many},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

上面例子中，图形的上部和下部都有横轴 x axis, 它们的刻度线一样的。如果想要改变刻度线的长度，需要用选项 /tikz/data visualization/low→P. 451 和 /tikz/data visualization/high→P. 452. 选项 low=⟨*dimension*⟩ 指的是刻度线的起点偏离轴线的尺寸，选项 height=⟨*dimension*⟩ 指的是刻度线的终点偏离轴线的尺寸。例如假设某一横轴的刻度线设置了 low=-5pt，high=10pt, 那么刻度线的起点将偏移到横轴之下 -5pt 处，起点将偏移到横轴之上 10pt 处，刻度线的总长度就是 15pt.

或者用下面的选项：

/tikz/data visualization/tick length⟨*dimension*⟩ (no default)

同时指定 low=-⟨*dimension*⟩ 和 high=⟨*dimension*⟩.

把前面例子中的刻度线改短：

```
\tikzset{
  data visualization/our system/.append style={
    x axis={visualize ticks={direction axis=left axis,high=0pt,left axis={goto=min}},
        visualize ticks={direction axis=left axis,low=0pt,left axis={goto=max}},
    }
  }
}
```

```
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=3cm, ticks=many},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

上面例子中，对上部横轴使用了选项 `low=0pt`, 对下部横轴使用了选项 `high=0pt`, 这就使得刻度线是指向绘图区域外部的。

为刻度线添加刻度值标签用到下面的选项：

/tikz/data visualization/tick text at low=⟨*true or false*⟩ (default `true`)

把这个选项会在刻度线的 low 端即起点之前添加刻度值标签。也就是说，对于横轴，该选项会把刻度值标签加在刻度线的下方；对于纵轴，该选项会把刻度值标签加在刻度线的左方。本选项的默认值是 `true`.

/tikz/data visualization/tick text at high=⟨*true or false*⟩ (default `true`)

把这个选项会在刻度线的 high 端即终点之后添加刻度值标签。也就是说，对于横轴，该选项会把刻度值标签加在刻度线的上方；对于纵轴，该选项会把刻度值标签加在刻度线的右方。本选项的默认值是 `true`.

```
\tikzset{
  data visualization/our system/.append style={
    x axis={
      visualize ticks={direction axis=left axis, left axis={goto=min}, high=0pt, tick text at low,
      ↪  stack},
      visualize ticks={direction axis=left axis, left axis={goto=max}, low=0pt, tick text at high,
      ↪  stack}
    }
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=3cm, ticks=some},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

/tikz/data visualization/no tick text (no value)

这个选项同时设置 `tick text at low=false` 和 `tick text at high=false`，即不添加刻度值标签。

82.5.5 坐标轴标签的可视化

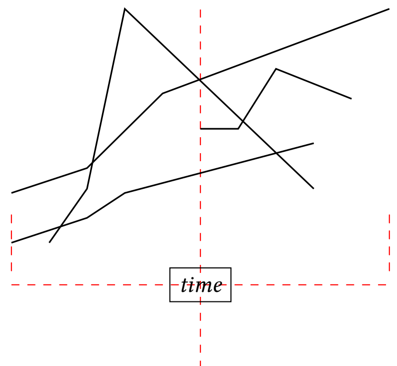
要使轴带有显示的标签，需要给轴加 `/tikz/data visualization/axis options/label`→P. 423 选项，但对于一般的自定义的轴来说，这个选项并不直接导致轴标签可视化，需要先用选项 `visualize label` 做有关设置后，才会令选项 `label` 有可视化效果。

`/tikz/data visualization/axis options/visualize label=`⟨*options*⟩ (no default)

⟨*options*⟩ 主要用于确定轴标签的位置、外观等，其中可以用选项 `node style=`. 轴标签位置的确定方式与轴位置的确定方式类似。以下横轴为例，下横轴标签在水平方向的位置可以用选项 `goto` 和 `goto pos=`⟨*fraction*⟩ 确定，这两个选项指定横轴标签在水平方向上（初始之下是沿着横轴）的偏移位置。`goto pos=0.5` 会把轴标签放在下横轴的中间位置；下横轴标签在竖直方向的偏移位置可以用其参照轴 `left axis` 来确定，如

```
x axis={visualize label={
          x axis={goto pos=.5},
          left axis={padding=1.5em, goto=padded min}}}
```

上面的代码设置轴 `x axis` 的标签位置，规定标签在水平方向的偏移位置是轴 `x axis` 的中间点，在竖直方向的偏移位置是：轴 `left axis` 对应的变量的最小值点还向下平移 1.5em 点处；这两个方向的偏移共同确定一个点，该点就是轴标签 node 的锚定点，默认轴标签 node 的中心处于该点。




```
\tikzdatavisualizationset{
  our system/.append style={
    x axis={
      visualize label={
        node style=draw,
        x axis={goto pos=.5}, % 横轴标签的列位置
        left axis={padding=1.5em, goto=padded min}}, % 横轴标签的行位置
      visualize axis={
        style={red,dashed}, % 画一个红色虚线横轴
        left axis={padding=1.5em, goto=padded min}},
      visualize grid={direction axis=left axis},
      visualize ticks={major={low=-1em,high=1em,style={red,dashed}}} % 给横轴定义主刻度线样式
    }
  }
}
\tikz \datavisualization [
    our system,
```

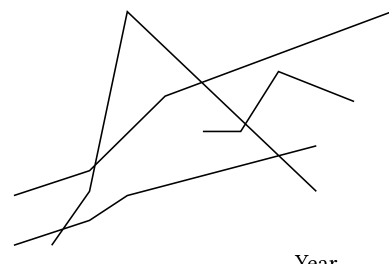
```
    x axis={attribute=time, label, ticks={major at={min,max}}, % 在横轴始末位置画刻度线
      grid={major at=1950 as[{low=-1,high=5,style={red,dashed}}]}}, % 在横轴中间画一根网格线
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

用两个偏移位置确定轴标签位置的句法有些繁琐，另有一种简捷一些的确定轴标签位置的方法，即使用 `node style={at=...}` 选项，其中 `at=...` 按 TikZ 的句法确定一个点，作为轴标签位置的锚定点，当然这个点应当与当前的可视化图形有关。考虑这样一个坐标系：以预定义的 `data visualization bounding box` 的左下角为原点创建一个直角坐标系，以 cm 为单位长度，这个坐标系中的点坐标就与当前的可视化图形有关了，在 `node style={at=...}` 中涉及的点的坐标就是这个坐标系中的坐标。

在 `node style={at=...}` 中可以使用多种 TikZ 句法来确定一个点，比较灵活，例如：

```
node style={at={($(data visualization bounding box.south)+(1,-1)$)}} % 坐标计算式
node style={at={($(0,0)!0.5!(-30:4)$)}}
node style={at={(0,0 |- data visualization bounding box.south)}, below} % 加平移选项
node style={at={($(0,0)+(1,-1)$)}, below right=-1cm and 2cm}
```

还需要注意的是，因为选项 `x axis={goto pos=`⟨*fraction*⟩`}` 指定轴标签在水平方向的偏移位置，所以如果某个轴同时带有 `x axis={goto pos=`⟨*fraction*⟩`}` 和 `node style={at=...}` 两个选项，则这两个选项的作用会叠加。




```
\tikzdatavisualizationset{
  our system/.append style={
    x axis={visualize label={
      x axis={goto pos=.8},
      node style={at={(0,0 |- data visualization bounding box.south)}, below}}}}}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, ticks=some, label=Year},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

上面例子中,选项 `x axis={goto pos=.8}` 确定标签 `Year` 在水平方向的位置,选项 `node style={at=...}` 确定标签 `Year` 在竖直方向的位置。

`/tikz/data visualization/axis option/anchor at min` (no value)

如果某个轴带有这个选项，则该轴的标签会位于轴的起点，即 `min` 或 `padded min` 对应的点，并且标签的 anchor 位置还会使得标签看上去处于轴的起始点的“之前”。例如通常情况下，对于横轴，标签会处于横轴左侧；对于纵轴，标签会处于纵轴下方。

/tikz/data visualization/axis option/anchor at max (no value)

类似 anchor at min.

82.5.6 完整的定义代码

下面是前文定义轴系统 our system 的完整代码:

```
\tikzdatavisualizationset{ our system/.style={
% 声明 3 个坐标轴
  new Cartesian axis=x axis, new Cartesian axis=left axis, new Cartesian axis=right
  ↪  axis,
% 指定轴的偏移、方向
  all axes={padding=.5em}, left axis={unit vector={(0cm,1pt)}}, right axis={unit
  ↪  vector={(0cm,1pt)}},
% 指定 x axis 对应的变量
  x axis={attribute=x},
% 指定轴的长度
  x axis ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/width}},
  left axis ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height
  ↪  }},
  right axis={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height
  ↪  }},
% 设置轴的样式——颜色
  every axis/.style={style=black!50}, % 设为默认颜色
% 将轴可视化
  left axis= {visualize axis={x axis= {goto=padded min}, style=red!75, padded}},
  right axis={visualize axis={x axis= {goto=padded max}, style=blue!75,padded}},
  x axis= {visualize axis={left axis={goto=padded min}, padded},
    visualize axis={left axis={goto=padded max}, padded}},
% 设置可视化网格线的默认项目
  x axis= {visualize grid={direction axis=left axis}},
  left axis= {visualize grid={direction axis=x axis, common={style=red!50}}},
  right axis={visualize grid={direction axis=x axis, common={style=blue!50}}},
% 设置可视化刻度线的默认项目
  left axis={visualize ticks={style={red!50!black}, direction axis=x axis, x axis=
  ↪  {goto=padded min}, high=0pt, tick text at low}},
  right axis={visualize ticks={style={blue!80!black}, direction axis=x axis, x axis=
  ↪  {goto=padded max}, low=0pt, tick text at high}},
  x axis={visualize ticks={direction axis=left axis, left axis={goto=padded min},
  ↪  high=0pt, tick text at low},
    visualize ticks={direction axis=left axis, left axis={goto=padded max}, low=0pt}
    ↪  },
% 默认各个轴都有刻度线
  all axes={ticks},
% 设置可视化的轴标签的默认项目
  x axis={visualize label={x axis={goto pos=.5}, node style={
    at={(0,0 |- data visualization bounding box.south)}, below}}},
  left axis={visualize label={left axis={goto pos=.5}, node style={
    at={(0,0 -| data visualization bounding box.west)}, rotate=90, anchor=south,
    ↪  red!50!black}}},
  right axis={visualize label={right axis={goto pos=.5}, node style={
```

```
    at={(0,0 -| data visualization bounding box.east)}, rotate=-90, anchor=south,
    ↪  blue!80!black}}},
}}
```



```
\tikz \datavisualization [
  our system,
  x axis={attribute=time, label=Year, ticks={tick text padding=2pt, style={/pgf/number format/set
  ↪  thousands separator=}}},
  left axis={attribute=money, label=Spending, padding min=0, include value=0, grid, ticks={tick
  ↪  prefix=\$, style={/pgf/number format/fixed, /pgf/number format/fixed zerofill, /pgf/number
  ↪  format/precision=2}}},
  right axis={attribute=people, label=Population, padding min=0, include value=0, ticks=
  ↪  {style=/pgf/number format/fixed}},
  visualize as line/.list={
    people 1, people 2, money 1, money 2},
    people 1={style={visualizer color=blue}},
    people 2={style={visualizer color=blue!50}},
    money 1={style={visualizer color=red}},
    money 2={style={visualizer color=red!50}} ]
data group {people and money};
```

下面是另一个例子。

先定义坐标系：

```
% 定义一个坐标系，#1 是坐标轴与绘图区域的间距，#2 是横轴的长度，#3 是纵轴的长度
\tikzdatavisualizationset{
  AB system/.style n args={3}{
    new Cartesian axis=x axis,
    new Cartesian axis=y axis,
    new Cartesian axis=xx axis,
    y axis={unit vector={(0cm,1pt)}},
    all axes={padding=#1},
    x axis={
      length=#2,
      attribute=expense,
      visualize axis={y axis={goto=padded min}, padded, style={->}},
      visualize ticks={direction axis=y axis, low=0pt, high=4pt,
        y axis={goto=padded min},tick text at low},
      visualize label={x axis={goto=padded max}, y axis={goto=padded min},
        node style={anchor=west}}
    },
    y axis={
      length=#3,
```

```
      attribute=price,
      visualize axis={x axis={goto=padded min}, padded, style={->}},
      visualize ticks={direction axis=x axis, low=0pt, high=4pt,
        x axis={goto=padded min},tick text at low},
      visualize label={y axis={goto=padded max}, x axis={goto=padded min},
        node style={anchor=south}}
    },
    xx axis={
      length=#2,
      visualize label={node style={at={(data visualization bounding box.south)},
      ↪  below=1.5em}}
    }
  }
}
```

再定义一个数据点组:

```
\tikz \datavisualization
% B 曲线上的点
  data group [set=lineB]{e-p1BL} = {
    data {
      expense,price
      0,100
      100,100
      100,15
      117747,15
      117747,100}}
% B 曲线上的标记为三角的点
  data group [set=scatterB]{e-p1BS} = {
    data {
      expense,price
      100,90
      100,70
      100,55
      100,40
      100,22
      19600,15
      39200,15
      58800,15
      78400,15
      98000,15
      117747,30
      117747,50
      117747,70
      117747,90}}
% A 曲线对应的点
  data group [set=lineA]{e-p1AL} = {
    data {
      expense,price
      100,30
```

```
      114336,30
      114336,100
      146000,100}}
% A 曲线上标记为圆点的点
  data group [set=scatterA]{e-p1AS} = {
    data {
      expense,price
      100,97
      100,80
      100,65
      100,48
      100,30
      9600,30
      29200,30
      48800,30
      68400,30
      100000,30
      114336,40
      114336,60
      114336,75
      114336,96
      126000,100
      136000,100}
  };
```

画出图形：

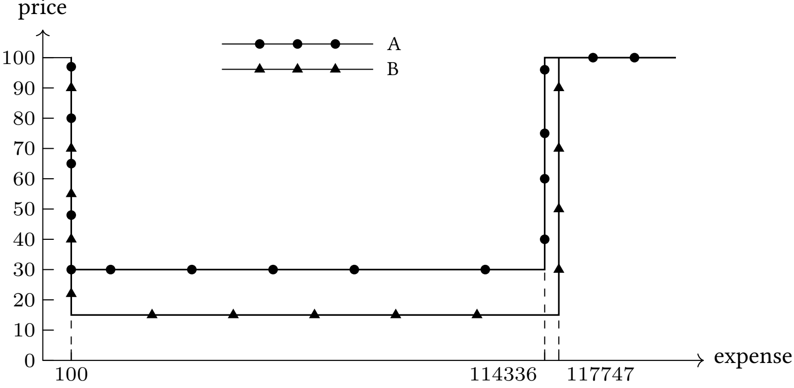


Figure 2: A and B

```
\begin{tikzpicture}
\datavisualization [
  AB system={1em}{8cm}{4cm},
  visualize as line=lineA,
  visualize as scatter=scatterA,
  visualize as line=lineB,
  visualize as scatter=scatterB,
  scatterA={style={mark=otimes*,mark options={color=black,scale=0.8}}},
  scatterB={style={mark=triangle*,mark options={color=black}}},
  x axis={
    label={expense},
```

```
    ticks={major at={
        100,
        114336 as [node style={anchor=north east}]$114336$,
        117747 as [node style={anchor=north west}]$117747$}}
  },
  y axis={
    label=price,
    padding min=0, include value=0,
    ticks={major at={0,10,...,100}}
  },
  new legend entry={
    text=A,
    visualizer in legend={\draw (0,0)--(-2,0) plot[mark=otimes*,mark options=
    ↪  {color=black,scale=0.8},only marks] coordinates {(-1.5,0) (-1,0) (-0.5,0)};}
  },
  new legend entry={
    text=B,
    visualizer in legend={\draw (0,0)--(-2,0);\draw plot[mark=triangle*,mark options={color=black
    ↪  },only marks] coordinates {(-1.5,0) (-1,0) (-0.5,0)};}
  },
  legend={at values={expense=58000, price=100}},
  xx axis={label={Figure 2: A and B}}
]
  data group {e-p1AL}
  data group {e-p1AS}
  data group {e-p1BL}
  data group {e-p1BS}
  info{
    \draw (visualization cs: expense=100,price=100)--++(-1em,0);
    \draw [dashed](visualization cs: expense=100,price=0)--(visualization cs:
    ↪  expense=100,price=15);
    \draw [dashed](visualization cs: expense=114336,price=0)--(visualization cs:
    ↪  expense=114336,price=30);
    \draw [dashed](visualization cs: expense=117747,price=0)--(visualization cs:
    ↪  expense=117747,price=15);
  };
\end{tikzpicture}
```

82.5.7 专用于创建新坐标系统的 key

前面例子中创建的轴系统 `our system` 不能使用 `our system={`⟨*options*⟩`}` 这种句法来对此轴系统设置 ⟨*options*⟩. 如果想使用句法 `our system={`⟨*options*⟩`}` 来设置轴系统，可以用下面的选项来声明轴系统。

`/tikz/data visualization/new axis system={`⟨*axis system name*⟩`}{`⟨*axis setup*⟩`}{`⟨*default options*⟩`}{`⟨*application options*⟩`}` (no default)

这个 key 有 4 个参数。⟨*axis system name*⟩ 轴系统的名称，本选项会创建下面的 key:

`/tikz/data visualization/`⟨*axis system name*⟩`=`⟨*options*⟩ (no default)

当使用 ⟨*axis system name*⟩ 时，下面的 keys 会被依次执行:

1. ⟨*axis setup*⟩ 中的 keys 会被按路径 `/tikz/data visualization/` 来执行。
2. ⟨*default options*⟩ 中的 keys 会被按路径 `/tikz/data visualization/` 来执行。
3. 执行下面的样式:

/tikz/data visualization/every ⟨*axis system name*⟩ (style, no default)

此样式中保存的 keys 会被按路径 /tikz/data visualization/⟨*axis system name*⟩/ 来执行。

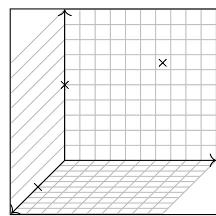
4. ⟨*options*⟩ 中的 keys 会被按路径 /tikz/data visualization/⟨*axis system name*⟩/ 来执行。
5. ⟨*application options*⟩ 中的 keys 会被按路径 /tikz/data visualization/ 来执行。

在上述执行过程中, ⟨*axis setup*⟩ 中的设置是"无需修改的基本设置", 例如在其中设置坐标轴 x axis, y axis, 但不设置标签, 因为标签 (内容、位置) 总是根据实际情况变化的; 在 ⟨*default options*⟩ 中设置一些默认值 (default values), 这些默认值会被传递给 ⟨*options*⟩; 而 ⟨*application options*⟩ 则把 ⟨*options*⟩ 中确定的各种选项参数用于绘制可视化轴线段。基本思路是, 让 ⟨*default options*⟩, ⟨*options*⟩, every ⟨*axis system name*⟩ 都有机会来对轴系统做出设置, 或者修改之前的设置。

可以参考文件《tikzlibrarydatavisualization.code.tex》中对 /tikz/data visualization/scientific axes→P. 425 的定义。

82.6 遇到的问题

xxxxxxxxxxxxxxxx



xxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxx

```
xxxxxxxxxxxxxxxx\\
xxxxxxxxxxxxxxxxxx
\begin{tikzpicture}
\tikz{
\datavisualization [
  xyz Cartesian cabinet,
  all axes={visualize axis={low=0, style=->}, grid=many},
  x axis={visualize grid={direction axis=z axis},
          visualize grid={direction axis=y axis}},
  y axis={visualize grid={direction axis=x axis},
          visualize grid={direction axis=z axis}},
  z axis={visualize grid={direction axis=x axis}},
  visualize as scatter]
data {
  x, y, z
  0, 0, 1
  0, 1, 0
  2, 2, 2
};
\draw (current bounding box.south west) rectangle (current bounding box.north east);
}
\end{tikzpicture}xxxxxxxxxxxxxxxxxxxxxxxxxx
```

83 Visualizers

83.1 Overview

显像器（visualizer）将数据点可视化，并决定以什么方式显示数据点。例如，显像器决定将数据点显示为散点或曲线，散点的标记符号是什么样式，曲线的线型、线宽、颜色，散点或曲线是否带有标签，等等。如果需要手工设置数据点的显示样式、外观，可以在显像器的参数中使用本节介绍的选项。自动设置数据点的显示样式、外观的方法是使用 style sheet，将在下节介绍。

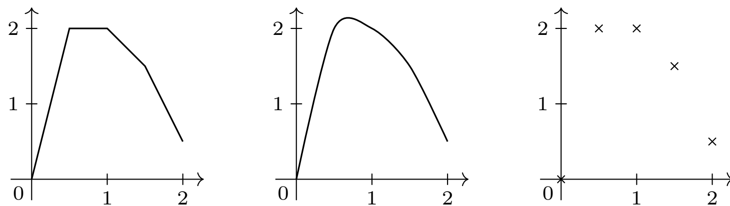
有多种不同的显像器，例如，line visualizer, 将数据点用直线段连接起来，这个显像器或者是在实际提供的数据点之间插入一些点构成线段，或者是按数据点出现的次序将它们连接起来。再如，scatter visualizer 或 mark visualizer, 用某种标记符号来标记数据点。在一个可视化过程中可以使用多个显像器，例如，一个可视化过程中有多组数据点，对各组数据使用不同的显像器，使它们的表现形式相互区别。对一个数据点可以使用多个显像器。

83.2 用法

83.2.1 使用一个显像器

在命令 `\datavisualization` 的选项中使用

```
visualize as line 用直线段连接数据点
visualize as smooth line 用直线段连接数据点并使得连接点处平滑
visualize as scatter 将数据点显示为散点
```




```
% 定义数据组
\tikz \datavisualization data group {example} = {
  data {
    x, y
    0, 0
    0.5, 2
    1, 2
    1.5, 1.5
    2, 0.5
  }};
\tikz \datavisualization [school book axes, visualize as line] data group {example};
\qquad
\tikz \datavisualization [school book axes, visualize as smooth line] data group {example};
\qquad
\tikz \datavisualization [school book axes, visualize as scatter] data group {example};
```

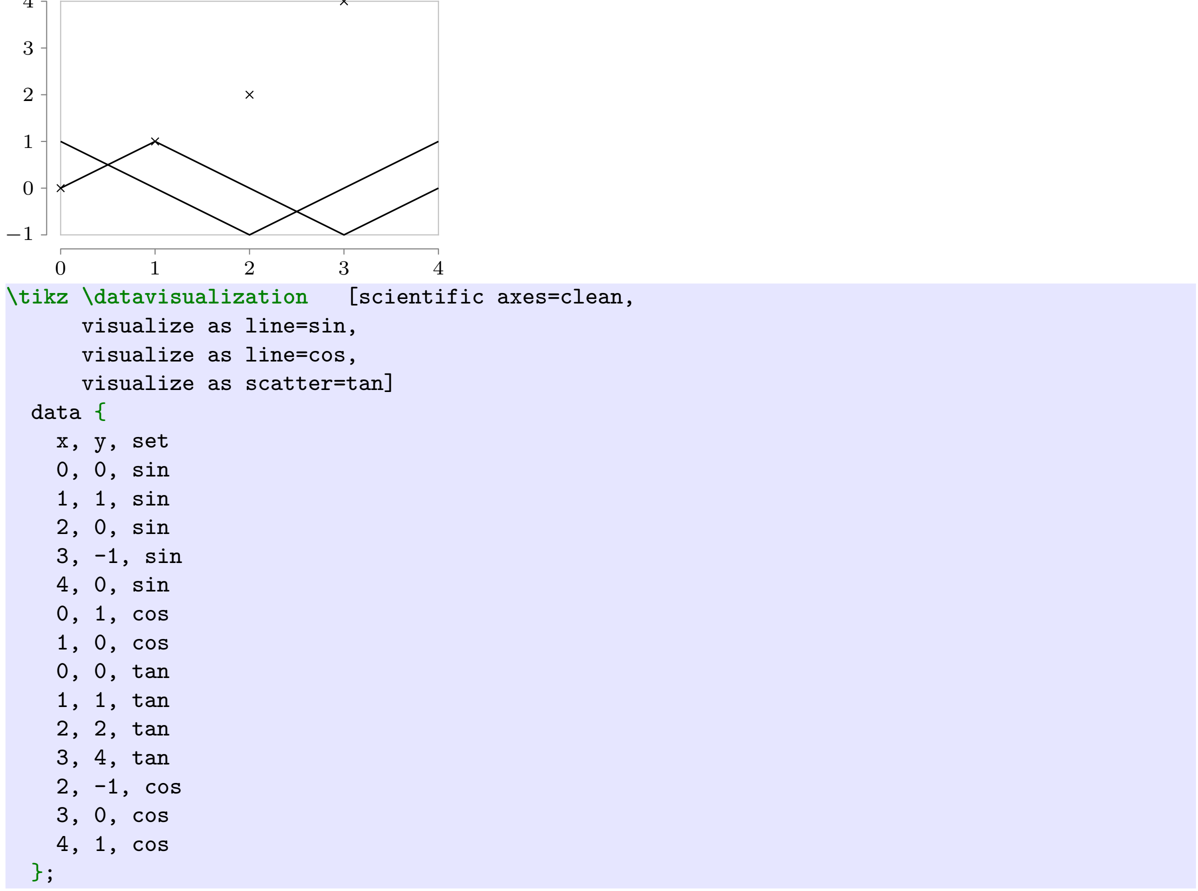
如果在命令 `\datavisualization` 的选项中只使用一个显像器选项，例如 `visualize as line`, 那么命令之内的所有数据点（包括所有 `data` 指令定义的数据点）都会被用直线段连起来，因此只能得到一个曲线。

83.2.2 使用多个显像器

为了能使每一组数据点都产生一个单独的曲线，需要使得每一组数据点都应用一个显像器。可以先给显像器命名，然后把显像器名称引入到数据点组中。一个显像器作为一个 key，可以带有一个参数，例如，`visualize as line=`⟨*name*⟩，然后可以将这个 ⟨*name*⟩ 添加到数据点中，作为一个标识，表示该数据点用 `visualize as line` 来显示。⟨*name*⟩ 对应的标识名称（变量名，attribute）是 `set`，而各个 ⟨*name*⟩ 则是 `set` 的参数。

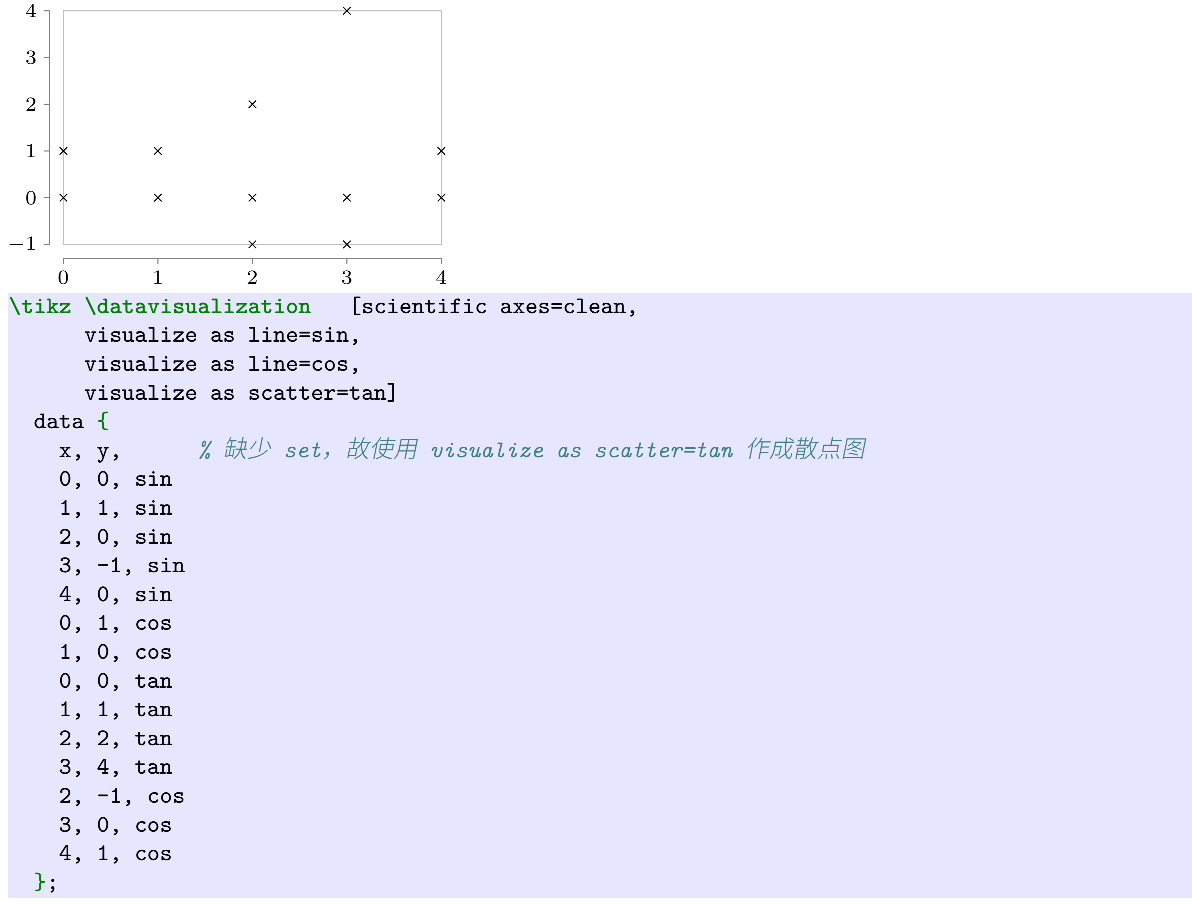
`/data point/set` (no value)

这个 key 的路径与表示数据点分量的标识（attribute, 变量名）`/data point/x`, `/data point/y` 是一样的，可以作为 attribute 用在数据点列表中。在数据点列表中，变量 `/data point/x` 代表的是数据，而标识符号 `set` 代表的是显像器的名称。当在数据点列表的头行中写出 `set` 时，每一行的数据点都要附带上一个显像器名称 ⟨*name*⟩ 作为一个元素数据，指示该点用名称为 ⟨*name*⟩ 的显像器来处理。具有相同显像器名称 ⟨*name*⟩ 的数据点组成一个集合，这个数据点集使用显像器 ⟨*name*⟩ 来统一处理。




```
\tikz \datavisualization   [scientific axes=clean,
      visualize as line=sin,
      visualize as line=cos,
      visualize as scatter=tan]
  data {
    x, y, set
    0, 0, sin
    1, 1, sin
    2, 0, sin
    3, -1, sin
    4, 0, sin
    0, 1, cos
    1, 0, cos
    0, 0, tan
    1, 1, tan
    2, 2, tan
    3, 4, tan
    2, -1, cos
    3, 0, cos
    4, 1, cos
  };
```

注意，如果在某一行数据点中不给出标识符号“`set`”，那么就默认该行数据点使用之前最近出现的显像器来处理。将下面的例子对比上面的例子：

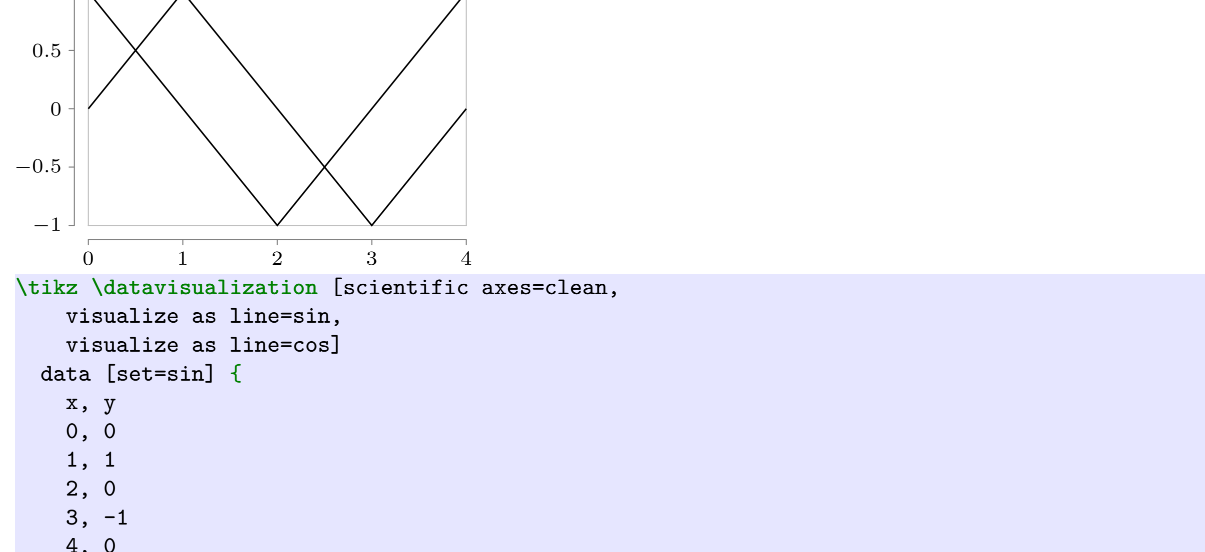



```
\tikz \datavisualization   [scientific axes=clean,
      visualize as line=sin,
      visualize as line=cos,
      visualize as scatter=tan]
  data {
    x, y,       % 缺少 set，故使用 visualize as scatter=tan 作成散点图
    0, 0, sin
    1, 1, sin
    2, 0, sin
    3, -1, sin
    4, 0, sin
    0, 1, cos
    1, 0, cos
    0, 0, tan
    1, 1, tan
    2, 2, tan
    3, 4, tan
    2, -1, cos
    3, 0, cos
    4, 1, cos
  };
```

上一个选项的用法有些繁琐，使用下面的选项也可以有相同的效果，而用法相对简捷一些。

`/pgf/data/set`=⟨*name*⟩ (no value)

这是 `/data point/set=`⟨*name*⟩ 的另写。这个选项作为 `data` 指令的选项，指示该 `data` 指令设置的数据点组用显像器 ⟨*name*⟩ 来处理。



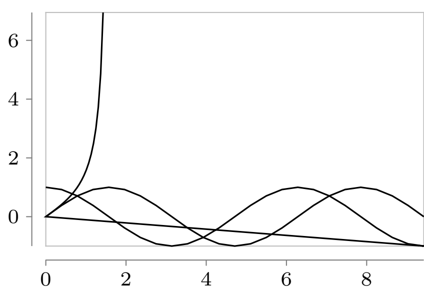

```
\tikz \datavisualization [scientific axes=clean,
    visualize as line=sin,
    visualize as line=cos]
  data [set=sin] {
    x, y
    0, 0
    1, 1
    2, 0
    3, -1
    4, 0
```

```
  }
  data [set=cos] {
    x, y
    0, 1
    1, 0
    2, -1
    3, 0
    4, 1
  };
```

当需要给一个显像器赋予多个不同名称时，可以使用 /.list 手柄，例如

```
visualize as line/.list={sin, cos, tan}
```

这样就可以令 visualize as line 依次取名 sin，cos，tan，这些名称应当依次用作各个 data 指令的选项，并且每个名称最好只用一次，不要重复使用。如果重复使用这些名称，例如，如果两个 data 指令都有 set=cos 选项，那么这两个指令定义的数据点之间会用（由显像器 visualize as line 产生的）直线段连接起来，这样一来，两个数据点组看上去就像是一个数据点组。




```
\tikz \datavisualization [scientific axes=clean,
    visualize as line/.list={sin, cos, tan}]
  data [set=sin, format=function] {
    var x : interval[0:3*pi];
    func y = sin(\value x r);
  }
  data [set=cos, format=function] {
    var x : interval[0:3*pi];
    func y = cos(\value x r);
  }
  data [set=cos, format=function] {    % 用了 set=cos，导致与 cos 曲线连起来了
    var x : interval[0:pi/2.2];
    func y = tan(\value x r);
  };
```

83.2.3 设置显像器的外观效果

/tikz/data visualization/⟨*visualizer name*⟩=⟨*options*⟩ (no default)

该选项可用作命令 \datavisualization 的选项，或者 \tikzset 的参数。当用选项

```
visualize as...=⟨visualizer name⟩
```

给显像器 visualize as... 命名后，程序会自动创建下面的 key 路径：

```
/tikz/data visualization/⟨visualizer name⟩
```

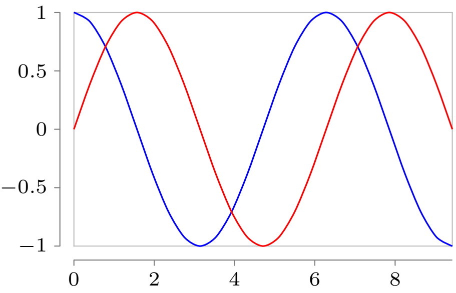
并且还会给 ⟨*visualizer name*⟩ 定义一个初始值。显像器有了名称，就可以把

⟨*visualizer name*⟩=⟨*options*⟩

作为可视化命令的选项来规定显像器的显示效果，⟨*options*⟩ 中的选项会被冠以前缀

```
/tikz/data visualization/visualizer options/
```

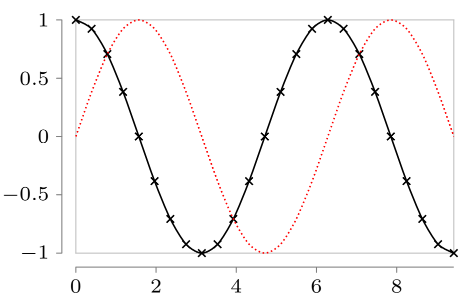
来执行。⟨*options*⟩ 中可以使用 `style`, `label in legend`, `label in data` 等选项来设置显像器的外观效果。




```
\tikz \datavisualization [scientific axes=clean,
    visualize as smooth line/.list={sin, cos},
    sin={style=red},
    cos={style=blue}]
  data [set=sin, format=function] {
    var x : interval[0:3*pi];
    func y = sin(\value x r);
  }
  data [set=cos, format=function] {
    var x : interval[0:3*pi];
    func y = cos(\value x r);
  };
```

`/tikz/data visualization/visualizer options/style=`⟨*options*⟩ (no default)

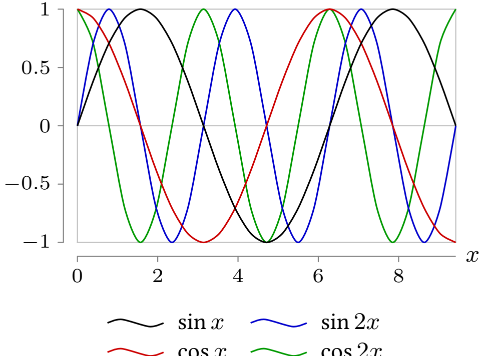
用在 ⟨*visualizer name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，设置显像器的外观效果，`style=`⟨*options*⟩ 中的选项应该是 TikZ 的选项。




```
\tikz \datavisualization [scientific axes=clean,
    visualize as smooth line=sin,
    sin={style={red, densely dotted}},
    visualize as smooth line=cos,
    cos={style={mark=x}}]
  data [set=sin, format=function] {
    var x : interval[0:3*pi];
    func y = sin(\value x r);
  }
  data [set=cos, format=function] {
    var x : interval[0:3*pi];
    func y = cos(\value x r);
```

```
};
```

可以在命令 \datavisualization 的选项中使用 style sheet 选项。



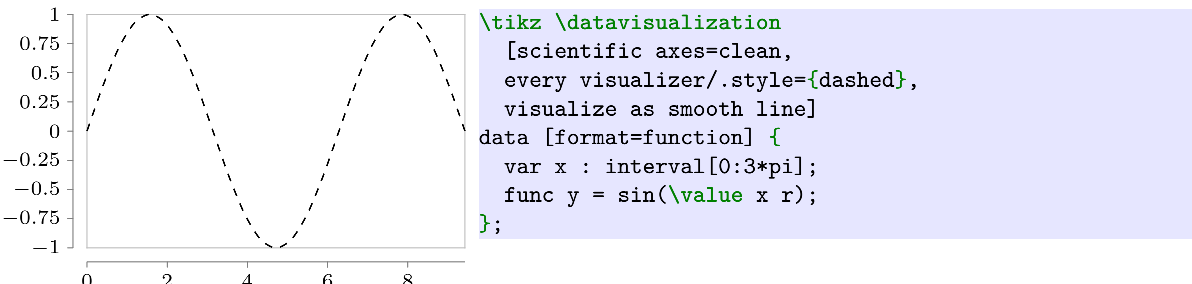

```
\tikz \datavisualization [scientific axes={clean, end labels},
  x axis={label=$x$}, y axis={grid={major also at=0}},
  visualize as smooth line/.list={sin,cos,sin 2,cos 2},
  legend={below, rows=2},
  sin={label in legend={text=$\sin x$}},
  cos={label in legend={text=$\cos x$}},
  sin 2={label in legend={text=$\sin 2x$}},
  cos 2={label in legend={text=$\cos 2x$}},
  style sheet=strong colors]
data [set=sin, format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
}
data [set=cos, format=function] {
  var x : interval[0:3*pi];
  func y = cos(\value x r);
}
data [set=sin 2, format=function] {
  var x : interval[0:3*pi];
  func y = sin(2*\value x r);
}
data [set=cos 2, format=function] {
  var x : interval[0:3*pi];
  func y = cos(2*\value x r);
};
```

`/tikz/data visualization/visualizer options/ignore style sheets` (no value)

用在 ⟨*visualizer name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，使得 style sheets 不用于显像器 ⟨*visualizer name*⟩.

`/tikz/data visualization/every visualizer` (style, no value)

这个样式中的选项应当是 TikZ 的选项。




```
\tikz \datavisualization
  [scientific axes=clean,
  every visualizer/.style={dashed},
  visualize as smooth line]
data [format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
};
```

83.3 基本的显像器

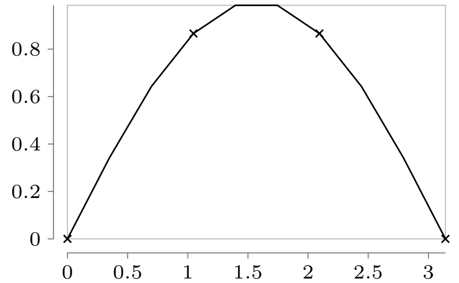
83.3.1 直线段或曲线显像器

`/tikz/data visualizers//visualize as line`=⟨*visualizer name*⟩ (default `line`)

本选项创建名称 ⟨*visualizer name*⟩, 这个名称指向（调用）显像器 `visualize as line`. 如果不给出名称 ⟨*visualizer name*⟩, 就默认名称为 line. 这个显像器会把那些带有 ⟨*visualizer name*⟩ 的数据点用直线段连接起来，或者把带有 `set=`⟨*visualizer name*⟩ 选项的数据点组用直线段连接起来。

本选项导致以下动作:

1. 创建一个 (属于 plot handler visualizer 类的) 新对象，用于分析带有 `set=`⟨*visualizer name*⟩ 选项的数据点的画布位置 (canvas positions)。
2. 在可视化过程的结尾处，使用 PGF 的 plot 机制 (见 plot 模块) 绘制数据流。这意味着，在 TikZ 中有效的图柄 (plot handlers) 有可能用于数据可视化过程，但是有的图柄不支持数据可视化引擎的“轴处理机制”，因此不能直接用于数据可视化过程。
3. 在 TikZ 中绘制点标记的选项是可用的。



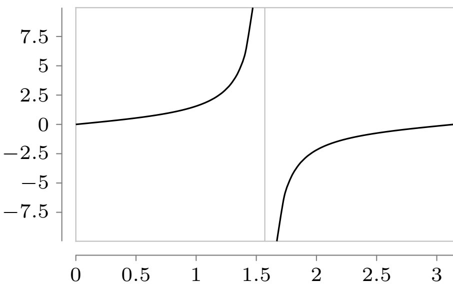

```
\tikz \datavisualization
    [scientific axes=clean,
     visualize as line=my data,
     my data={style={mark=x, mark repeat=3}}]
  data [format=function] {
    var x : interval [0:pi] samples 10;
    func y = sin(\value x r);
  };
```

有的可视化线条有间断点，表现为线条上的“缺口”，例如对于函数 $f(x) = \tan x, x \in [0, \pi]$, 点 $x = \frac{\pi}{2}$ 是它的无穷间断点，此时应该用区间 $[0, \frac{\pi}{2} - \epsilon] \cup [\frac{\pi}{2} + \epsilon, \pi]$ 代替区间 $[0, \pi]$, 绘制两段不相连的函数图。这种无穷间断点是 outlier 点，绘制线条的显像器 (例如 `visualize as line`, `visualize as smooth line`) 有自己的方法 (method) 处理这种 outlier 点，即下面的选项:

`/data point/outlier`=⟨*value*⟩ (default `true`, initially empty)

这个选项用作 `data point` 的选项，当 ⟨*value*⟩ 非空时创建一个特殊的 `outlier` 点。当绘制线条的显像器遇到这个特殊点时，不会把这个点与它之前或之后的点连起来，从而在图形上得到间断（有缺口）的效果。比方说，如果显像器 `visualize as smooth line` 遇到 3 个点 A, B, C，其中点 B 是 `data point[outlier]`, 那么显像器就不会在 A, C 之间连线，于是 A, C 之间出现间断。

下面的例子画的是 $[0, \pi]$ 之间的 $\tan x$ 的图像，以 $\frac{\pi}{2}$ 为间断点：




```
\tikz \datavisualization
  [scientific axes=clean,
    x axis={grid={major at=(pi/2)}},
    visualize as smooth line]
data [format=function] {
  var x : interval[0:pi/2-0.1];
  func y = tan(\value x r);
}
data point [outlier]
data [format=function] {
  var x : interval[pi/2+0.1:pi];
  func y = tan(\value x r);
};
```

`/tikz/data visualizers/visualize as smooth line=`⟨*visualizer name*⟩ (default `line`)

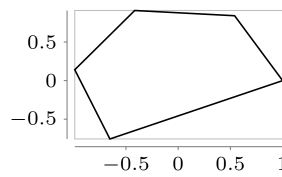
这是 `visualize as line=`⟨*visualizer name*⟩, ⟨*visualizer name*⟩`=smooth line` 的简捷用法。

`/tikz/data visualization/visualizer options/straight line` (no value)

用在 ⟨*visualizer name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，使得显像器的作用是用直线段连接数据点。

`/tikz/data visualization/visualizer options/straight cycle` (no value)

用在 ⟨*visualizer name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，使得显像器的作用是将数据点连接为封闭的多边形。




```
\tikz [scale=.55] \datavisualization
  [scientific axes=clean, all axes={ticks=few},
   visualize as smooth line=my data,
   my data={straight cycle}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

`/tikz/data visualization/visualizer options/polygon` (no value)

等效于前一个选项。

`/tikz/data visualization/visualizer options/smooth line` (no value)

用在 ⟨*visualizer name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，使得显像器的作用是用直线段连接数据点并使得连接点处平滑。

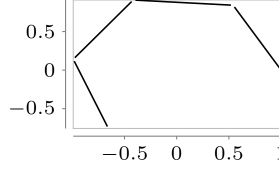
`/tikz/data visualization/visualizer options/smooth cycle` (no value)

用在 ⟨*visualizer name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，使得显像器的作用是将数据点连接为多边形并使得连接点处平滑。

```
\tikz [scale=.55] \datavisualization
  [scientific axes=clean, all axes={ticks=few},
   visualize as smooth line=my data,
   my data={smooth cycle}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/gap line (no value)

用在 ⟨*visualizer name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，使得显像器的作用是用直线段连接数据点，但直线段并不接触数据点位置，而是留有一段空缺。这个效果用命令 `\pgfplothandlergaplineto` 实现。




```
\tikz [scale=.55] \datavisualization
  [scientific axes=clean, all axes={ticks=few},
   visualize as smooth line=my data,
   my data={gap line}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/gap cycle (no value)

作用类似 `gap line`, 只是将数据点连接为多边形。

/tikz/data visualization/visualizer options/no lines (no value)

用在 ⟨*visualizer name*⟩=⟨*options*⟩ 的 ⟨*options*⟩ 中，使得显像器不画线。

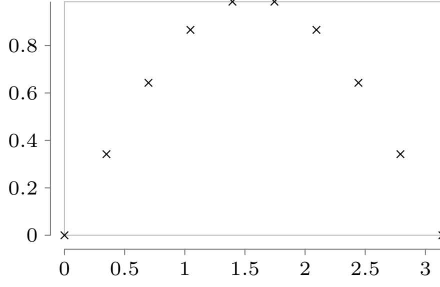
83.3.2 散点显像器

/tikz/data visualizers/visualize as scatter=⟨*visualizer name*⟩ (default `scatter`)

本选项创建名称 ⟨*visualizer name*⟩, 这个名称指向（调用）显像器 `visualize as scatter`. 如果不给出名称 ⟨*visualizer name*⟩, 就默认名称为 scatter. 本选项是

```
visualize as line=⟨visualizer name⟩,
⟨visualizer name⟩=no lines,
style={mark=...}
```

的简捷形式。




```
\tikz \datavisualization
  [scientific axes=clean,
   visualize as scatter]
data [format=function] {
  var x : interval [0:pi] samples 10;
  func y = sin(\value x r);
};
```

83.4 创建新的显像器

84 样式表与图例

84.1 Overview

一个可视化图形中可能有多个曲线，为了区别它们，每个曲线应该有单独的样式。为多个曲线设置单独样式的便捷方法是使用 style sheet，即“样式表”——由多个样式按次序组成的列表。在可视化命令中使用 style sheet 选项后，样式表中的第 1 个样式用于第 1 个曲线，第 2 个样式用于第 2 个曲线……，这都是自动完成的，无需手工设置曲线样式。

有数种类型的样式表，有的样式表专门设置颜色，有的样式表专门设置线型，所以在一个可视化命令中可以同时使用颜色 style sheet 和线型 style sheet. 例如

```
\datavisualization [...style sheet=strong colors, style sheet=vary dashing]...;
```

其中用了设置颜色的样式表 `strong colors` 和设置线型的样式表 `vary dashing`. 样式表 `strong colors` 中的颜色是那些“在白色背景下，具有最大对比度”的颜色，所以其颜色对比效果比较好。样式表 `vary dashing` 中的虚线则是那些尽量让虚线的实线段穿过数据点的虚线，尽量不让数据点处于虚线的空缺部分，所以一般情况下其效果会较好。

另外也有预定义的散点样式表，其中的散点样式都是易于相互区分的，即使两个点标记重合。所以推荐使用预定义的 style sheet.

前文提到，选项 ⟨*visualizer name*⟩`={style={`⟨*TikZ options*⟩`}}` 也可以设置显像器的外观效果，如果同时使用这个选项和样式表来规定显像器的外观效果，那么二者对显像器的规定会被累计，所以在使用样式表的时候，也可以使用这个选项来对显像器做某些调整。

84.2 Style Sheets 的例子

style sheet 可以与数据点标识符关联，即与 `/data point/`⟨*attribute*⟩ 关联，在默认情况下与 `/data point/set=`⟨*visualizer name*⟩ 关联，使得不同名称显像器的效果对应 style sheet 中的不同样式。也就是说，各个显像器名称组成集合 $A = \{a_1, a_2, \cdots\}$, 样式表中的各个样式名称组成集合 $S = \{s_1, s_2, \cdots\}$, 则在 A 与 S 之间有对应关系：显像器 a_1 的样式是 s_1, 显像器 a_2 的样式是 s_2 ……你只需要提供各个显像器名称 (集合 A)，这种对应可以由程序自动完成。

下面先定义一个数据点组，其中每个 `data` 指令都使用单独的显像器名称：

```
\tikz \datavisualization
  data group {function classes} = {
    data [set=log, format=function] {
      var x : interval [0.2:2.5];
      func y = ln(\value x);
    }
    data [set=lin, format=function] {
      var x : interval [-2:2.5];
      func y = 0.5*\value x;
    }
    data [set=squared, format=function] {
```

```
    var x : interval [-1.5:1.5];
    func y = \value x*\value x;
  }
  data [set=exp, format=function] {
    var x : interval [-2.5:1];
    func y = exp(\value x);
  }
};
```

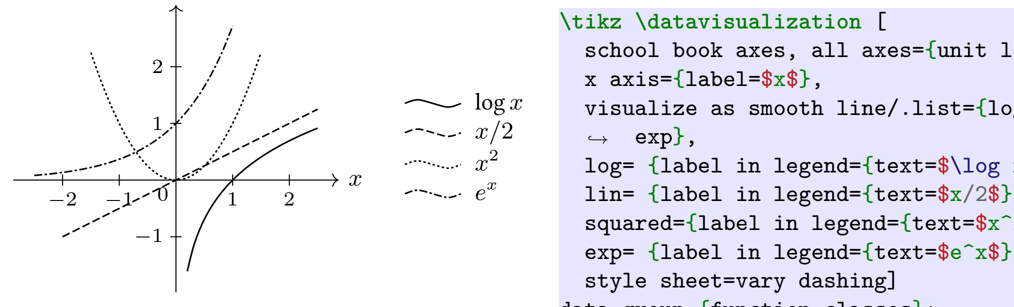
```
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  visualize as smooth line/.list={log, lin, squared, exp},
  style sheet=strong colors]
data group {function classes};
```

```
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  visualize as smooth line/.list={log, lin, squared, exp},
  style sheet=vary dashing]
data group {function classes};
```

```
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  visualize as smooth line/.list={log, lin, squared, exp},
  style sheet=vary dashing,
  style sheet=strong colors]
data group {function classes};
```

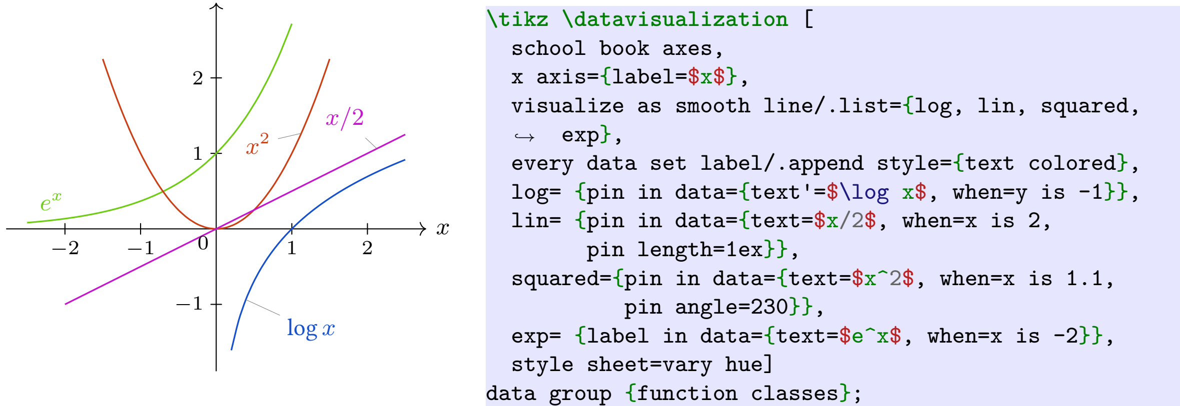
84.3 Legends 的例子

Legends，即“图例”，是对图形的注释。在可视化图形中，添加图例的方式比较灵活。下面的例子里，在 ⟨*visualizer name*⟩=⟨*options*⟩ 中使用 `label in legend={...}` 添加图例：




```
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared,
  ↪  exp},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

下面的例子里，在 ⟨*visualizer name*⟩=⟨*options*⟩ 中使用 `pin in data={...}` 和 `label in data={...}` 添加图例：




```
\tikz \datavisualization [
  school book axes,
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared,
  ↪  exp},
  every data set label/.append style={text colored},
  log= {pin in data={text'=$\log x$, when=y is -1}},
  lin= {pin in data={text=$x/2$, when=x is 2,
        pin length=1ex}},
  squared={pin in data={text=$x^2$, when=x is 1.1,
           pin angle=230}},
  exp= {label in data={text=$e^x$, when=x is -2}},
  style sheet=vary hue]
data group {function classes};
```

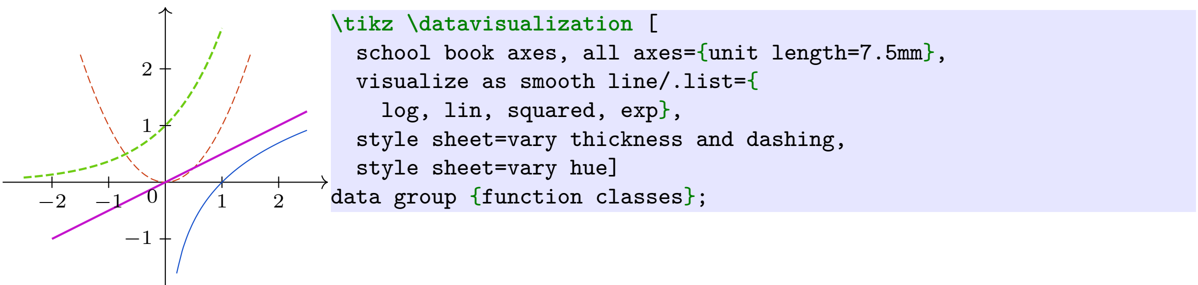
84.4 Style Sheet 的用法

84.4.1 引入一个 Style Sheet

Style Sheet 是与变量名 (attribute) 对应的。

`/tikz/data visualization/style sheet`=⟨*style sheet*⟩ (no default)

本选项引入名称为 ⟨*style sheet*⟩ 的样式表，在默认下样式表 ⟨*style sheet*⟩ 与标识符 `set` 相关联，使得 `set`=⟨*visualizer name*⟩ 指定的不同名称的显像器对应样式表中不同的样式。



```
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  visualize as smooth line/.list={
    log, lin, squared, exp},
  style sheet=vary thickness and dashing,
  style sheet=vary hue]
data group {function classes};
```

Key handler ⟨*attribute*⟩/.style sheet=⟨*style sheet*⟩

⟨*attribute*⟩ 的路径前缀是 /data point/. 这个手柄将标识符 ⟨*attribute*⟩ 与样式表 ⟨*style sheet*⟩ 关联起来，使得 ⟨*attribute*⟩ 的不同值与样式表中不同的样式相对应。上面的选项 style sheet 就是用这个手柄定义的：

```
style sheet/.style={/data point/set/.style sheet={#1}}
```

即将样式表 #1 与显像器名称标识符 set 相关联。

84.4.2 创建新的样式表

样式表作为 key 被创建，故要使用命令 \pgfkeys, 用下面的语句创建一个样式表：

```
\pgfkeys{
  /pgf/data visualization/style sheets/⟨style sheet name⟩/⟨style 1 name⟩/.style={⟨TikZ set⟩},
  /pgf/data visualization/style sheets/⟨style sheet name⟩/⟨style 2 name⟩/.style={⟨TikZ set⟩},
  /pgf/data visualization/style sheets/⟨style sheet name⟩/⟨style 3 name⟩/.style={⟨TikZ set⟩},
......
}
```

其中的 ⟨*style sheet name*⟩ 是创建的样式表的名称；⟨*style 1 name*⟩, ⟨*style 2 name*⟩, ... 是样式表名称 ⟨*style sheet name*⟩ 的“子键”（subkey），也是各个样式的名称；⟨*style 1 name*⟩ 是样式表中第 1 个样式的名称，使用手柄 /.style 来规定（其中采用 TikZ 的选项）。

创建样式表后，为了使用这个样式表，需要把 ⟨*style sheet name*⟩ 与某个 ⟨*attribute*⟩ 关联起来：

/data point/⟨*attribute*⟩/.style sheet=⟨*style sheet name*⟩

当 ⟨*attribute*⟩=⟨*style 1 name*⟩ 时，应用样式表中第 1 个样式；当 ⟨*attribute*⟩=⟨*style 2 name*⟩ 时，应用样式表中第 2 个样式……

还可以设置新建样式表的默认样式：

/pgf/data visualization/style sheets/⟨*style sheet name*⟩/default style=⟨*value*⟩ (style, no default)

这个选项设置样式表 ⟨*style sheet name*⟩ 的默认样式。当 ⟨*attribute*⟩=⟨值⟩ 的 ⟨值⟩ 不是样式表名称的子键时，即样式表中没有名称为 ⟨值⟩ 的样式时，就使用名称为 ⟨*value*⟩ 的样式。

下面定义一个名称为 traffic light 的样式表：

```
\pgfkeys{
  /pgf/data visualization/style sheets/traffic light/.cd,
  ↪ % 样式表名称为 traffic light
  1/.style={green!50!black},  % 第 1 个样式
  2/.style={yellow!90!black},  % 第 2 个样式
  3/.style={red!80!black},  % 第 3 个样式
  default style/.style={black}  % 默认样式
}
```

前文提到，当用选项 visualize as...=⟨*visualizer name*⟩ 给显像器命名时，程序会自动创建 key 路径

`/tikz/data visualization/`⟨*visualizer name*⟩

此时可以用 ⟨*visualizer name*⟩=⟨*options*⟩ 对这个显像器做设置，例如使其值等于样式表中的某个样式名称：

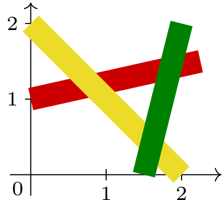
⟨*visualizer name*⟩=⟨*style i name*⟩

从而使得该显像器对应样式 ⟨*style i name*⟩.

在默认下，⟨*style sheet name*⟩ 与选项 `set` 关联，但实际上是通过 `set` 与显像器联系起来，联系的方式有下面三种：

1. ⟨*style i name*⟩ 是样式表中第 i 个样式的名称，同时也是个显像器名称，通过 `set=`⟨*style i name*⟩ 将这个显像器与第 i 个样式联系起来。
2. ⟨*style i name*⟩ 是样式表中第 i 个样式的名称，也是显像器名称的值 ⟨*visualizer name*⟩=⟨*style i name*⟩，通过 `set=`⟨*visualizer name*⟩ 将这个显像器与第 i 个样式联系起来。
3. 将样式表中各个样式的名称规定为 `1, 2, 3, ...`，显像器名称 `visualize as ...=`⟨*visualizer name*⟩ 随意，并且设置 `set=`⟨*visualizer name*⟩. 此时程序会自动为显像器名称 ⟨*visualizer name*⟩ 附带一个初始值，第一个显像器的初始值是 1, 第二个显像器的初始值是 2, ... 也就是说，自动为显像器名称编号，序号从 1 开始，这样就又回到了前一种方式。

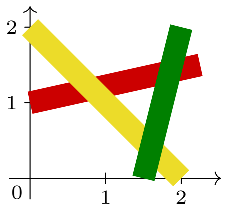
下面用前文定义的 traffic light 按第一个方式绘图：




```
\tikz \datavisualization [
  school book axes,
  visualize as line=1,  % 显像器名称与第 1 个样式一致，故使用第 1 个样式
  visualize as line=2,  % 显像器名称与第 2 个样式一致，故使用第 2 个样式
  visualize as line=3,  % 显像器名称与第 3 个样式一致，故使用第 3 个样式
  every visualizer/.style={line width=3mm},
  style sheet=traffic light]  % 使用样式表 traffic light
data point [x=1.5, y=0, set=1]  % 用第 1 个显像器画绿色线
data point [x=2, y=2, set=1]
data point [x=0, y=2, set=2]  % 用第 2 个显像器画黄色线
data point [x=2, y=0, set=2]
data point [x=0, y=1, set=3]  % 用第 3 个显像器画红色线
data point [x=2.25, y=1.5, set=3];
```

可见样式表中的第 1 个样式效果出现在图形的最上层，会遮挡其余样式的效果。

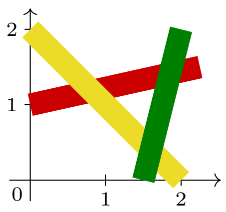
下面按第二个方式绘图：




```
\begin{tikzpicture}
\datavisualization data group {lines} = {
  data point [x=1.5, y=0, set=normal]  % 用第 1 个显像器
  data point [x=2, y=2, set=normal]
```

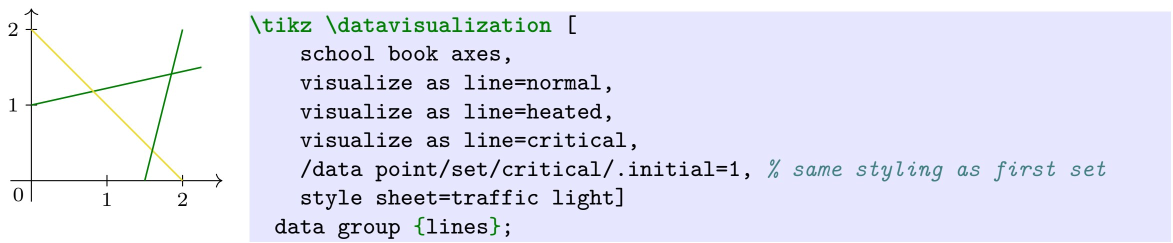
```
  data point [x=0, y=2, set=heated]  % 用第 2 个显像器
  data point [x=2, y=0, set=heated]
  data point [x=0, y=1, set=critical]  % 用第 3 个显像器
  data point [x=2.25, y=1.5, set=critical]};
\datavisualization [
  school book axes,
  visualize as line=normal,  % 第 1 个显像器名称
  visualize as line=heated,  % 第 2 个显像器名称
  visualize as line=critical,  % 第 3 个显像器名称
  every visualizer/.style={line width=3mm},
  /data point/set/normal/.initial=1,
  ↪  % 指定第 1 个显像器名称的初始值为 1，与样式表的第 1 个样式名称一致
  /data point/set/heated/.initial=2,
  ↪  % 指定第 2 个显像器名称的初始值为 2，与样式表的第 2 个样式名称一致
  /data point/set/critical/.initial=3,
  ↪  % 指定第 3 个显像器名称的初始值为 3，与样式表的第 3 个样式名称一致
  style sheet=traffic light]  % 使用样式表 traffic light
data group {lines};
\end{tikzpicture}
```

下面按第三种方式绘图：




```
\begin{tikzpicture}
\datavisualization data group {lines} = {
  data point [x=1.5, y=0, set=normal]  % 用第 1 个显像器
  data point [x=2, y=2, set=normal]
  data point [x=0, y=2, set=heated]  % 用第 2 个显像器
  data point [x=2, y=0, set=heated]
  data point [x=0, y=1, set=critical]  % 用第 3 个显像器
  data point [x=2.25, y=1.5, set=critical]};
\datavisualization [
  school book axes,
  visualize as line=normal,  % 第 1 个显像器名称
  visualize as line=heated,  % 第 2 个显像器名称
  visualize as line=critical,  % 第 3 个显像器名称
  every visualizer/.style={line width=3mm},
  style sheet=traffic light]  % 使用样式表 traffic light
data group {lines};
\end{tikzpicture}
```

修改上面图形中第三条线的颜色，使之与第一条线颜色相同：



```
\tikz \datavisualization [
    school book axes,
    visualize as line=normal,
    visualize as line=heated,
    visualize as line=critical,
    /data point/set/critical/.initial=1, % same styling as first set
    style sheet=traffic light]
  data group {lines};
```

还有一个专门用于创建样式表的命令：

\pgfdvdeclarestylesheet{⟨*style sheet name*⟩}{⟨*keys*⟩}

例如，前面用命令 \pgfkeys 创建的样式表 traffic light 也可以如下得到：

```
\pgfdvdeclarestylesheet{traffic light}{
  1/.style={green!50!black},
  2/.style={yellow!90!black},
  3/.style={red!80!black},
  default style/.style={black}
}
```

这个命令会自动添加路径前缀 /pgf/data visualization/style sheets/.

另外，在定义样式表时，其中各个样式的代码中可以含有一个变量，这个变量的值就是当前的显像器名称的序号。

```
\begin{tikzpicture}
\datavisualization data group {lines} = {
  data point [x=1.5, y=0, set=normal]  % 用第 1 个显像器
  data point [x=2, y=2, set=normal]
  data point [x=0, y=2, set=heated]  % 用第 2 个显像器
  data point [x=2, y=0, set=heated]
  data point [x=0, y=1, set=critical]  % 用第 3 个显像器
  data point [x=2.25, y=1.5, set=critical]};

\pgfdvdeclarestylesheet{my dashings}{
  default style/.style={dash pattern={on #1*2pt off #1*2pt}, line width=#1mm}
}  % 定义样式表 my dashings，其中只有默认样式

\datavisualization [
  school book axes,
  visualize as line=normal, normal={style=green!50!black},
  visualize as line=heated, heated={style=yellow!90!black},
  visualize as line=critical, critical={style=red!80!black},
  style sheet=my dashings]
data group {lines};
\end{tikzpicture}
```

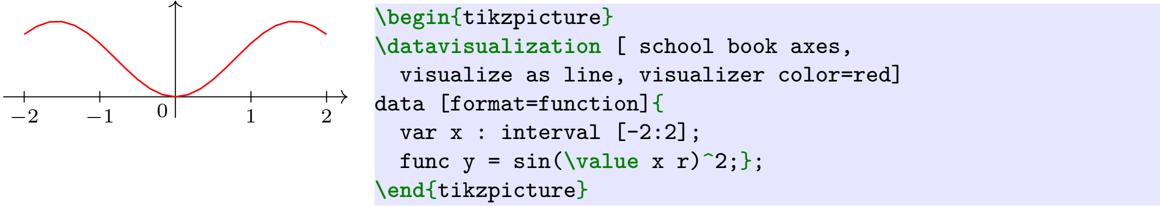
在上面例子中，样式表 my dashings 的默认样式的代码中，变量 #1 会以显像器名称的序号为值。

84.4.3 创建新的颜色样式表

创建颜色样式表时，最好用选项 visualizer color=⟨*color*⟩ 来规定颜色，因为这个选项不仅规定颜色，还有其它的作用。后文的选项 /tikz/data visualization/visualizer label options/text colored →P. 492 会调用样式表中相应样式的 visualizer color=⟨*color*⟩ 设置。涉及颜色的各种预定义样式表中，各个样式的颜色都用选项 visualizer color 规定，因此都能与选项 text colored 相配合。

/tikz/visualizer color=⟨*color*⟩ (no default)

这个选项可以直接用作可视化命令 \datavisualization 的选项，也可以用作选项 style={} 的参数，也可以用作自定义样式表中手柄 /.style 的参数，但不能直接用作显像器名称的参数，因为这个 key 的路径前缀是 /tikz/.



```
\begin{tikzpicture}
\datavisualization [ school book axes,
  visualize as line, visualizer color=red]
data [format=function]{
  var x : interval [-2:2];
  func y = sin(\value x r)^2;};
\end{tikzpicture}
```

下面的例子用 visualizer color 定义一个颜色样式表，并且画出前面例子定义的数据点组 data group {lines}.

```
\begin{tikzpicture}
\pgfdvdeclarestylesheet{my colors}{
  default style/.style={visualizer color=black},  % 直接用选项 visualizer color 规定颜色
    1/.style={visualizer color=black},
    2/.style={visualizer color=red!80!black},
    3/.style={visualizer color=blue!80!black},
}

\datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  style sheet=my colors]
data group {lines};
\end{tikzpicture}
```

还有一个专门用于创建颜色样式表的命令，其中使用颜色的思路来自宏包 xcolor, 先为第一个数据点（组）指定初始颜色，之后的数据点（组）的颜色都是将初始颜色做“偏移”后的颜色。

\tikzdvdeclarestylesheetcolorseries{⟨*name*⟩}{⟨*color model*⟩}{⟨*initial color*⟩}{⟨*step*⟩}

这个命令实际上使用命令 \pgfdvdeclarestylesheet 来工作。下面是个例子：

```
\tikzdvdeclarestylesheetcolorseries{greens}{hsb}
  {0.3,1.3,0.8} {0,-.4,-.1}
\tikz \datavisualization [
    school book axes,
    visualize as line=normal,
    visualize as line=heated,
    visualize as line=critical,
    style sheet=greens]
  data group {lines};
```

这个例子中，颜色样式表的名称是 greens, 颜色模式是 hsb, 在这个颜色模式下，第 1 个颜色样式

（初始颜色）由参数（向量）{0.3,1.3,0.8} 规定，第 2 个颜色由第 1 个颜色的参数（向量）加上步长（向量）{0,-.4,-.1} 得到，即 {0.3,.9,.7}.

84.5 预定义的线型样式表

预定义的线型样式表有 vary thickness，vary dashing，vary dashing and thickness.

为了展示这些预定义的线型样式表，先定义一组显像器以及一组函数数据。

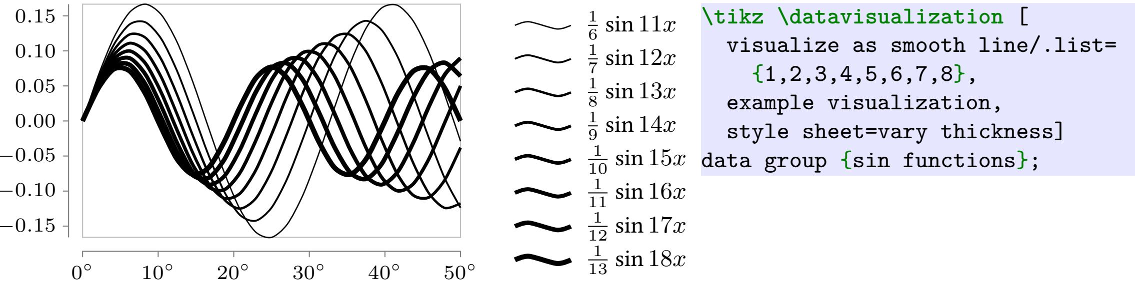
```
\tikzdatavisualizationset {
  example visualization/.style={
    scientific axes=clean,
    y axis={ticks={style={ % 设置 y 轴刻度值的外观
      /pgf/number format/fixed,
      /pgf/number format/fixed zerofill,
      /pgf/number format/precision=2}}},
    x axis={ticks={tick suffix=${}^\circ$}}, % 设置 x 轴刻度值的外观
    1={label in legend={text=$\frac{1}{6}\sin 11x$}}, % 设置序号为 1 的显像器的图例
    2={label in legend={text=$\frac{1}{7}\sin 12x$}},
    3={label in legend={text=$\frac{1}{8}\sin 13x$}},
    4={label in legend={text=$\frac{1}{9}\sin 14x$}},
    5={label in legend={text=$\frac{1}{10}\sin 15x$}},
    6={label in legend={text=$\frac{1}{11}\sin 16x$}},
    7={label in legend={text=$\frac{1}{12}\sin 17x$}},
    8={label in legend={text=$\frac{1}{13}\sin 18x$}}
  }
}

\tikz \datavisualization data group {sin functions} = {
  data [format=function] {
    var set : {1,...,8}; % 把标识符 \ttt{set} 看作是变量，规定其值
    var x : interval [0:50];
    func y = sin(\value x * (\value{set}+10))/(\value{set}+5);
  }
};
```

Style sheet vary thickness

这个样式表针对线宽，其中各个样式具有不同的线宽，第一个样式线宽最细。

这个样式表在文件《tikzlibrarydatavisualization.code.tex》中的定义是：

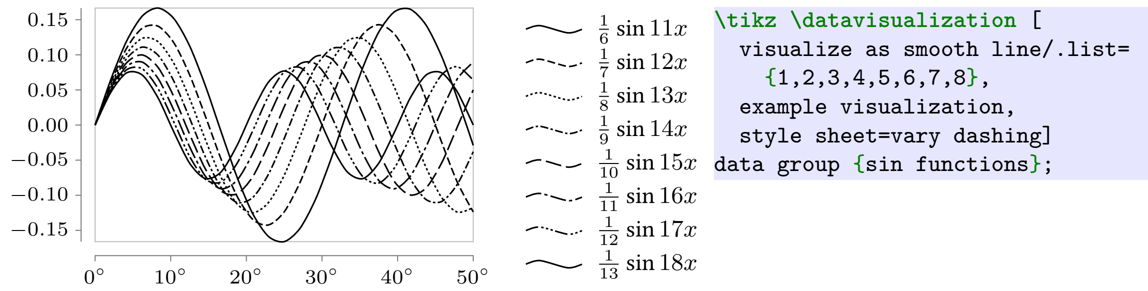
```
\pgfdvdeclarestylesheet{vary thickness}
{
  default style/.style={line width={0.3pt+#1*0.2pt}}
}%
```




```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary thickness]
data group {sin functions};
```

Style sheet vary dashing

这个样式表中的各个样式具有不同的线型，第一个样式是实线，第二个样式是虚线，第三个样式是点线，第四个样式是点划线，等等。这个样式表中共有 7 个线型样式，如果显像器个数超过 7 个，则第 8 个及以后的显像器都自动用实线 (solid)，此时需要手工设置这些显像器的线型。



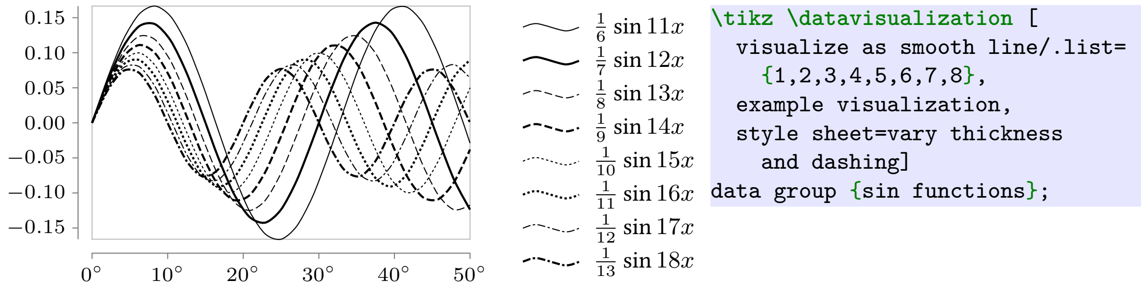

```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary dashing]
data group {sin functions};
```

Style sheet vary dashing and thickness

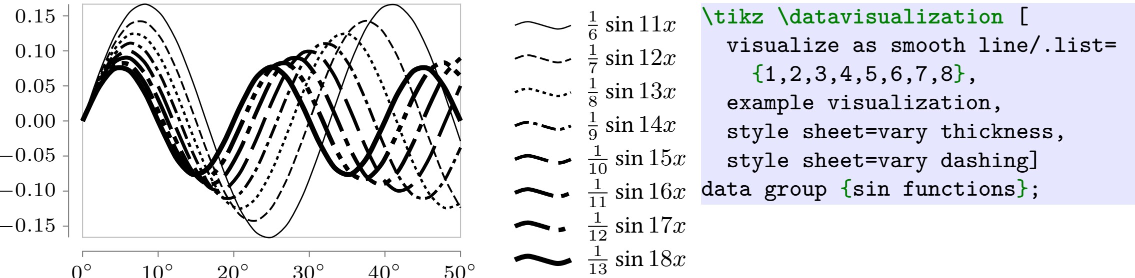
这个样式表中的各个样式在线型和线宽两方面相互区别，共有 14 个样式。注意这个样式表不是

```
style sheet=vary thickness
style sheet=vary dashing
```

这两个样式表的简单叠加。




```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary thickness
    and dashing]
data group {sin functions};
```

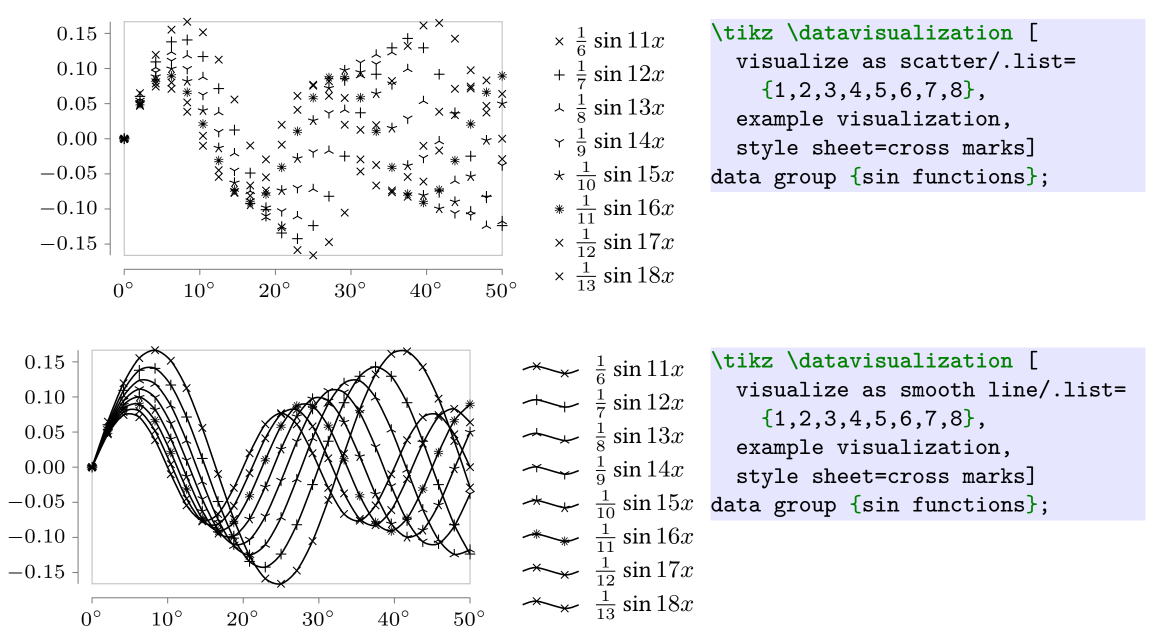


84.6 预定义的散点样式表

为了能使用多种点标记符号，在使用散点样式表之前，先在导言区调用程序库 plotmarks.

Style sheet `cross marks`

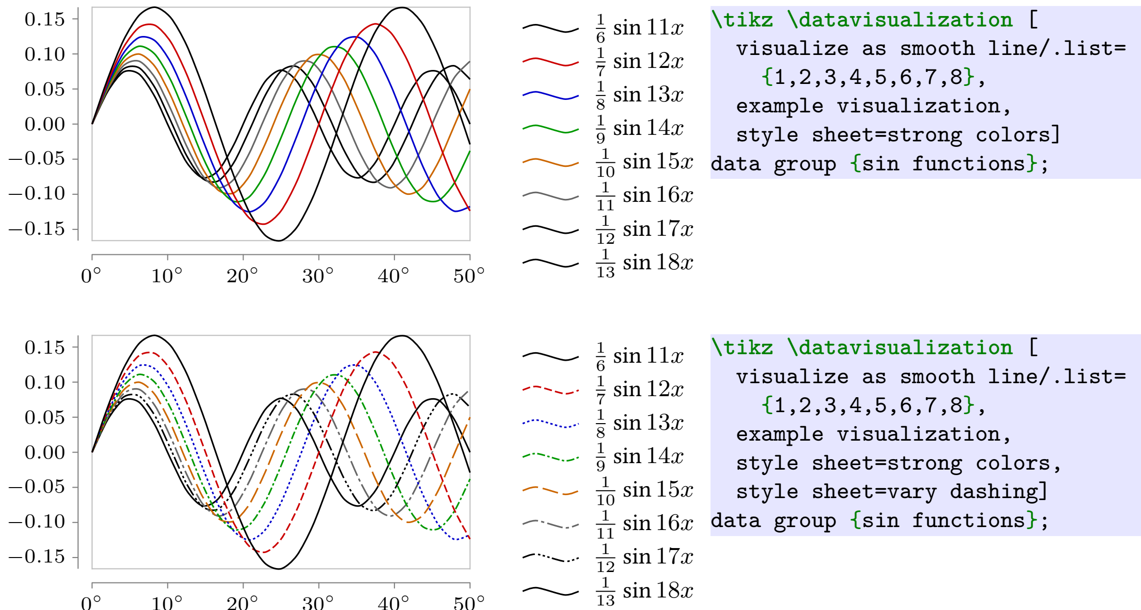
这个样式表中的各个样式的区别在于标记散点的标记符号不同，这些标记符号不是随意选择的。当两个样式的标记符号重叠时，仍然能分辨它们，所以在使用多个散点显像器时，建议使用这个样式表。这个样式表共有 6 个样式，分别提供一种散点标记符号。第 1 个样式是叉号，第 2 个样式是加号。如果显像器个数多于 6 个，则第 7 个以及以后的显像器都自动使用第一个样式的散点标记符号，此时可能需要手工设置这些显像器的标记符号。



84.7 预定义的颜色样式表

Style sheet `strong colors`

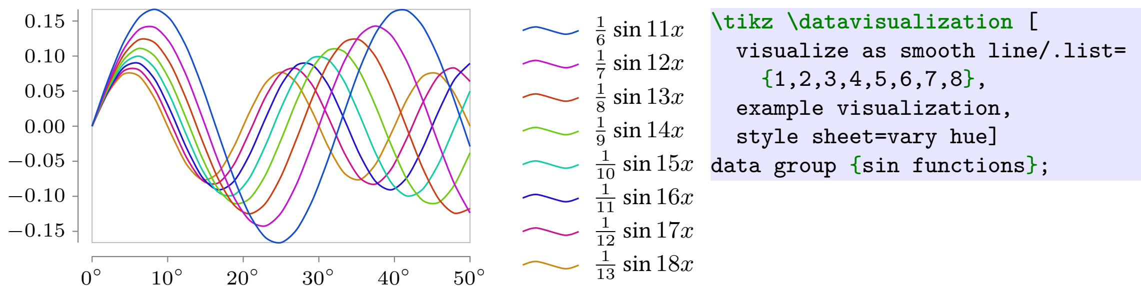
前面已经提到，这个样式表中各个样式使用的颜色具有（在白色背景下）最大的视觉对比效果。这个样式表共有 6 个样式，第 1 个样式是黑色，第 2 个样式是红色。如果显像器个数多于 6 个，则第 7 个以及以后的显像器都自动使用黑色。



与 `strong colors` 不同，下面几个颜色样式表是用 `\tikzdvdeclarestylesheetcolorseries` 定义的，其中的样式个数是可变的，要多少有多少。

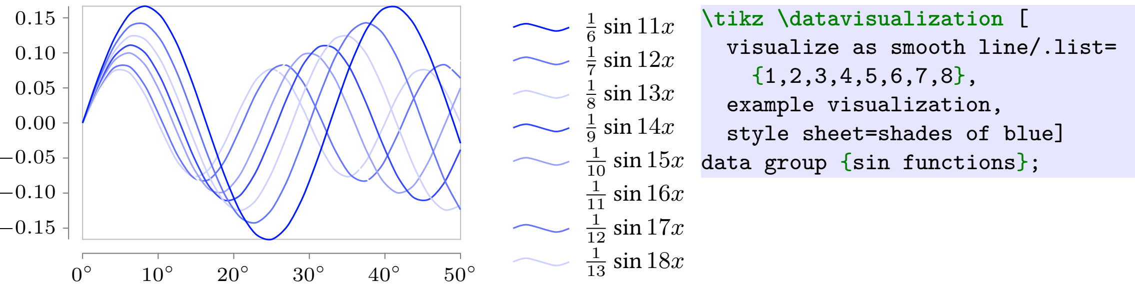
Style sheet `vary hue`

单词 hue 的意思是：色度，色彩，色调。这个样式表中的各个样式的区别在于 hue 不同。



Style sheet `shades of blue`

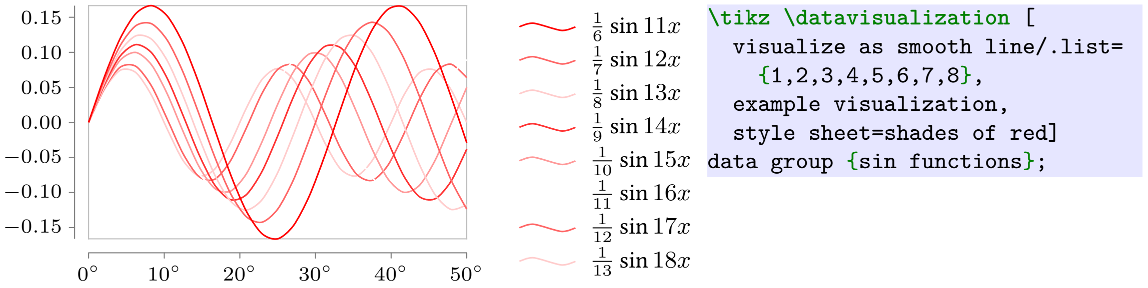
这个样式表中的各个样式都使用蓝色，区别在于蓝色的深浅不同。



```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=shades of blue]
data group {sin functions};
```

Style sheet `shades of red`

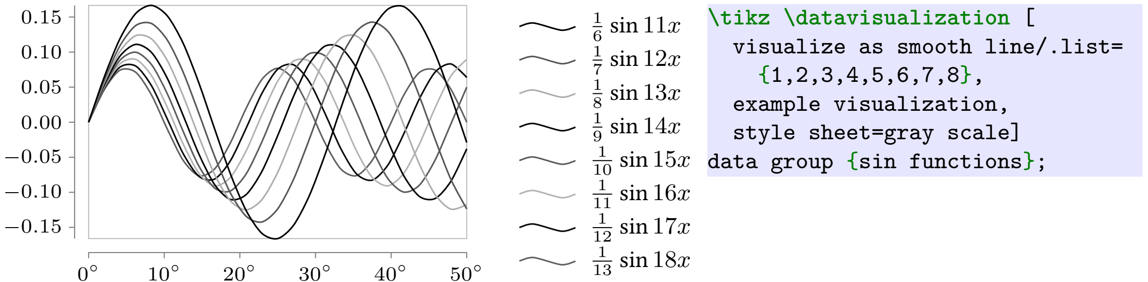
这个样式表中的各个样式都使用红色，区别在于红色的深浅不同。



```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=shades of red]
data group {sin functions};
```

Style sheet `gray scale`

这个样式表中的各个样式都使用灰色，区别在于灰色的深浅不同。



```
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=gray scale]
data group {sin functions};
```

84.8 显像器的标签

一组数据点对应一个显像器名称，相应的显像器将该数据点组可视化为一个图形，在二维之下这个图形通常是一组散点或一个曲线。由于一个显像器处理一组数据点，可以把一组数据点看作是属于相应显像器的数据点。显像器决定这组数据点的可视化外观，其中包括这组数据点的标签，也就是这组数据点的图形的标签。因为一组数据点对应一个显像器，数据点的标签也可以看作是显像器的标签。例如对于曲线来说，这种标签会放在该曲线上某一点的旁边，起到注释曲线的作用。

设置这种标签的选项一般应作为显像器的参数。

84.8.1 给一组数据点设置标签

下一选项设置某个显像器的标签。

/tikz/data visualization/visualizer options/label in data=⟨*options*⟩ (no default)

这个选项作为显像器的参数，给一组数据点设置标签。可以在一个显像器的参数中多次使用这个选项，给显像器设置多个标签。这里 ⟨*options*⟩ 是关于标签的设置，其中的选项会被冠以前缀

```
/tikz/data visualization/visualizer label options
```

来执行。在 ⟨*options*⟩ 中可以使用下面介绍的选项。

/tikz/data visualization/visualizer label options/text=⟨*text*⟩ (no default)

这个选项设置标签的文字内容，该文字标签通常位于图形的左侧。该选项的定义中使用了 `auto` 选项。

/tikz/data visualization/visualizer label options/text'=⟨*text*⟩ (no default)

这个选项设置标签的文字内容，该文字标签通常位于图形的右侧。该选项的定义中使用了 `auto,swap` 选项。

假设一组数据点可视化为一个曲线，其中各个数据点的先后次序决定曲线的方向。在给数据点添加标签时，通常采用某个方法选定某个数据点 A，记该点之后的数据点是 B，从 A 到 B 决定一个 (虚拟的) 有向线段，这个线段有“左侧”和“右侧”，标签就放在其左侧（对应选项 `text`）或者右侧（对应选项 `text'`）且位于 A 点附近，即标签以 A 为锚定点。所选的锚定点 A 会被储存在（`label visualizer coordinate`）中，而其后的点 B 会被储存在（`label visualizer coordinate'`）中。标签的样式则可以使用 `node style` 来设置。这种放置标签的机制体现在下面的选项中。

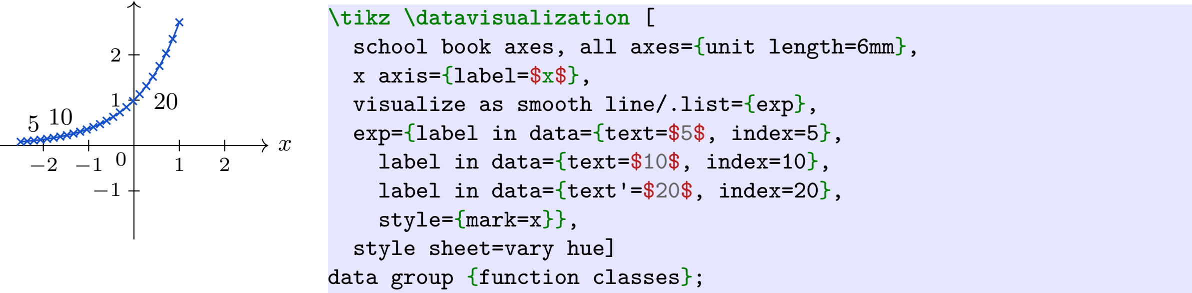
/tikz/data visualization/visualizer label options/when=⟨*attribute*⟩ `is` ⟨*number*⟩ (no default)

⟨*attribute*⟩ 是数据的标识符或者说变量名。以绘制函数曲线来讲，一般默认使用 25 个样本点（即数据点），假如设置 `when=x is 5.3`, 则规定曲线标签的锚定点是坐标分量 x 值不小于 `x=5.3` 的样本点。如果分量 x 值不小于 `x=5.3` 的样本点找不到，就把最后一个数据点（样本点）作为标签的锚定点。

```
\tikz \datavisualization [
  school book axes, all axes={unit length=6mm},
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  log= {label in data={text'=$\log x$, when=y is -1, text
  ↪  colored}},
  lin= {label in data={text=$x/2$, when=x is 2}},
  squared={label in data={text=$x^2$, when=x is 1.1}},
  exp= {label in data={text=$e^x$, when=x is -2, text colored}},
  style sheet=vary hue]
data group {function classes};
```

/tikz/data visualization/visualizer label options/index=⟨*number*⟩ (no default)

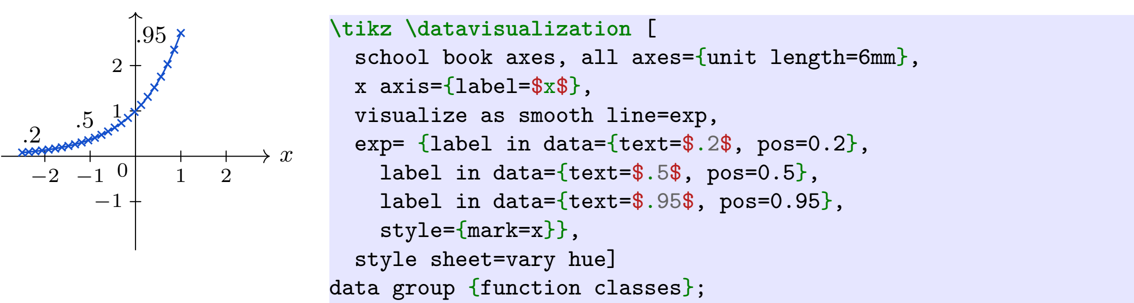
这里 ⟨*number*⟩ 是个正整数，作为序号来索引数据点。该选项选定第 ⟨*number*⟩ 个数据点（样本点）作为标签的锚定点。




```
\tikz \datavisualization [
  school book axes, all axes={unit length=6mm},
  x axis={label=$x$},
  visualize as smooth line/.list={exp},
  exp={label in data={text=$5$, index=5},
    label in data={text=$10$, index=10},
    label in data={text'=$20$, index=20},
    style={mark=x}},
  style sheet=vary hue]
data group {function classes};
```

`/tikz/data visualization/visualizer label options/pos=`⟨*fraction*⟩ (no default)

⟨*fraction*⟩ 是个小数，记该选项所在的显像器的数据点总数为 ⟨*total*⟩，则该选项使得索引序号 `index=`⟨*number*⟩ 不小于 ⟨*fraction*⟩ 与 ⟨*total*⟩ 的乘积。



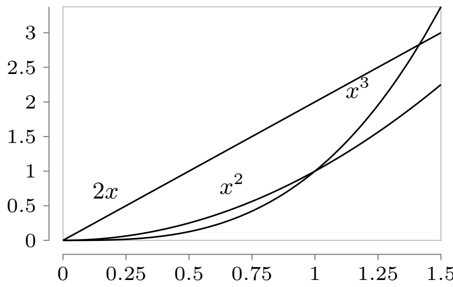

```
\tikz \datavisualization [
  school book axes, all axes={unit length=6mm},
  x axis={label=$x$},
  visualize as smooth line=exp,
  exp= {label in data={text=$.2$, pos=0.2},
    label in data={text=$.5$, pos=0.5},
    label in data={text=$.95$, pos=0.95},
    style={mark=x}},
  style sheet=vary hue]
data group {function classes};
```

`/tikz/data visualization/visualizer label options/auto` (no value)

这个选项是默认执行的选项，即把 `label in data` 作为显像器的参数后，如果不用其它选项指定标签的指向点，就用该选项来指定。假设该选项作为某个显像器的参数，对应的数据点组是 ⟨*data set*⟩. 在可视化过程中可能会有多组数据点，设共有 T 组数据点，而数据点组 ⟨*data set*⟩ 的索引序号是正整数 t, 也就是说数据点组 ⟨*data set*⟩ 是第 t 组数据点。利用下面的算式，本选项确定数据点组 ⟨*data set*⟩ 的标签的锚定点：

$$\texttt{pos} = (t - \tfrac{1}{2})/T$$

假设数据点共有 10 组，分别对应 10 个显像器，每个显像器对应一个数据点组，则该选用 `pos=(1-1/2)/10=0.05` 确定第 1 个数据点组的标签的锚定点；用 `pos=(10-1/2)/10=0.95` 确定第 10 个数据点组的标签的锚定点，这样可以尽量避免标签重叠，当然效果可能欠佳。



```
\tikz \datavisualization [
  scientific axes=clean,
  visualize as smooth line/.list={linear, squared, cubed},
  linear ={label in data={text=$2x$}},
  squared={label in data={text=$x^2$}},
  cubed ={label in data={text=$x^3$}}]
data [set=linear, format=function] {
  var x : interval [0:1.5];
  func y = 2*\value x;
}
data [set=squared, format=function] {
  var x : interval [0:1.5];
  func y = \value x * \value x;
}
data [set=cubed, format=function] {
  var x : interval [0:1.5];
  func y = \value x * \value x * \value x;
};
```

以上选项用于确定标签的位置，用下面的选项可以设置标签的样式外观。

`/tikz/data visualization/visualizer label options/node style=`⟨*options*⟩ (no default)

该选项的 ⟨*options*⟩ 会被传递给 `/tikz/data visualization/node style`. ⟨*options*⟩ 中的选项应该是 TikZ 的选项。

`/tikz/data visualization/visualizer label options/text colored` (no value)

该选项利用 `node style` 将文字的颜色设置为 `visualizer color` 所指定的颜色，即使得标签与数据点的显示颜色一致。所以要想使用这个选项就必须同时使用涉及颜色的样式表，并且该样式表中各个样式都使用 `visualizer color` 来规定颜色。各种预定义的涉及颜色的样式表都使用选项 `visualizer color` 来规定颜色。

注意尽管

```
⟨visualizer name⟩={style={visualizer color=⟨color⟩}}, label in data={text=⟨text⟩},
↪   text colored}
```

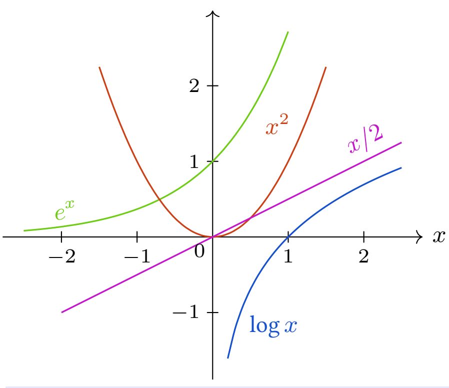
是有效的句法，但是其中的 `visualizer color` 不能被 `text colored` 调用。

另外，如果自定义的样式表中的某个样式的颜色没有使用 `visualizer color` 来规定，而是直接使用 `color=`⟨*color*⟩ 选项，那么这个样式传递给显像器，显像器会使得其中的标签文字也带有这个颜色 ⟨*color*⟩.

`/tikz/data visualization/every data set label` ((style, no value)

这是个样式（style），其中可以用 `node style` 来设置各组数据点的标签的样式。一般情况下，如果没有手工设置各标签的样式，那么各标签就使用预定义的默认样式。因为标签的默认样式有比较好的适用性，最好不要修改，所以通常为默认样式附加某些样式来做适当调整。

下面的例子用该选项为默认样式附加其它样式，其中使用的手柄是 `/.append style`：




```
\tikz \datavisualization [
  school book axes,
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  every data set label/.append style={text colored},
  log= {label in data={text'=$\log x$, when=y is -1}},
  lin= {label in data={text=$x/2$, node style=sloped, when=x is 2}},
  squared={label in data={text=$x^2$, when=x is 1.1}},
  exp= {label in data={text=$e^x$, node style=sloped, when=x is -2}},
  style sheet=vary hue]
data group {function classes};
```

`/tikz/data visualization/every label in data` (style, no value)

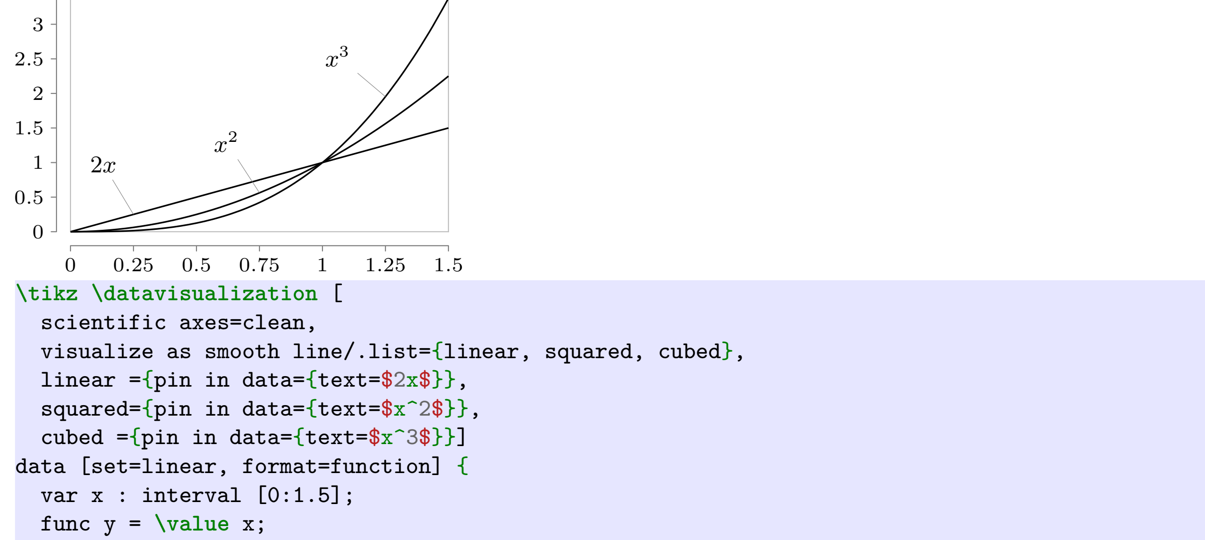
这是个样式（style），类似 `every data set label`，二者的区别是，在各个样式表（style sheet）被执行后才会执行该选项，这就多了一次修改各个标签样式的机会。

84.8.2 给一组数据点设置大头针标签

如果把 `pin in data` 作为显像器的参数，就产生大头针标签。

`/tikz/data visualization/visualizer options/pin in data=`⟨*options*⟩ (no default)

这个选项与 `label in data` 类似，⟨*options*⟩ 中也是使用前文介绍的 `text`，`text'` 选项来设置标签的文字内容（作为大头针的“头”），可以使用 `pos` 等选项来确定标签的锚定点（针尖的指向位置）。




```
\tikz \datavisualization [
  scientific axes=clean,
  visualize as smooth line/.list={linear, squared, cubed},
  linear ={pin in data={text=$2x$}},
  squared={pin in data={text=$x^2$}},
  cubed ={pin in data={text=$x^3$}}]
data [set=linear, format=function] {
  var x : interval [0:1.5];
  func y = \value x;
```

```
}
data [set=squared, format=function] {
  var x : interval [0:1.5];
  func y = \value x * \value x;
}
data [set=cubed, format=function] {
  var x : interval [0:1.5];
  func y = \value x * \value x * \value x;
};
```

上面这个例子就是前文中的例子，二者比较来说，使用大头针标签更清楚明确。
此外在 ⟨*options*⟩ 中还可以使用下面的选项。

`/tikz/data visualization/visualizer label options/pin angle=`⟨*angle*⟩ (no default)

这个选项会使得“大头针”绕它的针尖（标签的锚定点）旋转，⟨*angle*⟩ 用于规定旋转的角度。但是具体的旋转效果并不明显，一般正值角度对应逆时针旋转，负值角度对应顺时针旋转。如果不指定 ⟨*angle*⟩，那么大头针的状态默认为：标签的“针”与线段 `(label visualizer coordinate)--(label visualizer coordinate')` 垂直,垂足为 `(label visualizer coordinate)`; 如果使用选项 `text=`⟨*text*⟩, 则标签在该线段左侧；如果使用选项 `text'=`⟨*text*⟩, 则标签在该线段右侧。

`/tikz/data visualization/visualizer label options/pin length=`⟨*dimension*⟩ (no default)

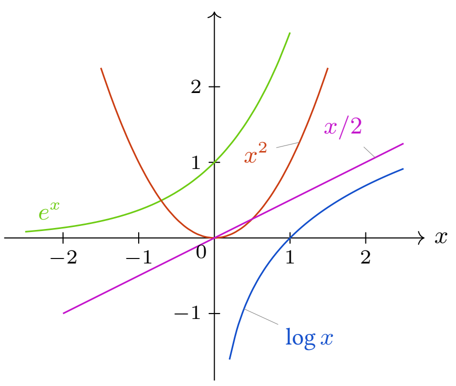
这个选项指定大头针标签的“针”的长度，在大头针围绕针尖的指向点旋转时会保持“针”的长度不变。注意 ⟨*dimension*⟩ 是带单位的尺寸。本选项的定义是：

```
pin length/.style={node style={pin distance={#1}}}
```

所以如果不指定 ⟨*dimension*⟩, 那么“针”的长度与选项 `/tikz/pin distance`→P. 124 有关。

关于大头针标签还可以参考样式 `/tikz/every pin edge`→P. 124 和选项 `/tikz/pin edge`→P. 124, 但是在可视化命令中这两种 key 都难以起作用，很难修改单个大头针标签的“针”的样式。如果要统一修改大头针标签的“针”的样式，可以用以下形式：

```
\tikzset {every pin edge/.style={...}}
或者
\tikz [every pin edge/.style={...}] \datavisualization...;
或者
\begin{tikzpicture}[every pin edge/.style={...}]
  ...
\end{tikzpicture}
```




```
\tikz \datavisualization [
  school book axes,
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  every data set label/.append style={text colored},
  log= {pin in data={text'=$\log x$, when=y is -1}},
  lin= {pin in data={text=$x/2$, when=x is 2, pin length=1ex}},
  squared={pin in data={text=$x^2$, when=x is 1.1, pin angle=230}},
  exp= {label in data={text=$e^x$, when=x is -2}},
  style sheet=vary hue]
data group {function classes};
```

84.9 为数据点组创建图例

图例要比标签复杂。一个可视化图形中可有多个图例，一般情况下，一个图例中含有数个条目，或者称为条目标签。一个条目是对某一个显像器产生的数据点图形的注释。也就是说，条目直接与显像器对应。使用图例的通常做法是：首先创建一个或者数个自命名的图例；然后给需要图例条目注释的显像器使用参数 `label in legend=`⟨*options*⟩ 来产生一个对应该显像器的条目，并在 ⟨*options*⟩ 中使用 `legend=`⟨*name*⟩ 来指定该条目属于那个图例；如果不创建自命名的图例，就默认使用名称为 `main legend` 的图例，并且不必在显像器的参数 `label in legend=`⟨*options*⟩ 中使用 `legend=main legend` 来指定该条目属于默认图例。

一个条目一般包含两部分，文字标签和图示标签。文字标签是用文字做的注释，通常由选项 `text=`⟨*text*⟩ 产生。图示标签是用图形做的注释，由“条目显像器”产生，在默认下是个之字形（zig-zag）线。图例在整体上是个矩阵（matrix）node，它的条目就是它的元素，对于图例的设置就分为对矩阵整体的设置和对各个条目的设置。图例涉及的 key 比较多，这里先概略梳理一下。下面用词“引起”表示开启某个操作模式，用词“针对”表示具体实施某个操作。

以下 3 个 key:

```
/tikz/data visualization/new legend=⟨legend name⟩
/tikz/data visualization/⟨legend name⟩=⟨options⟩
/tikz/data visualization/legend=⟨options⟩
```

其前缀路径都是 `/tikz/data visualization/`，故它们都直接用作可视化命令的选项。第 1 个 key 声明一个名称为 ⟨*legend name*⟩ 的图例，可以用这个选项声明多个图例，即可以在图形中使用多个图例。第 2 个 key 引起对图例 ⟨*legend name*⟩ 的整体设置或者其中所有条目的统一设置 (一个图例就是一个 matrix)。第 3 个 key 引起对默认图例 (即 `main legend`) 的整体设置或者其中所有条目的统一设置。

针对图例整体做设置的 key 一般都有前缀路径:

```
/tikz/data visualization/legend options/
```

它们都用作 ⟨*legend name*⟩ 或 `legend` 的参数。

引起单个条目设置的 key 是:

```
/tikz/data visualization/visualizer options/label in legend=⟨options⟩
```

按照这个 key 的路径，它只能用作显像器名称的参数。它引起与所在显像器相关联的图例条目的设置。

针对单个条目做设置的 key 都有前缀路径:

```
/tikz/data visualization/legend entry options/
```

它们可以作为 `label in legend` 的参数来设置单个条目；也可以用作 ⟨*legend name*⟩ 的参数来对该图例中的所有条目做统一设置；也可以用作 `legend` 的参数来对默认图例中的所有条目做统一设置。

另外还有一些以 `/tikz/data visualization/` 为前缀的样式也用于设置图例，它们直接用作可视化命令的选项。

84.9.1 创建图例，图例中的条目

在默认之下是不给各组数据点创建图例标签的，要想使用图例就要用相关选项做设置。

`/tikz/data visualization/new legend`=⟨*legend name*⟩ (default `main legend`)

这个选项创建一个名称为的作为矩阵整体的图例，其默认名称为 `main legend`. 如果用这个选项创建两个名称相同的图例，则第 2 个无效，也就是说一个图例名称只能用一次。

当使用这个选项后，程序会自动创建下面的键:

`/tikz/data visualization/`⟨*legend name*⟩=⟨*options*⟩ (no default)

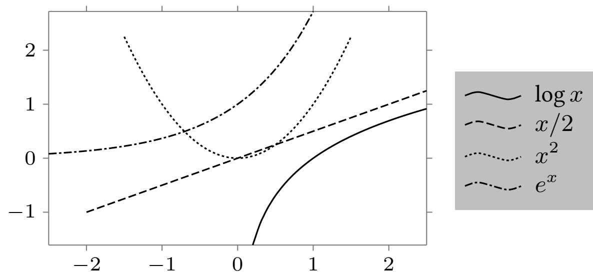
这里 ⟨*options*⟩ 中的选项会被冠以前缀

```
/tikz/data visualization/legend options
```

来执行，用以确定图例的位置，外观，元素条目的排布方式等等。这里的情况类似显像器，使用显像器时，给显像器命名，然后给显像器名称赋选项值，设置显像器的外观效果。

`/tikz/data visualization/legend options/matrix node style`=⟨*options*⟩ (no default)

这个选项设置图例的样式，针对作为 matrix node 的图例整体，而不是其中的各个条目。在 ⟨*options*⟩ 中是 TikZ 的选项。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin, squared, exp},
  legend={matrix node style={fill=black!25}},  % 图例的填充色为 25% 黑
```

```
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

/tikz/data visualization/legend options/every new legend (style, no value)

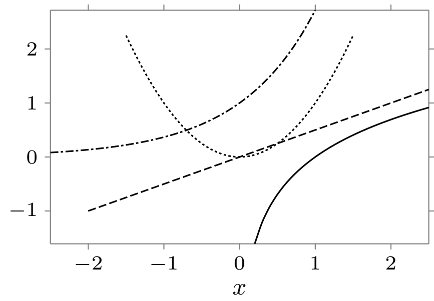
这个样式的默认设置是：`east outside, label style=text right`，即图例位于绘图区域之外且在绘图区的东方（右侧），各个条目中的文字在条目的右侧，如前面的例子所示。

/tikz/data visualization/legend=⟨*options*⟩ (no default)

这个选项是

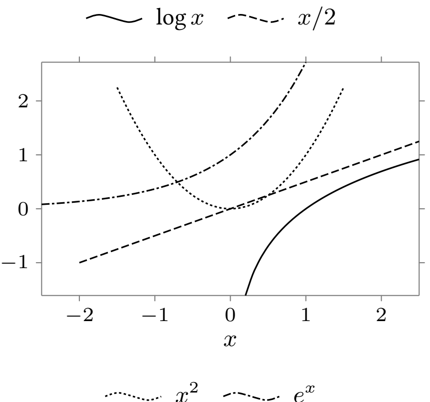
```
new legend=main legend, main legend=⟨options⟩
```

的简化，即这个选项创建一个名称为 ⟨*main legend*⟩ 的图例，并将 ⟨*options*⟩ 赋予这个图例。



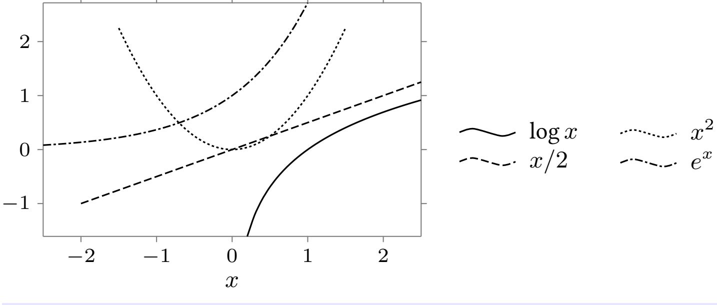

```
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  legend=below,
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

上面的例子中使用选项 `legend=below` 创建名称为 `main legend` 的图例，并将它放在绘图区的下方。




```
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  new legend={upper legend},
  new legend={lower legend},
  upper legend=above,
  lower legend=below,
  log= {label in legend={text=$\log x$, legend=upper legend}},
  lin= {label in legend={text=$x/2$, legend=upper legend}},
  squared={label in legend={text=$x^2$, legend=lower legend}},
  exp= {label in legend={text=$e^x$, legend=lower legend}},
  style sheet=vary dashing]
data group {function classes};
```

上面的例子中，设置了两个图例 ⟨*upper legend*⟩ 和 ⟨*lower legend*⟩，它们的位置分别是 `above` 和 `below`. 然后在显像器 `log` 的参数中使用 `legend=upper legend` 选项将该显像器的标签放在图例 ⟨*upper legend*⟩ 中。对比下面的例子：




```
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  log= {label in legend={text=$\log x$, legend=above}},
  lin= {label in legend={text=$x/2$, legend=above}},
  squared={label in legend={text=$x^2$, legend=below}},
  exp= {label in legend={text=$e^x$, legend=below}},
  style sheet=vary dashing]
data group {function classes};
```

在这个例子中只有一个图例，即选项 `legend` 所默认的图例 `main legend`，显像器中的图例选项 `legend=above` 和 `legend=below` 只是使得标签分为两列。

`/tikz/data visualization/visualizer options/label in legend=`⟨*options*⟩ (no default)

这个选项用在显像器的参数中，为该显像器的数据点生成图例条目，并放在某个指定的图例盒子（matrix node）中。在 ⟨*options*⟩ 中的选项会被冠以前缀

```
/tikz/data visualization/legend entry options
```

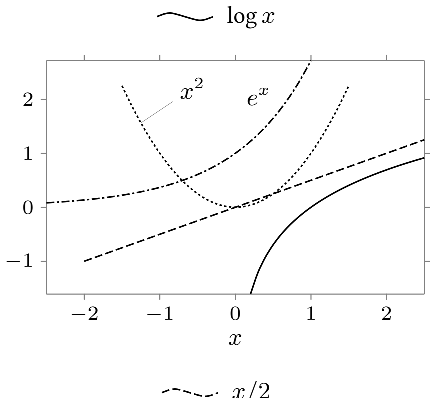
来执行。

在 ⟨*options*⟩ 中可以使用以下选项。

/tikz/data visualization/legend entry options/legend=⟨*name*⟩ (no default, initially main legend)

这个选项规定所生成的图例条目标签属于名称为 ⟨*name*⟩ 的图例（即放在这个图例盒子中）。这里的 ⟨*name*⟩ 可以是选项 `new legend=`⟨*name*⟩ 声明的图例名称。如果没有用选项 `new legend` 声明图例名称，那么可以使用默认名称 `main legend`；如果打算使用默认的图例名称 `main legend`，就无需明确写出 `legend=main legend`.

如果 `legend=`⟨*name*⟩ 的 ⟨*name*⟩ 是之前没有被声明的图例名称，则程序会自动创建这个图例名称。

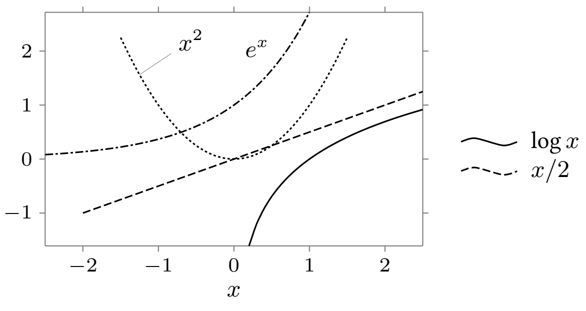



```
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  log= {label in legend={legend=upper legend, text=$\log x$}},
  upper legend=above,
  lin= {label in legend={legend=lower legend, text=$x/2$}},
  lower legend=below,
  squared={pin in data ={text=$x^2$, pos=0.1}},
  exp= {label in data ={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

上面例子中。选项 `legend=upper legend` 声明图例 `upper legend`，然后用 `upper legend=above` 规定了该图例的位置。因为图例名称的路径前缀是 `/tikz/data visualization/`，所以 `upper legend=above` 直接用作 `\datavisualization` 的选项，不能像 `legend=upper legend` 那样用作 `label in legend` 的参数。

/tikz/data visualization/legend entry options/text=⟨*text*⟩ (no default)

这个选项设置条目标签的文字内容。




```
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={pin in data ={text=$x^2$, pos=0.1}},
  exp= {label in data ={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

84.9.2 图例中条目的行列排布

一个图例是个 matrix node, 其中的条目作为它的元素排成行、列。图例条目有多个排布规则，每个规则对应一个 key，这些 key 一般应作为图例名称的参数，即选项 `new legend=`⟨*legend name*⟩ 所设置的图例名称 ⟨*legend name*⟩ 的参数，放在 ⟨*legend name*⟩`={`⟨*options*⟩`}` 中的 ⟨*options*⟩ 里。

如果不设置图例名称, 那么可以使用默认的图例名称 `main legend`, 即把这些 key 放在 `main legend = {`⟨*options*⟩`}` 中的 ⟨*options*⟩ 里；也可以把这些 key 作为选项 `legend` 的参数，即把这些 key 放在 `legend = {`⟨*options*⟩`}` 中的 ⟨*options*⟩ 里，因为选项 `legend` 会默认图例名称为 `main legend`, 所以这两个方式等效。

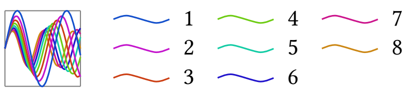
为了展示这些 key 的效果，先定义一个样式 legend example：

```
\tikzdatavisualizationset {
  legend example/.style={
    scientific axes, all axes={length=1cm, ticks=none},
    1={label in legend={text=1}}, % 设置序号为 1 的显像器的图例
    2={label in legend={text=2}},
    3={label in legend={text=3}},
    4={label in legend={text=4}},
    5={label in legend={text=5}},
    6={label in legend={text=6}},
    7={label in legend={text=7}},
    8={label in legend={text=8}}
  }
}
```

`/tikz/data visualization/legend options/down then right` (no value)

这个选项确定图例条目的排布方式是：先自上而下纵向排成一列，当该列达到某个行数后，转到该列的右侧再自上而下排一列，如此继续。

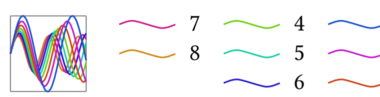
这个选项是默认选项。



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8
  ↪ },
  legend example, style sheet=vary hue,
  main legend={down then right, columns=3}]
data group {sin functions};
```

`/tikz/data visualization/legend options/down then left` (no value)

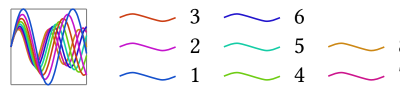
这个选项确定图例条目的排布方式是：先自上而下纵向排成一列，当该列达到某个行数后，转到该列的左侧再自上而下排一列，如此继续。



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8
  ↪ },
  legend example, style sheet=vary hue,
  legend={down then left, columns=3}]
data group {sin functions};
```

`/tikz/data visualization/legend options/up then right` (no value)

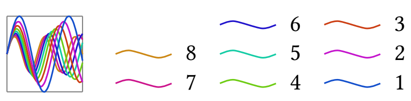
这个选项确定图例条目的排布方式是：先自下而上纵向排成一列，当该列达到某个行数后，转到该列的右侧再自下而上排一列，如此继续。



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8
  ↪ },
  legend example, style sheet=vary hue,
  legend={up then right, columns=3}]
data group {sin functions};
```

`/tikz/data visualization/legend options/up then left` (no value)

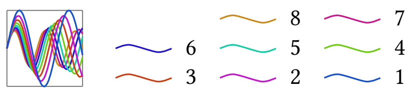
这个选项确定图例条目的排布方式是：先自下而上纵向排成一列，当该列达到某个行数后，转到该列的左侧再自下而上排一列，如此继续。



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8
  ↪ },
  legend example, style sheet=vary hue,
  legend={up then left, columns=3}]
data group {sin functions};
```

`/tikz/data visualization/legend options/left then up` (no value)

这个选项确定图例条目的排布方式是：先自右而左横向排成一行，当该行达到某个列数后，转到该行的上方再自右而左排一行，如此继续。



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8
  ↪ },
  legend example, style sheet=vary hue,
  legend={left then up, columns=3}]
data group {sin functions};
```

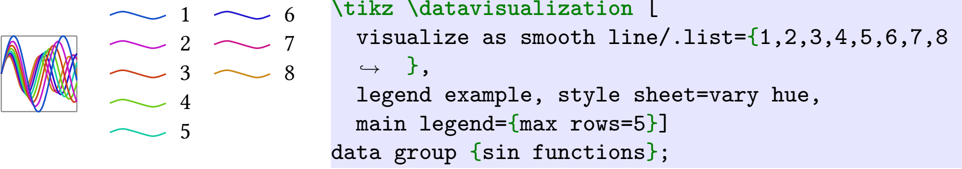
/tikz/data visualization/legend options/left then down (no value)

/tikz/data visualization/legend options/right then up (no value)

/tikz/data visualization/legend options/right then down (no value)

/tikz/data visualization/legend options/max rows=⟨*number*⟩ (no value)

这个选项规定图例矩阵的最大行数。



```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8
  ↪ },
  legend example, style sheet=vary hue,
  main legend={max rows=5}]
data group {sin functions};
```

/tikz/data visualization/legend options/max columns=⟨*number*⟩ (no value)

这个选项规定图例矩阵的最大列数。注意只有使用“先左右，后上下”的条目排布方式时，这个选项才有效。

```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8
  ↪ },
  legend example, style sheet=vary hue,
  legend={right then down, max columns=2}]
data group {sin functions};
```

/tikz/data visualization/legend options/ideal number of columns=⟨*number*⟩ (no default)

这个选项设置一个“理想的”列数，程序会尽量将条目排成这个列数，但如果还给出了选项 `max rows`, 那么选项 `max rows` 要比这个选项更有优先地位。

```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8
  ↪ },
  legend example, style sheet=vary hue,
  main legend={ideal number of columns=2}]
data group {sin functions};
```

```
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8
  ↪ },
  legend example, style sheet=vary hue,
  main legend={max rows=3,ideal number of
  ↪ columns=2}]
data group {sin functions};
```

/tikz/data visualization/legend options/rows=⟨*number*⟩ (no default)

这是 `ideal number of rows=`⟨*number*⟩ 的简写。

/tikz/data visualization/legend options/ideal number of rows=⟨*number*⟩ (no default)

这个选项设置一个“理想的”列数，程序会尽量将条目排成这个列数，但如果该给出了选项 `max columns`, 那么选项 `max columns` 要比这个选项更有优先地位。

/tikz/data visualization/legend options/columns=⟨*number*⟩ (no default)

这是 `ideal number of columns=`⟨*number*⟩ 的简写。

84.9.3 确定图例位置的一般方法

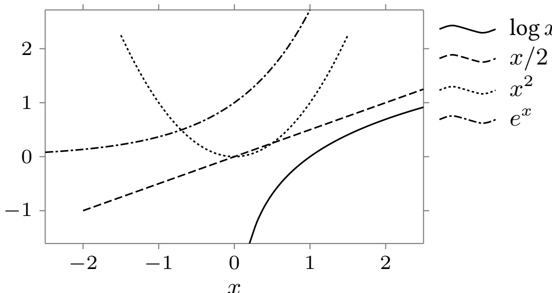
图例是个 matrix node，作为一个 node 它有自己的各种“部位”、“锚位”，也可以使用 `at` 选项来确定它的锚定点。

/tikz/data visualization/legend options/anchor=⟨*anchor*⟩ (no default)

这个选项用作图例名称的参数或者 `legend` 的参数，该选项把图例的锚位 ⟨*anchor*⟩ 放在其锚定点上。

/tikz/data visualization/legend options/at=⟨*coordinate*⟩ (no default)

这个选项用作图例名称的参数或者 `legend` 的参数，该选项确定图例的锚定点。不过注意这里的 ⟨*coordinate*⟩ 必须是与预定义的 `data bounding box` 或 `data visualization bounding box` 有关的点，不能直接使用数值坐标。




```
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  legend={anchor=north west, at=(data visualization bounding box.north east)},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

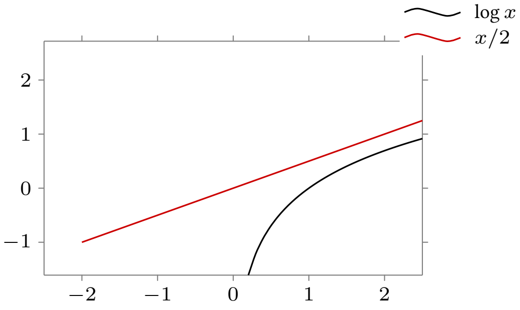
`at=`⟨*coordinate*⟩ 还可以使用如下的坐标计算格式：

```
at=($(data visualization bounding box.east)+(90:1)$)
at=($(data visualization bounding box.east)+(10pt, 1cm)$)
```

在这种计算格式中的坐标点的默认长度单位都是 cm，并不是可视化命令中轴系统中的坐标点。

/tikz/data visualization/legend options/at values={⟨*data point*⟩} (no default)

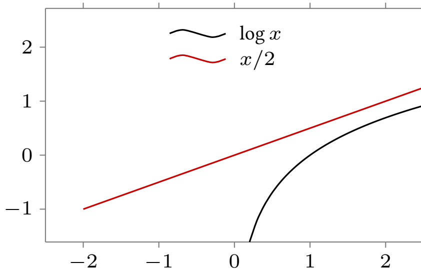
与 at= 选项不同，这里的 {⟨*data point*⟩} 是可视化命令中轴系统中的坐标点，例如 x=1，y=2，使用这个选项选定数据点后，默认图例的中心点处于这个数据点位置上。注意这里使用 ⟨*attribute*⟩=⟨*value*⟩ 的形式，要写出正确的 ⟨*attribute*⟩ 名称。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin},
  legend={at values={x=3, y=3}},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/right of=⟨*data point*⟩ (no default)

这里的 ⟨*data point*⟩ 是可视化命令中轴系统中的坐标点，例如 x=1，y=2，这个选项会使得图例处于该数据点的右侧，即图例的锚位 west 处于该数据点的位置上。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin},
  legend={right of={x=-1, y=2}},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/above right of=⟨*data point*⟩ (no default)

/tikz/data visualization/legend options/above of=⟨*data point*⟩ (no default)

/tikz/data visualization/legend options/above left of=⟨*data point*⟩ (no default)

/tikz/data visualization/legend options/left of=⟨*data point*⟩ (no default)

/tikz/data visualization/legend options/below left of=⟨*data point*⟩ (no default)

/tikz/data visualization/legend options/below of=⟨*data point*⟩ (no default)

/tikz/data visualization/legend options/below right of=⟨*data point*⟩ (no default)

84.9.4 在绘图区域之外放置图例

下面的选项作为图例名称的参数或者 legend 的参数，在绘图区域之外放置图例。

/tikz/data visualization/legend options/east outside (no value)

将图例放在绘图区域之外的右侧，这是默认的放置方式。

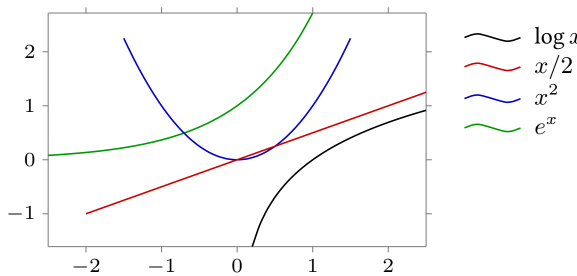
```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin, squared, exp},
  legend=east outside,
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/right (no value)

等价于 `east outside`.

/tikz/data visualization/legend options/north east outside (no value)

将图例放在绘图区域之外的右上角。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin, squared, exp},
  legend=north east outside,
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/south east outside (no value)

将图例放在绘图区域之外的右下角。

/tikz/data visualization/legend options/west outside (no value)

将图例放在绘图区域之外的左侧。

/tikz/data visualization/legend options/left (no value)

等价于 west outside.

/tikz/data visualization/legend options/north west outside (no value)

将图例放在绘图区域之外的左上角。

/tikz/data visualization/legend options/south west outside (no value)

将图例放在绘图区域之外的左下角。

/tikz/data visualization/legend options/north outside (no value)

将图例放在绘图区域之外的上部。

/tikz/data visualization/legend options/above (no value)

等价于 north outside.

/tikz/data visualization/legend options/south outside (no value)

将图例放在绘图区域之外的下部。

/tikz/data visualization/legend options/below (no value)

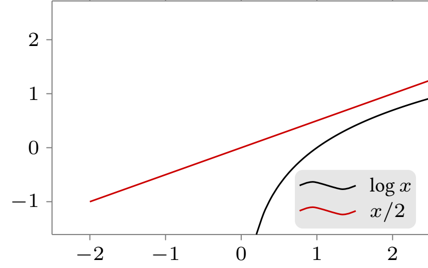
等价于 south outside.

84.9.5 在绘图区域之内放置图例

下面的选项作为图例名称的参数或者 legend 的参数，在绘图区域之内放置图例。在默认下，绘图区域之内的图例的外观样式与绘图区域之外的图例的外观样式有所不同。绘图区域之内的图例（作为 matrix node）会被用不透明的白色填充，因此可能会遮挡数据点，边界是圆角矩形，条目中文字的尺寸是脚注尺寸。

/tikz/data visualization/legend options/south east inside (no value)

将图例放在绘图区域之内的右下角。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin},
  legend={south east inside, opaque=gray!20},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/every legend inside (style, no value)

这个样式的默认设置是 opaque 并且标签文字尺寸是脚注尺寸。

/tikz/data visualization/legend options/opaque=⟨*color*⟩ (default white)

设置图例 (作为 matrix node) 的填充颜色为 ⟨*color*⟩, 默认颜色是白色，并且还使得图例 (作为 matrix node) 的边界为圆角，见上面的例子。

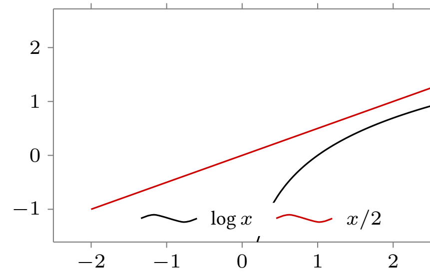
/tikz/data visualization/legend options/transparent (no value)

使得图例透明，即没有填充色：opaque=none.

/tikz/data visualization/legend options/east inside (no value)
/tikz/data visualization/legend options/north east inside (no value)
/tikz/data visualization/legend options/south west inside (no value)
/tikz/data visualization/legend options/west inside (no value)
/tikz/data visualization/legend options/north west inside (no value)

/tikz/data visualization/legend options/south inside (no value)

这个选项会使得条目排成一行。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin},
  legend=south inside,
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/north inside (no value)

这个选项会使得条目排成一行。

84.9.6 图例条目的一般样式

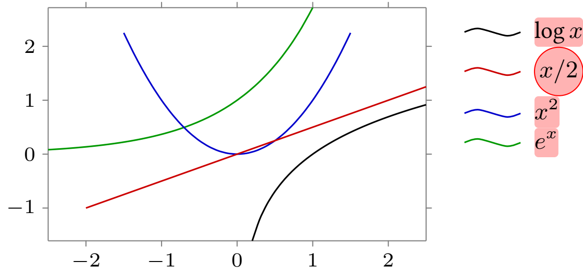
通常，一个条目由文字标签和图示标签两部分组成，对于条目的设置就是针对这两部分的设置。

/tikz/data visualization/every label in legend (style, no value)

这个样式针对所有图例中所有条目。该样式中的选项会被冠以前缀

```
/tikz/data visualization/legend entry options
```

来执行。其中可以用选项 `node style` 来设置文字标签的样式。

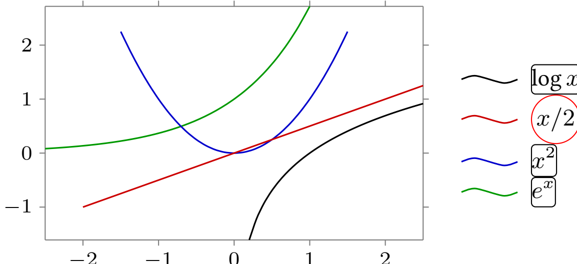



```
\tikz \datavisualization [
  scientific axes,
  every label in legend/.style={node style={fill=red!30}},
  visualize as smooth line/.list={log, lin, squared, exp},
  legend=north east outside,
```

```
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$,
  node style={circle, draw=red}}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/label style=⟨*options*⟩ (no default)

这个选项规定的样式会添加到样式 `every label in legend` 中。

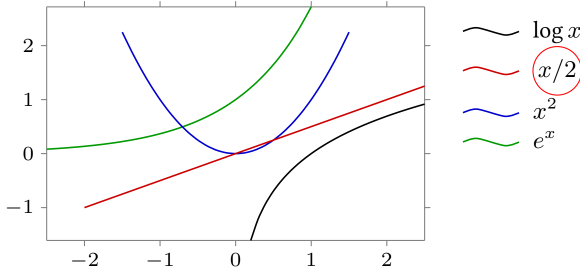



```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin, squared, exp},
  legend={label style={node style=draw}},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$, node style={circle, draw=red}}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

84.9.7 图例条目中的文字标签

/tikz/data visualization/legend entry options/node style=⟨*options*⟩ (no default)

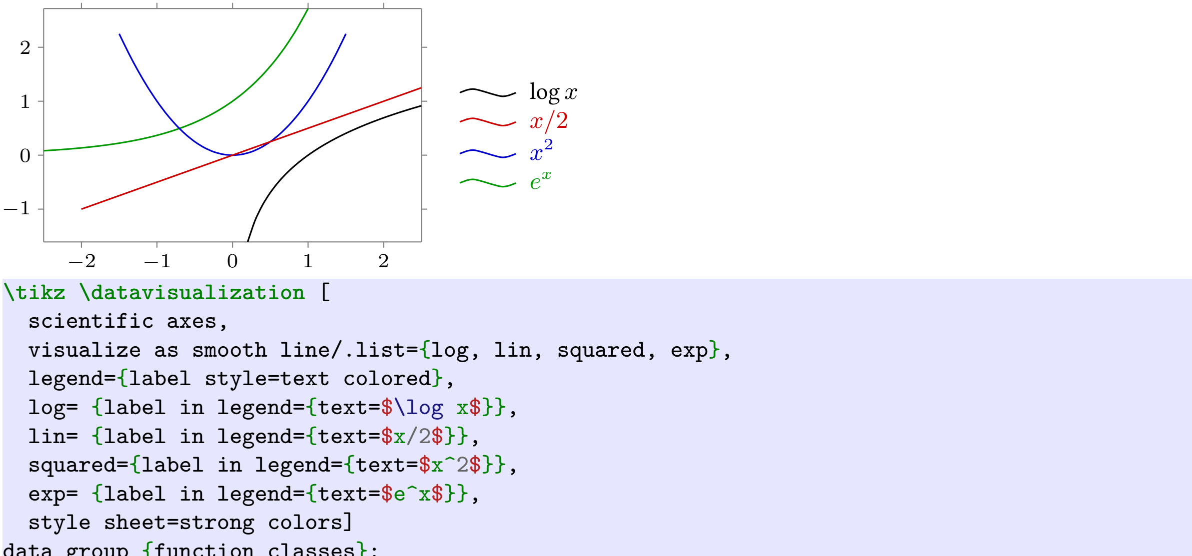
这个选项用于设置条目中的文字标签的样式。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin, squared, exp},
  legend=north east outside,
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$,
  node style={circle, draw=red}}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend entry options/text colored (no value)

这个选项的作用与标签的 `text colored` 选项一样，将与显像器关联的、样式表中的、相应样式的颜色设置 `visualizer color=`⟨*color*⟩ 与条目文字标签的 `node style` 选项联系起来，使得条目文字标签颜色是 ⟨*color*⟩, 与显像器所显示的数据点的颜色一致。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin, squared, exp},
  legend={label style=text colored},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

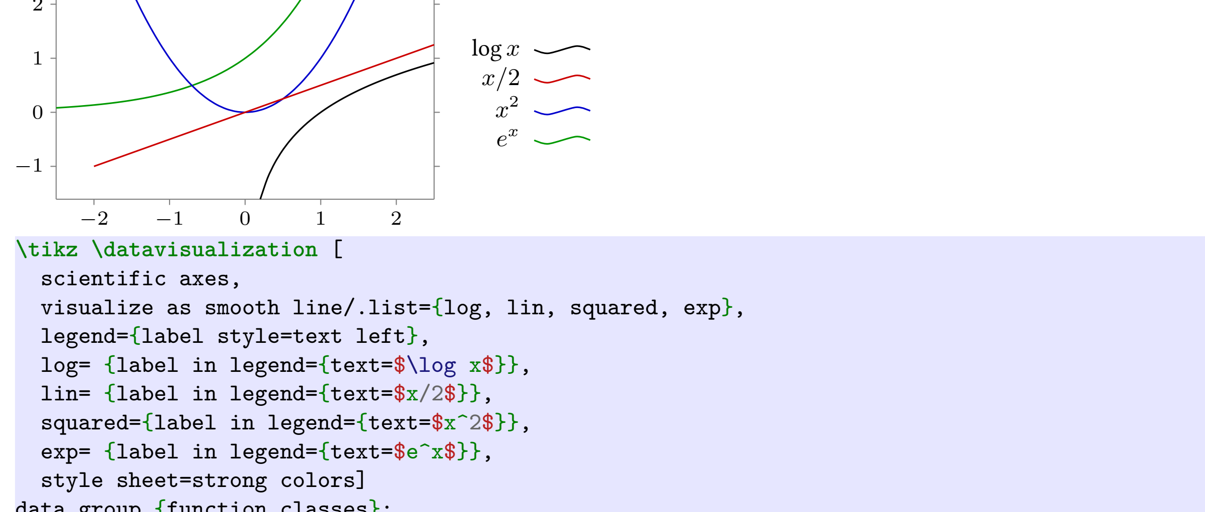
84.9.8 条目中文字标签与图示标签的相对位置

/tikz/data visualization/legend entry options/text right (no value)

这个选项使得条目中文字标签位于图示标签的右侧，在多数情况下这是默认的排列方式。

/tikz/data visualization/legend entry options/text left (no value)

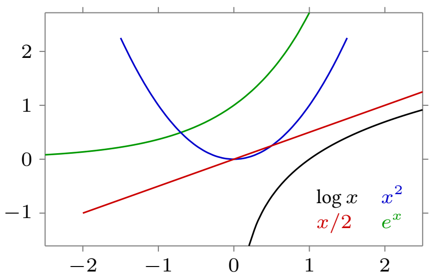
这个选项使得条目中文字标签位于图示标签的左侧。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin, squared, exp},
  legend={label style=text left},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend entry options/text only (no value)

这个选项只设置文字标签，取消图示标签，同时默认选项 `text colored` 有效，以利于把条目和数据点图形对应起来。




```
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin, squared, exp},
  legend={south east inside, rows=2,
  label style=text only},
  log= {label in legend={text=$\log x$}},
  lin= {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp= {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

84.9.9 手工添加条目

一般情况下，给某个显像器使用参数 `label in legend=`⟨*options*⟩ 产生一个对应该显像器的条目。也可以手工添加一个不与任何显像器对应的条目，并指定该条目属于哪个图例。

/tikz/data visualization/new legend entry=⟨*options*⟩ (no default)

这个选项创建一个条目，⟨*options*⟩ 设置该条目的具体内容和样式。⟨*options*⟩ 中的选项会被冠以前缀

```
/tikz/data visualization/legend entry options/
```

来执行。可以在 ⟨*options*⟩ 中使用 `legend=`⟨*name*⟩ 来指定该条目属于哪个图例。条目会按照先后次序添加到图例中。如果需要把该条目放入默认的图例 `main legend` 中，就不必写出 `legend=main legend`.

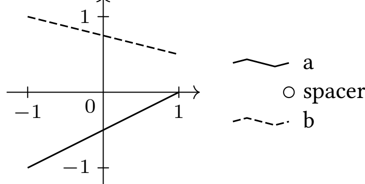
在 ⟨*options*⟩ 中仍然用

```
/tikz/data visualization/legend entry options/text
```

设置条目的文字标签，图示标签则用下一选项设置：

/tikz/data visualization/legend entry options/visualizer in legend=⟨*code*⟩ (no value)

这里的 ⟨*code*⟩ 是绘图代码，绘制当前条目的图示标签。



```
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},   style sheet=vary dashing,
  a={label in legend={text=a}},
  new legend entry={
    text=spacer,
    visualizer in legend={\draw[solid] (0,0) circle[radius=2pt];}
  },
  b={label in legend={text=b}}]
data point [x=-1, y=-1, set=a] data point [x=1, y=0, set=a]
data point [x=-1, y=1, set=b] data point [x=1, y=0.5, set=b];
```

一个图例就是一个 matrix node, 向图例中手工添加一个条目就是在 matrix node 中画出一个元素图形 (cell picture)。在画出这个元素图形前，下面的设置或操作会被依次执行:

1. `/tikz/data visualization/every data set label`→P. 492, 此样式中的 key 的路径是 `/tikz/data visualization`.
2. `/tikz/data visualization/every label in legend`→P. 507, 此样式中的 key 的路径是 `/tikz/data visualization/legend entry options`.
3. `new legend entry=`⟨*options*⟩ 的 ⟨*options*⟩.
4. 下面选项中的代码:

   `/tikz/data visualization/legend entry options/setup``={`⟨*set up code*⟩`}` (no value)

   在这个节骨眼上执行 `{`⟨*set up code*⟩`}`, 通常是把某个样式表与某个标识符 attribute 关联起来。
5. 发布一个 styling signal.
6. 针对文字标签执行 `node style` 的当前值。
7. 针对图示标签执行下面选项中的 ⟨*options*⟩:

   `/tikz/data visualization/legend entry options/visualizer in legend style``=`
   `{`⟨*options*⟩`}` (style, no default)

   ⟨*options*⟩ 中的选项会被冠以 `/tikz/` 来执行，即其中的选项都是 TikZ 的选项。

在内部的处理过程中，选项 `label in legend=`⟨*options*⟩ 会调用 `new legend entry` 的内容，此时下面的设置或操作会被传递给 `new legend entry`:

- 显像器自己的样式设置。
- 样式 `/tikz/data visualization/every label in legend`→P. 507.
- 样式 `/tikz/every label`→P. 123.
- 利用 `setup` 选项设置 `/data point/set=`⟨*name of the visualizer*⟩.
- 保存在显像器中的其它选项设置，这些设置可以用下面的选项修改:

  `/tikz/data visualization/visualizer options/label in legend options``=`⟨*options*⟩
  (no default)

  这个选项用作显像器的参数，设置对应该显像器的图例条目的内容、样式。在内部处理过程中，本选项通常用作 `visualizer in legend` 的参数。

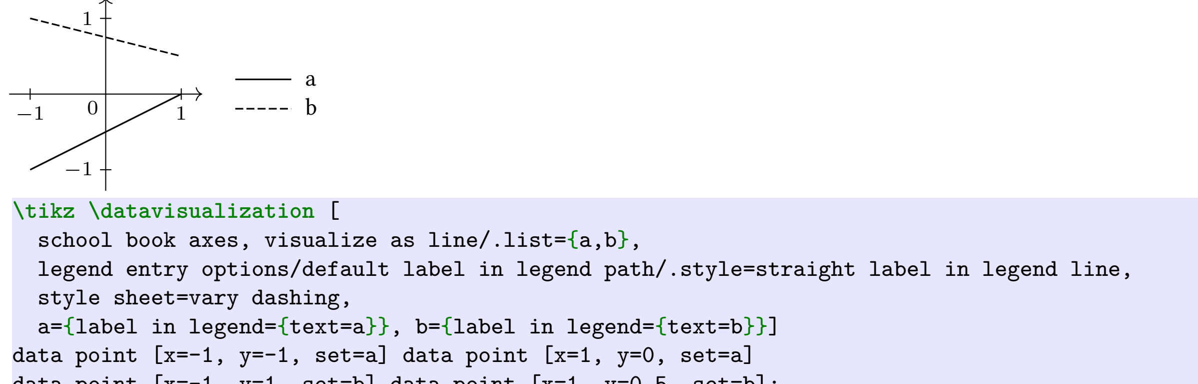
84.9.10 条目显像器

通常，图例条目中的图示标签会被自动画出（不是手工添加），自动把图示标签画出的是“条目显像器”（label in legend visualizer），它画出的图示标签的外观与它所对应的显像器具有一致性。注意“条

目显像器”所对应的显像器，就是条目所对应的显像器。下面的选项能对“条目显像器”做出设置。

/tikz/data visualization/legend entry options/default label in legend path (style, no value)

当显像器使用选项 /tikz/data visualization/visualizer options/straight line→ P. 475, smooth line, gap line 等 line 线条时，该显像器对应的条目图示标签会使用这个样式规定的线条。这个样式的默认设置是选项 zig zag label in legend line(见后文)，使得图示标签是之字形的，可以改为其它形式，例如改用选项 straight label in legend line, 即直线段形式的图示标签。




```
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},
  legend entry options/default label in legend path/.style=straight label in legend line,
  style sheet=vary dashing,
  a={label in legend={text=a}}, b={label in legend={text=b}}]
data point [x=-1, y=-1, set=a] data point [x=1, y=0, set=a]
data point [x=-1, y=1, set=b] data point [x=1, y=0.5, set=b];
```

/tikz/data visualization/legend entry options/default label in legend closed path (style, no value)

当显像器使用选项 /tikz/data visualization/visualizer options/smooth cycle→ P. 475, straight cycle 时，该显像器对应的条目图示标签会使用这个样式规定的线条。这个样式的默认设置是选项 circular label in legend line(见后文), 使得图示标签是环形的。

/tikz/data visualization/legend entry options/default label in legend mark (style, no value)

当显像器使用选项 /tikz/data visualization/visualizer options/no lines→ P. 476, 并且使用散点来标记数据点时，该显像器对应的条目图示标签会使用这个样式规定的点标记。这个样式的默认设置是选项 label in legend line one mark, 只有一个点标记符号，可以改为其它形式，例如改用选项 label in legend line three marks, 即使用 3 个点标记符号。

× × × example a
+ + + example b
⅄ ⅄ ⅄ example c

```
\tikz \datavisualization [
  visualize as scatter/.list={a,b,c},
  style sheet=cross marks,
  legend entry options/default label in legend mark/.style=
    label in legend three marks,
  a={label in legend={text=example a}},
  b={label in legend={text=example b}},
  c={label in legend={text=example c}}];
```

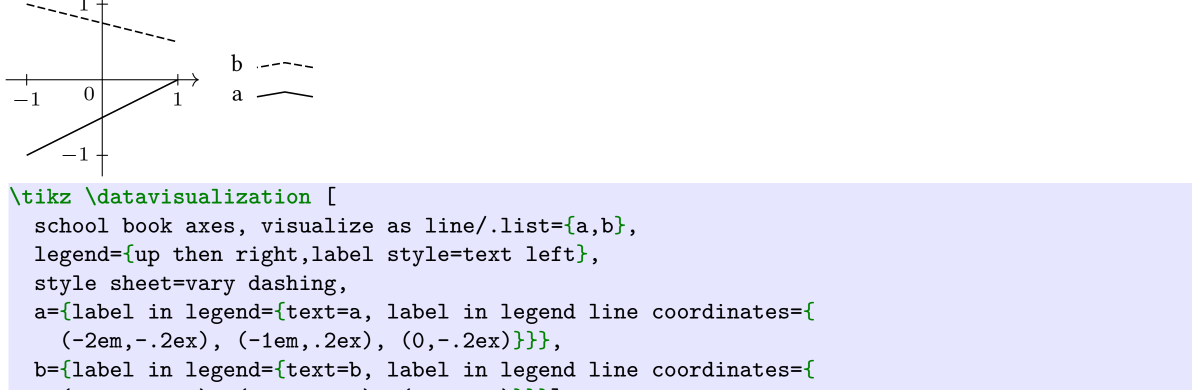
一个条目的图示标签通常是一段曲线（可能是环形），或者是一个或者数个散点标记符号，或者是带有标记符号的曲线。TikZ 会开启一个坐标系，在该坐标系中绘制图示标签，为了方便，称这个绘制图示标签的坐标系为“图示标签坐标系”。在默认下，文字标签位于图示标签的右侧，而“图示标签坐标

系”的原点 `(0,0)` 位于文字标签的左侧，并在文字基线之上 0.5ex 处。下面的选项利用“图示标签坐标系”中的点绘制图示标签。

`/tikz/data visualization/legend entry options/label in legend line coordinates=` `{`⟨*list of coordinates*⟩`}` (no default)

⟨*list of coordinates*⟩ 是个点坐标列表，其中的点坐标使用 TikZ 的格式，如 `(0,1)`，代表的是“图示标签坐标系”中的点。条目显像器会画出经过这些点的曲线，将其作为图示标签的曲线部分（图示标签还可能包含散点，需要用下面的选项画出），曲线的样式与显像器画出的数据点曲线的样式相同。例如，如果显像器使用选项 `smooth line` 和 `style=red`，则数据点曲线和图示标签曲线都是红色、光滑的。

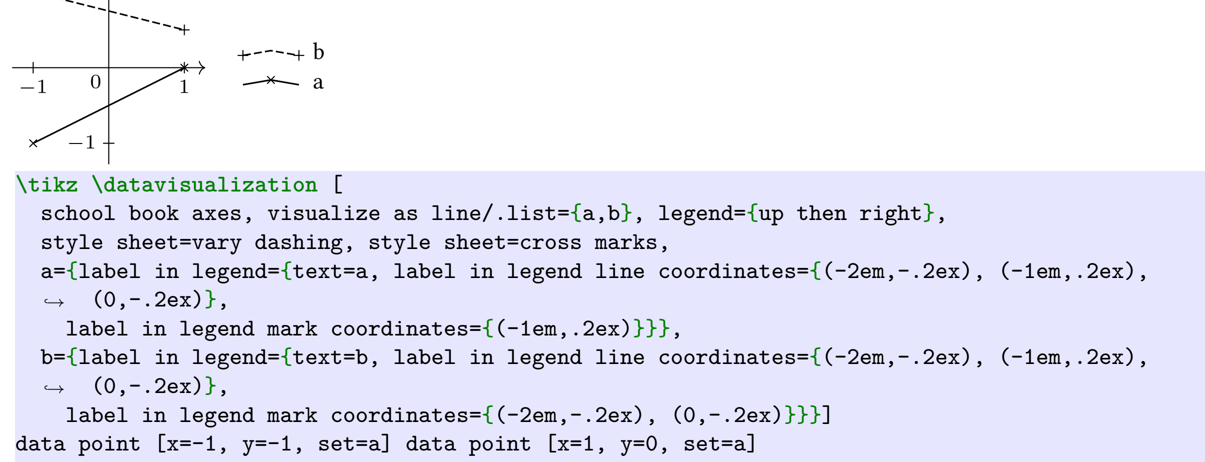
⟨*list of coordinates*⟩ 中的坐标的横坐标分量应当使用非正数，否则图示标签曲线可能会与文字标签重叠。当使用选项 `text left` 时，文字标签位于图示标签的左侧，但此时“图示标签坐标系”也会被翻过来（受到一个反射变换），所以 ⟨*list of coordinates*⟩ 中坐标的横坐标分量仍然应当使用非正数。




```
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},
  legend={up then right,label style=text left},
  style sheet=vary dashing,
  a={label in legend={text=a, label in legend line coordinates={
    (-2em,-.2ex), (-1em,.2ex), (0,-.2ex)}}},
  b={label in legend={text=b, label in legend line coordinates={
    (-2em,-.2ex), (-1em,.2ex), (0,-.2ex)}}}]
data point [x=-1, y=-1, set=a] data point [x=1, y=0, set=a]
data point [x=-1, y=1, set=b] data point [x=1, y=0.5, set=b];
```

`/tikz/data visualization/legend entry options/label in legend mark coordinates=` `{`⟨*list of coordinates*⟩`}` (no default)

与上面的选项 `label in legend line coordinates` 类似，只是用于画出图示标签的散点部分，也就是说，这两个选项可以同时使在一个条目中。




```
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b}, legend={up then right},
  style sheet=vary dashing, style sheet=cross marks,
  a={label in legend={text=a, label in legend line coordinates={(-2em,-.2ex), (-1em,.2ex),
  ↪  (0,-.2ex)},
    label in legend mark coordinates={(-1em,.2ex)}}},
  b={label in legend={text=b, label in legend line coordinates={(-2em,-.2ex), (-1em,.2ex),
  ↪  (0,-.2ex)},
    label in legend mark coordinates={(-2em,-.2ex), (0,-.2ex)}}}]
data point [x=-1, y=-1, set=a] data point [x=1, y=0, set=a]
```

```
data point [x=-1, y=1, set=b] data point [x=1, y=0.5, set=b];
```

上面两个选项都需要手工指定一个 ⟨*list of coordinates*⟩ 来绘制图示标签，手工指定坐标列表有时会比较麻烦，例如，如果显像器使用选项 `gap cycle`, 那么图示标签曲线当然也是 “gap cycle” 形式的，为了使得整个条目足够美观并且与其它条目大小相称，在手工指定一个坐标列表时就得仔细考虑一番，选择合适的坐标。如果想省去手工指定坐标列表的麻烦，可以使用下面的选项。

/tikz/data visualization/legend entry options/straight label in legend line (no value)

这个选项将图示标签曲线指定为直线段。如果还指定了点标记符号 (用选项 `style = {mark = `⟨*mark specification*⟩`}`), 那么直线段上还会有两个散点标记。

bad example a
bad example b
no mark example c

```
\tikz \datavisualization [visualize as line/.list={a,b,c},
  legend entry options/default label in legend path/.style=
    straight label in legend line,
  a={style={mark=+}, label in legend={text=bad example a}},
  b={style={mark=-}, label in legend={text=bad example b}},
  c={label in legend={text=no mark example c}}];
```

上面这个例子表明，有的点标记符号在直线段上是辨认不出来的，因此这个选项不是默认选项。

/tikz/data visualization/legend entry options/zig zag label in legend line (no value)

这个选项将图示标签曲线指定为之字形曲线，这是默认的形式。

good example a
good example b
good example c

```
\tikz \datavisualization [
  visualize as line/.list={a,b},
  visualize as smooth line=c,
  style sheet=cross marks,
  a={label in legend={text=good example a}},
  b={label in legend={text=good example b}},
  c={gap line, label in legend={text=good example c}}];
```

/tikz/data visualization/legend entry options/circular label in legend line (no value)

这个选项将图示标签曲线指定为环形,当显像器使用选项 `/tikz/data visualization/visualizer options/polygon`→P. 475, `smooth cycle`, `straight cycle` 时，可以使用这个选项，该选项是专为这个情况定制的。

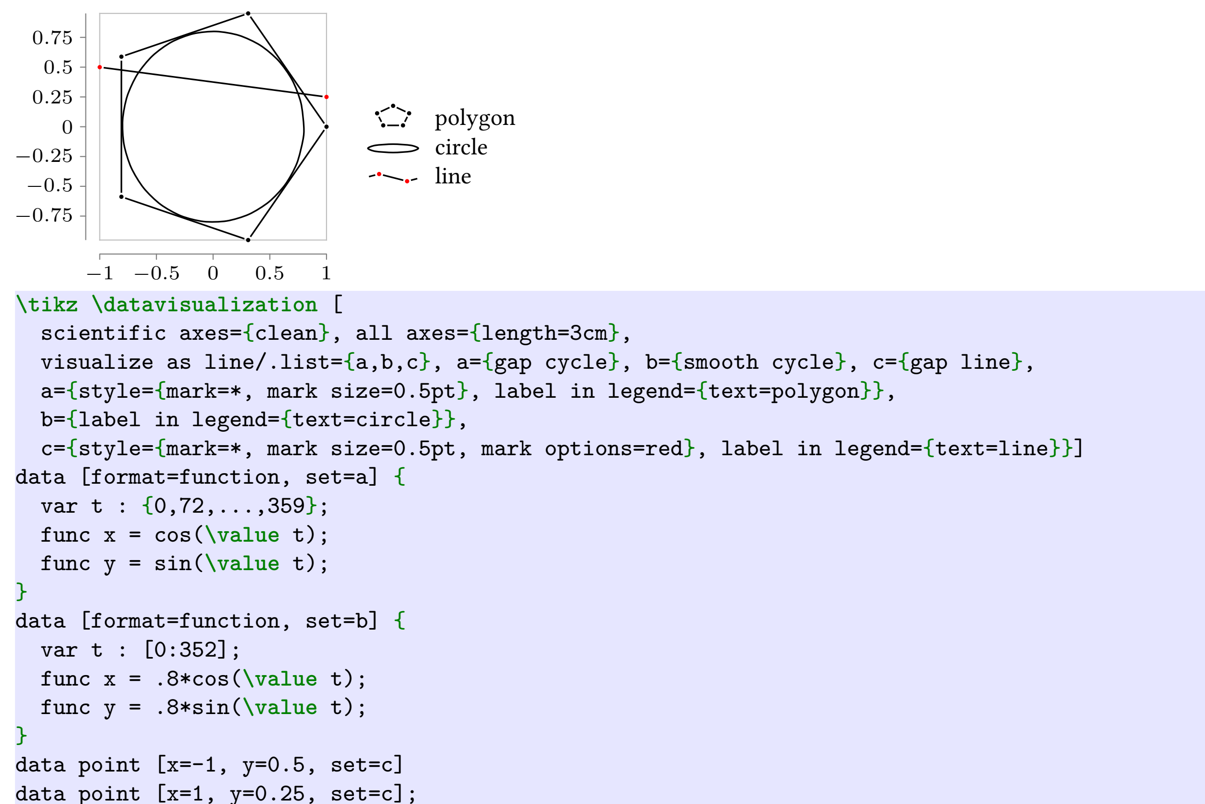
0.75
0.5
0.25
0
−0.25
−0.5
−0.75
−1 −0.5 0 0.5 1
polygon
circle
line

```
\tikz \datavisualization [
  scientific axes={clean}, all axes={length=3cm},
  visualize as line/.list={a,b,c}, a={polygon}, b={smooth cycle},
  style sheet=cross marks,
  a={label in legend={text=polygon}},
  b={label in legend={text=circle}},
```

```
  c={label in legend={text=line}}]
data [format=function, set=a] {
  var t : {0,72,...,359};
  func x = cos(\value t);
  func y = sin(\value t);
}
data [format=function, set=b] {
  var t : [0:2*pi];
  func x = .8*cos(\value t r);
  func y = .8*sin(\value t r);
}
data point [x=-1, y=0.5, set=c]
data point [x=1, y=0.25, set=c];
```

/tikz/data visualization/legend entry options/gap circular label in legend line (no value)

这个选项将图示标签曲线指定为带空隙的环形,当显像器使用选项 /tikz/data visualization/visualizer options/gap cycle →P. 476, gap line 时，可以使用这个选项，该选项是专为这个情况定制的。




```
\tikz \datavisualization [
  scientific axes={clean}, all axes={length=3cm},
  visualize as line/.list={a,b,c}, a={gap cycle}, b={smooth cycle}, c={gap line},
  a={style={mark=*, mark size=0.5pt}, label in legend={text=polygon}},
  b={label in legend={text=circle}},
  c={style={mark=*, mark size=0.5pt, mark options=red}, label in legend={text=line}}]
data [format=function, set=a] {
  var t : {0,72,...,359};
  func x = cos(\value t);
  func y = sin(\value t);
}
data [format=function, set=b] {
  var t : [0:352];
  func x = .8*cos(\value t);
  func y = .8*sin(\value t);
}
data point [x=-1, y=0.5, set=c]
data point [x=1, y=0.25, set=c];
```

/tikz/data visualization/legend entry options/label in legend one mark (no value)

这个选项适用于显像器绘制散点图，或者显像器使用 no lines 选项的情况，该选项只把一个点标记符号作为图示标签。

× example a
+ example b
⅄ example c

```
\tikz \datavisualization [visualize as scatter/.list={a,b,c},
  style sheet=cross marks,
  a={label in legend={text=example a}},
  b={label in legend={text=example b}},
  c={label in legend={text=example c}}];
```

`/tikz/data visualization/legend entry options/label in legend three marks` (no value)

这个选项适用于显像器绘制散点图，或者显像器使用 `no lines` 选项的情况，该选项把 3 个点标记符号作为图示标签。

× × × example a
+ + + example b
⅄ ⅄ ⅄ example c

```
\tikz \datavisualization [visualize as scatter/.list={a,b,c},
  style sheet=cross marks,
  a={label in legend={text=example a,
    label in legend three marks}},
  b={label in legend={text=example b,
    label in legend three marks}},
  c={label in legend={text=example c,
    label in legend three marks}}];
```

85 极坐标系

85.1 Overview

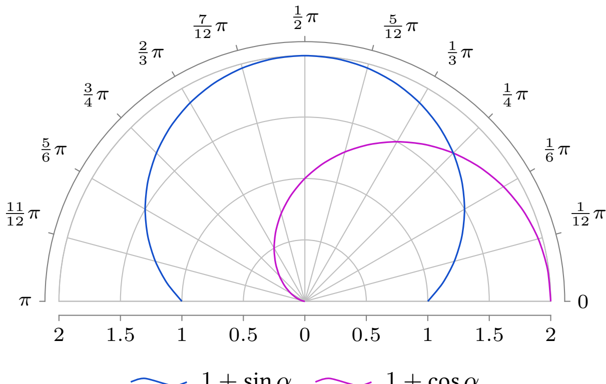
TikZ Library `datavisualization.polar`

```
\usetikzlibrary{datavisualization.polar} % LaTeX and plain TeX
\usetikzlibrary[datavisualization.polar] % ConTeXt
```

本程序库提供选项，能在极坐标系统下绘图。

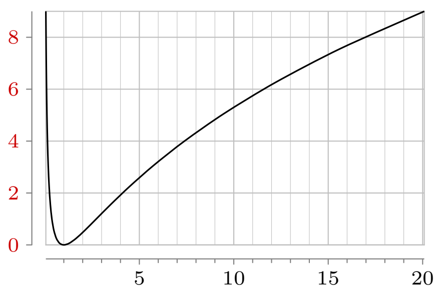
在极坐标系统中，数据点的一个数据标识符对应角度，另一个对应半径。极坐标系统的角度轴可以使用角度制、弧度制，也可以做对数化处理，或者在两个角度区间之间转换（映射）。

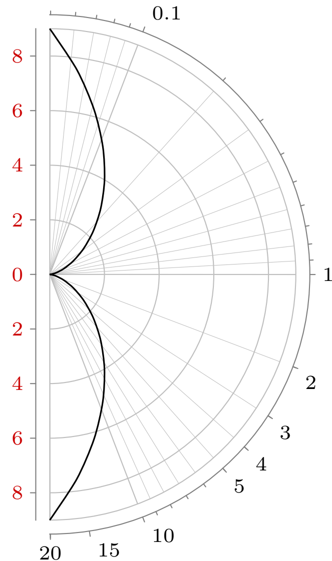
在可视化命令中使用选项 `scientific polar axes` 就可以开启极坐标系。这个选项可以带有一些参数 (keys) 来对极坐标系做某些设置。极坐标系统中的“轴”尽管比笛卡尔坐标系统中的“轴”抽象一些，但在可视化系统中二者都是通常的轴，在很多方面的处理上都是类似地，能用于笛卡尔坐标轴的那些选项也能用于极坐标轴。例如关于轴标签、刻度线、刻度值标签、网格线、轴的长度、style 等的选项。当然极坐标系统有自己的特殊选项。




```
\tikz \datavisualization [scientific polar axes={0 to pi, clean}, all axes=grid,
    style sheet=vary hue, legend=below]
  [visualize as smooth line=sin,
  sin={label in legend={text=$1+\sin \alpha$}}]
data [format=function] {
  var angle : interval [0:pi];
```

```
  func radius = sin(\value{angle}r) + 1;
}
  [visualize as smooth line=cos,
  cos={label in legend={text=$1+\cos\alpha$}}]
data [format=function] {
  var angle : interval [0:pi];
  func radius = cos(\value{angle}r) + 1;
};
```






```
\tikz[baseline] \datavisualization [
  scientific axes={clean},
  x axis={attribute=angle, ticks={minor steps between steps=4}},
  y axis={attribute=radius, ticks={some, style=red!80!black}},
  all axes=grid,
  visualize as smooth line=sin]
data [format=function] {
  var t : interval [-3:3];
  func angle = exp(\value t);
  func radius = \value{t}*\value{t};
};
\qquad
\tikz[baseline] \datavisualization [
  scientific polar axes={right half clockwise, clean},
  angle axis={logarithmic, ticks={minor steps between steps=8,
          major also at/.list={2,3,4,5,15,20}}},
  radius axis={ticks={some, style=red!80!black}},
  all axes=grid,
  visualize as smooth line=sin]
data [format=function] {
  var t : interval [-3:3];
  func angle = exp(\value t);
  func radius = \value{t}*\value{t};
};
```

85.2 Scientific Polar Axis System

`/tikz/data visualization/scientific polar axes`=⟨*options*⟩ (no default)

这个选项开启极坐标系统，其中两个轴的名称分别默认为 `angle axis`, `radius axis`. ⟨*options*⟩ 中

的选项会被冠以前缀

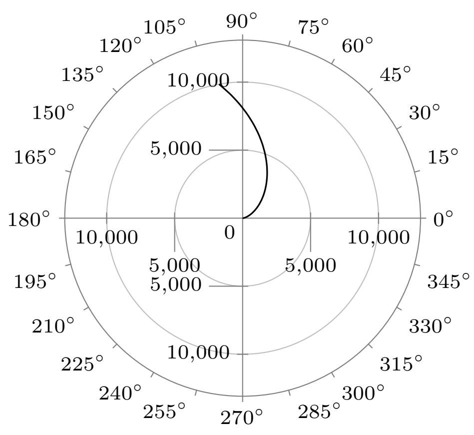
```
/tikz/data visualization/scientific polar axes
```

来执行。

与笛卡尔坐标系统一样，要把轴于数据点的数据标识符（变量名）对应起来，使用选项 `/tikz/data visualization/axis options/attribute`→P. 417.

对于半径轴 `radius axis`, 对数化选项 `/tikz/data visualization/axis options/logarithmic`→P. 421 无效，因为任何极径都包含极点，在对半径轴的实际长度做变换时都会涉及 0, 而对数函数在 0 处无定义。除此之外，半径轴都与笛卡尔坐标系统中的轴类似。在规定半径轴的实际长度时，可用

```
radius axis={scaling=0 at 0 and 999 at 2}
radius axis={length=2cm}
```




```
\tikz \datavisualization [scientific polar axes,
  radius axis={attribute=distance,
    ticks={step=5000, stack}, padding=1.5em,
    length=1.8cm, grid},
  visualize as smooth line]
data [format=function] {
  var angle : interval [0:100];
  func distance = \value{angle}*\value{angle};
};
```

85.2.1 角度轴的刻度线

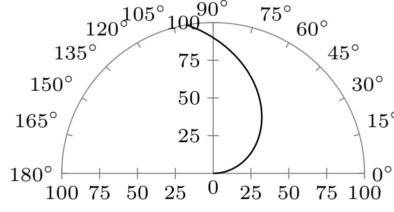
角度轴的刻度值标签处于“外侧”，刻度线有“向内”、“向外”两种选择。

`/tikz/data visualization/scientific polar axes/outer ticks` (no value)

这个选项使得角度轴的刻度线指向“外侧”，这是默认的。

`/tikz/data visualization/scientific polar axes/inner ticks` (no value)

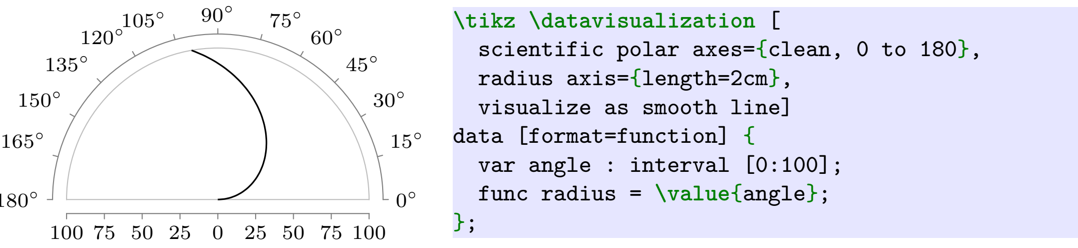
这个选项使得角度轴的刻度线指向“内侧”。




```
\tikz \datavisualization [
  scientific polar axes={inner ticks, 0 to 180},
  radius axis={length=2cm},
  visualize as smooth line]
data [format=function] {
  var angle : interval [0:100];
  func radius = \value{angle};
};
```

`/tikz/data visualization/scientific polar axes/clean` (no value)

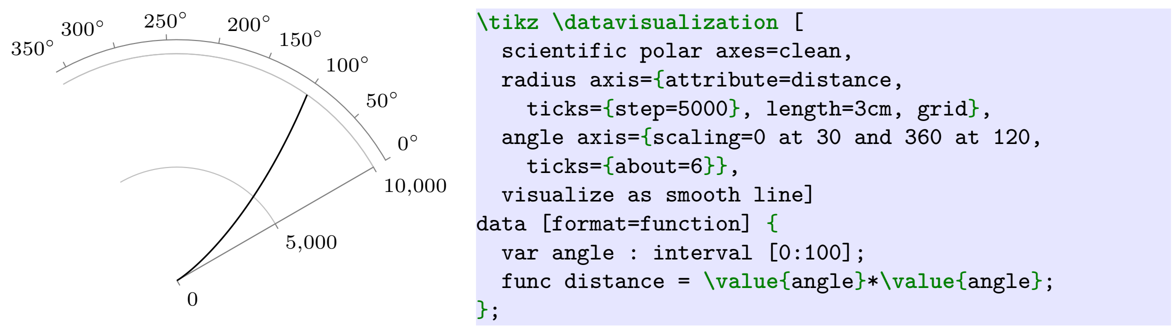
这个选项使得可视化的半径轴线段和可视化的角度轴圆弧脱离绘图区域，向外偏移一段距离，并且没有竖直的纵轴。这样可视化的轴就不会干扰读图。




```
\tikz \datavisualization [
  scientific polar axes={clean, 0 to 180},
  radius axis={length=2cm},
  visualize as smooth line]
data [format=function] {
  var angle : interval [0:100];
  func radius = \value{angle};
};
```

85.2.2 角度轴的角度范围

当说到角度轴的角度范围时，可能有两个意思：角度轴的刻度值标签所标示的角度范围，或者角度轴在页面上实际占用的角度范围。对于第 2 个意思，可在角度轴的选项中使用 `scaling=`⟨*a1*⟩ `at` ⟨*A1*⟩ `and` ⟨*a2*⟩ `at` ⟨*A2*⟩ 来设置。




```
\tikz \datavisualization [
  scientific polar axes=clean,
  radius axis={attribute=distance,
    ticks={step=5000}, length=3cm, grid},
  angle axis={scaling=0 at 30 and 360 at 120,
    ticks={about=6}},
  visualize as smooth line]
data [format=function] {
  var angle : interval [0:100];
  func distance = \value{angle}*\value{angle};
};
```

对于第 1 个意思，系统有多个预定义的选项，这些选项有 3 种类型

- 所规定的范围都是 90° 的整数倍。
- 所规定的范围都是 $\frac{\pi}{2}$ 的整数倍。
- 以角度轴处理的各数据点的最小角度为水平或竖直方向，并且各数据点的最大角度也是水平或竖直方向。

将这些预定义的选项简单列举如下，具体效果参考手册：

```
      0 to 90
    -90 to 0
      0 to 180
    -90 to 90
      0 to 360
   -180 to 180
      0 to pi half 范围同 0 to 90，但角度值标签使用弧度并带有 π
-pi half to 0
      0 to pi
-pi half to pi half
      0 to 2pi
    -pi to pi
quadrant
quadrant clockwise
fourth quadrant
fourth quadrant clockwise
upper half
```

```
upper half clockwise
lower half
lower half clockwise
left half
left half clockwise
right half
right half clockwise
```

85.3 创建新的极坐标系统

`/tikz/data visualization/new polar axes`={⟨*angle axis name*⟩}{⟨*radius axis name*⟩} (no default)

这个选项创建一个新的极坐标系统，并且分别把角度轴和半径轴命名为 ⟨*angle axis name*⟩, ⟨*radius axis name*⟩. 轴 ⟨*angle axis name*⟩ 对应的变量名 (attribute) 被默认为 `angle`, 轴 ⟨*radius axis name*⟩ 对应的变量名 (attribute) 被默认为 `radius`. 两个轴的默认名称是 `angle axis` 和 `radius axis`.

轴 ⟨*angle axis name*⟩ 会跟踪两个向量：**v** 和 **u**, 这两个向量（作为单位向量）的长度都是 `1pt`, 但方向不同。在初始之下，$\mathbf{v}$ = `(1pt,0pt)`, $\mathbf{u}$ = `(0pt,1pt)`. 数据点的半径 r 来自 TEX 尺寸 r`pt` 的数值部分。当给定半径 r 和角度 θ 时，由 $r \cdot \cos\theta \cdot \mathbf{v} + r \cdot \sin\theta \cdot \mathbf{u}$ 确定页面上一个点，这个点就是数据点 (r, θ) 对应的点。单位向量 **v** 和 **u** 的方向可以用下面的选项设置。

`/tikz/data visualization/axis options/unit vectors`={⟨*unit vector* **v** *degrees*⟩}{⟨*unit vector* **u** *degrees*⟩} (no default, initially `{(1pt,0pt)}{(0pt,1pt)}`)

该选项作为角度轴的选项。在初始之下，极坐标系统将 `(1pt,0pt)` 作为角度度量的始边方向，即 0 度方向；将 `(0pt,1pt)` 作为 90 度方向。其中 ⟨*unit vector* **v** *degrees*⟩ 和 ⟨*unit vector* **u** *degrees*⟩ 都是 TikZ 的坐标形式。

```
\tikz \datavisualization [
  new polar axes={angle axis}{radius axis},
  radius axis={unit length=1cm},
  angle axis={unit vectors={(10:1pt)}{(60:1pt)}},
  visualize as scatter]
data [format=named] {
  angle={0,90}, radius={0.25,0.5,...,2}
};
```

`/tikz/data visualization/axis options/degrees` (no value)

该选项作为角度轴的选项。这个选项决定角度的度量和计算采用角度制。

```
\tikz \datavisualization [
  new polar axes={angle axis}{radius axis},
  radius axis={unit length=1cm},
  angle axis={degrees},
  visualize as scatter]
data [format=named] {
  angle={10,90}, radius={0.25,0.5,...,2}
};
```

`/tikz/data visualization/axis options/radians` (no value)

该选项作为角度轴的选项。这个选项决定角度的度量和计算采用弧度制。

```
\tikz \datavisualization [
  new polar axes={angle axis}{radius axis},
  radius axis={unit length=1cm},
  angle axis={radians},
  visualize as scatter]
data [format=named] {
  angle={0,1.5}, radius={0.25,0.5,...,2}
};
```

86 The Data Visualization Backend

87 Key Management

```
\usepackage{pgfkeys} % LaTeX
\input pgfkeys.tex % plain TeX
\usemodule[pgfkeys] % ConTeXt
```

本节讲解 pgfkeys 宏包，本宏包可以独立于 PGF 使用，PGF 和 TikZ 会自动加载该宏包。本宏包的运行不依赖 PGF 的其它宏包。

87.1 简介

87.1.1 与其它类似宏包的比较

pgfkeys 宏包定义了一种键值管理系统，在某些方面与 keyval 宏包或其改进版 xkeyval 宏包类似。pgfkeys 宏包与 xkeyva 宏包的主要区别在于：

- pgfkeys 宏包将键（key）组织为“树”，而 xkeyval 宏包将键组织为“族”（family）。
- pgfkeys 宏包没有 save-stack 效应。
- pgfkeys 宏包比 xkeyval 宏包稍慢，但也慢不了很多。
- pgfkeys 宏包支持“样式”（style），也就是说，可以把多个键的作用可以保存在一个键中，使用这一个键就可以实现多个键的作用。
- pgfkeys 宏包可以定义有多个参数的键。
- pgfkeys 宏包支持“手柄”（handler）。

87.1.2 快速引导

pgfkeys 宏包将键（key）组织为“树”，从树的根节点到某个键所包含的各个层次的键叫作一个“键路径”，一个路径包含键名称和分隔键名称的斜线，一个完整的键路径以斜线开头，例如

```
/tikz/cs/x
```

这是坐标系中代表 x 分量的键的路径，有 3 个层次。键路径中，最后一个斜线之后的符号叫做“键名称”；最后一个斜线之前的所有符号叫作该键名称的“前缀路径”，例如上面的键名称 x 的前缀路径就是”/tikz/cs”. “前缀路径”＋“/”＋“键名称”组成一个完整的键路径。

当某个键正在被处理时，该键的名称保存在宏 `\pgfkeyscurrentname` 中，该键的前缀路径保存在宏 `\pgfkeyscurrentpath` 中。

命令 `\pgfkeys` 是处理 key 的主要命令，这个命令处理具有键路径形式的一串符号或列表。命令 `\pgfkeys` 会导致一系列的条件分支过程，对不同情况的键做不同的处理。当执行 `\pgfkeys{`⟨*key*⟩`}` 时，可以用 ⟨*key*⟩ 做某些事情，例如，可以返回一个错误，可以用 ⟨*key*⟩ 保存某个“值”，也可以用 ⟨*key*⟩ 保存或执行某些命令。具有某种作用的 key 通常是“手柄”，通常所谓的“手柄”就是包含“`/.`”的键，例如 ⟨*key*⟩`/.code`, ⟨*key*⟩`/.style` 等，见下面的例子。

如果执行 `\pgfkeys{`⟨*key*⟩`}` 的结果是将某个“值”或某些代码、命令保存到 ⟨*key*⟩ 中，那么就说这是“定义⟨*key*⟩”；如果执行 `\pgfkeys{`⟨*key*⟩`}` 的结果是导致保存在 ⟨*key*⟩ 中的代码、命令被执行，或者导致保存在 ⟨*key*⟩ 中的值被调出、利用，那么就说这是“执行⟨*key*⟩”. 这样规定的“定义”与“执行”并不严格，只是直观上的区分。

在执行键时，通常使用“键值对”（key–value pairs）的形式，这是一种赋值形式，例如：

The value is 'Hi!'.

```
\pgfkeys{/my key/.code=The value is '#1'.}
\pgfkeys{/my key=Hi!}
```

这个例子的第 1 行将代码 `The value is '#1'.` 保存到“手柄” `/my key/.code` 中，从而定义键 `/my key`. 第 2 行给键 `/my key` 赋值并执行所保存的代码，得到一个句子，这是执行键 `/my key`.

再如定义包含两个变量的键：

The values are 'a1' and 'a2'.

```
\pgfkeys{/my key/.code 2 args=The values are '#1' and '#2'.}
\pgfkeys{/my key={a1}{a2}}
```

下面例子定义一个包含变量的键，并用手柄 `/.default` 指定这个键的默认值：

hello
world!

```
\pgfkeys{/my key/.code=#1}
\pgfkeys{/my key/.default=hello}
\pgfkeys{/my key, /my key={\par world!}}
```

在使用键时，一般应写出完整的键路径，但是如果某个键有默认路径，就可以只写出键名称，省略路径前缀。在执行键时，默认的前缀路径会被自动添加。Ti*k*Z 中的键多数都以 `/tikz/` 为默认路径前缀，例如

```
/tikz/line width
```

在 `{tikzpicture}` 环境中绘图时，写出

```
\draw[/tikz/line width=10mm]...
\draw[line width=10mm]...
```

都是可以的。可以用手柄 `/.cd` 来设置键的默认路径，句法是：

`\pgfkeys{`⟨前缀⟩`/.cd, `⟨*key1*⟩`=`⟨*value1*⟩`, `⟨*key2*⟩`=`⟨*value2*⟩`, ...}`

这个句式中的键 ⟨*key1*⟩, ⟨*key2*⟩ 等会都有相同的前缀 ⟨前缀⟩，例如，

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

再如

B,N!

```
\pgfkeys{/a/b/.initial={B,},/m/n/.initial=N!}
\pgfkeys{/a/.cd,b,/m/.cd,n}
```

这个例子中，键 b 的前缀被设为 /a; 键 n 的前缀被设为 /m.

执行命令 \pgfkeys 时，命令中的各个键应当以斜线“/”开头，否则，命令会自动给键加上斜线，例如：

(a:foo)(b:bar)(a:wow)

```
\pgfkeys{a/.code=(a:#1)}
\pgfkeys{b/.code=(b:#1)}
\pgfkeys{my style/.style={a=foo,b=bar,a=#1}}
\pgfkeys{my style=wow}
```

当以 / 开头写出键时，就默认所写的键是带有完整路径的键，完整的键路径不会被“修正”。

用手柄 /.style 定义的键叫作“样式”，样式中可以使用自定义的键，即把某个键作为其它键的参数，也就是说此时可以把键套嵌使用。下面的例子利用手柄 /.style 把 /tikz 设为默认的路径前缀：

```
\pgfkeys{/tikz/.style=/tikz/.cd}
\pgfkeys{tikz,line width=1cm,draw=red}
```

还需要注意的是，命令 \pgfkeys 定义的键的有效范围受到 TEX 分组的限制，超出所在分组无效。TEX 分组是花括号、环境等限定的范围。

87.2 The Key Tree

前面提到可以给键定义默认路径，但是默认路径不同于搜索路径，pgfkeys 并不执行针对键路径的搜索。当一个键有默认路径时，可以写出该键的完整路径，也可以只写键名称，例如以下 2 个形式都可以接受：

```
/pgf/arrow keys/fill=red
fill=red
```

```
\tikz \draw[-{Stealth[fill=red,scale=5]}](0,0);
\tikz \draw[-{Stealth[/pgf/arrow keys/fill=red,scale=5]}](0,0);
```

一个键中可以储存某些记号 (tokens), 记号可以是字符、其它的键、命令、宏等。可以用命令 \pgfkeys 或 \tikzset 向键中存储记号。下面的命令也可以向键中储存某些记号或者读取键中所存储的记号，这些命令主要用于内部程序的处理过程。

\pgfkeyssetvalue{⟨*full key*⟩}{⟨*token text*⟩}

这个命令的定义是：

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

本命令定义 \csname pgfk@#1\endcsname.

先把参数 #2 保存到记号寄存器 \pgfkeys@temptoks 中，再用 \the 把这个寄存器展开，然后把展开值保存到 \csname pgfk@#1\endcsname 中。

本命令将 ⟨*token text*⟩ 保存到 ⟨*full key*⟩ 中，⟨*full key*⟩ 必须是完整的键路径，本命令不对“非完整键”执行添加默认前缀路径的操作。⟨*token text*⟩ 可以是字符，TEX 的 if 语句（可以是不平衡的条件句），或者其它命令（如绘图命令），其中可以含有变量符号 #. 当执行 ⟨*full key*⟩ 后，输出 ⟨*token text*⟩. 可以用命令 \pgfkeys 或 \pgfkeysvalueof 执行 ⟨*full key*⟩.

```
\pgfkeyssetvalue{/fw}{\tikz\draw circle(10pt);}
\pgfkeys{/fw}
```

此命令定义的键的有效范围受到 TeX 分组的限制。

\pgfkeyssetevalue{⟨*full key*⟩}{⟨*token text*⟩}

这个命令的定义是：

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

本命令类似 \pgfkeyssetvalue, 其中的 \scantokens 是 eTeX 的命令，此命令的作用是将它的参数（不能再进一步展开的文本）转换为字符串，把这个字符串当作内部文件（一个 pseudo-file），并读取这个伪文件，参考《LaTeX 2_ε 文类和宏包学习手册》,《The ε-TeXmanual》.

\pgfkeysaddvalue{⟨*full key*⟩}{⟨*pre code*⟩}{⟨*append code*⟩}

此命令的作用是对键 ⟨*full key*⟩ 进行重定义，使得保存在键 ⟨*full key*⟩ 中的值由三部分组成：首先是 {⟨*pre code*⟩}, 再是该键原来所保存的代码，再是 {⟨*append code*⟩}. 此命令的定义是：

```
\long\def\pgfkeysaddvalue#1#2#3{%
  {%
    \toks0{#2}%
    \pgfkeysifdefined{#1}
    {\pgfkeys@temptoks\expandafter\expandafter\expandafter{\csname pgfk@#1
    ↪   \endcsname}}%
    {\pgfkeys@temptoks{}}%
    \toks1{#3}%
    \xdef\pgfkeys@global@temp{\the\toks0 \the\pgfkeys@temptoks \the\toks1}
    ↪   % believe or don't: the spaces are important
  }%
  \pgfkeyslet{#1}\pgfkeys@global@temp%
}
```

此命令的处理过程是：

1. 把 ⟨*pre code*⟩ 保存到记号寄存器 \toks0 中。
2. 用命令 \pgfkeysifdefined 检查名称为 pgfk@⟨*full key*⟩ 的命令是否已定义。
   - 如果已定义,则把 \csname pgfk@⟨*full key*⟩\endcsname 保存的记号保存到寄存器 \pgfkeys@temptoks 中。
   - 如果未定义，则清空寄存器 \pgfkeys@temptoks 的内容。
3. 把 ⟨*append code*⟩ 保存到记号寄存器 \toks1 中。
4. 定义全局宏 \pgfkeys@global@temp, 定义内容是

   ⟨*pre code*⟩ \csname pgfk@⟨*full key*⟩\endcsname ⟨*append code*⟩

5. 执行 \pgfkeyslet{⟨*full key*⟩}\pgfkeys@global@temp 来重定义 \csname pgfk@⟨*full key*⟩\endcsname.

$x\uparrow$■$\downarrow y$

```
\pgfkeyssetvalue{/a/b/c}{\rule{1em}{1em}}
\pgfkeysaddvalue{/a/b/c}{$\uparrow$\,}{\,$\downarrow$}
$x$\pgfkeys{/a/b/c}$y$
```

\pgfkeyslet{⟨*full key*⟩}{⟨*macro*⟩}

这个命令的定义是:

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

本命令定义 \csname pgfk@#1\endcsname.

本命令用 \let 命令使得 {⟨*full key*⟩} 指向宏 ⟨*macro*⟩, 即执行 {⟨*full key*⟩} 就相当于执行宏 ⟨*macro*⟩. 如果 ⟨*macro*⟩ 是 \relax, 则执行 {⟨*full key*⟩} 不会出错也没有任何结果, 但是却有副作用, 即使得程序不能将 {⟨*full key*⟩} 与未定义的键区分开。

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

这个命令的定义是:

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

本命令定义 \csname pgfk@#1\endcsname.

本命令定义命令 ⟨*macro*⟩, 利用 \let 使得 ⟨*macro*⟩ 等于 \csname pgfk@#1\endcsname. 保存在 ⟨*full key*⟩ 中的记号 (tokens) 也会被赋予 ⟨*macro*⟩. 执行 ⟨*macro*⟩ 就相当于执行键 ⟨*full key*⟩. 如果 ⟨*full key*⟩ 尚未被 set, 则 ⟨*macro*⟩ 等于 \relax.

Hello, world!

```
\pgfkeyssetvalue{/my family/my key}{Hello, world!}
\pgfkeysgetvalue{/my family/my key}{\helloworld}
\helloworld
```

\pgfkeysvalueof{⟨*full key*⟩}

这个命令的定义是:

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

将保存在 ⟨*full key*⟩ 中的记号插入到当前位置。

false

```
\pgfkeyssetvalue{/ifstat}{\ifnum 1>10 true \else false \fi}
\pgfkeysvalueof{/ifstat}
```

\pgfkeysifdefined{⟨*full key*⟩}{⟨*if code*⟩}{⟨*else code*⟩}

本命令的定义是:

```
\long\def\pgfkeysifdefined#1#2#3{\pgfkeys@ifcsname pgfk@#1\endcsname#2\else#3\fi
↪   }
```

它检查名称为 pgfk@#1 的命令是否已经被定义,也就是检查 ⟨*full key*⟩ 是否已经用 \pgfkeyssetvalue 或 \pgfkeyslet 做了定义。如果已经定义，则执行 ⟨*if code*⟩, 否则执行 ⟨*else code*⟩. 本命令使用 eTEX 的 \ifcsname 命令，所以，当 ⟨*full key*⟩ 被设置为 \relax 时，本命令在 TEX 中的表现与在 eTEX 中的表现可能有所不同。

yes

```
\pgfkeyslet{/my family/my key}{\relax}
\pgfkeysifdefined{/my family/my key}{yes}{no}
```

87.3 执行 Keys

↓ ↓

下面分析命令 \pgfkeys 的处理过程。在文件《pgfkeys.code.tex》中定义了命令 \pgfkeys:

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

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

\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}}
```

可见命令 \pgfkeys 的直接定义是不含参数的，其主要功能由命令 \pgfkeys@parse 来实现。

下面针对 \pgfkeys{⟨*key-value*⟩}, 即只处理一个 ⟨*key*⟩ 的情况，分析

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

的解析处理过程，这个过程比较长，按处理的次序或条件分支情况，分成几个 paragraph 来叙述。

下面的叙述中，条件分支用 {itemize} 环境表达，先后步骤用 {enumerate} 环境表达。

判断解析过程是否结束 命令 \pgfkeys@parse 的定义是：

```
\def\pgfkeys@parse{\futurelet\pgfkeys@possiblerelax\pgfkeys@parse@main}
\def\pgfkeys@parse@main{%
  \ifx\pgfkeys@possiblerelax\pgfkeys@mainstop%
    \expandafter\pgfkeys@cleanup%
  \else%
    \expandafter\pgfkeys@normal%
  \fi%
}
```

从这个定义看,如果 \ifx\pgfkeys@possiblerelax\pgfkeys@mainstop 的判断为真,即没有“⟨*key-value*⟩,”这一部分，那么执行

```
\pgfkeys@parse\pgfkeys@mainstop
```

也就是

```
\pgfkeys@cleanup\pgfkeys@mainstop
```

这会把记号 \pgfkeys@mainstop “吃掉”(后文也有这个用法), 并结束解析过程。如果有 “⟨key-value⟩,” 这一部分, 那么执行 \pgfkeys@normal.

命令 \pgfkeys@normal 首先检查 ⟨key-value⟩ 是否属于 “首字符检测句法”, 如果是, 则按 “首字符检测句法” 来处理; 如果不是, 则执行 \pgfkeys@@normal.

检查 ⟨key-value⟩ 是否属于 “首字符检测句法” 执行 \pgfkeys@normal:

```
\pgfkeys@normal ⟨key-value⟩,\pgfkeys@mainstop
```

这导致条件分支命令 \ifpgfkeys@syntax@handlers, 这个条件命令检查 ⟨key-value⟩ 是不是属于 “首字符检测句法”:

问题 1

- 如果有真值 \pgfkeys@syntax@handlerstrue, 则执行

  ```
  \pgfkeys@syntax@handlers ⟨key-value⟩,\pgfkeys@mainstop
  ```

  这个执行过程会:

  1. 把 ⟨key-value⟩ 开头的空格去掉,假设(只是假设)去掉开头的空格后得到 ⟨(no pre space) key-value⟩.
  2. 利用 \futurelet 命令使得 \pgfkeys@first@char 等同于 ⟨(no pre space) key-value⟩ 的第一个非空格符号。
  3. 然后执行

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

     问题 2

     即定义宏 \pgfkeys@the@handler.
  4. 然后执行一个条件分支命令 \ifx:
     - 如果 \pgfkeys@the@handler 不等于 \relax 就执行

       ```
       \pgfkeys@use@handler ⟨(no pre space) key-value⟩,\pgfkeys@mainstop
       ```

       因为有定义

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

       所以命令 \pgfkeys@use@handler 会 “吃掉” 一个逗号。这个命令展开后就是

       ```
       \pgfkeys@the@handler{⟨(no pre space) key-value⟩}%
       \pgfkeys@parse\pgfkeys@mainstop
       ```

       这导致记号 “\pgfkeys@mainstop” 被 “吃掉”。
     - 如果 \pgfkeys@the@handler 等于 \relax 就执行

       ```
       \pgfkeys@@normal ⟨(no pre space) key-value⟩,\pgfkeys@mainstop
       ```

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

  ```
  \pgfkeys@@normal ⟨key-value⟩,\pgfkeys@mainstop
  ```

命令 \pgfkeys@@normal 的处理 这个处理过程有点长。

因为有定义

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

所以命令 \pgfkeys@@normal 会“吃掉”一个逗号。执行

```
\pgfkeys@@normal ⟨key-value⟩,\pgfkeys@mainstop%
```

这个命令的展开如下:

```
\pgfkeys@unpack ⟨key-value⟩=\pgfkeysnovalue=\pgfkeys@stop%
\pgfkeys@parse\pgfkeys@mainstop%
```

命令 \pgfkeys@parse\pgfkeys@mainstop 什么也不做，它只是“吃掉”记号 \pgfkeys@mainstop.

命令 \pgfkeys@unpack 的定义格式是:

```
\long\def\pgfkeys@unpack#1=#2=#3\pgfkeys@stop{%
...
}
```

所以,如果 ⟨*key-value*⟩ 是 ⟨*key*⟩=⟨*value*⟩ 的形式,那么此命令的参数 #1 对应 ⟨*key*⟩, 参数 #2 对应 ⟨*value*⟩. 如果 ⟨*key-value*⟩ 是 ⟨*key*⟩ 的形式，那么此命令的参数 #1 对应 ⟨*key*⟩, 参数 #2 对应 \pgfkeysnovalue.

命令 \pgfkeys@unpack 的处理是:

1. 先定义 \pgfkeyscurrentkey:

   ```
   \pgfkeys@spdef\pgfkeyscurrentkey{⟨key⟩}%
   \edef\pgfkeyscurrentkey{\pgfkeyscurrentkey}
   ```

2. 执行条件分支 \ifx:

   - 如果 \pgfkeyscurrentkey 等于 \pgfkeys@empty, 即 ⟨*key*⟩ 是空的，那么什么也不做，否则:
   - 如果 \pgfkeyscurrentkey 的值非空，则执行以下几个步骤:

     (a) 执行 \pgfkeys@add@path@as@needed, 此命令会导致一个 \ifx:

       – 如果 \pgfkeyscurrentkey 的值以斜线“/”开头，即 ⟨*key*⟩ 以斜线开头，那么执行

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

       – 如果 \pgfkeyscurrentkey 的值不以斜线“/”开头，那么执行

       ```
       \pgfkeysaddeddefaultpathtrue
       \def\pgfkeyscurrentkeyRAW{⟨ 当前 \pgfkeyscurrentkey 的展开值⟩}%
       \edef\pgfkeyscurrentkey{\pgfkeysdefaultpath ⟨key⟩}%
       ```

即把保存在 \pgfkeysdefaultpath 中的斜线“/”加到 ⟨*key*⟩ 前面，并以此重定义 \pgfkeyscurrentkey. 注意在执行 \pgfkeys@parse 前有规定:

```
\let\pgfkeysdefaultpath\pgfkeys@root%
```

因此只要不改变 \pgfkeys@root 的值,或者不在 \pgfkeys{⟨*key-value list*⟩} 的 ⟨*key-value list*⟩ 中使用键修改 \pgfkeysdefaultpath 的值,那么对于 \pgfkeys 来说,命令 \pgfkeysdefaultpath 的值就是斜线“/”. 如果在 ⟨*key-value list*⟩ 中使用键修改 \pgfkeysdefaultpath 的值为 ⟨*some char*⟩, 那么 ⟨*some char*⟩ 应该以斜线“/”开头、结尾，否则就没有机会再添加斜线“/”了。这里分析的是 \pgfkeys{⟨*key-value*⟩}, 即只处理一个 ⟨*key*⟩ 的情况，所以就认为 \pgfkeysdefaultpath 的值是斜线“/”.

可见执行 \pgfkeys@add@path@as@needed 后，保存在 \pgfkeyscurrentkey 中的就是以“/”开头的完整的键路径 ⟨*full key*⟩.

(b) 然后定义 \pgfkeyscurrentvalue, 这个宏保存 ⟨*key*⟩ 的当前值:

```
\pgfkeys@spdef\pgfkeyscurrentvalue{#2}
```

其中的参数 #2 或者是 ⟨*key*⟩=⟨*value*⟩ 中的 ⟨*value*⟩, 或者是 \pgfkeysnovalue.
命令 \pgfkeys@spdef#1#2 的作用是：如果 #2 保存的是空格，那么把 #2 作为记号保存到 #1 中；如果 #2 不是空格，那么先把 #2 的第一个记号（只是第一个）展开，这会把 #2 开头的空格吃掉（如果 #2 以空格开头的话），然后把这样展开后的记号保存到记号寄存器中，然后用 \the 展开这个寄存器，再将展开得到的记号序列保存到 #1 中。所以，如果 #2 是“a b”这种中间有空格的一串符号，那么中间的空格不会被忽略（连续的数个空格看作一个空格），会被保存到 #1 中。例如:

xa bx

```
\makeatletter
\pgfkeys@spdef\aaaa{    a    b}
x\expandafter\verb\expandafter!\aaaa!x
\makeatother
```

(c) 然后用 \ifx 检查 \pgfkeyscurrentvalue 与 \pgfkeysnovalue@text 的定义是否相同 (有 \def\pgfkeysnovalue@text{\pgfkeysnovalue}):

– 如果 \pgfkeyscurrentvalue 与 \pgfkeysnovalue@text 的定义相同,就说明 ⟨*key-value*⟩ 是 ⟨*key*⟩ 的形式，那么就执行命令 \pgfkeysifdefined 来检查名称为

pgfk@\pgfkeyscurrentkey/.@def 即 pgfk@⟨*full key*⟩/.@def

问题 3

的命令是否已经被定义，即检查 ⟨*full key*⟩ 是否有默认值:

  * 如果名称为 pgfk@⟨*full key*⟩/.@def 的命令已经被定义，则执行

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

    即把 ⟨*full key*⟩ 的默认值赋予 \pgfkeyscurrentvalue.

  * 如果名称为 pgfk@⟨*full key*⟩/.@def 的命令尚未被定义，则什么也不做。

– 如果 \pgfkeyscurrentvalue 与 \pgfkeysnovalue@text 的定义不相同,就说明 ⟨*key-value*⟩ 是 ⟨*key*⟩=⟨*value*⟩ 的形式，而 \pgfkeyscurrentvalue 的值就是 ⟨*value*⟩, 此时什么也不做。

(d) 然后用 \ifx 检查 \pgfkeyscurrentvalue 与 \pgfkeysvaluerequired 的定义是否相同 (有 \def\pgfkeysvaluerequired{\pgfkeysvaluerequired}):

- 如果 \pgfkeyscurrentvalue 与 \pgfkeysvaluerequired 的定义相同，则执行

```
\def\pgf@marshal{\pgfkeysvalueof{/errors/value required/.@cmd}}%
\expandafter\pgf@marshal\expandafter{\pgfkeyscurrentkey}{}\pgfeov%
```

问题 4 即执行命令 \pgf@marshal.

- 如果 \pgfkeyscurrentvalue 与 \pgfkeysvaluerequired 的定义不相同，则执行

```
\pgfkeys@case@one%
```

执行 \pgfkeys@case@one 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}%
}
```

此命令实际上执行命令 \pgfkeysifdefined, 检查名称为

pgfk@\pgfkeyscurrentkey/.@cmd 即 pgfk@⟨*full key*⟩/.@cmd

问题 5 的命令是否已经被定义。

- 如果已定义，则执行

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

问题 6 即执行 \pgfkeys@code, 其中的 \pgfeov 作为参数定界标志。

从上面代码的形式看，\pgfkeys@code 应该等于某个命令 \⟨*macro*⟩, 此命令保存在 \csname pgfk@\pgfkeyscurrentkey/.@cmd 中，并且执行此命令时应该使用格式: \⟨*macro*⟩⟨ 参数列举格式⟩\pgfeov. 例如:

1 add 2

```
\def\aaaa#1+#2\pgfeov{#1 add #2}
\expandafter\let\csname pgfk@/a/b/.@cmd\endcsname\aaaa
\pgfkeys{a/b=1+2}
```

这就是命令 \pgfkeysdef →P. 540 的思路。

- 如果未定义，则执行 \pgfkeys@case@two.

执行 \pgfkeys@case@two 此命令的定义是:

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

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

此命令实际上执行命令 `\pgfkeysifdefined`, 检查名称为

`pgfk@\pgfkeyscurrentkey` 即 `pgfk@`⟨*full key*⟩

问题 7 的命令是否已经被定义:

- 如果已定义，则执行 `\pgfkeys@case@two@extern`, 此命令导致一个 `\ifx` 语句:
  - 如果 `\pgfkeyscurrentvalue` 与 `\pgfkeysnovalue@text` 的定义相同 (都是`\pgfkeysnovalue`), 则执行

    ```
    \pgfkeysvalueof{\pgfkeyscurrentkey}
    ```

    问题 8 即执行 `\csname pgfk@\pgfkeyscurrentkey\endcsname`.
  - 如果 `\pgfkeyscurrentvalue` 与 `\pgfkeysnovalue@text` 的定义不相同，则执行

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

    这会导致重定义:

    ```
    \expandafter\let\csname pgfk@\pgfkeyscurrentkey\endcsname
    ↪   \pgfkeyscurrentvalue%
    ```

    把 `\pgfkeyscurrentvalue` 的当前值保存到 `\csname pgfk@\pgfkeyscurrentkey\endcsname` 中，注意此时 `\pgfkeyscurrentkey` 保存的是以 “/” 开头的完整键路径。
- 如果未定义，则执行 `\pgfkeys@case@three`.

执行 `\pgfkeys@case@three` 此命令的处理是:

1. 执行 `\pgfkeys@split@path`, 此命令的处理是:
   (a) 执行 `\pgfkeys@pathtoks{}`, 即清空该寄存器的内容。
   (b) 执行

   ```
   \expandafter\pgfkeys@splitter\pgfkeyscurrentkey//%
   ```

   此时 `\pgfkeyscurrentkey` 保存的是以 “/” 开头的完整键路径，所以这个命令等于

   ```
   \pgfkeys@splitter/⟨some char⟩//%
   ```

   命令 `\pgfkeys@splitter` 的定义是:

   ```
   \def\pgfkeys@splitter#1/#2/{%
     \def\pgfkeys@temp{#2}%
     \ifx\pgfkeys@temp\pgfkeys@empty%
   ```

```
    % Ah. done
    \def\pgfkeyscurrentname{#1}%
    \expandafter\pgfkeys@gobbletoslash%
  \else%
    \expandafter\pgfkeys@pathtoks\expandafter{\the\pgfkeys@pathtoks#1/}%
  \fi%
  \pgfkeys@splitter#2/%
}
\def\pgfkeys@gobbletoslash\pgfkeys@splitter/{%
  \if\relax\detokenize\expandafter{\the\pgfkeys@pathtoks}\relax\else
    \expandafter\pgfkeys@remove@slash\the\pgfkeys@pathtoks\relax
  \fi
}%
\def\pgfkeys@remove@slash#1/\relax{\pgfkeys@pathtoks{#1}}
```

按照这个定义，代码

```
\expandafter\pgfkeys@pathtoks\expandafter{\the\pgfkeys@pathtoks/}%
```

至少被执行一次。

所以命令 \pgfkeys@split@path 的作用是：将键的名称保存在 \pgfkeyscurrentname 中，将键名称之前的前缀路径保存在寄存器 \pgfkeys@pathtoks 中。从这个定义可知，完整的键路径包括三部分："前缀路径" + "/" + "键的名称"。

因为有 \def\pgfkeyscurrentpath{\the\pgfkeys@pathtoks}, 所以此时的宏 \pgfkeyscurrentpath 保存当前键的前缀路径。

2. 执行 \pgfkeysifdefined, 检查名称为

   pgfk@/handlers/\pgfkeyscurrentname/.@cmd 即 pgfk@/handlers/⟨*key name*⟩/.@cmd

问题 9 的命令是否已经被定义：

   • 如果已定义，则执行

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

问题 10 即执行 \pgfkeys@code, 其中 \pgfeov 是参数定界标志。

   • 如果未定义，则执行

```
\pgfkeys@unknown
```

此命令实际执行 \pgfkeysifdefined 来检查名称为

   pgfk@\pgfkeyscurrentpath/.unknown/.@cmd 即 pgfk@⟨*key path*⟩/.unknown/.@cmd

问题 11 的命令是否已经被定义：

   – 如果已定义，则执行

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

问题 12 即执行 \pgfkeys@code, 其中 \pgfeov 是参数定界标志。

– 如果未定义，则执行

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

问题 13 即执行 \pgfkeys@code, 其中 \pgfeov 是参数定界标志。

以上就是命令 \pgfkeys{⟨*key-value*⟩} 的主要处理过程，这个过程不是自足的，其中的某些要素不在过程之内确定，例如：

问题 1 如何确定 \ifpgfkeys@syntax@handlers 的真值。

问题 2 如何确定 \pgfkeys@the@handler 的值，即如何确定保存在

```
/handlers/first char syntax/\meaning\pgfkeys@first@char
```

中的值。

问题 3 如何定义名称为 pgfk@\pgfkeyscurrentkey/.@def 的命令。

问题 4 ……

当命令 \pgfkeys 处理键值对列表 {⟨*key-value list*⟩} 时，将逗号作为相邻 ⟨*key-value*⟩ 的分隔符号，按从左到右的次序，逐个处理列表中的 ⟨*key-value*⟩. 逗号的这个“分隔”作用是这样实现的——把逗号用作命令 \pgfkeys@use@handler 和命令 \pgfkeys@@normal 的参数定界标志。假设执行

\pgfkeys{⟨*key-value 1*⟩,⟨*key-value 2*⟩}

其中的 ⟨*key-value 1*⟩ 与 ⟨*key-value 2*⟩ 都不属于“首字符检测句法”，那么处理过程就是：

```
\pgfkeys{⟨key-value 1⟩,⟨key-value 2⟩}
导致
\expandafter\pgfkeys@@set\expandafter{\pgfkeysdefaultpath}{⟨key-value 1⟩,⟨key-value 2⟩}
导致
\pgfkeys@@set{/}{⟨key-value 1⟩,⟨key-value 2⟩}
导致
\let\pgfkeysdefaultpath\pgfkeys@root%
\pgfkeys@parse ⟨key-value 1⟩,⟨key-value 2⟩,\pgfkeys@mainstop%
\def\pgfkeysdefaultpath{/}
导致
\let\pgfkeysdefaultpath\pgfkeys@root%
\pgfkeys@normal ⟨key-value 1⟩,⟨key-value 2⟩,\pgfkeys@mainstop%
\def\pgfkeysdefaultpath{/}
导致
\let\pgfkeysdefaultpath\pgfkeys@root%
\pgfkeys@@normal ⟨key-value 1⟩,⟨key-value 2⟩,\pgfkeys@mainstop%
\def\pgfkeysdefaultpath{/}
导致
\let\pgfkeysdefaultpath\pgfkeys@root%
\pgfkeys@unpack ⟨key-value 1⟩=\pgfkeysnovalue=\pgfkeys@stop%
\pgfkeys@parse ⟨key-value 2⟩,\pgfkeys@mainstop%
\def\pgfkeysdefaultpath{/}
导致
\let\pgfkeysdefaultpath\pgfkeys@root%
\pgfkeys@unpack ⟨key-value 1⟩=\pgfkeysnovalue=\pgfkeys@stop%
\pgfkeys@normal ⟨key-value 2⟩,\pgfkeys@mainstop%
\def\pgfkeysdefaultpath{/}
```

```
导致
\let\pgfkeysdefaultpath\pgfkeys@root%
\pgfkeys@unpack ⟨key-value 1⟩=\pgfkeysnovalue=\pgfkeys@stop%
\pgfkeys@@normal ⟨key-value 2⟩,\pgfkeys@mainstop%
\def\pgfkeysdefaultpath{/}
导致
\let\pgfkeysdefaultpath\pgfkeys@root%
\pgfkeys@unpack ⟨key-value 1⟩=\pgfkeysnovalue=\pgfkeys@stop%
\pgfkeys@unpack ⟨key-value 2⟩=\pgfkeysnovalue=\pgfkeys@stop%
\pgfkeys@parse\pgfkeys@mainstop%
\def\pgfkeysdefaultpath{/}
导致
\let\pgfkeysdefaultpath\pgfkeys@root%
\pgfkeys@unpack ⟨key-value 1⟩=\pgfkeysnovalue=\pgfkeys@stop%
\pgfkeys@unpack ⟨key-value 2⟩=\pgfkeysnovalue=\pgfkeys@stop%
\def\pgfkeysdefaultpath{/}
```

另外，按命令 `\pgfkeys` 的定义，其处理过程的最后一步是 `\def\pgfkeysdefaultpath{#1}`, 这会把 `\pgfkeysdefaultpath` 的值恢复到执行 `\pgfkeys` 之前的状态，也就是之前的 `\pgfkeys@root` 的值。前面提到，`\pgfkeysdefaultpath` 的最初值（即 `\pgfkeys@root` 的初值）是斜线“/”，如果在执行 `\pgfkeys{`⟨*key-value list*⟩`}` 之前改变 `\pgfkeys@root` 的值，或者在 ⟨*key-value list*⟩ 中使用键修改 `\pgfkeysdefaultpath` 的值，都会影响内部命令 `\pgfkeys@add@path@as@needed` 所添加的“默认前缀路径”。

如果在 ⟨*key-value list*⟩ 中使用键修改 `\pgfkeysdefaultpath` 的值为 ⟨*some char*⟩, 那么 ⟨*some char*⟩ 应该以斜线“/”开头、结尾，并且这个修改只在 `\pgfkeys` 的处理过程之内有效，`\pgfkeys` 的最后一步动作仍然是把 `\pgfkeysdefaultpath` 恢复到执行 `\pgfkeys` 之前的状态。例如：

Ne^{V}

```
\expandafter\def\csname pgfk@/m/n/.@cmd\endcsname#1\pgfeov{#1}%
\expandafter\def\csname pgfk@/u/v/.@cmd\endcsname#1\pgfeov{$\mathrm e^{#1}$}%
\expandafter\def\csname pgfk@/handlers/change default path/.@cmd\endcsname#1\pgfeov{%
  \def\pgfkeysdefaultpath{#1}}%
\pgfkeys{change default path=/m/,n=N,change default path=/u/,v=V}
```

上面的例子能使用键来修改 `\pgfkeysdefaultpath` 的值。下面的例子也是这样，只是形式上简洁一些：

Ne^{V}

```
\pgfkeysdef{/m/n}{#1}
\pgfkeysdef{/u/v}{$\mathrm e^{#1}$}
\pgfkeysdef{/handlers/change default path}{\def\pgfkeysdefaultpath{#1}}%
\pgfkeys{change default path=/m/,n=N,change default path=/u/,v=V}
```

↑ ↑

命令 `\pgfkeys` 执行键时处理的是键值对，键值对是 ⟨*key*⟩=⟨*value*⟩ 或 ⟨*key*⟩ 这种形式。`\pgfkeys` 处理键值对时会执行以下操作：

1. 某个以 ⟨*value*⟩ 为参数的命令会被执行，该命令被储存在 ⟨*key*⟩ 的一个特殊的子键（subkey）中。
2. 将 ⟨*value*⟩ 储存入 ⟨*key*⟩ 中。
3. 如果 ⟨*key*⟩ 的名称（最后一个斜线之后的符号）是手柄（handler），则按该类手柄的规则处理。
4. 如果 ⟨*key*⟩ 是未知的，则调用 unknown key handlers 来给出错误提示。

另外，如果只给出键名称 ⟨*key*⟩ 而没有写出 ⟨*value*⟩, 则程序可能会尝试使用 ⟨*key*⟩ 的默认值。
下面是一些用于执行键的命令，其作用范围受到 TeX 分组的限制。

`\pgfkeys{`⟨*key list*⟩`}`

⟨*key list*⟩ 是用逗号分隔的键值对列表，键值对可以是 ⟨*key*⟩=⟨*value*⟩ 或 ⟨*key*⟩ 的形式。在 ⟨*key*⟩ 和 ⟨*value*⟩ 两侧的空格会被会略，也可以在 ⟨*key*⟩ 或 ⟨*value*⟩ 的两侧使用花括号。如果 ⟨*value*⟩ 中含有逗号、等号则必须用花括号将它括起来。键值对会按照它们出现的次序被依次处理。

在命令中，如果 ⟨*key*⟩=⟨*value*⟩ 中的 ⟨*key*⟩ 只是键名称，那么所设置的 (在当前有效的) 默认前缀路径会被加到 ⟨*key*⟩ 之前 (如手柄 `/.cd` 的作用，这个手柄只在当前命令之内有效)。默认前缀路径保存在命令 `\pgfkeysdefaultpath` 中。本命令结束后，键的默认路径会被重置为本命令之前的状态。

如果 `{`⟨*key list*⟩`}` 中的第一个键是不完整的手柄键，则只会给它添加斜线“/”前缀，除非事先修改 `\pgfkeys@root` 的值。

在 ⟨*value*⟩ 中可以使用命令 `\pgfkeys`, 即在执行 ⟨*key*⟩ 时命令 `\pgfkeys` 可以套嵌使用，而且套嵌时不会搞错“不完整键名称”的默认路径，即命令 `\pgfkeys` 会针对套嵌的情况自动调整所默认的前缀路径。

(a:(b:1))

```
\pgfkeys{/a/.code=(a:#1)}
\pgfkeys{/b/.code=(b:#1)}
\pgfkeys{/a={\pgfkeys{b=1}}}
```

在定义 ⟨*key*⟩ 时命令 `\pgfkeys` 可以套嵌使用，例如

a:, b:1

```
\pgfkeys{/a/.code={a:\pgfkeys{/b/.code={b:#1}}}}
\pgfkeys{/a=1}, \pgfkeys{/b}
```

但是下面的

```
\pgfkeys{/a/.code={a:{\pgfkeys{/b/.code={b:#1}}}}}
\pgfkeys{/a=1}, \pgfkeys{/b}
```

会导致错误，因为 `\pgfkeys{/b/.code={b:#1}}` 的外围多了一层花括号，使得 `/b` 的定义被放入一个组中，在这个组之外使用键 `/b` 是无效的。

⟨*key*⟩=⟨*value*⟩ 中的 ⟨*value*⟩ 可以是字符，在允许的情况下也可以是命令，例如绘图命令。

```
\pgfkeys{/fw/.code={#1}}
\pgfkeys{/fw={\tikz\draw circle(10pt);}}
```

`\pgfqkeys{`⟨*default path*⟩`}{`⟨*key list*⟩`}`

等效于 `\pgfkeys{`⟨*default path*⟩`/.cd,`⟨*key list*⟩`}`, 只是运行速度稍微快些。这个命令一般不用在“用户代码”（user code）中，而是用在 `\tikzset` 或 `\pgfset` 中。

本命令的定义是：

```
\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}}
```

从定义看，本命令首先修改 `\pgfkeysdefaultpath` 的值为 ⟨*default path*⟩, 这种修改的作用要通过命令 `\pgfkeys@add@path@as@needed` 才能体现出来。参数 ⟨*default path*⟩ 应该是“/⟨*some char*⟩”这种

以斜线“/”开头的形式。另外，本命令做的最后一步是把 \pgfkeysdefaultpath 的值恢复到执行本命令之前的状态。

x, /

```
\pgfkeys{/a/b/.initial=x,}
\pgfqkeys{/a}{b}, \pgfkeysdefaultpath
```

\pgfkeysalso{⟨*key list*⟩}

本命令的定义是：

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

本命令类似 \pgfkeys, 不过本命令不会“自动”重置（修改）键的默认路径。也就是说，如果在本命令之前或在本命令的 ⟨*key list*⟩ 中修改了键的默认路径，那么这个修改对下一个 \pgfkeysalso 命令也是有效的。观察下面的例子：

yes
no

```
\pgfkeys{/a/b/c/.code={#1}}
\pgfkeysalso{/a/b/.cd, c=yes} \par
\pgfkeysalso{c=no}
```

但是

```
\pgfkeys{/a/b/c/.code={#1}}
\pgfkeysalso{/a/b/.cd, c=yes}
\pgfkeys{c=no}
```

以及

```
\pgfkeys{/a/b/c/.code={#1}}
\pgfkeys{/a/b/.cd, c=yes}
\pgfkeys{c=no}
```

都是无效的。

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

本命令的定义是：

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

类似 \pgfkeysalso, 只是速度稍快。本命令的参数 ⟨*default path*⟩ 应该是“/⟨*some char*⟩”这种以斜线“/”开头的形式。而且，本命令不会在最后一步改变 \pgfkeysdefaultpath 的值。

87.3.1 首字符句法检测

命令 \pgfkeys 执行 key 时，其参数是键值对，键值对之间用逗号分隔。键值对可以是以某特殊字符开头的字符串（即只有键名称，没有“=⟨*value*⟩”这部分），可以让这个特殊字符引起“首字符句法检测”功能。这个功能的意思是：\pgfkeys 会将某个字符与某个命令联系起来，如果一个参数以某个特殊字符开头，那么该命令就会处理这个参数，并显示处理结果。例如 quotes 库提供的双引号就有这种功能。

使用首字符句法检测功能需要以下 3 点：

1. 开启首字符句法检测功能，使用下一选项：

/handlers/first char syntax=⟨*true or false*⟩ (default `true`, initially `false`)

这个键（选项）决定是否开启首字符句法检测功能，其初始值为 `false`, 默认为 `true`.

2. 选定要使用的特殊字符，用下一选项:

/handlers/first char syntax/⟨*meaning of character*⟩ (no value)

这里的 ⟨*meaning of character*⟩ 必须是 "`the character` ⟨字符⟩" 这种形式，例如，将小于号作为特殊首字符，那么 ⟨*meaning of character*⟩ 就必须是 `the character <`, 其原因在于下一步骤。

3. 可以用命令 `\pgfkeyssetvalue` 或者手柄 `/.initial`, 将某个宏与指定的字符联系起来，这个宏可以是已有的，也可以是稍后再定义的。

```
/handlers/first char syntax/the character ⟨ 字符⟩/.initial=⟨wanted macro⟩
```

这个选项使用了 TEX 的 `\let` 和 `\meaning` 命令。命令 `\let\`⟨*macro*⟩`=`⟨标识符⟩ 的作用是，将 "⟨标识符⟩" 的当前功能赋予宏 `\`⟨*macro*⟩, 例如，将小于号的作用赋予宏 `\mycharacter`, 就是

```
\let\mycharacter=<
```

命令 `\meaning\`⟨*macro*⟩ 的作用是显示 `\`⟨*macro*⟩ 的定义，例如

the character <

```
\let\mycharacter=<
\meaning\mycharacter
```

命令 `\meaning\`⟨*macro*⟩ 的结果就是 "`the character` ⟨字符⟩" 这种形式。

手柄键 `/.initial` 所指向的宏 ⟨*wanted macro*⟩ 就以 `the character` ⟨字符⟩ 开头的参数为参数，该宏所执行的操作是需要自定义的。

下面是使用首字符句法检测功能例子。

Quoted: "foo". Pointed: <bar>.

```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character "/.initial=\myquotemacro,
  /handlers/first char syntax/the character </.initial=\mypointedmacro,
}
\def\myquotemacro#1{Quoted: #1. }
\def\mypointedmacro#1{Pointed: #1. }
\ttfamily \pgfkeys{"foo", <bar>}
```

下面的例子中，单引号起到了给 node 加标签的作用:

'foo'

bar

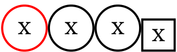
```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character '/.initial=
    \mysinglequotemacro
}
\def\mysinglequotemacro#1{\pgfkeysalso{label={#1}}}
\tikz \node [circle, 'foo', draw] {bar};
```

注意对于上面的例子来说，如果直接把命令 `\pgfkeysalso` 用作 `\node` 的选项:

```
\tikz \node [circle, \pgfkeysalso{label={'foo'}}, draw] {bar};
```

会导致错误：! TeX capacity exceeded, sorry [input stack size=5000].

下面的例子中，用左圆括号引起一个 if 判断:




```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character (/.initial=\myparamacro
}
\def\myparamacro#1{\myparaparser#1\someendtext}
\def\myparaparser(#1)#2\someendtext{
  \pgfmathparse{#1}
  \ifx\pgfmathresult\onetext
    \pgfkeysalso{#2}
  \fi
}
\def\onetext{1}
\foreach \i in {1,...,4}
\tikz \node [draw, thick, rectangle, (pi>\i) circle, (pi>\i*2) draw=red] {x};
```

下面几个小节介绍命令 \pgfkeys{⟨*key list*⟩} 对于 ⟨*key list*⟩ 中的键值对 ⟨*key*⟩=⟨*value*⟩ 或 ⟨*key*⟩ 的处理方式——首先确定 ⟨*key*⟩ 的值，然后针对不同的 ⟨*key*⟩ 分情况处理——所介绍的处理方式为某些预定义的命令或 keys 所利用，也可以为用户利用。首先注意，如果给出的 ⟨*key*⟩ 不是完整的路径，那么就先自动把它完善为完整的路径，所以下面就假定 ⟨*key*⟩ 是完整的路径。

87.3.2 默认参数值

命令 \pgfkeys 执行 key 时，其参数是键值对，键值对可以是 ⟨*key*⟩=⟨*value*⟩ 或 ⟨*key*⟩ 的形式。如果给出 ⟨*key*⟩=⟨*value*⟩, 那么 ⟨*key*⟩ 的值就是 ⟨*value*⟩. 如果只是给出 ⟨*key*⟩, 即 ⟨*key*⟩ 后没有等号 “=”，命令 \pgfkeys 会尝试为该键提供 “默认值”，如果该键有默认值 ⟨*default value*⟩, 则按 ⟨*key*⟩=⟨*default value*⟩ 来执行。键的值总是由宏 \pgfkeyscurrentkey 来保存。

当给出的 ⟨*key*⟩ 后没有等号 “=” 时，执行以下动作：

1. 输入的 ⟨*key*⟩ 被替换为 ⟨*key*⟩=\pgfkeysnovalue, 实际上，写出 \pgfkeys{⟨*key*⟩} 与写出 \pgfkeys{⟨*key*⟩ = \pgfkeysnovalue} 是等效的。
2. 检查名称为 pgfk@\pgfkeyscurrentkey/.@def(即 pgfk@⟨*full key*⟩/.@def) 的命令是否已经被定义。
3. 如果已定义，就把此命令中保存的记号作为 ⟨*key*⟩ 的 ⟨*value*⟩.
4. 如果未定义，则 ⟨*key*⟩ 的值保持为 \pgfkeysnovalue.
5. 如果 ⟨*key*⟩ 的 ⟨*value*⟩ 等于 \pgfkeysvaluerequired，则执行等效于下面命令的代码：

   ```
   \pgfkeys{/errors/value required=⟨key⟩{}}
   ```

   产生一个错误信息。

按照这个处理方式，用户只需直接定义名称为 pgfk@\pgfkeyscurrentkey/.@def 的命令，就可以将此命令所保存的内容作为 ⟨*key*⟩ 的默认值。

手柄 /.default 可以定义键的默认值，手柄 /.default 的定义如下：

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

可见手柄 /.default 会定义名称为 pgfk@\pgfkeyscurrentkey/.@def 的命令。例如，若：

```
\pgfkeys{/a/b/c/.default=ABC}
```

那么 /a/b/c 的默认值就是 ABC, 即 /a/b/c/.@def 的值是 ABC.

87.3.3 方法：定义“执行某个命令的键”并使用之

本小节介绍两种方法，可以定义“执行某个命令的 key”，以及如何使用这个 key. 这两种方法都利用了命令 \pgfkeys@case@one, \pgfkeys@case@two, \pgfkeys@case@three 的处理过程。

第一种方法 稍微复杂一点。

确定了 ⟨*key*⟩ 的值 ⟨*value*⟩ 后就执行 ⟨*key*⟩, 根据 ⟨*key*⟩ 的具体情况，又可以分成不同的执行情况。在各个情况中，宏 \pgfkeyscurrentkey 的展开值都是当前的完整 ⟨*key*⟩. 命令 \pgfkeys 会如下处理：

首先检查名称为 pgfk@\pgfkeyscurrentkey/.@cmd(即 pgfk@⟨*full key*⟩/.@cmd) 的命令是否已经被定义。如果已定义，则执行

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

其中 \pgfeov 的意思是 “end of value” ; 其中的 ⟨*value*⟩ 不用花括号括起来。执行完这个命令后，\pgfkeys 继续处理 ⟨*key list*⟩ 中的其它键。下面的例子是对以上处理过程的演示：

\a is hello, \b is world.

```
\def\mystore#1+#2\pgfeov{\def\a{#1}\def\b{#2}}
\pgfkeyslet{/my key/.@cmd}{\mystore}
\pgfkeys{/my key=hello+world}
\string\a\ is \a, \string\b\ is \b.
```

首先注意，上面例子中使用了符号 @, 但没有使用命令 \makeatletter 和 \makeatother. 上面例子中，\pgfkeys{/my key=hello+world} 会导致 \mystore hello+world\pgfeov 被执行。这个例子演示了第一种方法，这个方法的步骤是：

1. 首先要用命令 \def 定义一个宏 \⟨*macro*⟩, 并用这个宏保存所需要执行的代码，代码中可以使用变量。在该定义的变量列举之后要加上命令 \pgfeov, 这个命令提示变量列举完毕。例如

   ```
   \def\mystore(#1,#2)\pgfeov{\def\a{#1}\def\b{#2}}
   ```

2. 用命令 \pgfkeyslet 使得名称为 pgfk@\pgfkeyscurrentkey/.@cmd 的命令指向宏 \⟨*macro*⟩.

   ```
   \pgfkeyslet{/my key/.@cmd}{\mystore}
   ```

3. 执行 \pgfkeys{⟨*key*⟩=⟨*value*⟩}, 完成赋值并执行。

   ```
   \pgfkeys{/my key={(hello,world)}}
   ```

在这个方法中，⟨*key*⟩=⟨*value*⟩ 的 ⟨*value*⟩ 的形式就是 TeX 定义宏时的变量列举格式：

```
\def\⟨macro⟩⟨ 变量列举格式⟩{...}
```

关于变量列举格式的规则参考《The TeXbook》的第 20 章。

注意，在命令 \def 中列举变量时可以使用逗号作为变量之间的定界标志，但是在命令 \pgfkeys 的处理过程中，逗号用于分隔相邻的的两个 key-value, 具有特殊作用。因此，如果在 \def 的变量列举格式中使用逗号作为变量的定界标志，那么在命令 \pgfkeys 中，相应的 ⟨*key*⟩=⟨*value*⟩ 的整个 ⟨*value*⟩ 要用花括号括起来，这个办法依据的是内部命令 \pgfkeys@unpack 的定义。

hello world!

```
\def\mystore(#1,#2)\pgfeov{\def\a{#1} \def\b{#2} \a \ \b}
\pgfkeyslet{/my key/.@cmd}{\mystore}
\pgfkeys{/my key={(hello,world!)}}
```

第二种方法 稍微简捷一些，利用下面的命令。

`\pgfkeysdef{⟨full key⟩}{⟨code⟩}`

本命令先定义宏 `\pgfkeys@temp#1\pgfeov`, 其中的 `\pgfeov` 是命令 `\pgfkeys@temp` 的参数定界标志,此命令的定义内容是 ⟨*code*⟩. 然后,本命令使用 `\let` 使得 `\csname pgfk@⟨full key⟩/.@cmd\endcsname` 等于 `\pgfkeys@temp`.

使用本命令后, 执行 `\pgfkeys{⟨key⟩=⟨value⟩}` 会导致 ⟨*code*⟩ 被执行, 而且执行 ⟨*code*⟩ 时, 还以 ⟨*value*⟩ 替换 ⟨*code*⟩ 中的变量 `#1`. 命令 `\csname pgfk@⟨full key⟩/.@cmd\endcsname` 在执行 `\pgfkeys@case@one` 时发挥作用。

hello, hi!

```
\pgfkeysdef{/my key}{hello, hi!}
\pgfkeys{/my key}
```

hello, hello.

```
\pgfkeysdef{/my key}{#1, #1.}
\pgfkeys{/my key=hello}
```

这个命令可以套嵌使用，即可以在 ⟨*code*⟩ 中使用该命令，例如：

hello，world

```
\pgfkeysdef{/my key}{#1, \pgfkeysdef{/aaaa}{##1}}
\pgfkeys{/my key=hello} \pgfkeys{/aaaa=world}
```

上面例子中, 在执行键 `/my key=hello` 时会顺带执行 `\pgfkeysdef{/aaaa}{#1}`, 参考 `\pgfkeysdef` 的定义。

`\pgfkeysedef{⟨full key⟩}{⟨code⟩}`

这个命令类似 `\pgfkeysdef`, 只不过本命令使用 `\edef` 来定义 `\pgfkeys@temp`.

`\pgfkeysdefnargs{⟨full key⟩}{⟨argument count⟩}{⟨code⟩}`

这个命令类似 `\pgfkeysdef`, 不过本命令允许在 ⟨*code*⟩ 中使用多个变量，至多 9 个，变量个数要在 ⟨*argument count*⟩ 中声明。在本命令之后，执行`\pgfkeys{⟨key⟩=⟨value⟩}` 会导致 ⟨*code*⟩ 被执行，并且在执行时 ⟨*value*⟩ 中列举的参数会替换 ⟨*code*⟩ 中的相应变量。

\a is hello, \b is world.

```
\pgfkeysdefnargs{/my key}{2}{\def\a{#1}\def\b{#2}}
\pgfkeys{/my key={hello}{world}}
\string\a \ is \a, \string\b \ is \b.
```

`\pgfkeysedefnargs{⟨full key⟩}{⟨argument count⟩}{⟨code⟩}`

这是 `\pgfkeysdefnargs` 的 `\edef` 版本。

`\pgfkeysdefargs{⟨full key⟩}{⟨argument pattern⟩}{⟨code⟩}`

这个命令类似 `\pgfkeysdefnargs`, 只是 ⟨*argument pattern*⟩ 是变量列举格式。

注意：(1) 在 ⟨*argument pattern*⟩ 的末尾不能使用 `\pgfeov`, 否则出错；(2) 在 ⟨*argument pattern*⟩ 中最好不要出现逗号；(3) 在本命令之后执行 `\pgfkeys{⟨key⟩=⟨value⟩}` 时，⟨*value*⟩ 的格式要与 ⟨*argument pattern*⟩ 的格式一致。

\a is hello, \b is world.

```
\pgfkeysdefargs{/my key}{#1+#2}{\def\a{#1}\def\b{#2}}
\pgfkeys{/my key=hello+world}
\string\a\ is \a, \string\b\ is \b.
```

`\pgfkeysedefargs{⟨full key⟩}{⟨argument pattern⟩}{⟨code⟩}`

这是 \pgfkeysdefargs 的 \edef 版本。

↓ ↓

命令 \pgfkeysdef 的定义 此命令的定义是:

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

所以 \pgfkeysdef{⟨*full key*⟩}{⟨*code*⟩} 就是

```
\long\def\pgfkeys@temp#1\pgfeov{⟨code⟩}%
\pgfkeyslet{⟨full key⟩/.@cmd}{\pgfkeys@temp}%
\pgfkeyssetvalue{⟨full key⟩/.@body}{⟨code⟩}%
```

本命令:

1. 用 \def 定义命令 \pgfkeys@temp, 其定义内容是 ⟨*code*⟩, 以 \pgfeov 为参数定界标志。
2. 规定 \let\csname pgfk@⟨*full key*⟩/.@cmd\endcsname\pgfkeys@temp, 于是

```
\csname pgfk@⟨full key⟩/.@cmd\endcaname#1\pgfeov
等效于
\pgfkeys@temp#1\pgfeov
即执行
⟨code⟩
```

3. 用 \edef 把 \the\pgfkeys@temptoks{⟨*code*⟩} 保存到
   \csname pgfk@⟨*full key*⟩/.@body\endcsname 中

从定义看, ⟨*code*⟩ 中最多只能使用一个变量 #1, 因为定义的命令格式 \pgfkeys@temp 只有一个参数。在执行键值对 \pgfkeys{⟨*key-value*⟩} 的具体过程中, \pgfkeys@temp 之后的变量 #1, 以及 ⟨*code*⟩ 中的 #1 都会被替换为键值 ⟨*value*⟩(保存在 \pgfkeyscurrentvalue 中)。

g(s)

```
\pgfkeysdef{/a}{#1}
\pgfkeys{/a=g(s)}
```

上面例子的第一行看作是一个映射 F: /a $\to$ $f_{/\mathtt{a}}(\#1)$, 其中的 $f_{/\mathtt{a}}(\#1)$ 看作是以 #1 为变量的函数, 而且 $f_{/\mathtt{a}}(\#1) = \#1$ 是恒等函数。第二行的作用是计算 $f_{/\mathtt{a}}(\#1) = \#1$ 在 #1=g(s) 处的值, 得到 g(s).

$g(h(s))$

```
\pgfkeysdef{/a}{\pgfkeysdef{/b}{#1}}
\pgfkeys{/a=g(#1)}
$\pgfkeys{/b=h(s)}$
```

上面例子:

1. 第一行得到函数 $f_{/\mathtt{a}}(\#1)$ =\pgfkeysdef{/b}{#1}.

2. 第二行得到函数 $f_{/\mathtt{a}}(g(\#1))$ =\pgfkeysdef{/b}{g(\#1)}= $f_{/\mathtt{b}}(\#1) = g(\#1)$.
3. 第三行得到 $g(h(s))$.

可以把键 /a 和 /b 看作是一种“标签”，用这种标签可以找到相应的函数，并执行函数计算。

另外注意，命令 \pgfkeysdef 对 \pgfkeys 的影响出现在 \pgfkeys@case@one 那里。

命令 \pgfkeysedef 的定义 此命令的定义是:

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

它用 \edef 来定义 \pgfkeys@temp.

命令 \pgfkeysdefargs 的定义 此命令的定义是:

```
\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}%
}
```

所以 \pgfkeysdefargs{⟨*full key*⟩}{⟨*argument pattern*⟩}{⟨*code*⟩} 就是

```
\long\def\pgfkeysdefargs ⟨full key⟩⟨argument pattern⟩⟨code⟩{%
  \long\def\pgfkeys@temp ⟨argument pattern⟩\pgfeov{⟨code⟩}%
  \pgfkeyslet{⟨full key⟩/.@cmd}{\pgfkeys@temp}%
  \pgfkeyssetvalue{⟨full key⟩/.@args}{⟨argument pattern⟩\pgfeov}%
  \pgfkeyssetvalue{⟨full key⟩/.@body}{⟨code⟩}%
}
```

本命令:

1. 用 \def 定义命令 \pgfkeys@temp, 其定义内容是 ⟨*code*⟩.
2. 规定 \let\csname pgfk@⟨*full key*⟩/.@cmd\endcsname\pgfkeys@temp, 于是

   ```
   \csname pgfk@⟨full key⟩/.@cmd\endcaname#1\pgfeov
   等效于
   \pgfkeys@temp#1\pgfeov
   ```

3. 用 \edef 把 \the\pgfkeys@temptoks{⟨*argument pattern*⟩\pgfeov} 保存到 \csname pgfk@⟨*full key*⟩/.@args\endcsname 中
4. 用 \edef 把 \the\pgfkeys@temptoks{⟨*code*⟩} 保存到 \csname pgfk@⟨*full key*⟩/.@body\endcsname 中

文件中给出下面的例子:

1=‘1’, 2=‘ 2’

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

注意输出结果中“2=‘ 2’”里面的空格，对比下面的：

1=‘1’, 2=‘2’

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

这个例子的输出中没有这个空格。其中的原因有两个，第一个原因见下面的例子：

ab

a b

```
\def\aaaa#1#2{#1#2}
\aaaa{a} {b}\par
\def\aaaa#1#2\bbbb{#1#2}
\aaaa{a} {b}\bbbb
```

可见使用 `\bbbb` 作为参数定界标志时不会忽略参数之间的空格，这是 TEX 自己的规则。第二个原因在于定义命令 `\pgfkeys@spdef\pgfkeyscurrentvalue{#2}`, 这样定义的 `\pgfkeyscurrentvalue` 保存一串记号，其中可以有空格。

命令 `\pgfkeysedefargs` 的定义 此命令的定义是：

```
\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}%
}
```

命令 `\pgfkeysdefnargs` 与 `\pgfkeysedefnargs` 的定义

```
\long\def\pgfkeysdefnargs#1#2#3{\pgfkeysdefnargs@{#1}{#2}{#3}{\def}{\pgfkeyssetvalue
↪  }}%
\long\def\pgfkeysedefnargs#1#2#3{\pgfkeysdefnargs@{#1}{#2}{#3}{\edef}{
↪  \pgfkeyssetevalue}}%
\long\def\pgfkeysdefnargs@#1#2#3#4#5{%
  \ifcase#2\relax
    \pgfkeyssetvalue{#1/.@args}{}%
  \or
    \pgfkeyssetvalue{#1/.@args}{##1}%
  \or
    \pgfkeyssetvalue{#1/.@args}{##1##2}%
  \or
    \pgfkeyssetvalue{#1/.@args}{##1##2##3}%
  \or
    \pgfkeyssetvalue{#1/.@args}{##1##2##3##4}%
  \or
    \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5}%
  \or
```

```
    \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6}%
  \or
    \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7}%
  \or
    \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8}%
  \or
    \pgfkeyssetvalue{#1/.@args}{##1##2##3##4##5##6##7##8##9}%
  \else
    \pgfkeys@error{\string\pgfkeysdefnargs: expected  <= 9 arguments, got #2}%
  \fi
  \pgfkeysgetvalue{#1/.@args}\pgfkeys@tempargs
  \def\pgfkeys@temp{\expandafter\long\expandafter#4\csname pgfk@#1/.@@body\endcsname
  ↪  }%
  \expandafter\pgfkeys@temp\pgfkeys@tempargs{#3}%
  % eliminate the \pgfeov at the end such that TeX gobbles spaces
  % by using
  % \pgfkeysdef{#1}{\pgfkeysvalueof{#1/.@@body}##1}
  % (with expansion of '#1'):
  \edef\pgfkeys@tempargs{\noexpand\pgfkeysvalueof{#1/.@@body}}%
  \def\pgfkeys@temp{\pgfkeysdef{#1}}%
  \expandafter\pgfkeys@temp\expandafter{\pgfkeys@tempargs##1}%
  #5{#1/.@body}{#3}%
}
```

所以 `\pgfkeysdefnargs{⟨full key⟩}{3}{⟨code⟩}` 得到:

```
\pgfkeysdefnargs@{⟨full key⟩}{3}{⟨code⟩}{\def}{\pgfkeyssetvalue}
导致
\pgfkeyssetvalue{⟨full key⟩/.@args}{#1#2#3}%
\pgfkeysgetvalue{⟨full key⟩/.@args}\pgfkeys@tempargs
\def\pgfkeys@temp{\expandafter\long\expandafter\def\csname pgfk@⟨full key⟩/.@@body
↪  \endcsname}%
\expandafter\pgfkeys@temp\pgfkeys@tempargs{⟨code⟩}%
\edef\pgfkeys@tempargs{\noexpand\pgfkeysvalueof{⟨full key⟩/.@@body}}%
\def\pgfkeys@temp{\pgfkeysdef{⟨full key⟩}}%
\expandafter\pgfkeys@temp\expandafter{\pgfkeys@tempargs#1}%
\pgfkeyssetvalue{⟨full key⟩/.@body}{⟨code⟩}%
```

即

1. 用 `\edef` 把 `\the\pgfkeys@temptoks{#1#2#3}` 保存到 `\csname pgfk@⟨full key⟩/.@args\endcsname` 中
2. 用 `\let` 使得 `\pgfkeys@tempargs` 等于 `\csname pgfk@⟨full key⟩/.@args\endcsname`, 所以执行 `\pgfkeys@tempargs` 就得到“`#1#2#3`”.
3. 定义 `\pgfkeys@temp`, 使之能引起对 `\csname pgfk@⟨full key⟩/.@@body\endcsname` 的定义。
4. 然后定义 `\csname pgfk@⟨full key⟩/.@@body\endcsname`, 此命令有 3 个参数

   ```
   \expandafter\long\expandafter\def\csname pgfk@⟨full key⟩/.@@body\endcsname#1#2#3{
   ↪  ⟨code⟩}
   ```

5. 然后重定义 `\pgfkeys@tempargs`.
6. 然后重定义 `\pgfkeys@temp`, 这是为了方便在下一步使用 `\expandafter`.

7. 然后

```
\pgfkeysdef{⟨full key⟩}{\csname pgfk@⟨full key⟩/.@@body\endcsname#1}
```

这是处理过程的“核心”步骤。当执行 \pgfkeys{⟨full key⟩={⟨value1⟩}{⟨value2⟩}{⟨value3⟩}} 时，上面的 #1 会被替换为“{⟨value1⟩} {⟨value2⟩} {⟨value3⟩}”.

8. 用 \edef 把 \the\pgfkeys@temptoks{⟨code⟩} 保存到 \csname pgfk@⟨full key⟩/.@body\endcsname 中。

文件中给出下面的例子：

1=‘1’, 2=‘2’

```
\pgfkeysdefnargs{/a}{2}{1=`#1', 2=`#2'}
\pgfkeys{
  /a=
    {1}
    {2}
}
```

上面例子的输出中没有多余的空格，这与命令 \pgfkeysdefargs 不同。原因在于执行 \pgfkeys 会导致命令 \csname pgfk@⟨full key⟩/.@@body\endcsname#1 被执行，而其中的 #1 会被(保存在 \pgfkeyscurrentvalue 中的）键值替换，此命令没有参数定界标志，所以 \pgfkeyscurrentvalue 中的不必要空格会被忽略。

小结

定义 (重定义) \csname pgfk@\pgfkeyscurrentkey\endcsname 的命令有：

- \pgfkeyssetvalue
- \pgfkeyssetevalue
- \pgfkeysaddvalue（重定义）
- \pgfkeyslet

定义 \csname pgfk@\pgfkeyscurrentkey/.@cmd\endcsname 的命令有：

- \pgfkeysdef
- \pgfkeysedef
- \pgfkeysdefargs
- \pgfkeysedefargs

定义 \csname pgfk@\pgfkeyscurrentkey/.@body\endcsname 的命令有：

- \pgfkeysdef
- \pgfkeysedef
- \pgfkeysdefargs
- \pgfkeysedefargs
- \pgfkeysdefnargs

- \pgfkeysedefnargs

定义 \csname pgfk@\pgfkeyscurrentkey/.@args\endcsname 的命令有:

- \pgfkeysdefargs
- \pgfkeysedefargs
- \pgfkeysdefnargs
- \pgfkeysedefnargs

定义 \csname pgfk@\pgfkeyscurrentkey/.@@body\endcsname 的命令有:

- \pgfkeysdefnargs
- \pgfkeysedefnargs

定义 \csname pgfk@excpt@#1\endcsname 的命令有:

- \pgfkeysaddhandleonlyexistingexception

```
\def\pgfkeysaddhandleonlyexistingexception#1{\expandafter\def\csname
↪   pgfk@excpt@#1\endcsname{1}}%
```

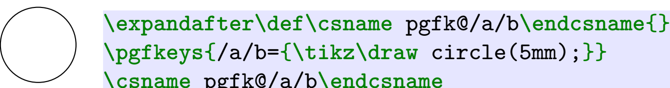
↑ ↑

87.3.4 Keys That Store Values

本小节内容对应命令 \pgfkeys@case@two 的处理过程,此时名称为 pgfk@\pgfkeyscurrentkey/.@cmd 的命令未被定义，而且会检查名称为 pgfk@\pgfkeyscurrentkey 的命令是否已经被定义。如果已经定义了此命令，那么就重定义此命令:

```
\expandafter\let\csname pgfk@\pgfkeyscurrentkey\endcsname\pgfkeyscurrentvalue%
```

例如



```
\expandafter\def\csname pgfk@/a/b\endcsname{}
\pgfkeys{/a/b={\tikz\draw circle(5mm);}}
\csname pgfk@/a/b\endcsname
```

87.3.5 定义手柄键

本小节内容对应命令 \pgfkeys@case@three 的处理过程,此时名称为 pgfk@\pgfkeyscurrentkey/.@cmd 的命令，名称为 pgfk@\pgfkeyscurrentkey 的命令都是未被定义的。\pgfkeyscurrentname 保存当前键的名称，\pgfkeyscurrentpath 保存当前键的前缀路径，\pgfkeyscurrentkey 保存完整的键名称。

检查名称为 pgfk@/handlers/\pgfkeyscurrentname/.@cmd 的命令是否已经被定义。如果已定义，则执行

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

观察下面的例子：

macro:->##1^##1

e^{2^2}

```
\pgfkeysdef{/handlers/.my code}{\pgfkeysdef{\pgfkeyscurrentpath}{$\mathrm
↪   e^{#1}$}}
\pgfkeys{/my key/.my code={#1^#1}}
\texttt{\meaning\pgfkeyscurrentvalue}\par
\pgfkeys{/my key={2}}
```

这个例子的处理过程是：

1. 第一行得到如下定义

```
\long\def\pgfkeys@temp#1\pgfeov{\pgfkeysdef{\pgfkeyscurrentpath}{$\mathrm e^{#1
↪   }$}}%
\pgfkeyslet{/handlers/.my code/.@cmd}{\pgfkeys@temp}%
\pgfkeyssetvalue{/handlers/.my code/.@body}{\pgfkeysdef{\pgfkeyscurrentpath}{$
↪   \mathrm e^{#1}$}}%
```

2. 第二行就是

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

这导致执行

```
\pgfkeys@temp\pgfkeyscurrentvalue\pgfeov
```

注意此时 `\pgfkeyscurrentvalue` 的值是等号后的记号序列“`#1^#1`”, 它成为 `\pgfkeys@code` 的参数，也会用来替换 `{$\mathrm e^{#1}$}` 中的“`#1`”, 导致

```
\pgfkeysdef{/my key}{$\mathrm e^{#1^#1}$}
```

又得到如下定义

```
\long\def\pgfkeys@temp#1\pgfeov{$\mathrm e^{#1^#1}$}%
\pgfkeyslet{/my key/.@cmd}{\pgfkeys@temp}%
\pgfkeyssetvalue{/my key/.@body}{$\mathrm e^{#1^#1}$}%
```

`/my key/.my code=` 的等号之后花括号里的代码就是它的键值，注意在这个键值中最多使用一个变量“`#1`”, 因为命令 `\pgfkeysdef` 在定义 `\pgfkeys@temp` 时，只为它规定了一个参数。

3. 第三行，显示当前 `\pgfkeyscurrentvalue` 的值。
4. 第四行，因为第三行已经定义了 `\csname pgfk@/my key/.@cmd\endcsname`, 所以这一行命令的处理过程主要在 `\pgfkeys@case@one` 那里进行，即执行

```
\pgfkeysgetvalue{/my key/.@cmd}{\pgfkeys@code}%
\expandafter\pgfkeys@code2\pgfeov
```

也就是执行 `$\mathrm e^{2^2}$`.

约定：当定义命令 `\csname pgfk@/handlers/`⟨*name*⟩`/.@cmd\endcsname` 后，就把 ⟨*name*⟩ 叫做“手柄 (handler)”，以 ⟨*name*⟩ 为名称的键叫做“键手柄 (key handler)”. 如上面例子中的 `.my code` 是手柄，为了强调它的特性，也写成 `/.my code`. 上面例子中的 `/my key/.my code` 是“手柄键”。

总结一下命令 `\pgfkeys@case@one`, `\pgfkeys@case@two`, `\pgfkeys@case@three` 的处理过程：

- 检查名称为 `pgfk@\pgfkeyscurrentkey/.@cmd` 的命令是否已经定义，若未定义
- 检查名称为 `pgfk@\pgfkeyscurrentkey` 的命令是否已经定义，若未定义
- 检查名称为 `pgfk@/handlers/\pgfkeyscurrentname/.@cmd` 的命令是否已经定义

这个过程会检查各种条件，根据检查结果做相应的处理。这些检查条件的次序是可以改变的，这用到下面的选项：

`/handler config=all|only existing|full or existing` (no default, initially `all`)

all 这是初始值，初始行为如前述。

only existing 此选项导致如下处理：

第一：首先执行 `\pgfkeys@split@path`，得到 `\pgfkeyscurrentname` 和 `\pgfkeyscurrentpath`.

第二：然后检查名称为 `pgfk@/handler/\pgfkeyscurrentname/.@cmd` 的命令是否已经定义：

- 若已定义，则检查名称为 `pgfk@excpt@\pgfkeyscurrentname` 的命令是否已经定义
  - 若已定义，则执行命令 `\pgfkeys@ifexecutehandler` 的参数 `#1`.
  - 若未定义，则检查名称为 `pgfk@\pgfkeyscurrentpath` 的命令是否已经定义
    - 若已定义，则检查名称为 `pgfk@\pgfkeyscurrentpath/.@cmd` 的命令是否已经定义
      - 若已定义，则执行命令 `\pgfkeys@ifexecutehandler` 的参数 `#1`.
      - 若未定义，则执行命令 `\pgfkeys@ifexecutehandler` 的参数 `#2`.
    - 若未定义，则什么也不做。
- 若未定义，则执行 `\pgfkeys@unknown`.

full or existing 此选项导致如下处理：

第一：首先执行 `\pgfkeys@split@path`，得到 `\pgfkeyscurrentname` 和 `\pgfkeyscurrentpath`.

第二：然后检查名称为 `pgfk@/handler/\pgfkeyscurrentname/.@cmd` 的命令是否已经定义：

- 若已定义，则检查 `\ifpgfkeysaddeddefaultpath` 的真值
  - 若有 `\pgfkeysaddeddefaultpathtrue`，则检查名称为 `pgfk@excpt@\pgfkeyscurrentname` 的命令是否已经定义
    - 若已定义，则执行命令 `\pgfkeys@ifexecutehandler` 的参数 `#1`.
    - 若未定义，则检查名称为 `pgfk@\pgfkeyscurrentpath` 的命令是否已经定义
      - 若已定义，则执行命令 `\pgfkeys@ifexecutehandler` 的参数 `#1`.
      - 若未定义，则检查名称为 `pgfk@\pgfkeyscurrentpath/.@cmd` 的命令是否已经定义 (a) 若已定义，则执行命令 `\pgfkeys@ifexecutehandler` 的参数 `#1`.
        (b) 若未定义，则执行命令 `\pgfkeys@ifexecutehandler` 的参数 `#2`.
  - 若有 `\pgfkeysaddeddefaultpathfalse`，则执行命令 `\pgfkeys@ifexecutehandler` 的参数 `#1`.

`/handler config/only existing/add exception={⟨key handler name⟩}` (no default)

这个选项设置的 ⟨*key handler name*⟩ 不受 `/handler config=only existing` 的影响。在初始之下，手柄 `/.cd`, `/.try`, `/.retry`, `/.lastretry`, `/.unknown`, `/.expand once`, `/.expand two once`, `/.expanded` 是不受影响的。

↓ ↓

命令 \pgfkeys@case@three@handleall 此命令的定义紧接在 \pgfkeys@case@three 的定义之后:

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

命令 \pgfkeys@ifexecutehandler

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

命令 \pgfkeys@ifexecutehandler@handleall

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

命令 \pgfkeys@case@three@handle@restricted 与 \pgfkeys@ifexecutehandler

1. 首先执行 \pgfkeys@split@path, 定义 \pgfkeyscurrentname 来保存当前键的名称; 定义 \pgfkeyscurrentpath 来保存当前键的前缀路径。
2. 执行命令 \pgfkeysifdefined 来检查名称为 pgfk@/handlers/\pgfkeyscurrentname/.@cmd 的命令是否已经被定义。
   - 如果已定义，则执行命令 \pgfkeys@ifexecutehandler{⟨*code1*⟩}{⟨*code2*⟩}. 按此命令最初的定义，它会导致其参数 ⟨*code1*⟩ 被执行，即执行

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

     此命令的第二个参数 ⟨*code2*⟩ 没有被执行，不过 ⟨*code2*⟩ 比较有意思:
     (a) 例如, 假设当前的完整的键是 /x/y/a/b/c, 参数 ⟨*code2*⟩ 重定义 \pgfkeyscurrentkey 来保存 /x/y/a/b/c; 重定义 \pgfkeyscurrentname 来保存 b/c; 重定义 \pgfkeyscurrentpath 来保存 /x/y/a.
     (b) 执行 \pgfkeys@unknown.
   - 如果未定义，则执行 \pgfkeys@unknown.

   其中的命令 \pgfkeys@ifexecutehandler 可以被重定义。

命令 \pgfkeys@ifexecutehandler@handleonlyexisting 此命令的定义是:

```
\def\pgfkeys@ifexecutehandler@handleonlyexisting#1#2{%
  \pgfkeys@ifcsname pgfk@excpt@\pgfkeyscurrentname\endcsname%
     #1%
  \else
     % implement the 'only existing' feature here:
     \pgfkeysifdefined{\pgfkeyscurrentpath}{#1}{%
       \pgfkeysifdefined{\pgfkeyscurrentpath/.@cmd}{#1}{#2}%
     }{}%
  \fi%
}%
```

此命令有两个参数 #1 和 #2, 它执行一个条件分支语句, 检查名称为 pgfk@excpt@\pgfkeyscurrentname 的命令是否已经被定义:

- 如果已定义，则执行第一个参数 #1.
- 如果未定义，则执行命令 \pgfkeysifdefined 来检查名称为 pgfk@\pgfkeyscurrentpath 的命令是否已经被定义:
  - 如果已定义,则执行命令 \pgfkeysifdefined 来检查名称为 pgfk@\pgfkeyscurrentpath/.@cmd 的命令是否已经被定义:
    - 如果已定义，则执行第一个参数 #1.
    - 如果未定义，则执行第二个参数 #2.
  - 如果未定义，则什么也不做。

命令 \pgfkeys@ifexecutehandler@handlefullorexisting 此命令的定义是:

```
\def\pgfkeys@ifexecutehandler@handlefullorexisting#1#2{%
  \ifpgfkeysaddeddefaultpath
    \pgfkeys@ifcsname pgfk@excpt@\pgfkeyscurrentname\endcsname%
       #1%
    \else
       % implement the 'only existing' feature here:
       \pgfkeysifdefined{\pgfkeyscurrentpath}{%
         #1%
       }{%
         \pgfkeysifdefined{\pgfkeyscurrentpath/.@cmd}{%
           #1%
         }{%
           #2%
         }%
       }%
    \fi%
  \else
    #1% ok, always true if the USER explicitly provided the full key path.
  \fi
}%
```

此命令有两个参数 #1 和 #2, 它用 \ifpgfkeysaddeddefaultpath 执行一个条件分支语句:
(参考命令 \pgfkeys@add@path@as@needed)

- 如果有 \pgfkeysaddeddefaultpathtrue, 则检查名称为 pgfk@excpt@\pgfkeyscurrentname 的命令是否已定义:
  - 如果已定义，则执行第一个参数 #1.
  - 如果未定义，则执行命令 \pgfkeysifdefined 来检查名称为 pgfk@\pgfkeyscurrentpath 的命令是否已经被定义:
    - 如果已定义，则执行第一个参数 #1.

    * 如果未定义,则执行命令 \pgfkeysifdefined 来检查名称为 pgfk@\pgfkeyscurrentpath/.@cmd 的命令是否已经被定义:
        · 如果已定义，则执行第一个参数 #1.
        · 如果未定义，则执行第二个参数 #2.

- 如果有 \pgfkeysaddeddefaultpathfalse, 则执行第一个参数 #1.

命令 \pgfkeysaddhandleonlyexistingexception 此命令的定义是:

```
\def\pgfkeysaddhandleonlyexistingexception#1{\expandafter\def\csname pgfk@excpt@#1
↪  \endcsname{1}}%
```

此命令引起对命令 \csname pgfk@excpt@#1\endcsname 的定义。

选项 /handler config/... 有下面的定义:

```
\pgfkeys{
  /handler config/.is choice,
  /handler config/all/.code={%
    \let\pgfkeys@case@three=\pgfkeys@case@three@handleall
    \let\pgfkeys@ifexecutehandler=\pgfkeys@ifexecutehandler@handleall
  },
  /handler config/only existing/.code={%
    \let\pgfkeys@case@three=\pgfkeys@case@three@handle@restricted
    \let\pgfkeys@ifexecutehandler=\pgfkeys@ifexecutehandler@handleonlyexisting
  },
  /handler config/full or existing/.code={%
    \let\pgfkeys@case@three=\pgfkeys@case@three@handle@restricted
    \let\pgfkeys@ifexecutehandler=\pgfkeys@ifexecutehandler@handlefullorexisting
  },
  /handler config/only existing/add exception/.code={
  ↪  \pgfkeysaddhandleonlyexistingexception{#1}},
}%
```

↑ ↑

87.3.6 设置未知键的提示信息

当执行 \pgfkeys{⟨*key-value*⟩} 时，如果
- 名称为 pgfk@\pgfkeyscurrentkey/.@cmd 的命令
- 名称为 pgfk@\pgfkeyscurrentkey 的命令
- 名称为 pgfk@/handlers/\pgfkeyscurrentname/.@cmd 的命令

都不存在,就执行 \pgfkeys@unknown(见 \pgfkeys@case@three). 例如,假设在此处对 /tikz/something strange 执行 \pgfkeys@unknown, 此时检查名称为 pgfk@/tikz/.unknown/.@cmd 的命令是否已定义:

- 如果已定义，则执行

```
\pgfkeysgetvalue{/tikz/.unknown/.@cmd}{\pgfkeys@code}%
\expandafter\pgfkeys@code\pgfkeyscurrentvalue\pgfeov
```

这需要提前定义键 /tikz/.unknown, 在文件《tikz.code.tex》中有此定义。

- 如果未定义，则执行

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

这需要提前定义键 /handlers/.unknown, 这个键的定义是:

```
\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}%
  }%
}
```

也就是说，实际上执行键 /errors/unknown key→P. 573 所保存的代码。

例如:

```
\pgfkeys{/my test path/.unknown/.code=
  {%
    \edef\aaa{发现 \pgfkeyscurrentkey 未知}
    \expandafter\errmessage\expandafter{\aaa}
  }%
}
\pgfkeys{/my test path/.cd,foo}
```

上面代码就导致编译中断，给出错误提示：! 发现 /my search path/foo 未知.

下面的例子利用以上机制做了点“其它工作”:

/b/c/foo

```
\pgfkeys{/b/c/foo/.code={\pgfkeyscurrentkey}}
\pgfkeys{/my search path/.unknown/.code=
  {%
    \let\searchname=\pgfkeyscurrentname%
    \pgfkeysalso{%
      /a/\searchname/.try=#1,
      /b/\searchname/.retry=#1,
      /b/c/\searchname/.retry=#1%
    }%
  }%
}
\pgfkeys{/my search path/.cd,foo}
```

当 /my search path/foo 导致执行 \pgfkeys@unknown 时，以上代码并不是给出错误信息，而是尝试执行 /a/foo, /b/foo, /b/c/foo, 会导致输出“\pgfkeyscurrentkey”的值。这种搜索机制参考手柄 /.search also.

87.3.7 搜索键的前缀路径，搜索手柄

大意如上一小节。

87.4 手柄

下面讲解预定义的手柄。

87.4.1 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/}
```

87.4.2 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⟩}
```

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

Key handler ⟨*key*⟩`/.value required`

此手柄关联选项 `/erros/value required`. 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 `/erros/value forbidden` 被执行，产生一个错误提示。

此手柄的定义是:

```
\pgfkeys{/handlers/.value forbidden/.code=\pgfkeys{\pgfkeyscurrentpath/.add
↪  code=%
{%
  \ifx\pgfkeyscurrentvalue\pgfkeysnovalue@text%
  \else%
    \def\pgf@marshal{\pgfkeysvalueof{/errors/value forbidden/.@cmd}}%
    \expandafter\expandafter\expandafter\pgf@marshal\expandafter\expandafter
    ↪  \expandafter{\expandafter\pgfkeyscurrentkey\expandafter}\expandafter{
    ↪  \pgfkeyscurrentvalue}\pgfeov%
  \fi%
}{}}}
```

87.4.3 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}}
```

这个定义得到如下定义

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

这个手柄的作用已经在前面有所分析，见 \pgfkeysdef. 简单地说，这个手柄确保以下两个步骤可行:

1. 使用 \pgfkeys{/my key/.code={⟨*code*⟩}} 将代码 ⟨*code*⟩ 与 /my key 关联起来，即导致执行

   ```
   \pgfkeysdef{/my key}{⟨code⟩}
   ```

   在 ⟨*code*⟩ 中最多能用一个变量 #1.

2. 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.

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}}
```

这得到如下定义:

```
\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@temp{⟨code with ⟨value1⟩ and ⟨value2⟩⟩}\pgfeov
```

导致

```
\pgfkeysdefargs{/my key}{#1#2}{{⟨code with ⟨value1⟩ and ⟨value2⟩⟩}}
```

这表明：执行

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

会导致执行

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

使用手柄有时会方便一些。

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}}
```

导致

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

这导致两个主要定义:

```
\expandafter\long\expandafter\def\csname pgfk@/handlers/.code args/.@@body
↪  \endcsname#1#2{\pgfkeysdefargs{\pgfkeyscurrentpath}{#1}{#2}}
以及
\pgfkeysdef{/handlers/.code args}{\csname pgfk@/handlers/.code args/.@@body
↪  \endcsname#1}
```

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

导致

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

这表明：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。

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。

87.4.4 定义样式的手柄

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

⟨*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}}}
```

这导致执行

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

按 \pgfkeysdef 的定义，有

```
\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
```

即执行

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

导致

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

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}
```

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}}}
```

这导致

```
\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.

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}}}}
```

这导致

```
\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⟩}}
```

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}}}}
```

这导致

```
\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 pattern⟩}{\pgfkeysalso{⟨key list⟩}}}
```

导致定义

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

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*⟩ 原来所保存的 ⟨*key list*⟩, 再是 ⟨*append key list*⟩.

手柄 `/.add style` 的定义是:

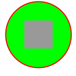
```
\pgfkeys{/handlers/.add style/.code 2 args=\pgfkeys{\pgfkeyscurrentpath/.add
↪   code={\pgfkeysalso{#1}}{\pgfkeysalso{#2}}}}%
```

这导致

```
\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。

87.4.5 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。

手柄 /.initial 的定义是：

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

这导致

```
\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 \csname pgfk@/my key\endcaname 定义为 \the\pgfkeys@temptoks{⟨*value*⟩}. 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 定义为 \the\pgfkeys@temptoks{⟨*Value*⟩}, 这只是赋值，并不输出 ⟨*Value*⟩.

Key handler ⟨*key*⟩/.get=⟨*marc*⟩

手柄 /.get 的定义是:

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

这会导致

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

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

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

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

red
blue

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

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}}
```

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*⟩.

Hello Gruffalo!

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

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，会先检查 ⟨*value*⟩ 的真值是 true 还是 false, 将 ⟨*value*⟩ 的真值赋予 ⟨*TeX-if name*⟩. 如果只写出 ⟨*key*⟩ 就默认 ⟨*value*⟩ 的真值是 true. 如果无法判断 ⟨*value*⟩ 的真值，就执行 `/errors/boolean expected` 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%
  }%
}
```

这会导致

```
\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}
  ```

  此命令执行一个条件语句:

  - 如果名称为 ⟨*TeX-if name*⟩ 的 if 命令已经被“声明”（定义），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”（定义），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}` 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。

87.4.6 键值的展开，多重键值

当写出 `\expandafter\aaaa\bbbb` 时,先展开 `\bbbb` 得到 ⟨*contents*⟩, 然后再顺次执行 `\aaaa`⟨*contents*⟩. 命令 `\expandafter` 在展开 `\bbbb` 时，只做“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
```

执行上面代码中的 `\ifnum` 时，先检查是否有与之对应的 `\fi`, 然后再执行条件判断。命令 `\expandafter` 作用于 (吃掉) `\else`, 而不是作用于 `\bbbb`.

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}}
```

87.4.7 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`, 检查键 `/secondary path/option` 是否存在, 然后执行之。但是注意下面的代码什么也不输出:

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

手柄 `/.search also` 的定义分两步:

1. 定义 \pgfkeys@searchalso@prepare@unknown@handler.
2. 执行

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

这会导致

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

所以执行 \pgfkeys{/my key/.search also={⟨*path list*⟩}} 就导致

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

命令 \pgfkeys@searchalso@prepare@unknown@handler 的定义格式是:

```
\def\pgfkeys@searchalso@prepare@unknown@handler#1{...}
```

下面看一下命令 \pgfkeys@searchalso@prepare@unknown@handler{⟨*path1*⟩,⟨*path2*⟩} 的处理过程:

1. 定义全局宏 \pgfkeys@global@temp

```
\global\def\pgfkeys@global@temp#1\pgfeov{}%
```

2. 执行命令

```
\pgfkeys@searchalso@parse ⟨path1⟩,⟨path2⟩,\pgfkeys@mainstop
```

其中的 \pgfkeys@mainstop 是命令 \pgfkeys@searchalso@parse 的参数定界标志。这个命令判断解析过程是否结束:

- 如果没有 "⟨*path1*⟩,⟨*path2*⟩," 这一部分，就执行

```
\pgfkeys@cleanup\pgfkeys@mainstop
```

结束解析过程。
- 如果有 "⟨*path1*⟩,⟨*path2*⟩," 这一部分，就执行 \pgfkeys@searchalso@appendentry.

3. 按上一步的结果，应该是执行

```
\pgfkeys@searchalso@appendentry ⟨path1⟩,⟨path2⟩,\pgfkeys@mainstop
```

命令 \pgfkeys@searchalso@appendentry 的定义格式是

```
\def\pgfkeys@searchalso@appendentry#1,#2{...}
```

所以命令 \pgfkeys@searchalso@appendentry 会把“#1,#2”吃掉，也就是会把“⟨*path1*⟩,”以及 ⟨*path2*⟩ 的第一个记号吃掉。暂且用 ⟨*t*⟩ 代表 ⟨*path2*⟩ 的第一个记号。按照命令 \pgfkeys@searchalso@appendentry 的定义，有下一步

4. 定义

```
\def\pgfkeys@searchalso@nexttok{⟨t⟩}
```

5. 定义

```
\pgfkeys@spdef\pgfkeys@temp{⟨path1⟩}
```

6. 然后用 { 开启一个 TeX 分组。

7. 定义

```
\toks0=\expandafter{\pgfkeys@global@temp#1\pgfeov}%
```

即

```
\toks0={}
```

8. 定义

```
\toks1=\expandafter{\pgfkeys@temp}
```

9. 用 \xdef 定义全局宏 \pgfkeys@global@temp

```
\xdef\pgfkeys@global@temp{%
    \the\toks0 % the space is important!
    \noexpand\ifpgfkeyssuccess\noexpand\else
      \noexpand\pgfqkeys{\the\toks1 }{\noexpand\pgfkeys@searchalso@name
               \ifx\pgfkeys@searchalso@nexttok\pgfkeys@mainstop\else/.try
               ↪  \fi /.expand once=\noexpand
               ↪  \pgfkeys@searchalso@temp@value}%
    \noexpand\fi}%
```

相当于

```
\gdef\pgfkeys@global@temp{%
    展开的\pgfkeys@global@temp#1\pgfeov%
    \ifpgfkeyssuccess\else
      \pgfqkeys{\the\toks1 }{\pgfkeys@searchalso@name
        \ifx\pgfkeys@searchalso@nexttok\pgfkeys@mainstop\else/.try\fi
        ↪  /.expand once=\pgfkeys@searchalso@temp@value}%
    \fi}%
```

10. 重定义全局宏 \pgfkeys@global@temp

```
\expandafter\gdef\expandafter\pgfkeys@global@temp\expandafter#\expandafter1
↪  \expandafter\pgfeov\expandafter{\pgfkeys@global@temp}%
```

也就是

```
\gdef\pgfkeys@global@temp#1\pgfeov{%
    \ifpgfkeyssuccess\else
      \pgfqkeys{⟨path1⟩}{\pgfkeys@searchalso@name
        /.try/.expand once=\pgfkeys@searchalso@temp@value}%
    \fi
}
```

11. 用 } 结束 TEX 分组。

12. 执行 \pgfkeys@searchalso@parse⟨*t*⟩. 因为之前 ⟨*t*⟩ 被吃掉了，这里补充上。也就是继续处理

```
\pgfkeys@searchalso@parse ⟨path2⟩,\pgfkeys@mainstop
```

得到

```
\gdef\pgfkeys@global@temp#1{%
    \ifpgfkeyssuccess\else
      \pgfqkeys{⟨path1⟩}{\pgfkeys@searchalso@name
        /.try/.expand once=\pgfkeys@searchalso@temp@value}%
    \fi
    \ifpgfkeyssuccess\else
      \pgfqkeys{⟨path2⟩}{\pgfkeys@searchalso@name
        /.expand once=\pgfkeys@searchalso@temp@value}%
    \fi
}
```

13. 执行 \pgfkeys@searchalso@parse\pgfkeys@mainstop, 结束解析。

14. 定义

```
\toks0=\expandafter{\pgfkeys@global@temp#1\pgfeov}%
```

即

```
\toks0={%
    \ifpgfkeyssuccess\else
      \pgfqkeys{⟨path1⟩}{\pgfkeys@searchalso@name
        /.try/.expand once=\pgfkeys@searchalso@temp@value}%
    \fi
    \ifpgfkeyssuccess\else
      \pgfqkeys{⟨path2⟩}{\pgfkeys@searchalso@name
        /.expand once=\pgfkeys@searchalso@temp@value}%
    \fi
}
```

15. 定义

```
\toks1={\pgfkeysalso{/handlers/.unknown/.@cmd/.expand once=
↪   \pgfkeys@searchalso@temp@value}}%
```

16. 重定义全局宏 \pgfkeys@global@temp

```
\xdef\pgfkeys@global@temp{%
  \noexpand\def\noexpand\pgfkeys@searchalso@temp@value{##1}%
  \noexpand\ifpgfkeysaddeddefaultpath
    \noexpand\pgfkeyssuccessfalse
    \noexpand\let\noexpand\pgfkeys@searchalso@name=\noexpand
    ↪  \pgfkeyscurrentkeyRAW
    \the\toks0
    ↪  % one or more /.try things; one for each path. The last element won't have a /.tr
    %\noexpand\ifpgfkeyssuccess
    %\noexpand\else
    %  \the\toks1 % invoke /handlers/.unknown handler
    %\noexpand\fi
  \noexpand\else
    \the\toks1 % invoke /handlers/.unknown handler
  \noexpand\fi
}%
```

相当于

```
\gdef\pgfkeys@global@temp{%
  \def\pgfkeys@searchalso@temp@value{##1}%
  \ifpgfkeysaddeddefaultpath
    \pgfkeyssuccessfalse
    \let\noexpand\pgfkeys@searchalso@name=\pgfkeyscurrentkeyRAW
    展开的\the\toks0
  \else
    \pgfkeysalso{/handlers/.unknown/.@cmd/.expand once=
    ↪  \pgfkeys@searchalso@temp@value}
  \fi
}%
```

17. 再次重定义全局宏 \pgfkeys@global@temp

```
\expandafter\gdef\expandafter\pgfkeys@global@temp\expandafter##
↪  \expandafter1\expandafter\pgfeov\expandafter{\pgfkeys@global@temp}%
```

也就是

```
\gdef\pgfkeys@global@temp#1\pgfeov{%
  \def\pgfkeys@searchalso@temp@value{#1}%
    \ifpgfkeysaddeddefaultpath
      \pgfkeyssuccessfalse
      \let\noexpand\pgfkeys@searchalso@name=\pgfkeyscurrentkeyRAW
      展开的\the\toks0
    \else
      \pgfkeysalso{/handlers/.unknown/.@cmd/.expand once=
      ↪  \pgfkeys@searchalso@temp@value}
    \fi
}
```

87.4.8 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`.

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，就用命令 `\pgfkeysifdefined` 检查名称为 `pgfk@\pgfkeyscurrentpath/.@def` 的命令是否已定义：
     - 如果已定义就执行

       ```
       \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。

3. 用命令 `\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，则用命令 `\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) 用命令 \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, 此命令的两个参数是：

i. 第一个参数是

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

ii. 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 其实是命令 \pgfkeys@case@one, \pgfkeys@case@two, \pgfkeys@case@three 这三个命令的综合修改版。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*⟩}.

以下情况之一设置 \pgfkeyssuccesstrue:

- 名称为 pgfk@\pgfkeyscurrentpath/.@cmd 的命令存在时
- 名称为 pgfk@\pgfkeyscurrentpath 的命令存在时
- 名称为 pgfk@/handlers/\pgfkeyscurrentname/.@cmd 的命令存在时

其它情况设置 \pgfkeyssuccessfalse.

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

87.4.9 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
```

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
```

`/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.
```

87.5 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}%
  }%
}
```

87.6 键筛选

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.

88 重复操作：foreach 句法

实现 foreach 句法的是宏包 pgffor，这个宏包会被 TikZ 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.

\foreach⟨*variables*⟩[⟨*options*⟩]in{⟨*list*⟩}⟨*commands*⟩

首先注意，\foreach 语句各个组成部分之间不能有空行。

在这个句法中，⟨*variables*⟩ 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).

[⟨*options*⟩] 是可选的。

{⟨*list*⟩} 是以逗号分隔的列表，要用花括号括起来。尽管 ⟨*list*⟩ 要用花括号括起来，但这对花括号只是起到“分界标志”的作用，用来提示列表的起止点，并不属于列表本身。也就是说，列表仅由“列表项”和“逗号”构成，不包含外围的那对花括号。{⟨*list*⟩} 也可以是一个宏，只要这个宏的值是逗号分隔的列表即可。其中列举的条目可以是数字，宏，字符串等等。注意 ⟨*list*⟩ 中某些地方的空格不会被忽略，所以不要为了增强代码的可读性而随意添加空格。

⟨*commands*⟩ 通常是用花括号括起来的一组命令、代码。

foreach 句法会将 {⟨*list*⟩} 中的诸条目，作为值依次赋予 ⟨*variables*⟩ 中的变量，对每次赋值，用 ⟨*commands*⟩ 执行一次。每次执行 ⟨*commands*⟩ 时，⟨*commands*⟩ 都会被放入一个 TEX 分组中。因此，如果 ⟨*commands*⟩ 中有受限于分组的命令，该命令的效果都会限于一次执行中，不会超出这次执行所在的分组。例如，假设 ⟨*commands*⟩ 中有将某个计数器加 1 的命令，在每次执行 ⟨*commands*⟩ 之前该计数器的值都是 n，那么每执行 ⟨*commands*⟩ 时该计数器的值就是 n+1；当各次执行结束后，该计数器的值还是 n. 如果希望某个命令不受分组限制，可以使用 \global.

[1][2][3][0]

```
\def\mylist{1,2,3,0}
\foreach \x in \mylist {[\x]}
```

88.1 ⟨*commands*⟩ 的句法

通常 ⟨*commands*⟩ 要用花括号括起来，foreach 命令就以这对花括号界定 ⟨*commands*⟩ 的起止位置。如果不用花括号将 ⟨*commands*⟩ 括起来，那么 \foreach 会检测分号，并以检测到的第一个分号为标志来结束 ⟨*commands*⟩.

```
\tikz \foreach \x in {0,1,2,3}
  \draw (\x,0) circle (0.2cm);
```

另外还有 \foreach 语句套嵌的情况。当两个 \foreach 语句直接套嵌时，内层的 \foreach 语句作为外层 \foreach 的 ⟨*commands*⟩, 可以不处于花括号内。

```
\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。

另外需要注意的是,如果 ⟨*commands*⟩ 中不出现所设定的变量 ⟨*variables*⟩, 那么程序就仅仅把 ⟨*commands*⟩ 执行若干次，所执行的次数是 ⟨*list*⟩ 中列表项的个数。

0.48586, 0.48692, 0.80148, 0.60384, 0.71593,

```
\foreach \x in {1,...,5} {\pgfmathparse{rnd}\pgfmathresult, }
```

88.2 ⟨*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,
```

88.3 在 ⟨*list*⟩ 中使用花括号包裹列举条目

如果 ⟨*list*⟩ 中的某个列举条目中含有逗号，则应当用花括号把该列举条目括起来；但如果该列举条目本身以开圆括号“(”开头，以闭圆括号“)”结尾，则可以不用花括号包裹该列举条目，例如 TikZ 的坐标。

```
\tikz \foreach \position in {(0,0), (1,1), (2,0), (3,1)}
    \draw \position rectangle +(.25,.5);
```

88.4 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) };
```

注意 node 和 pic 操作也支持 foreach 语句。

88.5 多个相互关联的变量

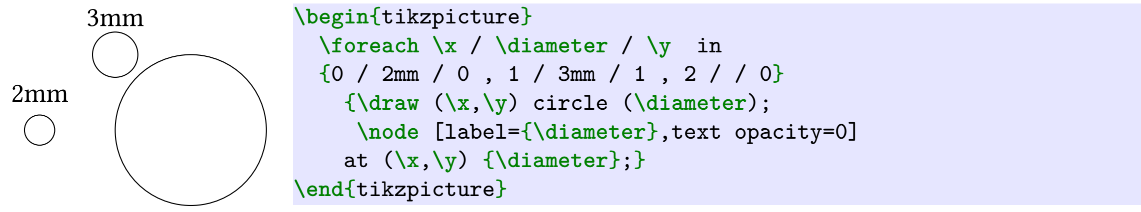
假设有 n 个状态点：$\{(x_1,y_1,z_1),\cdots,(x_n,y_n,z_n)\}$，每个状态点对应一个结果：$f((x_i,y_i,z_i))$，$i=1,2,\cdots,n$. 可以用 foreach 语句得到这 n 个结果，此时要用斜线“/”来分隔变量和变量值，即在 ⟨*variables*⟩ 中设置数个变量并用斜线“/”分隔它们，而在 ⟨*list*⟩ 中一个列举条目就是各变量的一个值，构成一个状态点，在条目中用斜线“/”分隔各个变量对应的值。在这里，状态点是 3 维向量，因此需要设置 3 个变量，例如 `\x/\y/\z`, 这 3 个变量不是相互独立的，也就是说，当 `\x` 取某个值时，另外两个变量不能随意取值，它们要关联起来构成这 n 个 3 维向量。下面举例来说明。

```
\foreach \x / \y in {1/2,a/b} { “\x\ and \y” }
得到 “1 and 2”“a and b”，这里 \x 与 \y 构成的向量是 (1,2)，(a,b)
```

关于斜线“/”需要注意以下 3 点：

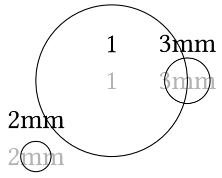
- 在 ⟨*variables*⟩ 中斜线“/”的前后可以有空格；在 ⟨*list*⟩ 中，斜线“/”的前后如果有空格，则空格会被当作变量值的一部分，有时会导致错误。

- 有的斜线“/”的前面或后面缺少变量值，就当作“空值”处理，在不同情况下程序会有不同反应。

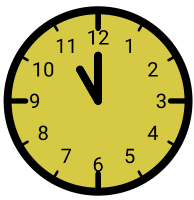



```
\begin{tikzpicture}
  \foreach \x / \diameter / \y  in
  {0 / 2mm / 0 , 1 / 3mm / 1 , 2 / / 0}
    {\draw (\x,\y) circle (\diameter);
     \node [label={\diameter},text opacity=0]
    at (\x,\y) {\diameter};}
\end{tikzpicture}
```

- 如果在 ⟨*list*⟩ 中，某个列表项缺少斜线“/”，那么程序会自动把该列表项补充完整。简单地说，如果某个列表项缺少斜线“/”，那么该列表项中一定缺少变量值。如果缺少第 1 个变量值，就把第 1 个变量值设为 1；如果缺少其它变量值（除第 1 个变量值以外），程序会把所缺少的变量值之前的一个变量值拿来做补充，并自动补充斜线，直到列表项中的斜线个数“够数”。




```
\begin{tikzpicture}
  \foreach \x / \diameter / \y  in
  {0 / 2mm / 0 , 2 / 3mm / 1 , /1 }
    {\draw (\x,\y) circle (\diameter);
     \node [label={\diameter},text opacity=0.3]
    at (\x,\y) {\diameter};}
\end{tikzpicture}
```




```
\begin{tikzpicture}[line cap=round,line width=3pt,scale=0.6]
  \filldraw [fill=yellow!80!black] (0,0) circle (2cm);
  \foreach \angle / \label in
    {0/3, 30/2, 60/1, 90/12, 120/11, 150/10, 180/9, 210/8, 240/7, 270/6, 300/5, 330/4}
  {
  \draw[line width=1pt] (\angle:1.8cm) -- (\angle:2cm);
  \draw (\angle:1.4cm) node[font=\footnotesize]{\textsf{\label}};
  }
\foreach \angle in {0,90,180,270}
  \draw[line width=2pt] (\angle:1.6cm) -- (\angle:2cm);
  \draw (0,0) -- (120:0.8cm); % 时针
  \draw (0,0) -- (90:1cm); % 分针
\end{tikzpicture}
```

```
\tikz[shading=ball,scale=0.5]
\foreach \x / \cola in {0/red,1/green,2/blue,3/yellow}
\foreach \y / \colb in {0/red,1/green,2/blue,3/yellow}
\shade[ball color=\cola!50!\colb] (\x,\y) circle (0.4cm);
```

下面的例子用 foreach 语句粗略地实现一种文字效果：

```
\tikz [scale=3]
{
  \foreach \i in {0,0.02,...,1.5}
    \node [transform shape,evaluate={\c=\i*60;}, text=red!\c,
      opacity=\i]at(-\i pt,-0.5*\i pt){好};
}
```

88.6 针对变量的选项

/pgf/foreach/var=⟨*variable*⟩ (no default)

这个选项用于声明变量名称，举例来说，以下两个叙述等效：

```
\foreach \x/\y
\foreach [var=\x,var=\y]
```

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。

/pgf/foreach/evaluate=⟨*variable*⟩as⟨*macro*⟩using⟨*formula*⟩ (no default)

在 ⟨*list*⟩ 中，列表项可能是算式，例如，

```
\foreach \x in {2^0,2^...,2^4}{$\x$, }
```

得到 $2^0, 2^1, 2^2, 2^3, 2^4$, 尽管算式 `2^0` 的值是 1，但是命令会把算式形式 `2^0` 赋予变量 `\x`，而不是把 1 赋予变量 \x. 如果想把 ⟨*list*⟩ 中列表项算式的值赋予变量，就使用这个选项。

等号后的 ⟨*variable*⟩as⟨*macro*⟩using⟨*formula*⟩ 都是可选的。

⟨*variable*⟩ 用于指定该选项针对哪个变量，也就是说，算式的运算结果将被保存在 ⟨*variable*⟩ 中。

as⟨*macro*⟩using⟨*formula*⟩ 的作用是：定义宏 ⟨*macro*⟩; 把 ⟨*variable*⟩ 的值带入公式 ⟨*formula*⟩ 中做计算，计算结果保存在宏 ⟨*macro*⟩ 中；变量 metavariable 的值仍然是列表项中的算式形式；如果不写出 using⟨*formula*⟩ 这一部分，就把列表项中算式的运算结果保存在宏 ⟨*macro*⟩ 中，变量 metavariable 的值仍然是列表项的算式形式。

宏 ⟨*macro*⟩ 储存了算式的值，可以用在 ⟨*commands*⟩ 中参与各种运算。

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}{$\x$, }
```

$[1.0, 3^0]$, $[2.0, 3^1]$, $[4.0, 3^2]$,

```
\foreach \x / \y [evaluate=\x] in {2^0/3^0, 2^1/3^1, 2^2/3^2} {$[\x, \y$], }
```

$2^0 = 1.0$, $2^1 = 2.0$, $2^2 = 4.0$, $2^3 = 8.0$, $2^4 = 16.0$, $2^5 = 32.0$, $2^6 = 64.0$, $2^7 = 128.0$, $2^8 = 256.0$,

```
\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 \x [evaluate=\x as \shade using \x*10] in {0,1,...,10}
  \node [fill=red!\shade!yellow, minimum size=0.65cm] at (\x,0) {\x};
```

注意在 ⟨*formula*⟩ 中可以使用数学程序库 math 的句法，例如：

0 1 2 3 4 5

```
\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. 615, 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/remember=⟨*variable*⟩as⟨*macro*⟩(initially ⟨*value*⟩) (no default)

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 $f(a_{n-1}, a_n)$，也就是说，⟨*commands*⟩ 是针对 \x 的当前值和前一个值的操作，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}
```

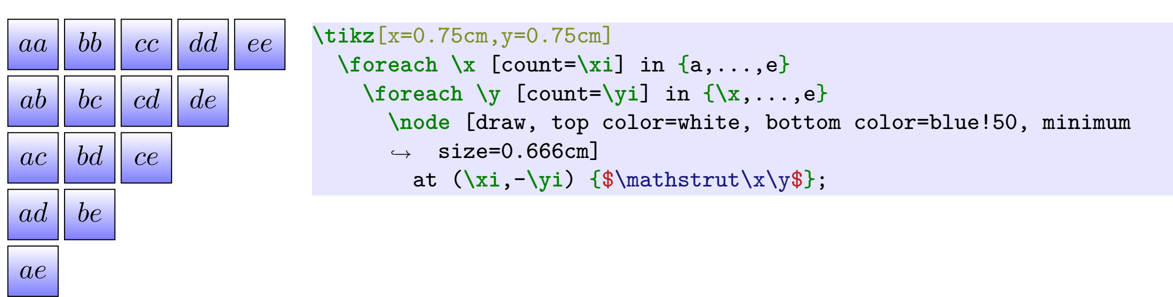
$\overrightarrow{AB}$, $\overrightarrow{BC}$, $\overrightarrow{CD}$, $\overrightarrow{DE}$, $\overrightarrow{EF}$,
$\overrightarrow{\text{初值}C}$, $\overrightarrow{CD}$, $\overrightarrow{DE}$, $\overrightarrow{EF}$,

```
\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=⟨*macro*⟩from⟨*value*⟩ (no default)

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={⟨*boolean*⟩} (default false)

如果本选项的值设为 true, 那么 ⟨*list*⟩ 中的列表项会被命令 \pgfmathparse 处理，所以列表项可以是很复杂的表达式，但表达式不能过于“特别”，因为已经知道某些个内部 TEX 寄存器可能会引起问题。

1 2 3 4 5 6 7 8 9
```
\foreach \x [parse=true] in {1,...,1.0e+1 - 1}{ \x }
```

`\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。

```
\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}
```

88.7 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 分组中。

观察下面的例子：

0

5

```
\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
```

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 count 等也不能用于此命令。

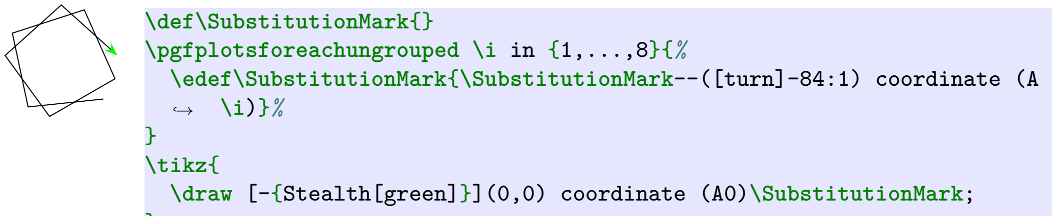
观察下面的例子：

--([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}
```

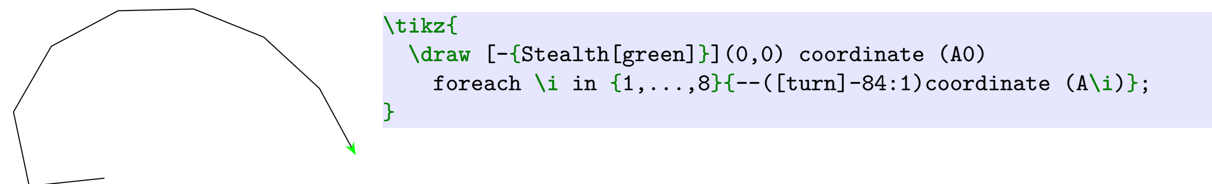
这个例子用一个迭代构造 \SubstitutionMark.

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：

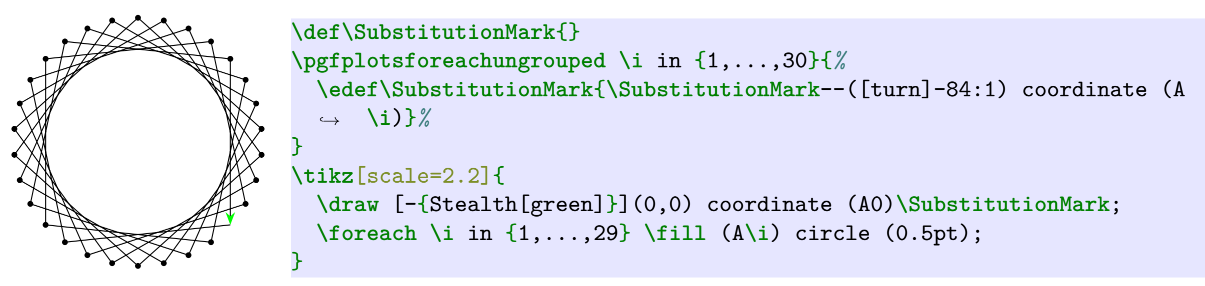


```
\def\SubstitutionMark{}
\pgfplotsforeachungrouped \i in {1,...,8}{%
  \edef\SubstitutionMark{\SubstitutionMark--([turn]-84:1) coordinate (A
  ↪  \i)}%
}
\tikz{
  \draw [-{Stealth[green]}](0,0) coordinate (A0)\SubstitutionMark;
}
```

如果使用 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)};
}
```



```
\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}
```

下面的例子得到 Fibonacci 数列的第 20, 21 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 21 项，因为 $6765 + 10946 = 17711 > 16383.99999$, 已经超出 TeX 关于单位 pt 的运算范围。要想得到更多的项，需要使用 fpu 程序库。

下面两个例子都调用 fpu 程序库，但计算结果不一样：

$8.3189 \cdot 10^5$

$1.346027 \cdot 10^6$

```
\pgfkeys{/pgf/fpu,/pgf/number format/precision=25}
\def\FibonacciFirst{1}
\def\FibonacciSecond{1}
\foreach \i in {1,...,29}
{
  \pgfmathsetmacro{\FibonacciTemp}{int(\FibonacciFirst +
  ↪  \FibonacciSecond)}
  \xdef\FibonacciFirst{\FibonacciSecond}
  \xdef\FibonacciSecond{\FibonacciTemp}
}
\pgfmathprintnumber[sci]{\FibonacciFirst}\par
\pgfmathprintnumber[sci]{\FibonacciSecond}
```

$8.3203993 \cdot 10^5$

$1.34626889 \cdot 10^6$

```
\def\FibonacciFirst{1}
\def\FibonacciSecond{1}
\pgfmathfloatparsenumber{\FibonacciFirst}
\xdef\FibonacciFirstTemp{\pgfmathresult}
\pgfmathfloatparsenumber{\FibonacciSecond}
\xdef\FibonacciSecondTemp{\pgfmathresult}
\foreach \i in {1,...,29}
{
  \pgfmathfloatadd{\FibonacciFirstTemp}{\FibonacciSecondTemp}
  \xdef\FibonacciFirstTemp{\FibonacciSecondTemp}
  \xdef\FibonacciSecondTemp{\pgfmathresult}
}
\pgfmathprintnumber[sci,precision=25]{\FibonacciFirstTemp}\par
\pgfmathprintnumber[sci,precision=25]{\FibonacciSecondTemp}
```

以上两个例子计算出来的结果相差较大。

91 扩展颜色支持

宏包 `xxcolor` 是 PGF 的一个子宏包,它扩展了宏包 `color` 的特性。宏包 `xxcolor` 提供环境 `{colormixin}` 和命令 `\colorcurrentmixin`.

调用宏包 `\usepackage{xxcolor}`.

`\begin{colormixin}{`⟨*mix-in specification*⟩`}`
⟨*environment content*⟩
`\end{colormixin}`

宏包 `xcolor` 定义了类似 `black!25!white` 这样的颜色表达式，意思是将 25% 的黑色与 75% 的白色混合。将这种颜色表达式的第一个颜色及第一个感叹号“`!`”去掉就是这里的 ⟨*mix-in specification*⟩, 例如 `25!white`，下面以此为例继续介绍。

在环境内容 ⟨*environment contents*⟩ 中可能有多种内容，如文字，表格线，图形，从外部插入的图形等等，这些内容都有自己原本的颜色。例如，假设文字的颜色原本是黑色 black, 那么本环境就会把文字的颜色变成 `black!25!white`; 如果某个线条的颜色是自定义颜色 ⟨*self-def-color*⟩, 那么这个线条

的颜色就变成 ⟨*self-def-color*⟩`!25!white`. 也就是说，某个颜色成分与环境内容的原有色相混合，从而修改了内容的颜色，为了方便，称这个颜色成分为“修改色”。

Red text,washed-out red text, washed-out blue text, dark washed-out blue text, dark washed-out green text, back to washed-out blue text, and back to red.

```
\begin{minipage}{4cm}\raggedright
  \color{red}Red text,%
  \begin{colormixin}{25!white}
    washed-out red text,
    \color{blue} washed-out blue text,
    \begin{colormixin}{25!black}
      dark washed-out blue text,
      \color{green} dark washed-out green text,%
    \end{colormixin}
    back to washed-out blue text,%
    \tikz \fill [green] circle(4pt);
  \end{colormixin}
  and back to red.
\end{minipage}%
```

注意本环境未必能改变所有内容的颜色，一般插入图形的颜色不会被本环境改变。本环境一定能改变命令 `\color` 规定的颜色。另外命令 `\pgfuseimage` 和 `\pgfuseshading` 支持本环境。

以下面的代码为例：

```
\begin{colormixin}{25!white}
  \begin{pgfpicture}
    \pgftext[at=\pgfpoint{1cm}{5cm},left,base] {\pgfuseimage{image}}
    \pgfusepath{stroke}
  \end{pgfpicture}
\end{colormixin}
```

上面的代码中，在 `{colormixin}` 环境内使用了 `{pgfpicture}` 环境，命令 `\pgfuseimage{image}` 指定插入名称为 `image` 的图形，但实际上 PGF 会先查找并插入名称为 `image.!25!white` 的图形，这个名称是原图名称与环境 `{colormixin}` 的参数用感叹号“`!`”连接起来的符号串。

当套嵌使用 `{colormixin}` 环境时，各环境的参数 ⟨*mix-in specification*⟩ 会按照次序，由感叹号“`!`”连接起来，构成“修改色”来修改环境内容的颜色。当多个 `{colormixin}` 环境套嵌时，“修改色”可能不太容易梳理清楚，可以用命令 `\colorcurrentmixin` 返回当前的“修改色”。

`\colorcurrentmixin`

本命令返回当前的“修改色”。

!75!white should be “!75!white”

!75!black!75!white should be “!75!black!75!white”

!50!white!75!black!75!white should be “!50!white!75!black!75!white”

!50!green!75!black!75!white should be “!50!green!75!black!75!white”

```
\begin{minipage}{\linewidth-6pt}\tt \raggedright
  \begin{colormixin}{75!white}
    \colorcurrentmixin\ should be “!75!white” \par
    \begin{colormixin}{75!black}
      \quad\colorcurrentmixin\ should be “!75!black!75!white” \par
      \begin{colormixin}{50!white}
        \qquad\colorcurrentmixin\ should be “!50!white!75!black!75!white” \par
      \end{colormixin}
      \begin{colormixin}{50!green}
```

```
        \qquad\colorcurrentmixin\ should be “!50!green!75!black!75!white”\par
      \end{colormixin}
    \end{colormixin}
  \end{colormixin}
\end{minipage}
```

92 解析器模块

```
\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。

对于给定的一串字母，解析器逐个扫描其中的字母：首先解析器处于初始状态 (`initial`，这是个关键词)，在这个状态下扫描第一个字母，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；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，如此继续，直到解析器达到终结状态 (`final`，这是个关键词)，结束解析过程。也就是说，在解析过程中的任何时候，解析器都处于某个状态，并在该状态下进行扫描、执行代码、切换状态的操作，直到 `final` 状态。

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}aabaabababbbbbabaabcccc
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 3 种状态，这 3 个状态的名称都是 `initial`，但它们所针对的字母不同 (分别针对字母 a, b, c)，所执行的代码也不同。上面例子中的字母串的结束处有 4 个字母 `c`，第一个 `c` 被解析器扫描并使得解析器切换到 `final` 状态，从而结束解析过程，剩余 3 个字母 `c` 被打印到屏幕上。

`\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。

`\pgfparserdef`{⟨*parser name*⟩}{⟨*state*⟩}{⟨*symbol meaning*⟩}[⟨*arguments*⟩]{⟨*action*⟩}

这个命令定义一个名称为 ⟨*parser name*⟩ 的解析器，并且为 ⟨*parser name*⟩ 添加一个状态 ⟨*state*⟩，可以多次使用本命令为 ⟨*parser name*⟩ 添加多个状态，一个状态包括状态名称 ⟨*state*⟩ 以及 ⟨*symbol meaning*⟩ 和 ⟨*action*⟩.

⟨*symbol meaning*⟩ 是状态 ⟨*state*⟩ 在扫描过程所针对的字母符号。

⟨*action*⟩ 是某些代码，当在状态 ⟨*state*⟩ 下扫描出 ⟨*symbol meaning*⟩ 时就执行 ⟨*action*⟩.

注意这里 ⟨*symbol meaning*⟩ 的格式应当是 TEX 命令 `\meaning` 的输出结果的形式，例如，假设状态 ⟨*state*⟩ 是针对字母 `A` 设置的，那么这里的 ⟨*symbol meaning*⟩ 就应该是 `the letter A`，因为命

令 `\meaning A` 的输出就是 `the letter A`. 一个空格产生一个 blank space. 也可以去掉 ⟨*symbol meaning*⟩ 外围的花括号，直接写出符号 ⟨*symbol meaning*⟩, 例如:

```
\pgfparserdef{myparser}{initial}{the letter a}{foo}
等效于
\pgfparserdef{myparser}{initial}a{foo}
```

⟨*arguments*⟩ 是选项格式，类似 xparse 宏包所规定的 argument specification, 其中列举的是选项类型标示符号（不用逗号分隔）。在 ⟨*arguments*⟩ 中至多使用 9 个选项类型标示符号，可用的选项类型标示符号如下:

`m` 代表强制选项，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*⟩ 代表强制选项，⟨*delim*⟩ 作为定界符，当遇到 ⟨*delim*⟩ 时就认为选项列举完毕。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`.

`o` 代表可选项，它对应以 "`[`⟨*something*⟩`]`" 这种形式给出的可选参数，这里的 ⟨*something*⟩ 被默认为某个 "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*⟩`}` 代表可选项，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*⟩ 代表可选项, ⟨*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*⟩} 代表可选项，⟨*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
```

- 如果 ⟨*action*⟩ 中有两个必选项，则 [⟨*arguments*⟩] 就是 [mm].
- 如果想在 [⟨*arguments*⟩] 中把星号 * 作为一个 token, 然后设置一个用方括号给出的可选项（若不给出这个可选项就自动使用某个 mark），然后是一个必选项，然后设置一个用圆括号给出的可选项（如果不给出这个可选项就把 something 作为这个可选项），那么 [⟨*arguments*⟩] 就是 [t*omD()something].
- 如果 argument 是分号之前的哪些符号、代码，那么 [⟨*arguments*⟩] 就是 [r;].
- 如果把 [⟨*arguments*⟩] 设置为 [r m], 那么 argument 会一直延续，直到遇到字母 m 才结束。
- 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}].

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*⟩ 中不能使用解析器，除非将该解析器限制在一个域 (scope) 中。

若 ⟨*state*⟩ 被设置为 all(这是个关键词)，那么相应的 ⟨*action*⟩ 就会在所有状态下被执行，无论各个状态是否扫描到 ⟨*symbol meaning*⟩.

注意，如果被扫描的字母串 ⟨*text*⟩ 中出现了解析器不能识别的字母，解析器就会发出错误信息。

\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，可以有以下三种形式：

- 第一种，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⟩
  ```

  就使得解析器 ⟨*parser name 1*⟩ 的状态 ⟨*state 1*⟩ 针对 ⟨*symbol meaning 1*⟩ 的操作等同于操作 ⟨*deal*⟩.
- 第二种，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⟩
  ```

  就使得解析器 ⟨*parser name 1*⟩ 的状态 ⟨*state 1*⟩ 针对 ⟨*symbol meaning 1*⟩ 的操作等同于操作 ⟨*deal*⟩, 也就是定义状态 s_1.
- 第三种，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*⟩ 的操作等同于操作 ⟨*deal*⟩, 也就是定义状态 s_1.

`\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 letter 时，就执行 ⟨*action*⟩.

3

```
\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
```

`\pgfparserdeffinal{⟨parser name⟩}{⟨action⟩}`

本命令的作用是，当解析器转到终结状态 `final` 后，执行 ⟨*action*⟩.

`\pgfparserswitch{⟨state⟩}`

如前面的例子所示，本命令用在 ⟨*action*⟩ 中，使得解析器切换至状态 ⟨*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母串中的) 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`

此命令可以用在 `\pgfparserdef` 或 `\pgfparserdefunknown` 规定的 ⟨*action*⟩ 中的末尾，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`

这个命令的展开值是当前状态的名称。

`\pgfparsertoken`

这个命令的展开值是当前的 letter.

`\pgfparserletter`

这个命令的展开值是当前的 letter. 例如，如果

```
\pgfparserparse{foo}a
```

那么就有定义 `\def\pgfparserletter{a}`, 这个定义发生在 a 引起的 ⟨*action*⟩ 之前。

`\pgfparserset{`⟨*key list*⟩`}`

此命令的定义是:

```
\long\def\pgfparserset#1%
  {%
    \pgfset{/pgfparser/.cd,#1}%
  }
```

92.1 Parser 模块的选项

/pgfparser/silent=⟨*boolean*⟩ (no default, initially false)

在本选项值为 true 的情况下，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)

在本选项值为 true 的情况下，每当执行 ⟨*action*⟩ 时，就发布一个状态信息，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)

在本选项值为 true 的情况下，解析器 ⟨*parser name*⟩ 会自动忽略未定义的 letter.

92.2 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*O{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};
```

93 数学引擎概略

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。

94 数学表达式

在解析数学表达式时，在任何时刻都应保证计算范围不超过 ±16383.99999，这是 TEX 允许的尺寸。先解释一下 TEX 的寄存器（register）。TEX 的寄存器分为多种，每一种都有自己的名称和编号，每个寄存器都可以保存某种特别类型的值。例如，

- 有 256 个整数寄存器：`\count0`，`\count1`，……`\count255`，每个都可以保存一个整数，但整数范围限制为 ±2147483647. 用 `\count0` 可以引用保存在该寄存器中的整数值。从 `\count0` 到 `\count22` 这 23 个整数寄存器是系统自己使用的，最好不要随意改变它们的值。
- 有 256 个（刚性）尺寸寄存器：`\dimen0`，`\dimen1`，……`\dimen255`，每个都可以保存一个尺寸。
- 有 256 个弹性尺寸寄存器：`\skip0`，`\skip1`，……`\skip255`，每个都可以保存一个弹性尺寸。
- 有 256 个弹性数学尺寸寄存器：`\muskip0`，`\muskip1`，……`\muskip255`，每个都可以保存一个弹性数学尺寸。
- 有 256 个盒子寄存器：`\box0`，`\box1`，……`\box255`，每个都可以保存一个盒子。
- 有 256 个记号列（token list）寄存器：`\toks0`，`\toks1`，……`\toks255`，每个都可以保存一个记号（token）。

几个命令：

`\wd` 这个命令的句法是 `\wd`⟨*number*⟩`=`⟨*dimension*⟩，将编号为 ⟨*number*⟩ 的盒子的宽度设为 ⟨*dimension*⟩，如果不给出"⟨*dimension*⟩"就默认其值为盒子的自然宽度。而 `\wd`⟨*number*⟩ 得到编号为 ⟨*number*⟩ 的盒子的宽度。

`\dp` 这个命令与 `\wd` 类似，只是针对盒子深度。

`\ht` 这个命令与 `\wd` 类似，只是针对盒子高度。

`\dimexpr` 这个命令的句法是 `\dimexpr`⟨*dimension expression*⟩. ⟨*dimension expression*⟩ 是关于（带单位的）尺寸的算术表达式（即只用加减乘除构造的算式），该命令计算这个表达式的值。

`\numexpr` 这个命令的句法是 `\numexpr`⟨*integer expression*⟩. ⟨*integer expression*⟩ 是关于整数的算术表达式，该命令计算这个表达式的值。

`\muexpr` 这个命令的句法是 `\muexpr`⟨*mu expression*⟩. ⟨*mu expression*⟩ 是关于数学长度（带有单位 mu 的尺寸）的算术表达式，该命令计算这个表达式的值。

`\the` 这个命令的句法是 `\the< 命令 >`，可以输出保存在 < 命令 > 中的数值、尺寸、代码等。

`\countdef` 这个命令的句法是 `\countdef`⟨*command*⟩`=`⟨*number*⟩. 这个句法创建命令 ⟨*command*⟩，故 ⟨*command*⟩ 应是以反斜线开头的命令形式。这个句法将命令 ⟨*command*⟩ 等同于编号为 ⟨*number*⟩ 的整数寄存器 `\count`⟨*number*⟩，故对 ⟨*number*⟩ 进行操作等同于对 `\count`⟨*number*⟩ 进行操作。

`\dimendef` 这个命令的句法是 `\dimendef`⟨*command*⟩`=`⟨*number*⟩. 这个句法创建命令 ⟨*command*⟩，故 ⟨*command*⟩ 应是以反斜线开头的命令形式。这个句法将命令 ⟨*command*⟩ 等同于编号为 ⟨*number*⟩ 的尺寸寄存器 `\dimen`⟨*number*⟩，故对 ⟨*command*⟩ 进行操作等同于对 `\dimen`⟨*number*⟩ 进行操作。

`\skipdef` 这个命令的句法是 `\skipdef`⟨*command*⟩`=`⟨*number*⟩. 这个句法创建命令 ⟨*command*⟩，故 ⟨*command*⟩ 应是以反斜线开头的命令形式。这个句法将命令 ⟨*command*⟩ 等同于编号为 ⟨*number*⟩ 的弹性尺寸寄

存器 \skip⟨*number*⟩, 故对 ⟨*command*⟩ 进行操作等同于对 \skip⟨*number*⟩ 进行操作。

\muskipdef 这个命令的句法是 \muskipdef⟨*command*⟩=⟨*number*⟩. 这个句法创建命令 ⟨*command*⟩, 故 ⟨*command*⟩ 应是以反斜线开头的命令形式。这个句法将命令 ⟨*command*⟩ 等同于编号为 ⟨*number*⟩ 的弹性数学尺寸寄存器 \muskip⟨*number*⟩, 故对 ⟨*command*⟩ 进行操作等同于对 \muskip⟨*number*⟩ 进行操作。

\setbox 这个命令的句法是 \setbox⟨*number*⟩=⟨*box*⟩. 这个句法将盒子 ⟨*box*⟩ 中的内容赋予编号为 ⟨*number*⟩ 的盒子寄存器 \box⟨*number*⟩.

\toksdef 这个命令的句法是 \toksdef⟨命令⟩=⟨*number*⟩. 这个句法将第 ⟨*number*⟩ 个记号列寄存器 \toks⟨*number*⟩ 等同于⟨命令⟩.

\newbox 这个命令的句法是 \newbox⟨*command*⟩, 用于自定义盒子命令 ⟨*command*⟩, 故 ⟨*command*⟩ 应是以反斜线开头的命令形式。这个句法将 ⟨*command*⟩ 创建为一个盒子，系统会自动为它分配一个盒子寄存器，即它会占用一个盒子寄存器。自定义盒子命令占用的盒子寄存器是从 \box26 到 box233, 共 208 个，即最多可以自定义 208 个盒子命令，过多的自定义盒子命令会导致错误信息：No room for a new \box.

\newcount 这个命令的句法是 \newcount⟨*command*⟩, 用于自定义一个整数寄存器命令 ⟨*command*⟩, 故 ⟨*command*⟩ 应是以反斜线开头的命令形式。系统会自动在 \count79，……，\count233 中挑选一个寄存器分配给 ⟨*command*⟩.

\newdimen 这个命令的句法是 \newdimen⟨*command*⟩, 用于自定义一个刚性尺寸寄存器命令 ⟨*command*⟩, 故 ⟨*command*⟩ 应是以反斜线开头的命令形式。系统会自动在 \dimen102，……，\dimen233 中挑选一个寄存器分配给 ⟨*command*⟩.

\newtoks 这个命令的句法是 \newtoks⟨*command*⟩, 将 ⟨*command*⟩ 创建为一个记号列寄存器，即它会占用一个记号列寄存器。

\multiply 这个命令的句法是 \multiply ⟨*a*⟩ by ⟨*b*⟩, 本命令计算 ⟨*a*⟩ 与 ⟨*b*⟩ 的乘积，其中 ⟨*a*⟩ 可以是数值、长度、代表数值或长度的命令，⟨*b*⟩ 应该是数值、整数、代表数值或整数的命令。

94.1 解析一个表达式

94.1.1 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。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，使用这个宏可以输出结果数字。如果 ⟨*expression*⟩ 中没有长度单位，那么解析的结果（而不是过程）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。

例如：

5.5 5.5

```
\pgfmathparse{2+3.5pt} \pgfmathresult \qquad
\pgfmathparse{2pt+3.5pt} \pgfmathresult
```

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。

\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），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*⟩.

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：

1.0pt plus 1.0fil

```
\skipdef\myskip=0
\pgfmathsetlength{\myskip}{+1pt plus 1fil} \the\myskip
```

`\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*⟩.

94.1.2 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. 这个条件判断命令是针对全局的，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

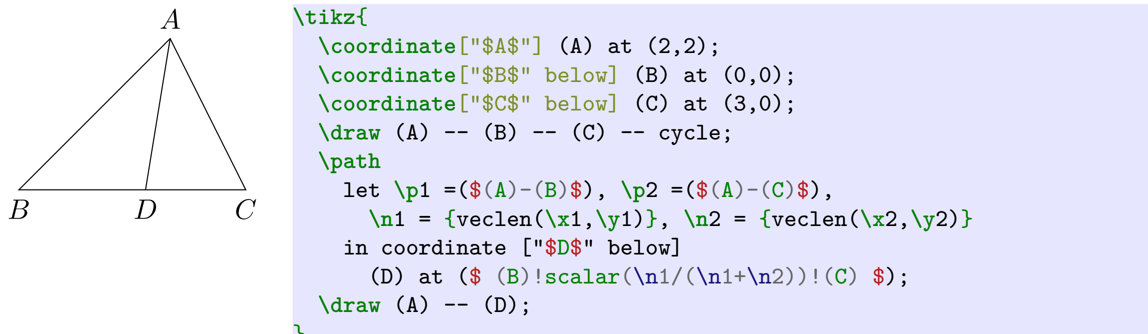
```
\pgfmathparse{scalar(1pt/2pt)} \pgfmathresult\
\ifpgfmathunitsdeclared with \else without \fi unit
```

注意 1pt+scalar(1pt) 与 scalar(1pt)+1pt 是不同的表达式：

2.0 without unit
2.0 with unit

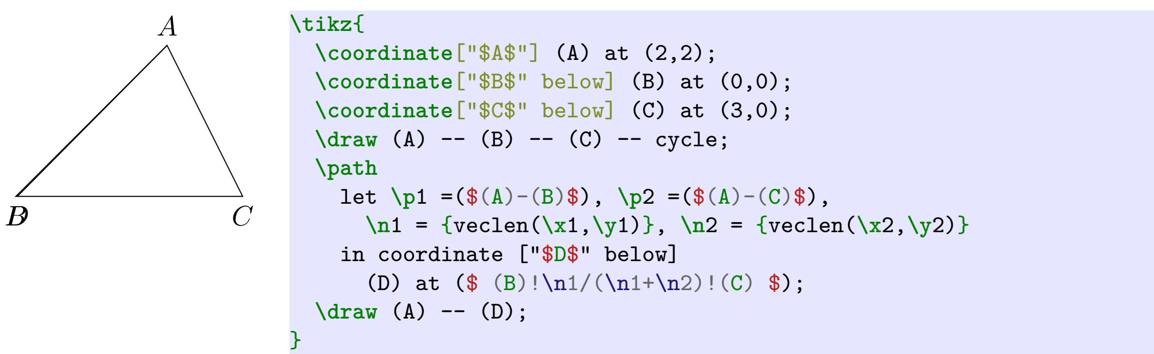
```
\pgfmathparse{1pt+scalar(1pt)} \pgfmathresult\
\ifpgfmathunitsdeclared with \else without \fi unit

\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，`\n1/(\n1+\n2)` 是带单位的，用 `scalar` 将其包裹起来得到无单位的数值，否则构造的坐标算式无效，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 ["$D$" below]
      (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。

94.2　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。

`xr` (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, hkai 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
```

贰

3 c 叁
2 b 贰
1 a 壹

```
\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]}};
}
```

`"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
6

```
\def\agroup{{96}}
\pgfmathsetmacro{\onenine}{\agroup[0]}\onenine\\
\pgfmathsetmacro{\onesix}{\agroup[1]}\onesix
```

94.3 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
```

94.3.1 基本算术函数

关于取整函数、除法余数的概念，参考

```
https://en.wikipedia.org/wiki/Floor_and_ceiling_functions
https://en.wikipedia.org/wiki/Truncation
https://en.wikipedia.org/wiki/Modulo_operation
```

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

$$r = a - \mathrm{sign}(b) \cdot b \cdot \left\lfloor \frac{a}{|b|} \right\rfloor.$$

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

$$\frac{a}{|b|} = \begin{cases} q + \frac{r}{b}, \ b > 0, \\ -q - \frac{r}{b}, \ b < 0, \end{cases} \quad 故 \ \left\lfloor \frac{a}{|b|} \right\rfloor = \begin{cases} q, \ b > 0, \\ -q, \ b < 0, \end{cases}$$

     所以

$$r = a - \mathrm{sign}(b) \cdot b \cdot \left\lfloor \frac{a}{|b|} \right\rfloor.$$

   - 若 $a < 0$，则 $r < 0$ 且 $|r| < |b|$，此时

$$\frac{a}{|b|} = \begin{cases} q + \frac{r}{b}, \ b > 0, \\ -q - \frac{r}{b}, \ b < 0, \end{cases} \quad 故 \ \left\lceil \frac{a}{|b|} \right\rceil = \begin{cases} q, \ b > 0, \\ -q, \ b < 0, \end{cases}$$

     所以

$$r = a - \mathrm{sign}(b) \cdot b \cdot \left\lceil \frac{a}{|b|} \right\rceil.$$

4. 令 r 的符号与 b 相同，这个情况叫作 floored division，

$$\frac{a}{b} = q + \frac{r}{b}, \ 故 \ \left\lfloor \frac{a}{b} \right\rfloor = q,$$

   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.

`add(x,y)`

`\pgfmathadd{⟨x⟩}{⟨y⟩}`

加法。

81.0

```
\pgfmathparse{add(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。

94.3.2 舍入函数

`round(x)`

`\pgfmathround{⟨x⟩}`

四舍五入。

`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
```

94.3.3 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.

94.3.4 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{$\langle x\rangle$}

假定 x 是角度制的数，并将它转换为弧度制的数。

deg(x)

\pgfmathdeg{$\langle x\rangle$}

假定 x 是弧度制的数，并将它转换为角度制的数。

sin(x)

\pgfmathsin{$\langle x\rangle$}

正弦函数，默认 x 是角度制的数。

cos(x)

\pgfmathcos{$\langle x\rangle$}

tan(x)

\pgfmathtan{$\langle x\rangle$}

sec(x)

\pgfmathsec{$\langle x\rangle$}

cosec(x)

\pgfmathcosec{$\langle x\rangle$}

cot(x)

\pgfmathcot{$\langle x\rangle$}

asin(x)

\pgfmathasin{$\langle x\rangle$}

反正弦函数，值域是 $[-90°, 90°]$.

acos(x)

\pgfmathacos{$\langle x\rangle$}

反余弦函数，值域是 $[0°, 180°]$.

atan(x)

\pgfmathatan{$\langle x\rangle$}

反正切函数，计算出来的函数值默认为角度制的数。

atan2(y,x)

\pgfmathatantwo{$\langle y\rangle$}{$\langle x\rangle$}

计算实数 $y \div x$ 所对应的反正切数值（角度制的），若 $x = 0$ 则返回 $90°$ 或 $-90°$.

-45.0

```
\pgfmathparse{atan2(-4,4)} \pgfmathresult
```

/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.

\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。

94.3.5 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⟩}{⟨y⟩}`

如果 x 的解析结果是 0 值，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.

94.3.6 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。

0.60799, 0.98637, 0.26668, 0.23878, 0.78108, 0.54295, 0.02867, 0.93219, 0.86432, 0.30621,

```
\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：

.0.0

```
\pgfmathparse{real(2)}
\pgfmathparse{random(\pgfmathresult,10)}
```

上面例子中出现了多余的“`.0`”, 对比下面的：

5

```
\pgfmathparse{int(real(2))}
\pgfmathparse{int(random(\pgfmathresult,10))}\pgfmathresult
```

94.3.7 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。

94.3.8 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
```

注意函数 min 有特别的规则。

- 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.

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。

78.12405

```
\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。

95 其它数学命令

95.1 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。

95.2 比较与逻辑函数

\pgfmathapproxequalto{⟨x⟩}{⟨y⟩}

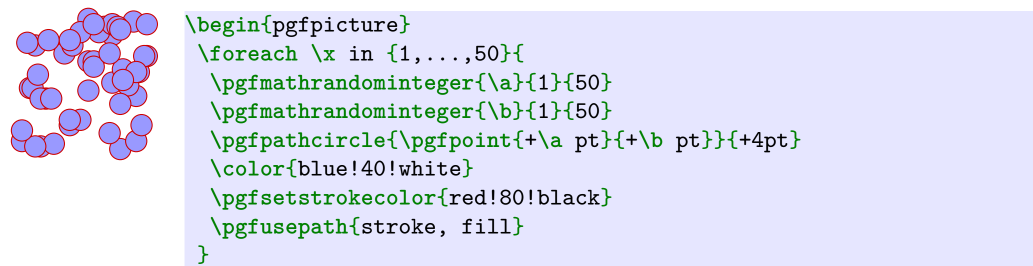
如果 ⟨x⟩ 与 ⟨y⟩ 的绝对值之差满足 $|x - y| < 0.0001$，该命令会把 1.0 保存在 \pgfmathresult 中；除了这个情况外，把 0.0 保存在 \pgfmathresult 中. 该命令还会使得 TEX 的条件判断命令 \ifpgfmathcomparison 的真值被相应地设定，这个条件判断命令是全局命令。

0.0
false

```
\pgfmathapproxequalto{0.01}{0.002} \pgfmathresult \\
\ifpgfmathcomparison true \else false \fi
```

`\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。



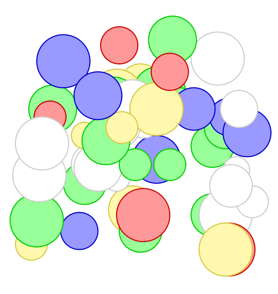
```
\begin{pgfpicture}
 \foreach \x in {1,...,50}{
  \pgfmathrandominteger{\a}{1}{50}
  \pgfmathrandominteger{\b}{1}{50}
  \pgfpathcircle{\pgfpoint{+\a pt}{+\b pt}}{+4pt}
  \color{blue!40!white}
  \pgfsetstrokecolor{red!80!black}
  \pgfusepath{stroke, fill}
 }
\end{pgfpicture}
```

`\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 `\tim`×`\year`.

95.3 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。

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 metainteger 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}
```

95.4 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
```

96 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，则带星号“*”的

命令会将函数的定义改写为“新版”。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。

⟨*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）后可以保存在命令 \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。

下面用一个例子来展示本命令的用法。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，定义函数的 ⟨*code*⟩ 是个 TEX 分组内容。\pgf@x 是 PGF 的尺寸寄存器，PGF 的尺寸寄存器还有 \pgf@y, 整数寄存器有 \c@pgf@counta, \c@pgf@countb 等等。

命令 \multiply 计算乘积。

注意宏\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 会把展开 \pgf@x 后的结果（去掉单位）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*⟩, 实际上就是“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。这个宏按 ⟨*code*⟩ 执行计算，然后将计算结果（不带单位）保存在 \pgfmathresult 中。

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. 604 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。例如函数 min 的定义是（见文件《pgfmathfunctions.misc.code.tex》)：

```
\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}%
```

可见函数 min 采用的是逐项比较的办法来确定最小值的。
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。
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
```

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。

下面的键（key）用于自定义函数，可以用在 Ti*k*Z 中。

`/pgf/declare function=`⟨*function definitions*⟩ (no default)

这个键用来自定义一个局部有效的、不太复杂的函数，先看一个例子：

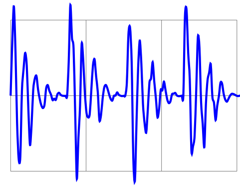
```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw [blue, thick, x=0.0085cm, y=1cm,
    declare function={
      sines(\t,\a,\b)=1 + 0.5*(sin(\t)+sin(\t*\a)+sin(\t*\b));
    }]
    plot [domain=0:360, samples=144, smooth] (\x,{sines(\x,3,5)});
\end{tikzpicture}
```

- ⟨*function definitions*⟩ 中定义一个函数所用的格式是:

  ⟨*name*⟩`(`⟨*arguments*⟩`)=`⟨*definition*⟩`;`

  注意其中末尾的分号。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}
```

97 输出数值的格式

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。

97.1 基本的命令与选项

`\pgfmathprintnumber[`⟨*options*⟩`]{`⟨*x*⟩`}`

注意在文件《pgfmathfloat.code.tex》中定义此命令，此命令要比 `\pgfmathprint`→P. 592 复杂很多。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`→P. 615 以及 `/pgf/number format/std` 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. 616.

/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。

$\frac{1}{3}$ $\frac{1}{2}$ $\frac{16}{75}$ $\frac{3}{25}$ $\frac{2}{75}$ $-\frac{1}{75}$ $\frac{18}{25}$ $\frac{1}{15}$ $\frac{2}{15}$ $-\frac{1}{75}$ $3\frac{1}{3}$ $1\frac{22657}{96620}$ 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{1.3333333333333333e-01}\hspace{1em}
\pgfmathprintnumber{-1.3333333333333333e-02}\hspace{1em}
\pgfmathprintnumber{3.3333333}\hspace{1em}
\pgfmathprintnumber{1.2345}\hspace{1em}
\pgfmathprintnumber{1}\hspace{1em}
\pgfmathprintnumber{-6}
```

/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。

$\frac{3}{5}$, $\dfrac{3}{5}$, $\sqrt{3}5$,

$1^{3}5$, $1_{3}5$, $\ln_{3} 5$,

```
\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。

$\frac{3}{5}$, $\frac{1}{2}$,

$\frac{7}{12}$, $\dfrac{-12}{-20}$,

```
\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)

/pgf/number format/precision={⟨*number*⟩} (no default)

本选项针对尾数（mantissa），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}
```

/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}
```

97.2 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：

1,234.56
1234.56
$1,234.56$
1,234.56

```
\pgfmathprintnumber{1234.56} \par
\pgfmathprintnumber[set thousands separator={}]{1234.56} \par
\pgfmathprintnumber[set thousands separator={,}]{1234.56} \par
\pgfmathprintnumber[set thousands separator={{{,}}}]{1234.56}
```

/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.

$1.234\&123\&456\&7 \cdot 10^3$

```
\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 $e \geqslant$ ⟨*number*⟩ 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 $s\ m$ 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，其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。

$1.23 \times \text{拾}^1$; $1.23 \times \text{拾}^{-4}$

```
\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。

$1.23\quad\text{玩一下}_1^{1.23}\text{拾}^1$; $-1.23\quad\text{玩一下}_2^{1.23}\text{拾}^{-4}$

```
\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。

$1.05 \cdot 10^1$; $1 \cdot 10^1$; $1.01 \cdot 10^3$; $-1 \cdot 10^3$;

```
\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& $\cdot$ 10^{1}
1& $\cdot$ 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}
```

99 基本层（basic layer）概略

本节讲解 PGF 的基本层（basic layer），它以系统层（system layer）为基础而建立。系统层提供了绘图所必须的少量内容，基本层提供了大量命令来直接绘图。与系统层相比，基本层的绘图操作要便捷一些。

但是与前端层（例如 TikZ ）相比，基本层的绘图句法用起来也不是那么方便，因此基本层通常会被其它程序利用。例如，beamer 宏包利用基本层来扩展其功能。

基本层的设计主要包括以下内容：

1. 内核和模块。
2. 能在 TEX 中使用的绘图命令。
3. 以路径（path）为核心的构图方式。
4. 坐标变换矩阵。

99.1 内核和模块

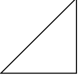
基本层的内核宏包叫作 pgfcore, 它由好几个子文件构成，提供大量基本命令。

用命令 \usepgfmodule 载入模块。

当执行命令：\usepackage{pgf} 或者 \input pgf.tex 或者 \usemodule[pgf] 时，模块 plot, 模块 shapes, 内核，以及系统层都会被载入。

99.2 基本层的宏

基本层提供大量的宏，通过这些宏来使用基本层的功能。这些宏都以 \pgf 开头，大部分宏都必须用在 {pgfpicture} 环境中。注意 {tikzpicture} 环境会开启一个 {pgfpicture} 环境，因此基本层的宏可以用在 {tikzpicture} 环境中。基本层的宏可以做很多事情，例如可以用命令使得当前点移动到别的地方，通过计算确定一个点，等等。



```
\newdimen\myypos  % 定义一个尺寸，没有给这个尺寸赋值，默认为 0pt.
\begin{pgfpicture}  % 开启 pgf 绘图环境
% 指定横标为 0cm，纵标为 \myypos 的一个点，将该点作为当前点并在该点开启一个路径。
  \pgfpathmoveto{\pgfpoint{0cm}{\myypos}}
% 指定横标为 1cm，纵标为 \myypos 的一个点，将当前点移动到该点并在移动时画线段。
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
% 将尺寸 \myypos 增加 1cm.
  \advance \myypos by 1cm%
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
  \pgfpathclose  % 使得路径成为闭合路径
  \pgfusepath{stroke}  % 画出刚才构造的路径，并结束该路径
\end{pgfpicture}  % 结束 pgf 绘图环境
```

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。

99.3 以路径为核心的构图方式

路径是 PGF 中最重要的概念。所有图形都是由一个或数个路径构成的。对路径可以进行多种操作，如画出（stroke），填充颜色（fill），带上阴影（shade），剪切（clipp）等等。一个路径可以是闭合的，非闭合的，自交的，或者是由数个不相连的部分构成。

路径先被构建，然后才能被“使用”。使用以 \pgfpath 开头的命令构建路径，每当使用这种命令，当前路径就会被延伸。当使用命令 \pgfusepath 时，就会结束刚才构建的路径，并且可以对刚才构建的路径进行某种操作，例如画出、填充、剪切等 (即“使用”路径)。

99.4 坐标变换与画布变换

PGF 提供两种变换：坐标变换与画布变换。PGF 本身具有坐标变换矩阵，PDF 和 PostScript 有画布变换矩阵。这两种变换很不一样。当对图形做画布变换时，图形中的一切要素，例如，线条的线宽，文字的笔画都会改变。打比方说，在一个气球上写下文字，画上图画，当气球充气时，文字的笔画、图画中的线条都会有变化。当对图形做坐标变换时，线条的线宽、文字的笔画（文字尺寸）都不会改变。

在默认下，只使用坐标变换。使用命令 \pgflowlevel 可以引入画布变换。

100 层级结构：宏包，环境，子环境，文字

100.1 Overview

PGF 有两种层级结构：一种是关于宏包的，即有的宏包会包含“子宏包”；另一种是关于绘图的，即绘图环境之内可以有“子环境”。

100.1.1 宏包的层级结构

PGF 由几个层次构成：

System layer. 这是最低的层，或者可以称之为“驱动层”、“后台层”，这个层次可以操作 .dvi 文件。系统层由宏包 pgfsys 实现，它能够按需要自动调用驱动文件。

Basic layer. 基本层由宏包 pgfcore 和数个模块组成，用命令 \usepgfmodule 载入模块。

Frontend layer. 前端层有多种，例如 Ti*k*Z, pgfpict2e, 都是基本层的前端。

每个层次都会自动加载所需要的文件。另外作为层次宏包的补充，还有许多程序库宏包，例如有的程序库宏包定义了多种样式的箭头，有的定义了绘图手柄。

100.1.2 图形的层级结构

一个图形可以具有层次，每个层次可以看作一个“组”，图形的各个要素被放入各个组中，可以对同一组中的要素做“统一处理”。例如可以把多个路径放入同一组中，并用红色线条画出它们，然后你

可以把红色线条改成蓝色线条——只需改动一个参数。

这里的"组"对应"域"（scope）这个概念。一般情况下，一个域内的参数只在该域内有作用。有多种"开启"、"关闭"一个域的方法，但是这些方法之间可能会有冲突，同时使用多种方法也会造成混乱。我们使用两种域，即"环境"和"TEX 分组"，规则如下：

1. PGF 的最外层的域是 {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，即 {pgfscope} 环境是"子环境"。

   注意 TEX 分组也是一种域，可以把 {pgfscope} 环境放入一个 TEX 分组内，也可以把一个 TEX 分组放入 {pgfscope} 环境内。凡是涉及"图形状态"（graphic state）的参数都接受 {pgfscope} 环境的限制，而不接受 TEX 分组的限制。有的图形参数（如箭头）、命令（如坐标变换），接受 TEX 分组的限制。

3. 注意 {pgfscope} 环境会自动创建一个 TEX 分组，因此接受 TEX 分组的限制的图形参数、命令也会受到 {pgfscope} 环境的限制。而且，不能将 {pgfscope} 环境与 TEX 分组交错使用。

4. 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 分组内，分别做变换。

5. 命令 \pgftext 会创建一个域，这个域是通常的 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。

100.2 宏包的层次

100.2.1 内核宏包

```
\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 TikZ），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。

100.2.2 模块

`\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`，因此你可以自己编辑一个这种文件，放在 TEX 能找到的地方，自己使用。

下面的模块能与 `pgfcore` 配合使用：

- `plot` 模块提供绘图命令，见 §112.
- `shapes` 模块提供绘图形状和 node，见 §106.
- `decorations` 模块提供装饰路径，见 §103.
- `matrix` 模块提供命令 `\pgfmatrix`，见 §107.

100.2.3 程序库宏包

程序库与模块的区别在于，程序库提供基本层的“附加”内容，基本层的功能可以作用于这些附加内容；而模块则提供新的功能。例如，程序库 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。

`\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`.

100.3 图形的层级

100.3.1 主要的环境

多数（不是全部）PGF 宏包的命令都要放在环境 `{pgfpicture}` 中。插入外部图形（如 `\pgfuseimage`）、创建阴影（如 `\pgfuseshading`）的命令要放在环境 `{pgfpicture}` 之外。但命令 `\pgfshadepath` 要放在环境内。

```
\begin{pgfpicture}
  ⟨environment content⟩
\end{pgfpicture}
```

这个环境会在当前位置插入一个 TEX 盒子，把该环境构造的图形放入这个盒子中。

边界盒子的尺寸。 盛放图形的盒子的尺寸用这种方式确定：当 PGF 解析 ⟨*environment contents*⟩ 时，会跟踪图形的边界盒子，也就是说，在处理绘图代码的同时，不断刷新图形的边界盒子。边界盒子内只包含绘图命令中“直接”涉及的坐标点，例如画圆的命令所画的圆。
当完成对 ⟨*environment contents*⟩ 的解析后，图形的边界盒子就确定了，此时 PGF 在页面的当前位置创建一个与边界盒子尺寸相同的 TEX 盒子，把图形放入这个 TEX 盒子中。

Hello ▭ World!

```
Hello \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture} World!
```

如果你希望能控制边界盒子的尺寸，可以使用命令 `\pgfusepath{use as bounding box}`，见 §104.6.

边界盒子的基线。 当一个图形插入到上下文中时，图形的基线通常位于图形的最底部。下面的例子中，两个矩形的长宽一样，尽管画出它们的纵坐标不同，但仍然处于同一水平线上：

Rectangles ▭ and ▭.

```
Rectangles \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture} and \begin{pgfpicture}
  \pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture}.
```

如果想控制图形基线的位置，可以使用命令 `\pgfsetbaseline`，例如：

Rectangles ▭ and ▭.

```
Rectangles \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
  \pgfsetbaseline{0pt}  % 将直线 y=0pt 作为基线
\end{pgfpicture} and \begin{pgfpicture}
  \pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
  \pgfsetbaseline{0pt}
\end{pgfpicture}.
```

上面例子中，命令 `\pgfsetbaseline` 是针对整个图形的，所以它可以用于 `\pgfusepath` 之后。

在图形中插入文字、外部图形。 不能在 `{pgfpicture}` 环境中直接插入文字，或者直接使用命令 `\includegraphics` 或 `\pgfimage` 插入外部图形。应当将文字和插入图形的命令作为 `\pgftext` 的参数，将它们插入到 `{pgfpicture}` 环境中，从而添加到图形中。

记住一个图形的位置并在后文中引用。 当一个图形编辑完成后，PGF 和 TEX 会忘记它在页面中的位置。这意味着你不能在后文中引用前面图形中的坐标点，不能在前文图形中的 node 与后文图形中的 node 之间连线，尽管有时候你想这么做。

为了让 PGF 记住一个图形的位置，需要将 TeX-if 判断命令 `\ifpgfrememberpicturepositiononpage` 的真值设为 true. 这个命令一定要用在 `{pgfpicture}` 环境内，且在环境的结尾处，而不是环境的开头处。将这个 TeX-if 的真值设为 true 后，PGF 就会记住该图形的位置。也可以全局地将这个 TeX-if 的真值设为 true, 那么 PGF 就会记住文档中所有图形的位置。

在默认下，这个 TeX-if 的真值是 false, 这是因为：第一，有的驱动不支持这一功能，目前这一功能只与 pdfTeX 配合工作；第二，当使用这一功能时，需要编译 TeX 两次才会得到跨图连线的效果，在第一次编译时，会搞乱所有图形的位置；第三，每记住一个图形，就会在 `.aux` 文件中添加一行代码。尽管有这些不便之处，当文档中的图形比较少时，全局地开启这一功能不会导致 TeX 变得很慢。

`\pgfpicture`
⟨*environment contents*⟩
`\endpgfpicture`

这是在 plain TeX 中使用的绘图环境，这个版本也会开启一个 TeX 分组。

`\startpgfpicture`
⟨*environment contents*⟩
`\stoppgfpicture`

这是在 ConTeXt 中使用的绘图环境。

`\ifpgfrememberpicturepositiononpage`

这个条件判断命令的真值决定是否让 PGF 记住图形在页面中的位置。注意，这个命令一定要用在 `{pgfpicture}` 环境内，且在环境的结尾处，而不是环境的开头处。使用这个选项后，尽管 PGF 能记住图形在页面中的位置，但图形中的坐标还是不能直接引用，目前只能引用图形中关于 node 的位置。见 §106.3.2，§17.13.

`\pgfsetbaseline{`⟨*dimension*⟩`}`

在默认下，图形的基线在图形的底部。本命令将直线 $y = \langle dimension \rangle$ 作为图形的基线，注意 ⟨*dimension*⟩ 是带单位的尺寸。

`\pgfsetbaselinepointnow{`⟨*point*⟩`}`

直接指定图形的基线为过点 ⟨*point*⟩ 的水平线。

`\pgfsetbaselinepointlater{`⟨*point*⟩`}`

直接指定图形的基线为过点 ⟨*point*⟩ 的水平线，不过 ⟨*point*⟩ 在绘图过程的结尾处确定下来。

Hello world.

```
Hello
\begin{pgfpicture}
  \pgfsetbaselinepointlater{\pgfpointanchor{X}{base}}
  ↪  % 注意此时还没有点 X
% 创建一个 node, 名称为 X, 内容是 "world.", 形状为叉号 cross out,
% 叉号位于文字的中心 center, 并且画出叉号
  \pgfnode{cross out}{center}{world.}{X}{\pgfusepath{stroke}}
\end{pgfpicture}
```

100.3.2 绘图子环境

\begin{pgfscope}
⟨*environment content*⟩
\end{pgfscope}

该环境内的“图形状态”参数只在该环境内起作用。“图形状态”参数包括以下内容：

- 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 分组的限制。当然，{pgfscope} 环境创建一个 TEX 分组。

```
\begin{pgfpicture}
  \begin{pgfscope}
    {  % 开启一个 TeX 分组
      \pgfsetlinewidth{2pt}  % 图形状态参数
      \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{2ex}}
      \pgfusepath{stroke}  % 画出刚才的矩形
    }  % 结束 TeX 分组
    \pgfpathrectangle{\pgfpoint{3ex}{0ex}}{\pgfpoint{2ex}{2ex}}
    \pgfusepath{stroke}
  \end{pgfscope}
  \pgfpathrectangle{\pgfpoint{6ex}{0ex}}{\pgfpoint{2ex}{2ex}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

```
\begin{pgfpicture}
  \begin{pgfscope}
    {
      \pgfsetarrows{-to}
      \pgfpathmoveto{\pgfpointorigin}\pgfpathlineto{\pgfpoint{2ex}{2ex}}
      \pgfusepath{stroke}
    }
    \pgfpathmoveto{\pgfpoint{3ex}{0ex}}\pgfpathlineto{\pgfpoint{5ex}{2ex}}
    \pgfusepath{stroke}
  \end{pgfscope}
  \pgfpathmoveto{\pgfpoint{6ex}{0ex}}\pgfpathlineto{\pgfpoint{8ex}{2ex}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

```
\begin{pgfpicture}
  \pgfsetarrows{-Stealth}
  \begin{pgfscope}
    \pgfpathmoveto{\pgfpointorigin}\pgfpathlineto{\pgfpoint{2ex}{2ex}}
    \pgfusepath{stroke}
  {
    \pgfsetarrows{-to}
    \pgfpathmoveto{\pgfpoint{3ex}{0ex}}\pgfpathlineto{\pgfpoint{5ex}{2ex}}
    \pgfusepath{stroke}
  }
  \end{pgfscope}
  \pgfpathmoveto{\pgfpoint{6ex}{0ex}}\pgfpathlineto{\pgfpoint{8ex}{2ex}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

注意，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。

`\pgfscope`
⟨*environment contents*⟩
`\endpgfscope`

这是 Plain TeX 的用法。

`\startpgfscope`
⟨*environment contents*⟩
`\stoppgfscope`

这是 ConTeXt 的用法。

`\begin{pgfinterruptpath}`
⟨*environment content*⟩
`\end{pgfinterruptpath}`

在构建路径的过程中插入这个环境，可以把当前路径的构建状态暂时封存起来，待本环境结束后，再调出封存的路径状态，继续构建路径。本环境不会开启 {pgfscope} 环境，也不会调用任何 \pgfsys 命令。

`\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，它会暂时打断当前环境，可以在 {pgfinterruptpicture} 环境中插入一个新的 {pgfpicture} 环境。但这个环境不能直接放到 {pgfpicture} 环境中。本环境必须放入一个 TEX 盒子中，然后将该盒子作为命令 \pgfqbox 的参数，引入 {pgfpicture} 环境中。

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。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。

\pgfinterruptboundingbox
⟨*environment contents*⟩
\endpgfinterruptboundingbox

这是 Plain TeX 的用法。

\startpgfinterruptboundingbox
⟨*environment contents*⟩
\stoppgfinterruptboundingbox

这是 ConTeXt 的用法。

100.3.3 插入文字和图形

将文字或者外部图形作为命令 \pgftext 的参数，可以插入到 {pgfpicture} 环境中。

\pgftext[⟨*options*⟩]{⟨*text*⟩}

这个命令会开启一个盒子，在这个盒子内部是通常的 TeX 状态，本命令把 ⟨*text*⟩ 放入盒子中，因此 ⟨*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。

坐标变换命令对此命令插入的 ⟨*text*⟩ 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）。

/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：

lovely

```
\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。

lovely

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[base] {lovely}}
```

以上选项可以配合使用，例如：

lovely

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[bottom,right] {lovely}}
```

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*⟩.

lovely

```
\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。

lovely

```
\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：

lovely

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[rotate=30,left,x=-1cm,y=-0.5cm,] {lovely}}
```

lovely

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
  \pgftext[left,x=-1cm,y=-0.5cm,rotate=30,] {lovely}}
```

100.4 Object Identifiers

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*⟩.

100.4.1 创建图形对象的命令

下面的系统层命令可以用 id 创建对象：

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。

以上命令可以被以下命令调用：

- \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, 这个环境具有 id, 其 id 用命令 \pgfuseid 添加。

\pgfidscope
⟨*environment contents*⟩
\endpgfidscope

PlainTeX 中的环境。

\startpgfidscope
⟨*environment contents*⟩
\stoppgfidscope

ConTeXt 中的环境。

100.4.2 设置、引用 identifier

\pgfuseid{⟨*name*⟩}

⟨*name*⟩ 是一串符号，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 分组中。实际上，在内部处理过程中，系统层命令 \pgfsys@new@id 会为这个 graphic object 创建一个“内部”id, 即 system layer identifier. 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.

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。

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; 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.
- 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。

在 Ti*k*Z 中：

- 当一个路径使用了 name 选项时，它可以有 .path 类型（针对该路径）。
- 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 fill, 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。

\pgfpoptype

将保存在一个内部的、全局定义的栈的顶层的类型调出。

\pgfclearid

将 local scope 中的当前的 id 以及 type 都清除掉。

\pgfidrefnextuse{⟨*macro*⟩}{⟨*name*⟩}

本命令将 ⟨*name*⟩, 即一个 system layer identifier 保存到 ⟨*macro*⟩ 中。这里的 ⟨*name*⟩ 是即将被用作 \pgfuseid 的参数的 id 名称（也就是说还没有用实际使用的名称）。

\pgfidrefprevuse{⟨*macro*⟩}{⟨*name*⟩}

本命令将之前最近使用的 id 名称 ⟨*name*⟩ 保存到 ⟨*macro*⟩ 中。

\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*⟩}

如果 ⟨*name*⟩ 被索引到了，就执行 ⟨*then code*⟩, 否则执行 ⟨*else code*⟩.

100.5 Resource Description Framework Annotations (RDFa)

100.6 错误信息与警告

\pgferror{⟨*message*⟩}

本命令可以中断文档的处理，显示错误信息 ⟨*message*⟩.

\pgfwarning{⟨*message*⟩}

本命令不中断文档的处理，但会显示错误信息 ⟨*message*⟩.

以上两个命令参考文件《pgfutil-common.tex》。

101 指定坐标

101.1 Overview

在 {pgfpicture} 环境中提供的坐标只在本环境中有效，一般情况下，你不能引用页面上的一个绝对位置。使用命令 \pgfpoint 指定一个点。

有的命令不必用在 {pgfpicture} 环境中，例如：

3.0pt
5.0pt

```
\newlength{\aaa} \newlength{\bbb} \newlength{\ccc}
\pgfmathsetlength{\aaa}{1pt} \pgfmathsetlength{\bbb}{2pt}
\pgfmathsetlength{\ccc}{3pt}
\pgfpointadd{\pgfpoint{\aaa}{\bbb}}{\pgfpoint{\bbb}{\ccc}}
\makeatletter
\the\pgf@x\\ \the\pgf@y
\makeatother
```

101.2 基本的坐标命令

\pgfpoint{⟨*x coordinate*⟩}{⟨*y coordinate*⟩}

这个命令产生一个点，⟨*x coordinate*⟩ 和 ⟨*y coordinate*⟩ 都是 TEX 尺寸，可以是带有长度单位的函数表达式，如 1cm + 2pt * cos(30). 如果 ⟨*x coordinate*⟩ 或 ⟨*y coordinate*⟩ 中的表达式不带长度单位，那么就默认其长度单位是 pt.

在文件《pgfcorepoints.code.tex》中此命令的定义是：

```
\def\pgfpoint#1#2{%
  \pgfmathsetlength\pgf@x{#1}%
  \pgfmathsetlength\pgf@y{#2}\ignorespaces}
```

可见这个定义主要使用了 \pgfmathsetlength→P. 593.

`\pgfqpoint`{⟨*x dimen expression*⟩}{⟨*y dimen expression*⟩}

此命令直接把 ⟨*x dimen expression*⟩ 赋予 `\pgf@x`; 直接把 ⟨*y dimen expression*⟩ 赋予 `\pgf@y`. 此命令的这两个赋值是全局赋值。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
```

在文件《pgfcorepoints.code.tex》中此命令的定义是:

```
\def\pgfqpoint#1#2{\global\pgf@x=#1\relax\global\pgf@y=#2\relax}
```

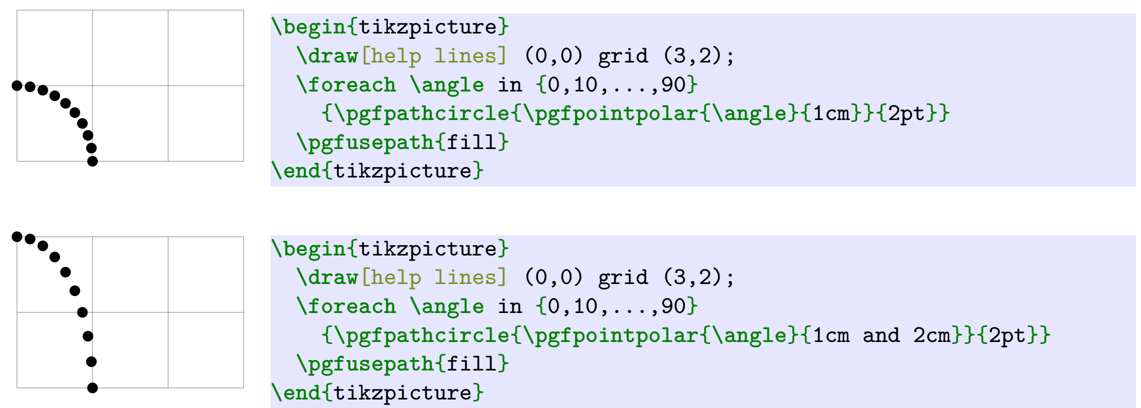
`\pgfpointorigin`

原点，等效于 `\pgfpoint{0pt}{0pt}`.

`\pgfpointpolar`{⟨*degree*⟩}{⟨*radius*⟩ and ⟨*y-radius*⟩}

这个命令产生极坐标点,其中 ⟨*degree*⟩ 是角度制下的数值,目前还不支持弧度制. 其中的 `and` ⟨*y-radius*⟩ 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*⟩.

观察下面的例子:



```
\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}
```

101.3 XY-坐标系统中的坐标

在通常情况下，x 轴的单位向量方向水平向右，长度是 1cm, y 轴的单位向量方向竖直向上，长度是 1cm. 可以重设坐标轴的单位向量。

`\pgfpointxy`{⟨s_x⟩}{⟨s_y⟩}

该命令确定一个点，这个点在 x 轴的分量等于 ⟨s_x⟩ 乘以 x 轴的单位向量，在 y 轴的分量等于 ⟨s_y⟩ 乘以 y 轴的单位向量。所以 ⟨s_x⟩ 和 ⟨s_y⟩ 都是数值，可以是数学表达式。

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfsetxvec{⟨point⟩}`

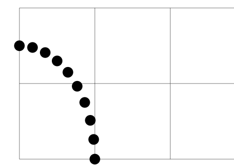
将点 ⟨*point*⟩ 作为 xyz 坐标系统的 x 轴的单位向量。

`\pgfsetyvec{⟨point⟩}`

将点 ⟨*point*⟩ 作为 xyz 坐标系统的 y 轴的单位向量。

`\pgfpointpolarxy{⟨degree⟩}{⟨radius⟩ and ⟨y-radius⟩}`

这个命令产生一个极坐标点，这里 ⟨*radius*⟩ 和 ⟨*y-radius*⟩ 都是数值，不是尺寸，是与坐标轴的单位向量相乘的因子。and ⟨*y-radius*⟩ 是可选的。



```
\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}
```

101.4 三维坐标

`\pgfpointxyz{⟨s_x⟩}{⟨s_y⟩}{⟨s_z⟩}`

这个命令产生一个点，该点的 x 轴分量是 $\langle s_x\rangle$ 与 x 轴单位向量的乘积，该点的 y 轴分量是 $\langle s_y\rangle$ 与 y 轴单位向量的乘积，该点的 z 轴分量是 $\langle s_z\rangle$ 与 z 轴单位向量的乘积。

```
\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}
```

`\pgfsetzvec{⟨point⟩}`

将点 ⟨*point*⟩ 作为 xyz 坐标系统的 z 轴的单位向量，这里 ⟨*point*⟩ 是二维点的形式。

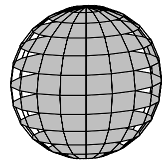
`\pgfpointcylindrical{⟨degree⟩}{⟨radius⟩}{⟨height⟩}`

这个命令产生一个圆柱坐标系点，等效于

```
\pgfpointadd{\pgfpointpolarxy{⟨degree⟩}{⟨radius⟩}}{\pgfpointxyz{0}{0}{⟨height⟩}}
```

`\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。



```
\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}
```

101.5 用已有坐标构建新的坐标

101.5.1 基本的坐标计算

`\pgfpointadd{⟨v1⟩}{⟨v2⟩}`

本命令得到向量 $\langle v_1\rangle$ 与 $\langle v_2\rangle$ 的和。

`\pgfpointscale{⟨factor⟩}{⟨coordinate⟩}`

本命令得到数值 ⟨*factor*⟩ 与向量 ⟨*coordinate*⟩ 的乘积。

`\pgfpointdiff{⟨start⟩}{⟨end⟩}`

本命令得到向量 ⟨*end*⟩ 减去 ⟨*start*⟩ 的差，即“终点”减去“起点”。

`\pgfpointnormalised{⟨point⟩}`

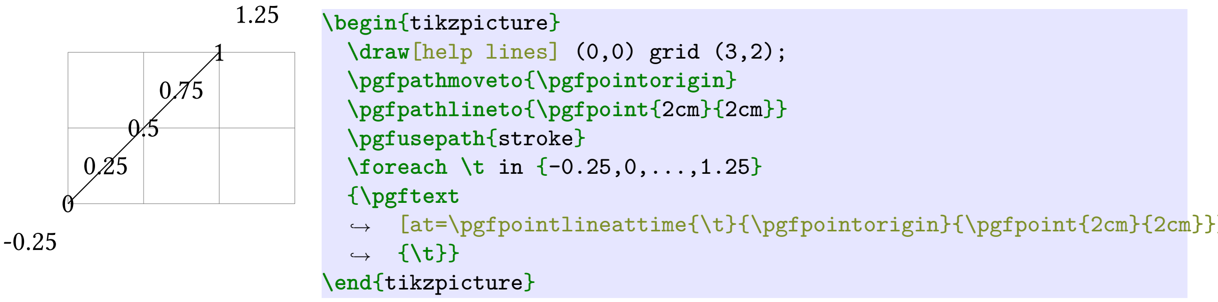
本命令将向量 ⟨*point*⟩ 单位化，即方向不变，长度变成 1pt. 如果 ⟨*point*⟩ 是 0 向量或者是长度极短的向量，那么本命令得到的是方向竖直向上，长度是 1pt 的向量。

101.5.2 直线或曲线上的点

设想一个点在直线或曲线上移动，在时刻 $t=0$，该点位于 p；在时刻 $t=1$，该点位于 q；在时刻 $t=\frac{1}{2}$，该点位于 p 与 q 之间的某个位置，具体位置取决于该点的“速度”变化，详细内容参考关于 Bézier 曲线的资料。

`\pgfpointlineattime`{⟨*time t*⟩}{⟨*point p*⟩}{⟨*point q*⟩}

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 $p + t(q - p)$. 如果 t 小于 0 或大于 1, 则本命令得到的点将处于线段 pq 之外。




```
\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*⟩}

⟨*distance*⟩ 是带单位的尺寸。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。

70pt
40pt
20pt
0pt
-20pt

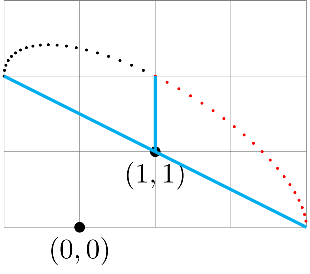
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \d in {-20pt,0pt,20pt,40pt,70pt}
  {\pgftext
  ↪   [at=\pgfpointlineatdistance{\d}{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}]
  ↪   {\d}}
\end{tikzpicture}
```

`\pgfpointarcaxesattime`{⟨*time t*⟩}{⟨*center*⟩}{⟨*0-degree axis*⟩}{⟨*90-degree axis*⟩}{⟨*start angle*⟩}{⟨*end angle*⟩}

设想一个椭圆 E, 初始下，E 的中心在原点，以 $(1, 0)$ 为横半轴，以 $(0, 1)$ 为纵半轴，以逆时针方向为角度的正方向。以原点为始点，以 ⟨*start angle*⟩ 为方向的射线 l_1 交椭圆于点 P; 以原点为始点，以 ⟨*end angle*⟩ 为方向的射线 l_2 交椭圆于点 Q. 假设一个动点在 E 上匀速运动，在时刻 $t = 0$ 时，动点位于点 P, 在时刻 $t = 1$ 时，动点位于点 Q, 那么在时刻 ⟨*time t*⟩ 时动点的位置 M 是容易确定的，因为此时的椭圆就是个圆。

对椭圆、射线、各个点做变换：先将原点平移到点 ⟨*center*⟩, 然后做一个仿射变换，使得椭圆的“横半轴”：$(1, 0)$ 变成向量 ⟨*0-degree axis*⟩; 椭圆的“纵半轴”：$(0, 1)$ 变成向量 ⟨*90-degree axis*⟩. 这样得到椭圆 E', 射线 l'_1, l'_2, 射线与椭圆的交点 P', Q', 以及 M', 这个点 M' 就是本命令确定的点。

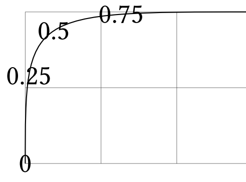
在这个变换下，椭圆以 ⟨*center*⟩ 为中心，以 ⟨*0-degree axis*⟩ 为度量角度的起始方向，以自 ⟨*0-degree axis*⟩ 旋转到 ⟨*90-degree axis*⟩ 的角度为正。




```
\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}
```

101.5.3 矩形或椭圆边界上的点

`\pgfpointborderrectangle`{⟨*direction point*⟩}{⟨*corner*⟩}

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，那么程序会把它“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（方向偏差越小）。

`\pgfpointborderellipse`{⟨*direction point*⟩}{⟨*corner*⟩}

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。

101.5.4 两直线的交点

`\pgfpointintersectionoflines`{⟨*p*⟩}{⟨*q*⟩}{⟨*s*⟩}{⟨*t*⟩}

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。

101.5.5 两个圆的交点

`\pgfpointintersectionofcircles`{⟨p_1⟩}{⟨p_2⟩}{⟨r_1⟩}{⟨r_2⟩}{⟨*solution*⟩}

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。

101.5.6 两个路径的交点

首先调用程序库 intersections.

TikZ Library `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
```

这个程序库计算两个路径的交点。限于 TEX 的精度，路径不能太复杂，尤其不要包含很多小线段，如装饰路径。

`\pgfintersectionofpaths`{⟨*path 1*⟩}{⟨*path 2*⟩}

本命令计算路径 ⟨*path 1*⟩ 和 ⟨*path 2*⟩ 的交点，将交点保存起来，可供稍后引用。其中 ⟨*path 1*⟩ 和 ⟨*path 2*⟩ 的代码被放入 TEX 分组中处理，代码中可以使用坐标变换命令。在 ⟨*path 1*⟩ 和 ⟨*path 2*⟩ 的代码中不要使用命令 `\pgfusepath{...}`, 否则无法计算两个路径的交点，除非在命令 `\pgfusepath{...}` 之前将路径另存，并在命令 `\pgfusepath{...}` 后调出路径（见下面的例子）。PGF 会对计算出来的交点做特殊处理，使交点位置“绝对化”，不再接受变换命令的作用。

`\pgfintersectionsolutions`

使用命令 `\pgfintersectionofpaths` 后，所得到的交点的个数会被保存在这个宏中。

`\pgfpointintersectionsolution`{⟨*number*⟩}

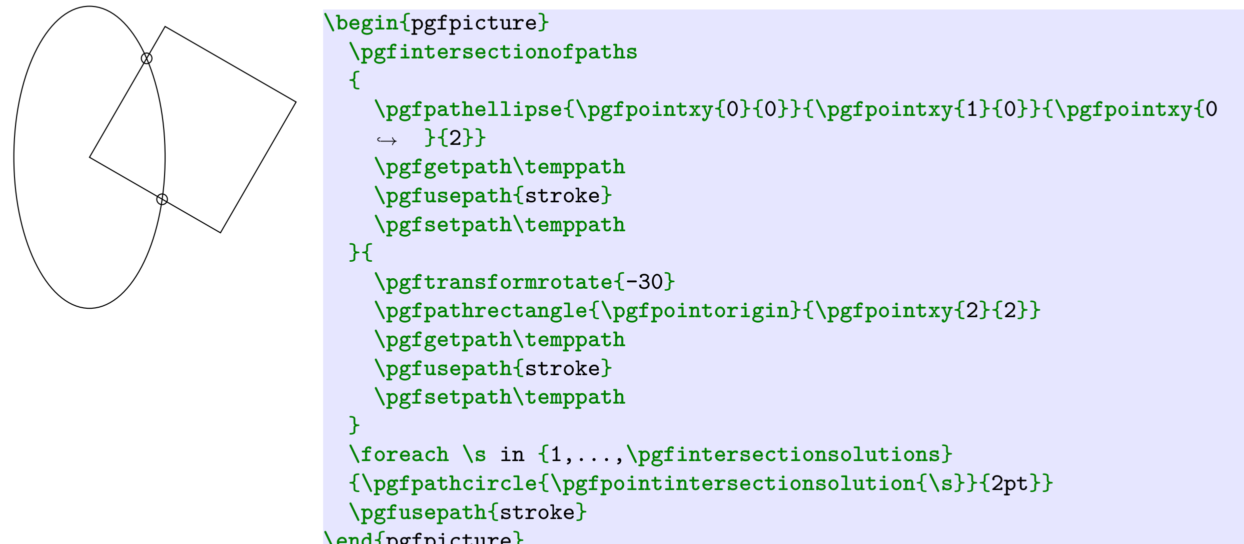
这里 ⟨*number*⟩ 是正整数。使用命令 `\pgfintersectionofpaths` 后，本命令确定第 ⟨*number*⟩ 个交点。如果第 ⟨*number*⟩ 个交点找不到，就返回原点。程序按照交点被找到的次序来规定各个交点的序号，但是寻找交点的算法比较复杂，所得到的交点次序也不易直观理解。

`\pgfintersectionsortbyfirstpath`

使用命令 `\pgfintersectionofpaths` 后，本命令重排交点的次序，使得交点按照第一个路径的方向排序，即用第一个路径像穿珠子似的将交点串起来。

`\pgfintersectionsortbysecondpath`

使用命令 \pgfintersectionofpaths 后，本命令使得交点按照第二个路径的方向排序。



```
\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}
```

在上面例子中用到了命令 \pgfgetpath 和 \pgfsetpath, 在文件《pgfcorepathconstruct.code》中对这两个命令的定义是:

```
\let\pgfgetpath=\pgfsyssoftpath@getcurrentpath
\let\pgfsetpath=\pgfsyssoftpath@setcurrentpath
```

命令 \pgfsyssoftpath@getcurrentpath⟨*macro*⟩ 的作用是将当前的软路径（soft path）保存到宏 ⟨*macro*⟩ 中；命令 \pgfsyssoftpath@setcurrentpath⟨*macro*⟩ 的作用是使得保存在宏 ⟨*macro*⟩ 中的软路径成为当前软路径。命令 \pgfusepath 会把软路径变成硬路径 (hard path), 不能计算两个硬路径的交点，所以如果把上面例子中的命令 \pgfgetpath 和 \pgfsetpath 都删去就得不到两个路径的交点。

101.6 坐标分量

可以得到一个坐标点的 x 分量和 y 分量。

`\pgfextractx`{⟨*dimension*⟩}{⟨*point*⟩}

⟨*dimension*⟩ 是个已经声明的 TEX 宏，本命令将点 ⟨*point*⟩ 的 x 分量，即一个长度尺寸（单位转换为 pt）赋予宏 ⟨*dimension*⟩.

注意：(1)⟨*dimension*⟩ 必须提前声明；(2) 本命令可以不用在 {pgfpicture} 环境中。

56.9055pt

```
\newdimen\mydim
\pgfextractx{\mydim}{\pgfpoint{2cm}{4pt}}
\the\mydim
```

`\pgfextracty`{⟨*dimension*⟩}{⟨*point*⟩}

⟨*dimension*⟩ 是个已经声明的 TEX 宏，本命令将点 ⟨*point*⟩ 的 y 分量，即一个长度尺寸（单位转换为 pt）赋予宏 ⟨*dimension*⟩.

注意：(1)⟨*dimension*⟩ 必须提前声明；(2) 本命令可以不用在 {pgfpicture} 环境中。

`\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 分量分别赋予宏 ⟨*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’ .
```

101.7 坐标点命令的工作方式

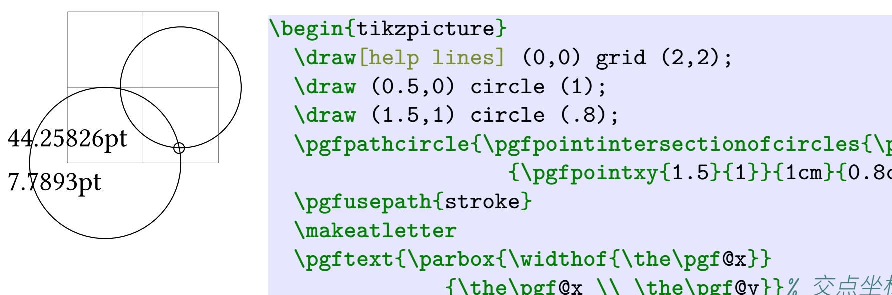
执行定义坐标点的命令，如 \pgfpoint{1cm}{2pt}, 会把 TeX 尺寸寄存器 \pgf@x 和 \pgf@y 分别赋值为 1cm 和 2pt. 这两个寄存器是分配给 PGF 的寄存器，参考 §117. 以坐标点为参数的命令，例如 \pgfpathmoveto, 会先确定 \pgf@x 和 \pgf@y 的值，然后将画笔移动到指定的坐标位置。

当给定一个坐标点，或者以某一组命令确定一个坐标点后，\pgf@x 和 \pgf@y 的值就是当前坐标点的分量值（带单位的尺寸）。例如：

28.45274pt
2.0pt

```
\begin{tikzpicture}
  \draw[help lines] (-1,-1) grid (1,1);
  \pgfmoveto{\pgfpoint{1cm}{2pt}}
  \makeatletter
  \pgftext[top]{\parbox{\widthof{\the\pgf@x}}
                {\the\pgf@x \\ \the\pgf@y}}
  \makeatother
\end{tikzpicture}
```

再如：




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \draw (0.5,0) circle (1);
  \draw (1.5,1) circle (.8);
  \pgfpathcircle{\pgfpointintersectionofcircles{\pgfpointxy{.5}{0}}
                {\pgfpointxy{1.5}{1}}{1cm}{0.8cm}{1}} {2pt}
  \pgfusepath{stroke}
  \makeatletter
  \pgftext{\parbox{\widthof{\the\pgf@x}}
           {\the\pgf@x \\ \the\pgf@y}}% 交点坐标
  \makeatother
\end{tikzpicture}
```

`\pgf@process{⟨code⟩}`

这是个使用率很高的内部命令，在文件《pgfcorepoints.code.tex》中对这个命令的定义是：

```
\def\pgf@process#1{{#1\global\pgf@x=\pgf@x\global\pgf@y=\pgf@y}}
```

本命令执行 ⟨*code*⟩ 后，就全局性地给 \pgf@x 和 \pgf@y 赋了值。因为当前坐标点是经常变化的，故 \pgf@x 和 \pgf@y 的值也是频繁变化的。如果你需要临时使用尺寸寄存器来保存 \pgf@x 和 \pgf@y 的值，可以使用 \pgf@xa, \pgf@ya 等等，参考 §117.

下面是命令 \pgfpointadd 的定义（见文件《pgfcorepoints.code》)：

```
\def\pgfpointadd#1#2{%
  \pgf@process{#1}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{#2}%
  \advance\pgf@x by\pgf@xa%
  \advance\pgf@y by\pgf@ya}
```

102 构建路径

102.1 Overview

在 PGF 中，最基本的绘图概念是“路径”（path），一个路径可以由点、直线段、控制曲线组成，可以包含不相连的数个部分，可以是开的，也可以是闭的。

一个路径可以看作一个“点集”，构建一个路径就是指定一个点集。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，只有用某种标记符号来标记路径上的点时，路径才会显示出来。用“画笔”画出路径的操作是 stroking 或者 drawing, 画笔可以描点，也可以画出线条。用某种颜色填充路径的操作是 filling, 填充路径时程序会把路径当作是闭的。

在 PGF 中，创建路径的命令都以 `\pgfpath` 开头，使用路径的命令都以 `\pgfusepath` 开头。

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，具体参考 §102.13.

创建路径的命令会延伸“当前路径”。“当前路径”是一个全局概念，不受 TEX 分组的限制。因此在构建一个路径的过程中可以插入 TEX 分组来完成某些工作。当遇到命令 `\pgfusepath` 时当前路径才会结束，也有其它结束当前路径的方式，参考 §121.

102.2 Move-To 路径操作

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`. Move-To 操作常用于路径开头。

`\pgfpathmoveto`{⟨*coordinate*⟩}

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）。

本命令接受当前的坐标变换矩阵。一般情况下，本命令会刷新当前路径和图形的边界盒子。

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathmoveto{\pgfpoint{2cm}{1cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathmoveto{\pgfpoint{2cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

比较以上 3 个例子，可见填充一个路径时，程序会把路径的各个不相连部分（子路径）分别看作是“闭的”，对各个子路径分别填充颜色。

102.3 Line-To 路径操作

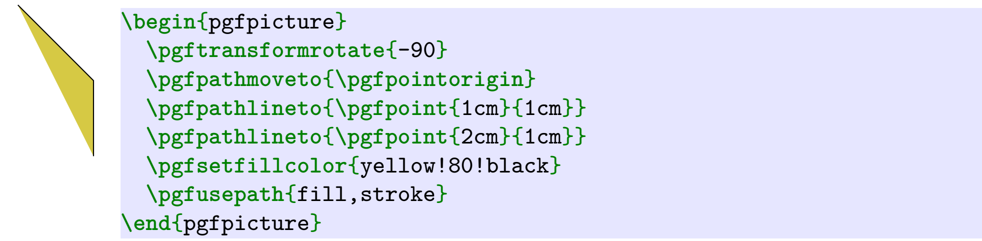
`\pgfpathlineto{`⟨*coordinate*⟩`}`

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。

注意本命令的操作需要事先有当前坐标点存在，因此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，而不以其它命令，如 move-to 操作开头，TeX 不会认为是个错误，但有的阅读器在渲染图形时，可能对此产生一个错误信息。如果以本命令开头构建“路径”，得到的不是真正的路径，使用命令 `\pgfusepath{stroke}` 也不能画出这个“路径”，但是这个假“路径”会被算入边界盒子内。类似这个命令，还有一些命令的操作需要事先有当前坐标点存在，这些命令都不能用于开启一个路径。

本命令会先将当前的坐标变换矩阵（如果本命令之前有坐标变换命令的话）用于 ⟨*coordinate*⟩, 然后确定当前点的位置。也就是说，如果本命令之前有坐标变换命令，那么输入的是⟨*coordinate*⟩, 得到的是变换 ⟨*coordinate*⟩ 之后的点。

一般情况下，本命令会刷新当前路径和图形的边界盒子。



```
\begin{pgfpicture}
  \pgftransformrotate{-90}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

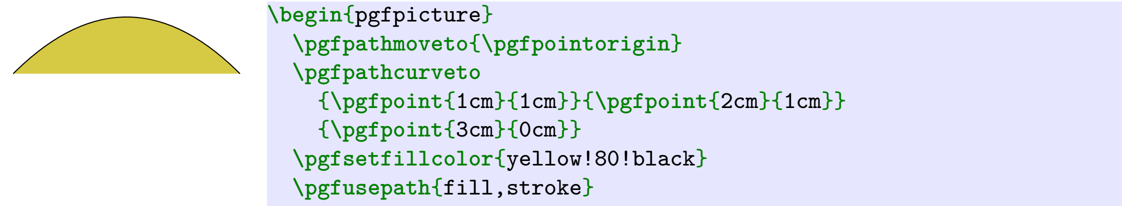
102.4 Curve-To 路径操作

`\pgfpathcurveto`{⟨*support 1*⟩}{⟨*support 2*⟩}{⟨*coordinate*⟩}

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(如果本命令之前有坐标变换命令的话)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}
```

`\pgfpathquadraticcurveto`{⟨*support*⟩}{⟨*coordinate*⟩}

本命令创建一段 2 次 Bézier 曲线来延伸当前路径。以本命令之前最近出现的点为始点, 以 ⟨*coordinate*⟩ 为终点，以 ⟨*support*⟩ 为控制点，构建一段 2 次控制曲线。在形式上本命令构造的是 2 次控制曲线，但实际上程序会自动重新选择控制点，把这个 2 次控制曲线转成 3 次控制曲线。而边界盒子会把重新选择的控制点包括在内，未必把 ⟨*support*⟩ 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，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。有一个动点在这段控制曲线上移动，在时刻 $t = 0$ 该动点位于始点 ⟨*point* p⟩, 在时刻 $t = 1$ 该动点位于终点 ⟨*point* q⟩.

本命令构建的曲线是动点在时刻 ⟨*time* t_1⟩ 到 ⟨*time* t_2⟩ 之间的轨迹。

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}
```

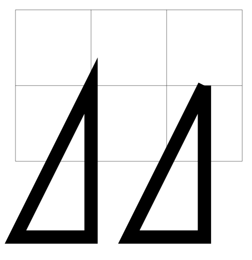
`\pgfpathcurvebetweentimecontinue`{⟨*time* t_1⟩}{⟨*time* t_2⟩}{⟨*point* p⟩}{⟨*point* s_1⟩}{⟨*point* s_2⟩}{⟨*point* q⟩}

类似上一个命令，只是本命令不把 move-to 操作作为自己的开头。

102.5 Close 路径操作

`\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}
```

102.6 Arc, Ellipse, Circle 路径操作

`\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。然后将这段弧平移，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。

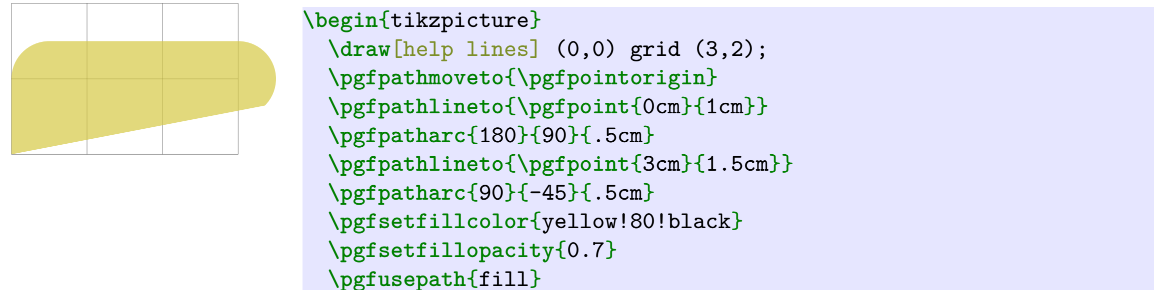
如果本命令之前有旋转变换，本命令先将弧围绕原点旋转，然后再将弧平移，使得弧的起点与当前坐标点重合。这种“绕原点旋转——平移”的效果，等于“平移——绕弧的始点旋转”。

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。

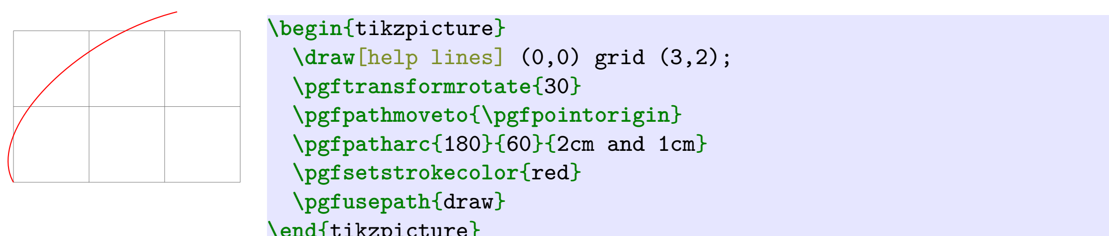
注意命令 `\pgfpatharc{0}{360}{1cm}` 给出的不是一个完整的圆，因为它不闭合，在它后面跟上命令 `\pgfpathclose` 使之闭合。

本命令会刷新边界盒子。

本命令的定义见文件《pgfcorepathconstruct.code》.



```
\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}
```

`\pgfpatharcaxes`{⟨*start angle*⟩}{⟨*end angle*⟩}{⟨*first axis*⟩}{⟨*second axis*⟩}

这个命令与上一个命令类似，以当前坐标点为始点构建一段椭圆弧。这里 ⟨*first axis*⟩ 与 ⟨*second axis*⟩ 都是坐标（向量），用于指定椭圆的横半轴和纵半轴。

假设一个单位圆 $x^2 + y^2 = 1$，在这个圆上取一段弧，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。然后做仿射变换，使得：$(1, 0)$ 变成向量 ⟨*first axis*⟩, $(0, 1)$ 变成向量 ⟨*second axis*⟩. 这样圆弧就变成了椭圆弧，然后将这段弧平移，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。这就是本命令的效果。

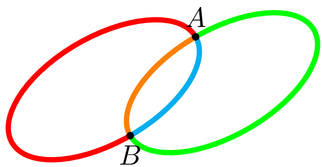
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2cm,5mm) (0,0) -- (0cm,1cm);
  \pgfpathmoveto{\pgfpoint{2cm}{5mm}}
  \pgfpatharcaxes{0}{90}{\pgfpoint{2cm}{5mm}}
                        {\pgfpoint{0cm}{1cm}}
  \pgfsetstrokecolor{red}
  \pgfusepath{draw}
\end{tikzpicture}
```

`\pgfpatharcto`{⟨*x-radius*⟩}{⟨*y-radius*⟩}{⟨*rotation*⟩}{⟨*large arc flag*⟩}{⟨*counterclockwise flag*⟩}{⟨*target point*⟩}

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，⟨*rotation*⟩ 是个代表角度的数值，⟨*large arc flag*⟩ 与 ⟨*counterclockwise flag*⟩ 是整数，⟨*target point*⟩ 是个坐标点。

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*⟩.

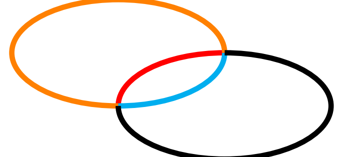
令 θ 等于角度 ⟨*rotation*⟩, 将椭圆旋转 θ 角度，然后再做一个平移变换，我们要求变换后的椭圆经过点 A，B. 如果能满足这个要求，那么一般会有两个（形状一样的）椭圆满足这个要求（两个椭圆可能重合，但仍然看作是两个）。下图展示了这样的两个椭圆：



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]
  \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。

`\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。

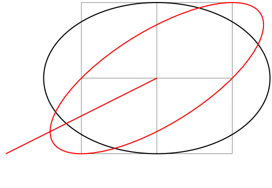
⟨*center point*⟩ 是椭圆弧所在椭圆的中心点。椭圆弧以当前的为起点，以 ⟨*end point*⟩ 为终点。⟨*start angle*⟩ 是从点 ⟨*center point*⟩ 到起点的方向角。⟨*end angle*⟩ 是从点 ⟨*center point*⟩ 到终点 ⟨*end point*⟩ 的方向角。⟨*x-radius*⟩ 是椭圆横半轴的长度（带单位）。⟨*y-radius*⟩ 是椭圆纵半轴的长度（带单位）。⟨*ratio x-radius/y-radius*⟩ 是椭圆横半轴长度与纵半轴长度的比值。⟨*ratio y-radius/x-radius*⟩ 是椭圆纵半轴长度与横半轴长度的比值。

`\pgfpatharctomaxstepsize`

`\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”，如果这两个向量不是正交的，则意味着其中有仿射变换。



```
\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}}
  \pgfpathlineto{\pgfpoint{-1cm}{0cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

如果本命令之前有坐标变换命令，那么本命令先按照参数构建椭圆，然后再把坐标变换施加于椭圆上的点，得到变换后的椭圆。

本命令会刷新边界盒子。

`\pgfpathcircle{⟨center⟩}{⟨radius⟩}`

本命令构建一个圆，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。如果 ⟨*radius*⟩ 是负值尺寸，则当作正值尺寸看待。

102.7 Rectangle 路径操作

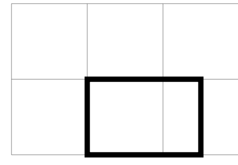
下面两个命令构建矩形，它们可以开启一个路径，或者延伸当前路径，它们构建的矩形是其所在路径的一个孤立部分，即所构建的矩形本身是闭的。

\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。

102.8 Grid 路径操作

网格是由平行于坐标轴的直线构成的，并且总是以原点为一个格点。

\pgfpathgrid[⟨*options*⟩]{⟨*first corner*⟩}{⟨*second corner*⟩}

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。当本命令结束时，当前坐标点是 ⟨*second corner*⟩.

本命令先创建网格，再对网格施加坐标变换矩阵（如果事先设置坐标变换命令的话）。

本命令刷新边界盒子。

⟨*options*⟩ 中可以使用以下选项:

/pgf/stepx=⟨*dimension*⟩ (no default, initially 1cm)

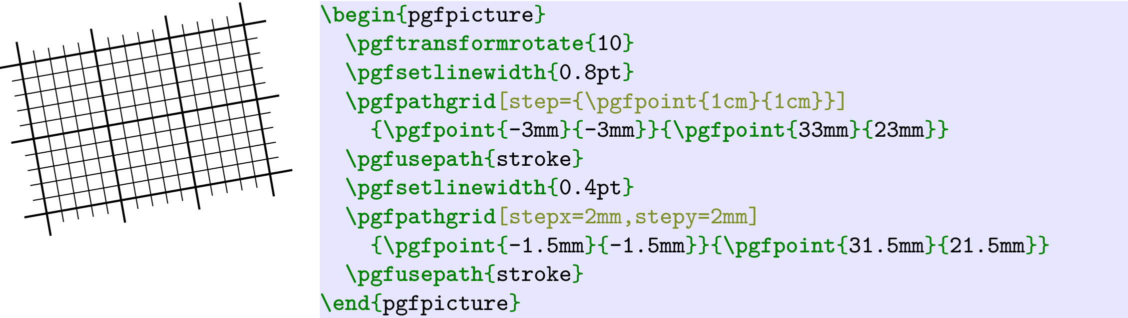
本选项指定网格的水平线之间的间距。

/pgf/stepy=⟨*dimension*⟩ (no default, initially 1cm)

本选项指定网格的竖直线之间的间距。

/pgf/step=⟨*vector*⟩ (no default)

将向量 ⟨*vector*⟩ 作为一个网格单元的对角向量。



```
\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}
```

102.9 Parabola 路径操作

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。

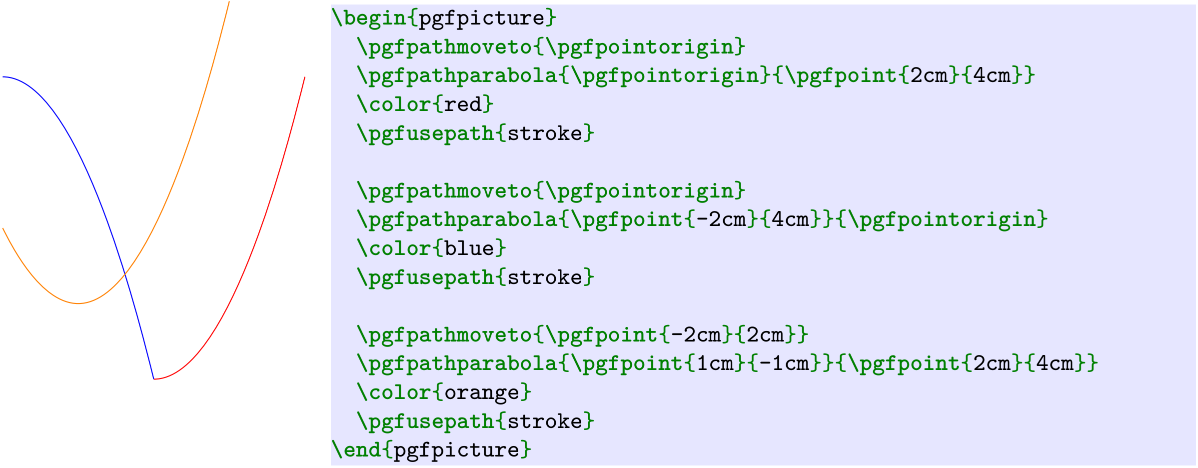
`\pgfpathparabola{`⟨*bend vector*⟩`}{`⟨*end vector*⟩`}`

本命令构建两段抛物线，第一段抛物线以当前坐标点为始点，以点 ⟨*bend vector*⟩ 为顶点以及终点；第二段抛物线以点 ⟨*bend vector*⟩ 为顶点以及始点，以 ⟨*end vector*⟩ 为终点。

如果 ⟨*end vector*⟩ 为空，则只构建第一段抛物线。如果 ⟨*bend vector*⟩ 为空，则本命令以当前坐标点为始点以及顶点，以 ⟨*end vector*⟩ 为终点构建一段抛物线。

不能用本命令构建一段不含顶点的抛物线。

本命令接受坐标变换矩阵，也会刷新边界盒子。



```
\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}
```

102.10 Sine 和 Cosine 路径操作

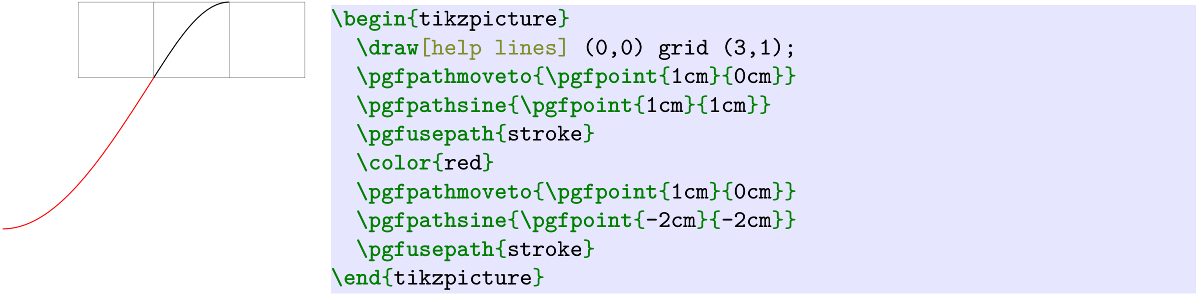
`\pgfpathsine{`⟨*vector*⟩`}`

本命令在当前坐标点与点 ⟨*vector*⟩ 之间构建一段正弦曲线，延伸当前路径。本命的效果相当于：令首先构建一段“标准的”正弦曲线路径：

$$\sin x, \quad x \in [0, \frac{\pi}{2}],$$

然后对这一段正弦曲线做平移变换、伸缩变换、上下对称变换，但不做左右对称变换，也不做旋转变换，使得正弦曲线上原来的点 $(0, \sin 0) = (0, 0)$ 变成当前坐标点，原来的点 $(\frac{\pi}{2}, \sin\frac{\pi}{2}) = (\frac{\pi}{2}, 1)$ 变成点 ⟨*vector*⟩.

本命令接受坐标变换矩阵，也会刷新边界盒子。



```
\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}
```

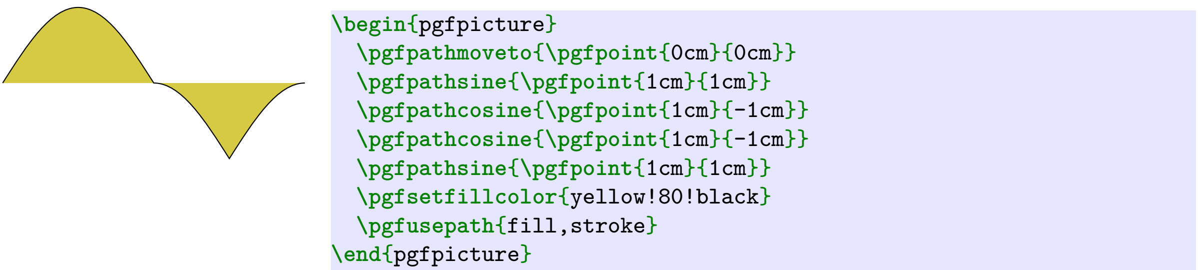
`\pgfpathcosine{`⟨*vector*⟩`}`

本命令在当前坐标点与点 ⟨*vector*⟩ 之间构建一段余弦曲线，延伸当前路径。本命令的效果相当于：首先构建一段“标准的”余弦曲线路径：

$$\cos x, \quad x \in [0, \frac{\pi}{2}],$$

然后对这一段余弦曲线做平移变换、伸缩变换、上下对称变换，但不做左右对称变换，也不做旋转变换，使得余弦曲线上原来的点 $(0, \cos 0) = (0, 1)$ 变成当前坐标点，原来的点 $(\frac{\pi}{2}, \cos\frac{\pi}{2}) = (\frac{\pi}{2}, 0)$ 变成点 ⟨*vector*⟩.

本命令接受坐标变换矩阵，也会刷新边界盒子。



```
\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}
```

102.11 Plot 路径操作

具体参考 §112.

102.12 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。

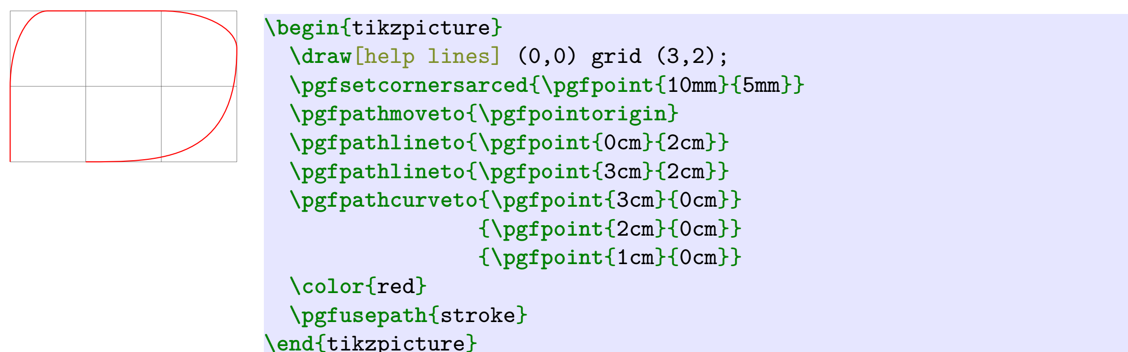
`\pgfsetcornersarced{⟨point⟩}`

本命令将其后的所有尖角变成圆角。注意圆角是“图形状态参数”。

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。

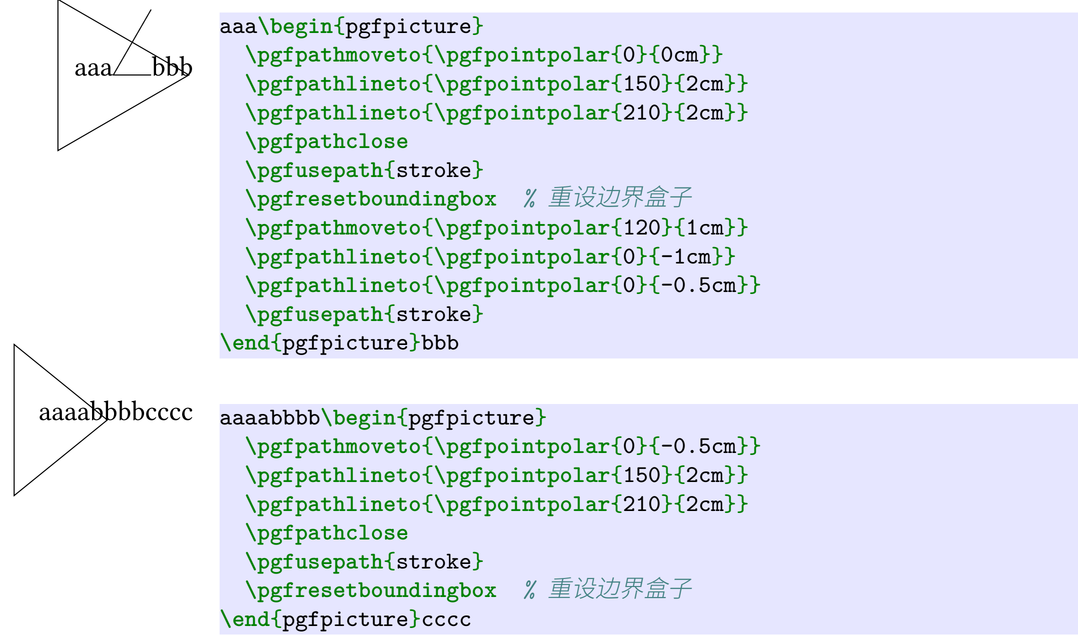
102.13 跟踪路径或图形的边界盒子

在构建路径时，程序会跟踪当前路径的边界盒子以及整个图形的边界盒子。

\pgfresetboundingbox

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
```

图形的边界盒子上的点不能用通常形式的宏来引用。下面的宏可以引用边界盒子上的点。

\pgf@pathminx

考虑当前路径中涉及的坐标点，这些点的 x 分量的最小值就是这个宏的值。这个宏的初始值是 16000pt.

16000.0pt

```
\tikz;
\makeatletter
\the\pgf@pathminx
\makeatother
```

\pgf@pathmaxx

考虑当前路径中涉及的坐标点，这些点的 x 分量的最大值就是这个宏的值。这个宏的初始值是 -16000pt.

-16000.0pt

```
\tikz;
\makeatletter
\the\pgf@pathmaxx
\makeatother
```

\pgf@pathminy

考虑当前路径中涉及的坐标点，这些点的 y 分量的最小值就是这个宏的值。这个宏的初始值是 16000pt.

\pgf@pathmaxy

考虑当前路径中涉及的坐标点，这些点的 y 分量的最大值就是这个宏的值。这个宏的初始值是 −16000pt.

\pgf@picminx

考虑当前图形中涉及的坐标点，这些点的 x 分量的最小值就是这个宏的值。这个宏的初始值是 −16000pt.

\pgf@picmaxx

考虑当前图形中涉及的坐标点，这些点的 x 分量的最大值就是这个宏的值。这个宏的初始值是 16000pt.

\pgf@picminy

考虑当前图形中涉及的坐标点，这些点的 y 分量的最小值就是这个宏的值。这个宏的初始值是 −16000pt.

\pgf@picmaxy

考虑当前图形中涉及的坐标点，这些点的 y 分量的最大值就是这个宏的值。这个宏的初始值是 16000pt.

每当使用构建路径的命令时，程序就会刷新以上 8 个宏的值，从而改变边界盒子。如果想手工控制边界盒子的尺寸和位置，可以使用下面的命令：

\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。本命令使得边界盒子包含这个点。使用该命令时可能需要配合下面的命令。

一个点或者一个路径是否计入边界盒子，由 TeX-if 命令：\ifpgf@relevantforpicturesize 决定，只有它的真值为 true 时，它之后的点或者路径才计入边界盒子。例如，当使用剪切路径时，程序把该命令的值设为 true，然后构建起“剪切”作用的路径，这个路径会被计入边界盒子；然后程序把该命令的值设为 false, 然后再构建那些“被剪切”的路径，那些被剪切路径不计入边界盒子。

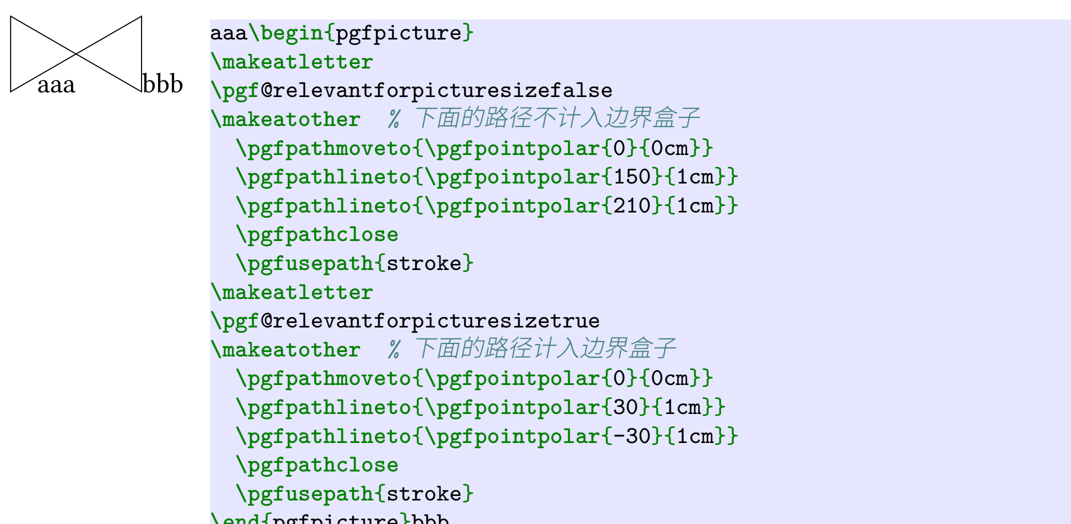
\pgf@relevantforpicturesizefalse

使用此命令后，此命令之后的路径不计入边界盒子。

```
aaa\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointpolar{0}{0cm}}
  \pgfpathlineto{\pgfpointpolar{150}{1cm}}
  \pgfpathlineto{\pgfpointpolar{210}{1cm}}
  \pgfpathclose
  \pgfusepath{stroke}
\makeatletter
\pgf@relevantforpicturesizefalse
\makeatother  % 下面的路径不计入边界盒子
  \pgfpathmoveto{\pgfpointpolar{0}{0cm}}
  \pgfpathlineto{\pgfpointpolar{30}{1cm}}
  \pgfpathlineto{\pgfpointpolar{-30}{1cm}}
  \pgfpathclose
  \pgfusepath{stroke}
\end{pgfpicture}bbb
```

`\pgf@relevantforpicturesizetrue`

使用此命令后，此命令之后的路径计入边界盒子。




```
aaa\begin{pgfpicture}
\makeatletter
\pgf@relevantforpicturesizefalse
\makeatother  % 下面的路径不计入边界盒子
  \pgfpathmoveto{\pgfpointpolar{0}{0cm}}
  \pgfpathlineto{\pgfpointpolar{150}{1cm}}
  \pgfpathlineto{\pgfpointpolar{210}{1cm}}
  \pgfpathclose
  \pgfusepath{stroke}
\makeatletter
\pgf@relevantforpicturesizetrue
\makeatother  % 下面的路径计入边界盒子
  \pgfpathmoveto{\pgfpointpolar{0}{0cm}}
  \pgfpathlineto{\pgfpointpolar{30}{1cm}}
  \pgfpathlineto{\pgfpointpolar{-30}{1cm}}
  \pgfpathclose
  \pgfusepath{stroke}
\end{pgfpicture}bbb
```

103 路径装饰

```
\usepgfmodule{decorations} % LaTeX and plain TeX and pure pgf
\usepgfmodule[decorations] % ConTeXt and pure pgf
```

这个模块定义了路径装饰功能，所以需要载入它才能装饰路径。各个路径装饰程序库会自动加载这个模块。

103.1 Overview

103.2 装饰自动化（Decoration Automata）

Decoration automata 以及相关的 meta-decoration automata, 是沿着路径创建图形的方法。

103.2.1 约定路径名称

为了表述清楚，约定几个名称。

- 对于一个完整路径来说，如果它开始的一段子路径不需要装饰，之后的子路径需要被装饰，那么开始的那一段不需要装饰的子路径称为“前路径”(preexisting path)，注意前路径是没有使用 \pgfuse 命令结尾的路径。前路径可以是未被装饰过的，也可以是被之前的装饰操作装饰过的。
- 被装饰的路径称为“输入路径”(input path)。输入路径本身也可能由数个“子输入路径”构成。一个路径的“子路径”可能指的是被 moveto 操作分隔的各个部分，也可能是单纯指当前路径的某个“子集”。但在这里“子输入路径”指的是由创建路径的基本操作（如 line-to 操作）所创建的路径。计算某个路径的长度时，先把路径“拆分”为数个子输入路径，计算每个子输入路径的长度，然后加起来得到整个路径的长度。当装饰自动化过程结束后，输入路径就会被“忘记”，不能再用。
- 起到装饰作用的路径或者其它内容称为“输出”。输出的可能是路径，如程序库 decorations.pathmorphing 提供的装饰路径 zigzag；输出的也可能是一串符号，如程序库 decorations.text 提供的功能；也可能什么也没有输出。

一般情况下，子输入路径就是由 moveto, lineto, curveto, closepath 等操作构建的子路径，不同操作构建不同类型的子输入路径。圆弧、椭圆弧、抛物线、函数曲线都是由一段或数段控制曲线（即 curve-to 操作）构建的。由 move-to 操作产生的“跳跃”只包含一个点，故其构建的子路径长度是 0. 例如下面图形中：

B

A C

```
\tikz \draw (0,0)node[below]{$A$}--
            (1,1)node[above]{$B$}--
            (1,0)node[below]{$C$}--cycle;
```

有向线段 CA 就是由 closepath 操作产生的子输入路径。

文件《pgflibrarydecorations.pathmorphing.code》对装饰样式 bent 的定义如下：

```
\pgfdeclaredecoration{bent}{bent}
{
  \state{bent}[width=+\pgfdecoratedinputsegmentremainingdistance]
  {
    \pgfpathcurveto
    {\pgfqpoint{\pgfdecorationsegmentaspect
     \pgfdecoratedinputsegmentremainingdistance}{\pgfdecorationsegmentamplitude}}
    {\pgfpointadd{\pgfqpoint{\pgfdecoratedinputsegmentremainingdistance}{0pt}}
       {\pgfqpoint{-\pgfdecorationsegmentaspect
        \pgfdecoratedinputsegmentremainingdistance}{
        \pgfdecorationsegmentamplitude}}}
    {\pgfqpoint{\pgfdecoratedinputsegmentremainingdistance}{0pt}}
  }
  \state{final}
  {}
}
```

可见 bent 只有一个非空状态，这个状态的片段是一段控制曲线，曲线的宽度就是当前子输入路径的长度。观察下面例子：

```
\begin{tikzpicture}
  \draw [red](0,0) arc (0:180:1);
  \draw [decorate,decoration=bent][cyan](0,0) arc (0:180:1);
  \draw (0,0)--++(90:{pi/2});
  \draw (-1,1)--++(180:{pi/2});
\end{tikzpicture}
```

可以看出来，圆弧 (0,0) arc (0:180:1) 有两个子输入路径。用同样的方法可以测试出由 circle 创建的圆有 4 个子输入路径；操作 rectangle 生成的矩形有 4 个子输入路径；每个画直线段的符号“--”创建一个子输入路径；控制曲线“⟨*p1*⟩ .. controls ⟨*p2*⟩ and ⟨*p3*⟩ .. ⟨*p4*⟩”（即 curve-to 操作）创建一个子输入路径。下面的例子表明 sin，cos 操作都创建一个“子输入路径”：

```
\tikz \draw [red, postaction={draw,cyan,decorate,decoration=bent}]
  (0,0) sin (1,1) cos (2,0);
```

装饰过程的前路径可能是空的，也可能非空。若前路径非空，则装饰路径就是前路径的延伸，即前路径、装饰路径都是当前路径的子路径。在装饰路径中可以使用 \pgfusepath 命令来执行 draw，fill 等操作，注意此时的操作针对前路径、装饰路径。

103.2.2 片段（segment）与状态（state）

有的情况下，装饰路径只是某一个或数个片段（segment）的不断重复，如装饰路径 zigzag 的主要片段是：，由下面的代码规定：

```
\pgfpathlineto{\pgfpoint{5pt}{5pt}}
\pgfpathlineto{\pgfpoint{15pt}{-5pt}}
\pgfpathlineto{\pgfpoint{20pt}{0pt}}
```

沿着输入路径重复这一片段就得到装饰路径 zigzag. 注意这里没有使用 \pgfpathmoveto{\pgfpointorigin}，因为这个命令会把一个片段分离为数个子片段。

```
\tikz \draw decorate[decoration=zigzag] {(0,0) -- (1,0)};
```

上面例子中，输入路径是一个 1cm 长的线段，装饰路径是 zigzag 类型的，其中一开始有 2 个主要片段，之后是一个不完整片段，最后是一段很小的线段。

一个装饰路径可以由一个片段重复构成，也可以由多个不同片段构成。决定什么情况下使用什么片段的规则叫作“状态”(state)，一个状态规定一个片段。装饰自动化实现的就是“有限状态自动化”(finite state automata)。通常，初始状态和终结状态所规定的片段都是必须添加的。初始状态是一个特殊的状态，它规定装饰路径的第一个片段。装饰路径的最后一个片段对应“终结状态”（名称为 final，这是个关键词，终结状态的名称必须使用这个词），当这个状态完成后，装饰自动化过程就结束了，得到输出路径。

通常，程序根据状态规定，逐步选择片段添加到输入路径上，逐步装饰输入路径。也就是说，在装饰过程中输入路径分为“已装饰部分”和“未装饰部分”。通常程序会计算“未装饰部分”的长度，根据这个长度选择需要添加的片段，每添加一个片段就会减小这个长度。程序计算“未装饰部分”的长度时会用到一个“参考点”，这个点在输入路径上移动。初始之下参考点与输入路径的始点重合，即初始之下整个输入路径都是“未装饰”的，“未装饰部分”的长度等于整个输入路径的长度。每个片段的定义中都含有一个“宽度”参数 (width, 指定片段的宽度)，每添加一个片段，参考点会沿着输入路径移动一

次，移动的轨迹长度是所添加片段的“宽度”，同时“未装饰部分”的长度也会减去这个片段的“宽度”。

输入路径也可能由数个子输入路径组成，此时对于输入路径的装饰就是依次对各个子输入路径进行装饰。程序会计算各个子输入路径的长度，加起来得到整个输入路径的长度。在装饰一个子输入路径时，前一个片段的终点与后一个片段的起点之间的联系方式决定于后一片段的定义代码。注意，前后相继的两个片段的“连接点”未必是前面说的“参考点”，或者说“参考点”未必就是片段的起点或终点。对于多数装饰类型来说，如果添加某个片段会导致参考点超出输入路径，那么就不添加这个片段，而是直接添加终结状态（final）的片段，所以 final 片段的代码中应当包含输入路径的终点。完成 final 片段后，参考点就与输入路径的终点重合，输入路径的“未装饰部分”的长度为 0.

打比方说，一条公路可以看作是输入路径，路边的路灯投下来的光看作是片段。这样想：为了得到观赏效果，采用数种不同的路灯；不同种类的路灯投下来的光有不同的发光模式（外观不同）；不同种类的路灯间距也有差别；相邻两个路灯的灯光（片段）可能有重叠，也可能没有重叠；路灯的间距决定（公路上的）“参考点”的位置，而不是灯光的照射范围决定“参考点”的位置，也就是说，相邻两个参考点之间有一个路灯；公路的起点和终点是“参考点”；灯光的装饰效果非常好，以至于让人忘记了原来的公路。

假设输入路径是 p，按 p 的路径方向，有 p 上的点：P_0，P_1，……，P_n，其中 P_0 是 p 的起点，P_n 是 p 的终点。假设给 p 添加的装饰片段依次是 S_1，……，S_n，其中 S_1 是初始片段，S_n 是终结片段。一开始，参考点位于 P_0，在创建片段 S_1 时，程序会开启一个“片段坐标系”，这个坐标系的原点就是参考点 P_0，在这个片段坐标系中绘制出片段 S_1，故 S_1 以 P_0 为“当前参考点”。当添加初始片段 S_1 后，参考点移动到 P_1，从 P_0 到 P_1 的路径长度就是“S_1 的宽度”，P_1 是创建 S_2 时的“参考点”。当添加片段 S_2 后，参考点位于 P_2，从 P_1 到 P_2 的路径长度就是“S_2 的宽度”……如此等等，注意 P_{n-1} 是创建 S_n 时的“当前参考点”。

103.3 自定义装饰路径

有多个装饰程序库，它们提供了多种装饰类型，这些是预定义的装饰类型。你也可以自定义一种装饰类型，下面介绍自定义装饰类型的方法，以此展示装饰自动化过程。

```
\pgfdeclaredecoration{⟨name⟩}{⟨initial state⟩}{⟨states⟩}
```

这个命令用于自定义一种装饰类型。⟨*name*⟩ 是自定义装饰类型的名称。⟨*initial state*⟩ 是初始状态的名称，规定装饰类型的第一个片段。⟨*states*⟩ 是一个或数个“状态”，决定在什么情况下使用什么片段。通常，这些状态都会根据输入路径的“未装饰部分”的长度来规定所添加的片段。

由于 TEX 在数学计算方面的能力较弱，关于路径的计算精度并不很高。除了输入路径是水平或竖直的状态外，对装饰路径的计算精度都不高。

在 ⟨*states*⟩ 中的各个状态需要使用命令 `\state` 来定义。

```
\state{⟨name⟩}[⟨options⟩]{⟨code⟩}
```

本命令声明一个名称为 ⟨*name*⟩ 的状态。当装饰自动化过程处于状态 ⟨*name*⟩ 时会产生以下动作：

1. 解析 ⟨*options*⟩. ⟨*options*⟩ 中的选项可能会导致状态切换，即从当前状态切换到其它状态。若某个选项导致切换状态，则切换是立即发生的，此选项之后的选项不会被执行，以下的动作也不

会被执行。⟨*options*⟩ 中的选项按次序逐个被执行，即前一个选项的作用实现后，再执行下一个选项。

2. 代码 ⟨*code*⟩ 是一段绘图代码，即本状态规定的片段图形。作为绘图代码，⟨*code*⟩ 有自己的坐标系（片段坐标系），这个坐标系的默认构建方式如下。

   ⟨*code*⟩ 会被放入一个 TEX 分组内执行，在这个分组内使用坐标变换，使得当前的参考点是其坐标系的原点；当读取第一个片段时，当前参考点当然就是输入路径的起点；前面已经提到，参考点 P_{n-1} 是创建片段 S_n 时的“当前参考点”，也就是说，参考点 P_{n-1} 是片段 S_n 的 ⟨*code*⟩ 坐标系的原点。

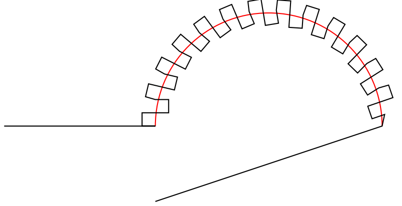
   假设 S_i 与 S_{i+1} 是前后相连的两个片段，添加 S_i 后，输入路径上的“参考点”（见前文）变成 P_i，输入路径曲线在 P_i 处有切向量 l_i，那么 S_{i+1} 的 ⟨*code*⟩ 坐标系会被围绕它的原点（即当前参考点 P_i）旋转，使其 x 轴的方向是切向量 l_i 的方向，而 y 轴方向在 x 轴的左侧构成右手系。

   片段 S_i 的终点与 S_{i+1} 的起点之间的联系方式决定于片段 S_{i+1} 的 ⟨*code*⟩ 中的第一个创建路径的命令，如果它的第一个创建路径的命令是 line-to 操作，则用直线段连接两个点；如果它的第一个创建路径的命令是 move-to 操作，则片段 S_i 与 S_{i+1} 之间有间断；如果它的第一个路径命令是 curve-to 操作，则用曲线连接两个点。

   通常，你需要在 ⟨*options*⟩ 中使用选项 `width=`⟨宽度⟩ 为片段规定一个“宽度”，即本片段所装饰的那一段输入路径的长度。假设 S_i 的参考点是 P_{i-1}，添加 S_i 后参考点变成 P_i，那么沿着输入路径从 P_{i-1} 到 P_i 的长度就是片段 S_i 的 ⟨宽度⟩。

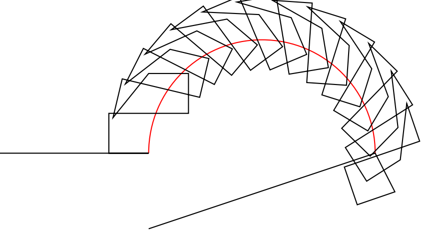
3. 当本状态的 ⟨*code*⟩ 执行完毕后（即本状态规定的片段添加完后），如果本状态的 ⟨*options*⟩ 中有 `next state` 选项，则会切换到下一个状态，解析下一个状态的选项和代码，否则程序会重复添加当前状态规定的片段，直到出现切换状态（或结束装饰过程）的条件。

状态片段的 ⟨*code*⟩ 作为一个图形有自己的固有宽度，其固有宽度不必与 `width=`⟨宽度⟩ 指定的 ⟨宽度⟩ 一致；如果二者不一致，可能会让情况变得复杂，比较下面两个图形：



```
\pgfdeclaredecoration{example}{initial}
  {
% 选项 width 的作用是，指定片段的装饰长度，若输入路径的未装饰部分的长度小于 10pt，则切换到状态 final

    \state{initial}[width=10pt] % 状态 initial 的片段的固有宽度正是 10pt.
    {
      \pgfpathlineto{\pgfpoint{0pt}{5pt}}
      \pgfpathlineto{\pgfpoint{5pt}{5pt}}
      \pgfpathlineto{\pgfpoint{5pt}{-5pt}}
      \pgfpathlineto{\pgfpoint{10pt}{-5pt}}
      \pgfpathlineto{\pgfpoint{10pt}{0pt}}
    }
    \state{final}  % 状态 final 的片段是连接到圆弧终点的线段
    {
      \pgfpathlineto{\pgfpointdecoratedpathlast}
    }
```

```
  }
\tikz[decoration=example]
{
  \draw [red](2,0) arc(180:0:1.5cm);
  \draw (0,0) -- (2,0) decorate {arc(180:0:1.5cm)} -- (2,-1);
}
```

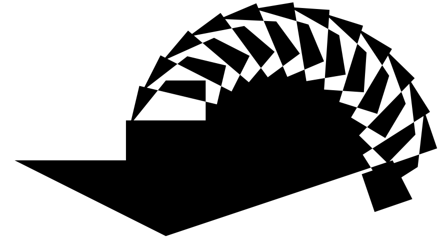


```
\pgfdeclaredecoration{example}{initial}
  {
    \state{initial}[width=10pt]  % 本状态的片段的固有宽度是 30pt，与选项 width 指定的宽度不一致。
    {
      \pgfpathlineto{\pgfpoint{0pt}{15pt}}
      \pgfpathlineto{\pgfpoint{15pt}{15pt}}
      \pgfpathlineto{\pgfpoint{15pt}{-15pt}}
      \pgfpathlineto{\pgfpoint{30pt}{-15pt}}
      \pgfpathlineto{\pgfpoint{30pt}{0pt}}
    }
    \state{final}  % 状态 final 的片段是连接到圆弧终点的线段
    {
      \pgfpathlineto{\pgfpointdecoratedpathlast}
    }
  }
\tikz[decoration=example]
{
  \draw [red](2,0) arc(180:0:1.5cm);
  \draw (0,0) -- (2,0) decorate {arc(180:0:1.5cm)} -- (2,-1);
}
```

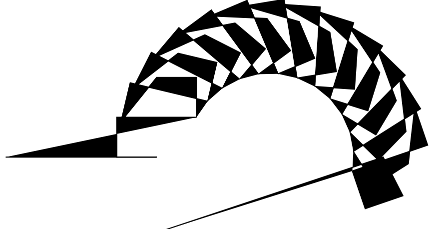
上面例子中定义的装饰类型名称是 example，初始状态名称 `initial` 是个关键词。第 1 个图形中的初始状态定义的片段是：⌐⌙，输入路径是一个半圆弧。上面第 2 个图形中的初始片段的尺寸是第 1 个图形中的初始片段的尺寸的 3 倍，显然第 2 个图形中的初始片段已经严重走样（除了第一个片段），并且倒数第 2 个片段的终点还超出了输入路径的终点（最后一个片段，即 `final` 状态的片段是连接到圆弧终点的线段）。

注意，在装饰自动化过程中，前路径是始终被“记住”的，故添加到输入路径上的片段实际上是前路径的延伸。因此，如果在 ⟨*code*⟩ 中使用命令 `\pgfusepath{fill}`，那么被填充的路径就包括原来的前路径，以及延伸到当前的片段（装饰路径）。

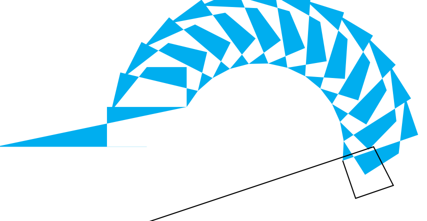
下面的图形是上面图形的修改，比较下面几个图形：



```
\pgfdeclaredecoration{example}{initial}
  {
    \state{initial}[width=10pt]
    {
      \pgfpathlineto{\pgfpoint{0pt}{15pt}}
      \pgfpathlineto{\pgfpoint{15pt}{15pt}}
      \pgfpathlineto{\pgfpoint{15pt}{-15pt}}
      \pgfpathlineto{\pgfpoint{30pt}{-15pt}}
      \pgfpathlineto{\pgfpoint{30pt}{0pt}}
    }
    \state{final}
    {
      \pgfpathlineto{\pgfpointdecoratedpathlast}
    }
  }
\tikz[decoration=example]  % 使用命令 \fill，看一下填充颜色的效果
{
  \fill (0,0) -- (2,0) decorate {arc(180:0:1.5cm)} -- (2,-1);
}
```



```
\pgfdeclaredecoration{example}{initial}
  {
    \state{initial}[width=10pt]
    {
      \pgfpathlineto{\pgfpoint{0pt}{15pt}}
      \pgfpathlineto{\pgfpoint{15pt}{15pt}}
      \pgfpathlineto{\pgfpoint{15pt}{-15pt}}
      \pgfpathmoveto{\pgfpoint{15pt}{-15pt}}  % 这里使用了 move-to 操作，将片段分离开来
      \pgfpathlineto{\pgfpoint{30pt}{-15pt}}
      \pgfpathlineto{\pgfpoint{30pt}{0pt}}
    }
    \state{final}
    {
      \pgfpathlineto{\pgfpointdecoratedpathlast}
    }
  }
\tikz[decoration=example]
{
  \filldraw (0,0) -- (2,0) decorate {arc(180:0:1.5cm)} -- (2,-1);
}
```



```
\pgfdeclaredecoration{example}{initial}
  {
    \state{initial}[width=10pt]
```

```
    {
      \pgfpathlineto{\pgfpoint{0pt}{15pt}}
      \pgfpathlineto{\pgfpoint{15pt}{15pt}}
      \pgfpathlineto{\pgfpoint{15pt}{-15pt}}
      \pgfsetfillcolor{cyan}
      \pgfusepath{fill}  % 结束一段路径
      \pgfpathmoveto{\pgfpoint{15pt}{-15pt}}  % 开启一段路径，与前一段路径是分离的
      \pgfpathlineto{\pgfpoint{30pt}{-15pt}}
      \pgfpathlineto{\pgfpoint{30pt}{0pt}}
    }
    \state{final}
    {
      \pgfpathlineto{\pgfpointdecoratedpathlast}
    }
  }
\tikz[decoration=example]  % 画出路径
{
  \draw (0,0) -- (2,0) decorate {arc(180:0:1.5cm)} -- (2,-1);
}
```

下面的例子中，将 node 作为片段添加到路径上，而且 node 的边界线条颜色和填充颜色都是随机决定的：

```
\pgfdeclaredecoration{stars}{initial}{
  \state{initial}[width=15pt]
  {
    \pgfmathparse{round(rnd*100)}
    \pgfsetfillcolor{yellow!\pgfmathresult!orange}
    \pgfsetstrokecolor{yellow!\pgfmathresult!red}
    \pgfnode{star}{center}{}{}{\pgfusepath{stroke,fill}}
  }
  \state{final}
  {
    \pgfpathmoveto{\pgfpointdecoratedpathlast}
  }
}
\tikz\path[rotate=90, decorate, decoration=stars, star point ratio=2, star points=5,
          inner sep=0, minimum size=rnd*10pt+2pt]
  (0,0) .. controls (0,2) and (3,2) .. (3,0) .. controls (3,-3) and (0,0) .. (0,-3) .. controls
  ↪  (0,-5) and (3,-5) .. (3,-3);
```

命令 `\state` 的 ⟨*options*⟩ 中的选项都有路径前缀 `/pgf/decoration automaton/`，在 ⟨*options*⟩ 中可以使用以下选项。

`/pgf/decoration automaton/switch if less than`=⟨*dimension*⟩ `to` ⟨*new state*⟩ (no default)

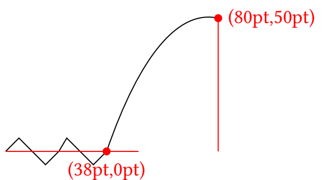
本选项的作用是：当 PGF 读取这个选项时，会检查输入路径的“未装饰部分”的长度，如果长度小于尺寸 ⟨*dimension*⟩ 就立即切换到状态 ⟨*new state*⟩，不再解析本选项之后的选项或者代码（本状态规定的片段当然也不会被添加到输入路径上）；否则继续解析本选项之后的选项或者代码。

```
\pgfdeclaredecoration{illus}{initial}
{
  \state{initial}[switch if less than=10cm to final]
% 若输入路径的未装饰部分的长度小于 10cm，则切换到状态 final
  {
    \pgfpathlineto{\pgfpoint{5pt}{5pt}}
    \pgfpathlineto{\pgfpoint{10pt}{0pt}}
  }
  \state{final} % 状态 final 的片段是一段曲线
  {
    \pgfpathcurveto
      {\pgfpointadd
        {\pgfpointscale{0.3333}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointadd
        {\pgfpointscale{0.6666}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointdecoratedpathlast}
  }
}

\tikz [decoration=illus] \draw [decorate] (0,0)--(3,0);  % 输入路径的长度是 3cm
```

上面例子中，状态 `initial` 的选项规定：若输入路径的未装饰部分的长度小于 10cm，则切换到状态 `final`, 而 `\draw` 命令中的输入路径长度是 3cm, 所以只有 `final` 的片段画出。

`/pgf/decoration automaton/switch if input segment less than`=⟨*dimension*⟩ `to` ⟨*new state*⟩ (no default)

当 PGF 读取这个选项时，会计算当前的子输入路径的“未装饰部分”的长度，即参考点与当前子输入路径的终点之间的路径长度，若此长度小于 ⟨*dimension*⟩，则切换到状态 ⟨*new state*⟩. 在装饰子输入路径时，这有利于避免装饰路径过分地突出子输入路径之外。




```
\pgfdeclaredecoration{illus}{initial}
{
  \state{initial}[width=18pt,switch if input segment less than=15pt to final,]
  {
    \pgfpathlineto{\pgfpoint{5pt}{5pt}}
    \pgfpathlineto{\pgfpoint{15pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{20pt}{0pt}}
  }
  \state{final}
  {
    \pgfpathcurveto
      {\pgfpointadd
        {\pgfpointscale{0.3333}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointadd
```

```
        {\pgfpointscale{0.6666}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointdecoratedpathlast}
  }
}

\tikz [decoration=illus]
{
  \draw [decorate] (0,0)--(50pt,0pt) (80pt,0pt)--(80pt,50pt);
  \draw [red] (0,0)--(50pt,0pt) (80pt,0pt)--(80pt,50pt);
  \fill [red] (38pt,0pt) circle(1.5pt) node [below] {\footnotesize(38pt,0pt)}
            (80pt,50pt) circle(1.5pt) node [right] {\footnotesize(80pt,50pt)};
}
```

上面例子中，状态 `initial` 的片段的宽度规定为 18pt，片段的固有宽度是 20pt，第一个子输入路径是的长度为 50pt 的线段。当在第一个子输入路径上添加两个初始状态的片段后，第二个片段的终点横标是 18pt+20pt=38pt，当前参考点的横标是 18pt+18pt=36pt，第一个子输入路径的“未装饰部分”长度是 50pt-36pt=14pt，小于选项 `switch if input segment less than` 规定的 15pt，因此切换到 `final` 状态，于是以第二个片段的终点 (38pt,0pt) 为始点，以输入路径的终点 (80pt,50pt) 为终点，构建一段控制曲线。

`/pgf/decoration automaton/width=`⟨*dimension*⟩ (no default)

本选项指定的 ⟨*dimension*⟩ 规定了本状态的片段所占据的输入路径上的一段宽度，即片段所装饰的那一部分输入路径的长度。当程序读取这个选项时，PGF 计算输入路径的“未装饰部分”的长度，如果此长度大于本选项指定的 ⟨*dimension*⟩，则添加本状态的片段；如果此长度小于本选项指定的 ⟨*dimension*⟩，则直接切换到状态 `final`，因此本选项相当于

```
switch if less than=⟨dimension⟩ to final
```

状态的 ⟨*code*⟩ 规定了片段图形，这个片段作为一个图形有自己的固有宽度，如果这个固有宽度与本选项指定的 ⟨*dimension*⟩ 不相同就会让情况变得复杂一些。观察下面的例子。

```
\pgfdeclaredecoration{illus}{initial}
{
  \state{initial}[width=12pt,next state=second]
  {
    \pgfpathlineto{\pgfpoint{5pt}{5pt}}
    \pgfpathlineto{\pgfpoint{15pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{20pt}{0pt}}
  }
  \state{second}[switch if less than=10pt to final]
  {
    \pgfpathlineto{\pgfpoint{0pt}{5pt}}
    \pgfpathlineto{\pgfpoint{5pt}{5pt}}
    \pgfpathlineto{\pgfpoint{5pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{10pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{10pt}{0pt}}
  }
  \state{final}
  {
    \pgfpathcurveto
```

```
      {\pgfpointadd
        {\pgfpointscale{0.3333}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointadd
        {\pgfpointscale{0.6666}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointdecoratedpathlast}
  }
}

\tikz [decoration=illus]
{
  \draw [red] (0,0)--(12pt,0)--++(0,5pt);
  \draw [decorate] (0,0)--(50pt,0pt);
}
\hspace{2cm}
\tikz [decoration=illus]
{
  \draw [red] (0,0)--(12pt,0)--++(0,5pt);
  \draw [decorate] (0,0)--(60pt,0pt);
}
```

在上面的例子中，自定义的装饰路径包括 3 个状态：

- 状态 initial 的片段是“之字形”折线 ⌃⌄.
- 原来所设想的状态 second 的片段是 ⌐⌞.
- 状态 final 的片段是一段 curve-to 曲线。

状态 initial 的片段的固有宽度是 20pt，但是该状态的选项 width=12pt 规定这个片段只占据 12pt 的宽度，即所装饰的那一部分输入路径的长度只有 12pt，这导致状态 second 片段的外观与原来设想的不一样。

左右两个图的代码的差别仅仅是输入路径的长度不一样，一个是 50pt，一个是 60pt，但是二者的最后一个片段差别很大。

`/pgf/decoration automaton/repeat state`=⟨*repetitions*⟩ (no default, initially 0)

这个选项的参数 ⟨*repetitions*⟩ 是个整数。某个计数器的值会被设为 ⟨*repetitions*⟩, 如果当前状态的片段被添加，那么该计数器的值减 1, 然后检查是否满足切换状态的条件；如果满足切换状态的条件就立即切换，否则再次添加当前状态的片段，并且让该计数器的值减 1; 如此继续，直到该计数器的值减为 0 时，再次添加当前状态的片段，然后切换到选项 `next state` 指定的状态；如果没有选项要求切换到其它状态，就继续添加当前状态的片段，直到出现满足切换状态的条件。也就是说，本选项会导致当前状态的片段至多被添加 ⟨*repetitions*⟩+1 次。

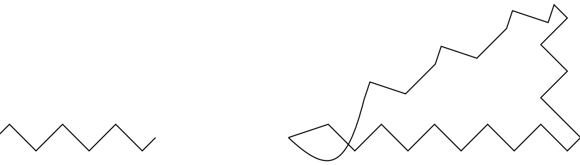
`/pgf/decoration automaton/next state`=⟨*new state*⟩ (no default)

在当前状态的 ⟨*code*⟩ 被执行最后一次后（即当前状态的片段被最后一次添加后），就切换到状态 ⟨*new state*⟩. 如果不给出本选项，就继续执行当前状态的 ⟨*code*⟩, 直到出现其它切换状态的条件。

`/pgf/decoration automaton/if input segment is closepath`=⟨*options*⟩ (no default)

当 PGF 读到这个选项时，会检查当前的子输入路径是否为 closepath 操作创建的路径，如果是，则针对 closepath 执行 ⟨*options*⟩; 如果不是，则没有动作，继续读取之后的选项。你可以利用这个选项

的 ⟨*options*⟩ 对 closepath 做某种特殊操作，也可以切换到其它状态。

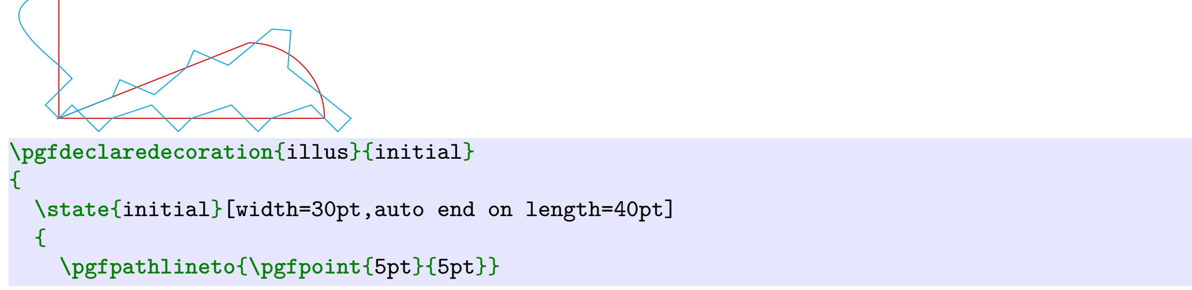


```
\pgfdeclaredecoration{illus}{initial}
{
  \state{initial}[width=20pt,if input segment is closepath={width=30pt},]
  {
    \pgfpathlineto{\pgfpoint{5pt}{5pt}}
    \pgfpathlineto{\pgfpoint{15pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{20pt}{0pt}}
  }
  \state{final}
  {
    \pgfpathcurveto
      {\pgfpointadd
        {\pgfpointscale{0.3333}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointadd
        {\pgfpointscale{0.6666}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointdecoratedpathlast}
  }
}

\tikz [decoration=illus]
{
  \draw [decorate]  (-50pt,0pt)--(-100pt,0pt) (0,0)--(100pt,0pt)--(100pt,50pt)--cycle;
}
```

`/pgf/decoration automaton/auto end on length`=⟨*dimension*⟩ (no default)

设计这个选项是为了便利，本选项的作用也可由其它选项实现。本选项的作用是：(1) 当 PGF 读取本选项时，如果输入路径的“未装饰部分”的长度不大于 ⟨*dimension*⟩, 那么在当前坐标点与输入路径的终点之间画一个直线段并结束装饰路径；(2) 如果当前子输入路径是由闭合路径操作 closepath 生成的线段，并且当前子输入路径的未装饰部分的长度不大于 ⟨*dimension*⟩, 则在当前坐标点与子输入路径的终点之间画一个直线段并且把装饰路径作成闭合的，然后继续在本状态下处理之后的输入路径，直到出现切换状态（或结束装饰过程）的条件。

其它情况下，本选项无作用。

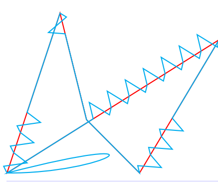


```
\pgfdeclaredecoration{illus}{initial}
{
  \state{initial}[width=30pt,auto end on length=40pt]
  {
    \pgfpathlineto{\pgfpoint{5pt}{5pt}}
```

```
    \pgfpathlineto{\pgfpoint{15pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{20pt}{0pt}}
  }
  \state{final}
  {
    \pgfpathcurveto
      {\pgfpointadd
        {\pgfpointscale{0.3333}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointadd
        {\pgfpointscale{0.6666}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointdecoratedpathlast}
  }
}

\tikz [decoration=illus]
{
  \draw [red,postaction={draw,decorate,cyan}] (0,0)--(100pt,0pt) arc
  ↪  (0:90:1cm)--cycle--(0pt,50pt);
}
```

`/pgf/decoration automaton/auto corner on length`=⟨*dimension*⟩ (no default)

这个选项的作用是：首先，如果 TEX-if 的条件判断宏 `\ifpgfdecoratepathhascorners` 的值是 false，则什么也不做；否则，检查当前子输入路径的“未装饰部分”的长度是否不大于 ⟨*dimension*⟩. 如果是，则用直线段将当前坐标点与当前子输入路径的终点连起来，然后在当前状态下继续装饰后面的子输入路径，直到出现切换状态（或结束装饰过程）的条件。注意本选项针对的是“子输入路径”，设计本选项的目的是为了避免装饰路径“忽略”子输入路径之间的转角。



```
\pgfdeclaredecoration{illus}{initial}
{
  \state{initial}[width=8pt,auto corner on length=40pt]
  {
    \pgfpathlineto{\pgfpoint{2pt}{2pt}}
    \pgfpathlineto{\pgfpoint{6pt}{-6pt}}
    \pgfpathlineto{\pgfpoint{8pt}{0pt}}
  }
  \state{final}
  {
    \pgfpathcurveto
      {\pgfpointadd
        {\pgfpointscale{0.3333}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointadd
        {\pgfpointscale{0.6666}{\pgfpointdecoratedpathlast}}
        {\pgfpoint{0pt}{20pt}}
      }
      {\pgfpointdecoratedpathlast}
```

```
  }
}

\tikz [decoration=illus]
{
  \pgfdecoratepathhascornerstrue
  \draw [red,postaction={draw,decorate,cyan}]
  ↪  (0,0)--(20pt,60pt)--(30pt,20pt)--(50pt,0pt)--(80pt,50pt)--cycle;
}
```

`/pgf/decoration automaton/persistent precomputation=`⟨*precode*⟩ (no default)

前面提到，如果当前状态的片段代码 ⟨*code*⟩ 被执行，则 ⟨*code*⟩ 会被放入一个 TEX 分组中来执行，并默认在这个 TEX 分组中对 ⟨*code*⟩ 使用坐标变换（即在“片段坐标系”中执行 ⟨*code*⟩）。使用本选项后，若 ⟨*code*⟩ 被执行，则本选项的 ⟨*precode*⟩ 会先被执行，然后再执行 ⟨*code*⟩, 并且 ⟨*precode*⟩ 会在容纳 ⟨*code*⟩ 的 TEX 分组之前被执行。但是注意，在执行 ⟨*precode*⟩ 时，不对 ⟨*precode*⟩ 施加变换，这一点与执行 ⟨*code*⟩ 的情况不同。

`/pgf/decoration automaton/persistent postcomputation=`⟨*postcode*⟩ (no default)

类似前一个选项。使用本选项后，当前状态的片段代码 ⟨*code*⟩ 被执行后，则本选项的 ⟨*postcode*⟩ 会被执行，并且 ⟨*postcode*⟩ 会在容纳 ⟨*code*⟩ 的 TEX 分组之后被执行。

下面的宏保存的值可能很有用。

`\pgfdecoratedpathlength`

这个宏保存当前的输入路径的长度。如果输入路径包含数个子输入路径，则输入路径的长度是各个子输入路径的长度之和。

`\pgfdecoratedinputsegmentlength`

这个宏保存当前的子输入路径的长度。

`\pgfpointdecoratedpathlast`

这个宏保存当前的输入路径的终点。

`\pgfpointdecoratedinputsegmentlast`

这个宏保存当前的子输入路径的终点。

`\pgfdecoratedangle`

前面提到，如果当前状态的片段代码 ⟨*code*⟩ 要被执行，则会把 ⟨*code*⟩ 放入一个 TEX 分组中来执行，并且在这个分组内进行平移和旋转变换，其中旋转变换的转角就是这个宏保存的值（使用角度制）。

`\pgfdecoratedremainingdistance`

这个宏保存的值是，当下的，输入路径的未装饰部分的长度。

`\pgfdecoratedcompleteddistance`

这个宏保存的值是，当下的，输入路径的已装饰部分的长度。

\pgfdecoratedinputsegmentremainingdistance

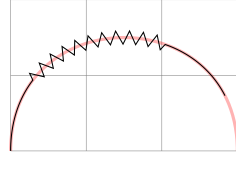
这个宏保存的值是，当下的，子输入路径的未装饰部分的长度。

\pgfdecoratedinputsegmentcompleteddistance

这个宏保存的值是，当下的，子输入路径的已装饰部分的长度。

103.4 {pgfdecoration} 环境

当定义了装饰类型后，就可以使用它。下面介绍的 {pgfdecoration} 环境要用在 {tikzpicture} 环境或 {pgfpicture} 环境中，用于构建装饰路径。



```
\begin{tikzpicture}[decoration={segment length=5pt}]
  \draw [help lines] grid (3,2);
  \draw [red,very thick,opacity=0.3]
    (0,0) .. controls (0,2) and (3,2) .. (3,0);
  \begin{pgfdecoration}{{curveto}{1cm},{zigzag}{2cm},{curveto}{1cm}}
    \pgfpathmoveto{\pgfpointorigin}
    \pgfpathcurveto{\pgfpoint{0cm}{2cm}} {\pgfpoint{3cm}{2cm}}
                   {\pgfpoint{3cm}{0cm}}
  \end{pgfdecoration}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\begin{pgfdecoration}{⟨*decoration list*⟩}
 ⟨*environment content*⟩
\end{pgfdecoration}

⟨*environment contents*⟩ 是路径命令构建的"输入路径"。⟨*decoration list*⟩ 是用逗号分隔的装饰类型的列表，列表项的格式是

{⟨*decoration*⟩}{⟨*length*⟩}{⟨*before code*⟩}{⟨*after code*⟩}

其中 ⟨*decoration*⟩ 是装饰类型的名称，⟨*length*⟩ 是一个尺寸，规定长度为 ⟨*length*⟩ 的一段输入路径用 ⟨*decoration*⟩ 来装饰。⟨*before code*⟩ 与 ⟨*after code*⟩ 是可选的。⟨*before code*⟩ 在执行 ⟨*decoration*⟩ 的片段图形代码 ⟨*code*⟩ 之前被执行；⟨*after code*⟩ 在执行 ⟨*decoration*⟩ 的片段图形代码 ⟨*code*⟩ 之后被执行。在前面的例子中，装饰类型列表是

```
{{curveto}{1cm},{zigzag}{2cm},{curveto}{1cm}}
```

其中没有 ⟨*before code*⟩ 和 ⟨*after code*⟩, 这个列表规定：输入路径开始的长度为 1cm 的一段用 curveto 型路径装饰，之后长度为 2cm 的一段用 zigzag 型路径装饰，再后的长度为 1cm 的一段用 curveto 型路径装饰，超出这 4cm 的输入路径没有装饰，最终会被"丢弃"。

当 PGF 处理这个环境时，发生以下动作：

1. 保存前路径。
2. 执行 ⟨*environment contents*⟩ 中的路径命令得到输入路径，保存之。假设 ⟨*decoration list*⟩ 中各个列表项是

   {⟨*decoration 1*⟩}{⟨*length 1*⟩}, . . . ,{⟨*decoration n*⟩}{⟨*length n*⟩}

   PGF 会从输入路径的起点开始，参照这些长度值 ⟨*length 1*⟩, . . . , ⟨*length n*⟩ 将输入路径分割为数个部分，PGF 会在分割点处将输入路径"打断"，从而得到数个前后相继的新的输入路径。第

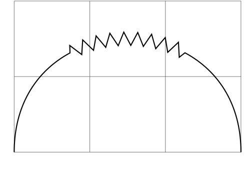
一段输入路径由 ⟨*decoration 1*⟩ 来装饰；第二段输入路径由 ⟨*decoration 2*⟩ 来装饰……假如各长度之和 $L = \langle length\ 1\rangle + \cdots + \langle length\ n\rangle$ 大于原来的输入路径的长度，那么超出原输入路径的那一段装饰路径会被忽略。假如各长度之和 L 小于原来的输入路径的长度，那么最后一段输入路径就没有装饰，最终会被“丢弃”。

3. 完成前路径。
4. 沿着输入路径构建装饰路径，得到输出路径。

使用 {pgfdecoration} 环境时注意以下几点：

- 如果 ⟨*environment contents*⟩ 不以 \pgfpathmoveto 开头，那么前路径的最后一个点就是输入路径的起点。
- 多余的 \pgfpathmoveto 命令会被忽略。
- ⟨*environment contents*⟩ 结尾处的 \pgfpathmoveto 命令会被忽略。
- ⟨*environment contents*⟩ 中的 \pgfpathclose 命令会被解释成一个直线段，是输入路径的一部分。
- 在装饰自动化过程中，程序不会自动使用 move-to 操作来造成分离的装饰路径，因此在需要间断的地方，装饰路径可能没有间断。此时可能需要在输入路径中使用 \pgfpathmoveto 命令或其它办法（如修改状态规定）。如果某个装饰路径（状态的片段）以 line-to 或 curve-to 开头，而装饰过程的前路径是空的，那么就在此装饰路径之前插入一个 move-to 操作。
- 如果在状态规定的片段（装饰路径）中或者在 ⟨*after code*⟩ 中使用 \pgfusepath 命令，那么该命令使用的路径包括前路径和直到当前的装饰路径。

下面的例子中使用了 \pgfdecoratedpathlength/3 这样的长度算式：



```
\begin{tikzpicture}[decoration={segment length=5pt}]
  \draw [help lines] grid (3,2);
  \begin{pgfdecoration}{
      {curveto}{\pgfdecoratedpathlength/3},
      {zigzag}{\pgfdecoratedpathlength/3},
      {curveto}{\pgfdecoratedremainingdistance}
    }
    \pgfpathmoveto{\pgfpointorigin}
    \pgfpathcurveto
      {\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{\pgfpoint{3cm}{0cm}}
  \end{pgfdecoration}
  \pgfusepath{stroke}
\end{tikzpicture}
```

宏 \pgfpointdecoratedpathfirst 可用于 ⟨*before code*⟩ 中：

\pgfpointdecoratedpathfirst

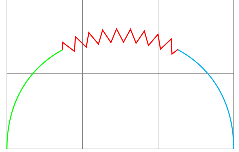
这个宏保存当前输入路径的起点。

宏 \pgfpointdecoratedpathlast 可用于 ⟨*after code*⟩ 中：

\pgfpointdecoratedpathlast

这个宏保存当前输入路径的终点。

下面的例子中，使用 ⟨*before code*⟩ 或者 ⟨*after code*⟩ 达到某种效果：



```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \begin{pgfdecoration}
  {
      {curveto}{\pgfdecoratedpathlength/3}{\pgfsetstrokecolor{green}}{\pgfusepath{stroke}},
      {zigzag}{\pgfdecoratedpathlength/3}
              {\pgfpathmoveto{\pgfpointdecoratedpathfirst}\pgfdecorationsegmentlength=5pt}
              {
               \pgfsetstrokecolor{red}
               \pgfusepath{stroke}
               \pgfpathmoveto{\pgfpointdecoratedpathlast}
               \pgfsetstrokecolor{cyan}
              },
      {curveto}{\pgfdecoratedremainingdistance}
  }
    \pgfpathmoveto{\pgfpointorigin}
    \pgfpathcurveto
      {\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{\pgfpoint{3cm}{0cm}}
  \end{pgfdecoration}
  \pgfusepath{stroke}
\end{tikzpicture}
```

当 `{pgfdecoration}` 环境结束后，下面的宏可用：

`\pgfdecorateexistingpath`

这个宏保存 `{pgfdecoration}` 环境之前的前路径。

`\pgfdecoratedpath`

这个宏保存 `{pgfdecoration}` 环境之内的原来的输入路径，即由 ⟨*environment contents*⟩ 创建的路径。

`\pgfdecorationpath`

这个宏保存 `{pgfdecoration}` 环境产生的输出路径。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。

`\pgfpointdecoratedpathlast`

这个宏保存 `{pgfdecoration}` 环境内原来的输入路径的终点。

`\pgfpointdecorationpathlast`

这个宏保存 `{pgfdecoration}` 环境产生的输出路径的终点。

下面的样式是针对每一段装饰路径的，你可以改变它的默认设置。

`/pgf/every decoration` (style, initially empty)

此样式针对每一段装饰路径。

如果要在 plain-TEX 中使用 {pgfdecoration} 环境，可以使用下面的命令组合：

```
\pgfdecoration{⟨name⟩}
  ⟨environment contents⟩
\endpgfdecoration
```

如果要在 ConTEXt 中使用 {pgfdecoration} 环境，可以使用下面的命令组合：

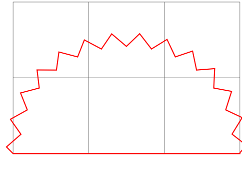
```
\startpgfdecoration{⟨name⟩}
  ⟨environment contents⟩
\stoppgfdecoration
```

下面的命令都是以 {pgfdecoration} 环境为基础定义的，用起来简洁一些。

`\pgfdecoratepath{⟨name⟩}{⟨path commands⟩}`

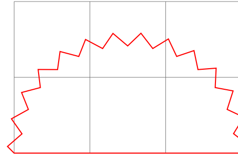
⟨*name*⟩ 是装饰类型的名称，⟨*path commands*⟩ 构建输入路径。用 ⟨*name*⟩ 类型的装饰路径来装饰 ⟨*path commands*⟩. 此命令的定义是

```
\long\def\pgfdecoratepath#1#2{%
  \pgfdecoration{{#1}{\pgfdecoratedpathlength}{\pgfdecoratebeforecode}{
  ↪   \pgfdecorateaftercode}}%
  #2%
  \endpgfdecoration}
```



```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \def\pgfdecorateaftercode{\pgfpathclose\pgfsetstrokecolor{red}
  ↪   \pgfusepath{stroke}}
  \pgfdecoratepath{zigzag}
    {\pgfpathmoveto{\pgfpointorigin}
    \pgfpathcurveto{\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{
    ↪   \pgfpoint{3cm}{0cm}}}
\end{tikzpicture}
```

上面例子等效于：



```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \pgfdecoratepath{zigzag}
    {\pgfpathmoveto{\pgfpointorigin}
    \pgfpathcurveto{\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{
    ↪   \pgfpoint{3cm}{0cm}}}
  \pgfpathclose\pgfsetstrokecolor{red}\pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfdecoratecurrentpath{⟨name⟩}`

用 ⟨*name*⟩ 类型的装饰路径来装饰当前的路径。

`\pgfdecoratebeforecode`

若定义 `\def\pgfdecoratebeforecode{⟨code⟩}`, 那么 ⟨*code*⟩ 将处于 ⟨*before code*⟩ 的位置上。

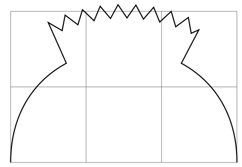
`\pgfdecorateaftercode{⟨code⟩}`

若定义 `\def\pgfdecoratebeforecode{⟨code⟩}`, 那么 ⟨*code*⟩ 将处于 ⟨*after code*⟩ 的位置上。

前面提到，片段（装饰路径）的代码会被放入一个 TEX 分组内执行，并且在分组内会进行默认的坐标变换，得到一个用于绘制片段的“片段坐标系”。除了默认的坐标变换，还可以用下面的命令规定其它的变换（相对于“片段坐标系”）：

\pgfsetdecorationsegmenttransformation⟨*code*⟩

⟨*code*⟩ 是坐标变换命令，这个命令针对的是当前装饰类型的各个片段（装饰路径），所做的变换以各片段的“片段坐标系”为参照。注意本命令的有效范围仅限于当前装饰类型，当更换装饰类型时，本命令会被重设 (reset)。



```
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \begin{pgfdecoration}{
      {curveto}{\pgfdecoratedpathlength/3},
      {zigzag}{\pgfdecoratedpathlength/3}
              {
                \pgfdecorationsegmentlength=5pt
                \pgfsetdecorationsegmenttransformation{\pgftransformyshift{.5cm}}
              },
      {curveto}{\pgfdecoratedremainingdistance}
  }
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto
      {\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{\pgfpoint{3cm}{0cm}}
  \end{pgfdecoration}
  \pgfusepath{stroke}
\end{tikzpicture}
```

上面例子中，命令 \pgfsetdecorationsegmenttransformation 仅用于第二个装饰类型 zigzag，只对这个装饰类型的各个片段有效。

103.5 Meta-Decorations

Meta-Decorations 就是用“已定义的装饰类型”来定义的“新的装饰类型”。

103.5.1 定义一个 Meta-Decorations

\pgfdeclaremetadecorate{⟨*name*⟩}{⟨*initial state*⟩}{⟨*states*⟩}

本命令的用法与 \pgfdeclaredecoration 类似。本命令定义一个 meta-decoration，其名称为 ⟨*name*⟩. ⟨*initial state*⟩ 是初始状态的名称。⟨*states*⟩ 是状态列表，各个状态使用命令 \state 定义。

\state{⟨*name*⟩}[⟨*options*⟩]{⟨*code*⟩}

⟨*name*⟩ 是状态的名称。meta-decoration 的状态片段代码 ⟨*code*⟩ 不会被放入 TEX 分组内执行，否则会有多余的计算。这里的 ⟨*options*⟩ 中可以使用以下选项。

/pgf/meta-decoration automaton/switch if less than=⟨*dimension*⟩ to ⟨*new state*⟩ (no default)

当 PGF 读取这个选项时,会检查输入路径的“未装饰部分”的长度,如果这个长度小于 ⟨*dimension*⟩ 则立即切换到状态 ⟨*new state*⟩. 在 ⟨*dimension*⟩ 中可以使用宏 `\pgfmetadecoratedpathlength` → P. 679.

`/pgf/meta-decoration automaton/width=`⟨*dimension*⟩ (no default)

本选项规定片段的“宽度”，即片段所装饰的路径长度。如果输入路径的“未装饰部分”的长度小于 ⟨*dimension*⟩, 则立即切换到终结状态 `final`.

`/pgf/meta-decoration automaton/next state=`⟨*new state*⟩ (no default)

当本状态的片段添加完毕后，切换到这个选项指定的状态 ⟨*new state*⟩，如果没有这个选项，就会一直添加本状态的片段，直到出现切换状态（或结束装饰过程）的条件。

在本命令的 ⟨*code*⟩ 中只能使用以下命令:

`\decoration`⟨*name*⟩

⟨*name*⟩ 是已经定义的或已有的装饰类型的名称，指定当前状态的片段是 ⟨*name*⟩ 类型的。如果没有给出这个命令，就默认使用 moveto 装饰类型。

`\beforedecoration`⟨*before code*⟩

⟨*before code*⟩ 是添加本状态的片段之前所需要执行的代码，例如，可以设置该片段中的线段长度、振幅等。可以没有这个命令。

`\afterdecoration`⟨*after code*⟩

⟨*after code*⟩ 是添加本状态的片段之后所需要执行的代码。可以没有这个命令。

在定义状态时，下面的宏可能用得上。

`\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。

`\pgfmetadecoratedpathlength`

这个宏保存输入路径的长度。

`\pgfmetadecoratedcompleteddistance`

这个宏保存当下的输入路径的“已装饰部分”的长度。

`\pgfmetadecoratedremainingdistance`

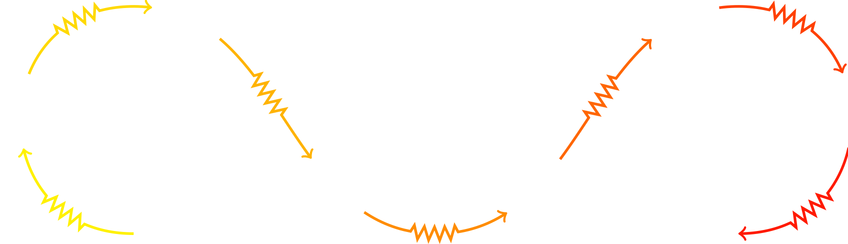
这个宏保存当下的输入路径的“未装饰部分”的长度。

`\pgfmetadecoratedinputsegmentcompleteddistance`

这个宏保存当下的子输入路径的“已装饰部分”的长度。

\pgfmetadecoratedinputsegmentremainingdistance

这个宏保存当下的子输入路径的“未装饰部分”的长度。



```
\pgfdeclaremetadecoration{arrows}{initial}
{
  \state{initial}[width=0pt, next state=arrow]
  {
    \pgfmathdivide{100}{\pgfmetadecoratedpathlength}
    \let\factor\pgfmathresult
    \pgfsetlinewidth{1pt}
    \pgfset{/pgf/decoration/segment length=4pt}
  }
  \state{arrow}[
      switch if less than=\pgfmetadecorationsegmentlength to final,
      width=\pgfmetadecorationsegmentlength/3,
      next state=zigzag]
  {
    \decoration{curveto}
    \beforedecoration
    {
      \pgfmathparse{\pgfmetadecoratedcompleteddistance*\factor}
      \pgfsetcolor{red!\pgfmathresult!yellow}
      \pgfpathmoveto{\pgfpointmetadecoratedpathfirst}
    }
  }
  \state{zigzag}[width=\pgfmetadecorationsegmentlength/3, next state=end arrow]
  {
    \decoration{zigzag}
  }
  \state{end arrow}[width=\pgfmetadecorationsegmentlength/3, next state=move]
  {
    \decoration{curveto}
    \beforedecoration{\pgfpathmoveto{\pgfpointmetadecoratedpathfirst}}
    \afterdecoration
    {
      \pgfsetarrowsend{to}
      \pgfusepath{stroke}
    }
  }
  \state{move}[width=\pgfmetadecorationsegmentlength/2, next state=arrow]{}
  \state{final}{}
}
\tikz[rotate=90] \draw[decorate,decoration={arrows,meta-segment length=2cm}]
  (0,0) .. controls (0,2) and (3,2) .. (3,0)
        .. controls (3,-2) and (0,-2) .. (0,-4)
        .. controls (0,-6) and (3,-6) .. (3,-8)
        .. controls (3,-10) and (0,-10) .. (0,-8);
```

103.5.2 预定义的 Meta-decorations

目前没有预定义的 Meta-decorations.

103.5.3 {pgfmetadecoration} 环境

{pgfmetadecoration} 环境与 {pgfdecoration} 环境的用法是类似的。

在 LaTeX 中如下使用 {pgfmetadecoration} 环境:

```
\begin{pgfmetadecoration}{⟨name⟩}
 ⟨environment content⟩
\end{pgfmetadecoration}
```

在 plainTeX 中如下使用 {pgfmetadecoration} 环境:

```
\pgfmetadecoration{⟨name⟩}
  ⟨environment contents⟩
\endpgfmetadecoration
```

在 ConTeXt 中如下使用 {pgfmetadecoration} 环境:

```
\startpgfmetadecoration{⟨name⟩}
  ⟨environment contents⟩
\stoppgfmetadecoration
```

104 使用路径

104.1 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:

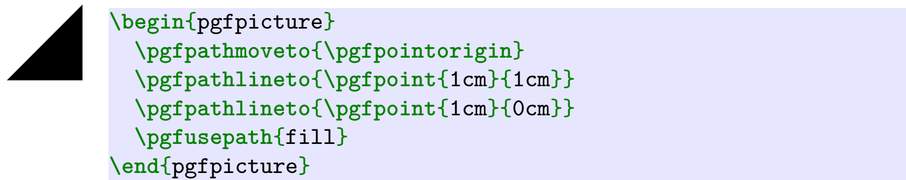
1. 画出路径,stroke 或 draw.
2. 给路径添加箭头,arrow tips.
3. 填充颜色,fill.
4. 作为剪切路径来剪切之后的路径,clip.
5. 画阴影,shade.
6. 用作边界盒子,use as bounding box.

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。

104.2 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 分组的限制。不过在将来的版本中，可能会改变这一状况。

104.2.1 图形参数：线宽 Line Width

`\pgfsetlinewidth`{⟨*line width*⟩}

⟨*line width*⟩ 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. 593 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。

`\pgflinewidth`

这是个 TEX 尺寸（不是用 `\def` 定义的宏），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
```

104.2.2 图形参数：线冠 Caps 与交接 Joins

`\pgfsetbuttcap`

规定线冠为 butt cap，见 §15.3.1.

`\pgfsetroundcap`

规定线冠为 round cap，见 §15.3.1.

`\pgfsetrectcap`

规定线冠为 square cap，见 §15.3.1.

`\pgfsetroundjoin`

规定交接为 round join，见 §15.3.1.

`\pgfsetbeveljoin`

规定交接为 bevel join，见 §15.3.1.

`\pgfsetmiterjoin`

规定交接为 miter join，见 §15.3.1.

\pgfsetmiterlimit{⟨*miter limit factor*⟩}

设置交接的极限因子，见 §15.3.1.

104.2.3 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”。参考 §15.3.2.

使用命令 \pgfsetdash{}{0pt} 得到实线。

104.2.4 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 分组的限制。如果使用 LaTeX 的颜色命令 \color{⟨*color*⟩} 设置颜色，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。

\pgfsetcolor{⟨*color*⟩}

这个命令同时设置线条颜色的填充颜色，其有限范围受到绘图环境的限制（而不是 TeX 分组）。

104.2.5 线条透明度

使用命令 \pgfsetstrokeopacity{⟨*value*⟩} 设置线条透明度，见 §115.

104.2.6 双线的内线

“双线功能”参考 §15.3.4. 如果给双线加箭头，那么程序会先画出外线，再画内线，然后画箭头，而箭头的尺寸参照内线的线宽。

\pgfsetinnerlinewidth{⟨*dimension*⟩}

程序读取这个命令时，就默认对当前路径启用双线功能，本命令规定内线的线宽。在默认下这里的 ⟨*dimension*⟩ 是 0pt，在这个尺寸下根本不会有画内线的动作（注意区别“不画线”与“画出线宽为 0pt 的线”），此时如果加箭头就加在“外线”上。也就是说，⟨*dimension*⟩ 是 0pt 的时候就会关闭双线功能。

这个命令的有限范围受到 TEX 分组的限制，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 分组的限制。

104.3 给路径加箭头

关于箭头的详细规定参考 §16.2. 注意不能给封闭路径或者剪切路径加箭头。

`\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 分组的限制。

`\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 分组的限制。

```
\begin{pgfpicture}
  \pgfsetarrowsstart{Latex[length=10pt]}
  \pgfsetarrowsend{Computer Modern Rightarrow}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfsetarrows`{⟨*argument*⟩}

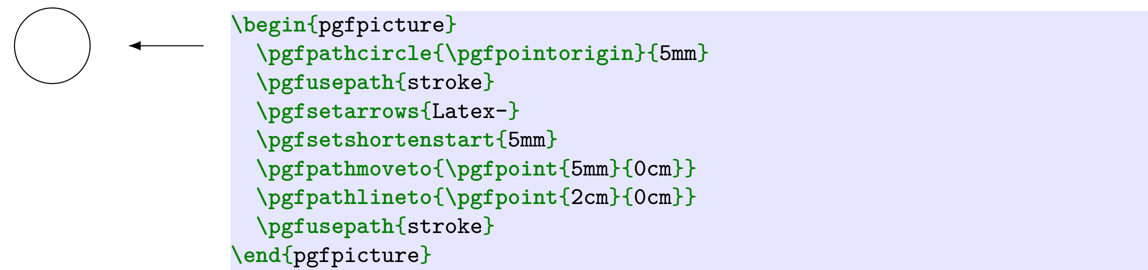
这个命令指定箭头的样式，⟨*argument*⟩ 是指定箭头样式的那些句法格式，参考 §16.4.

```
\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. 94 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*⟩.

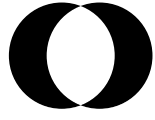
104.4 填充路径

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（§15.5）。

104.4.1 图形参数：判断内部点的规则

`\pgfseteorule`

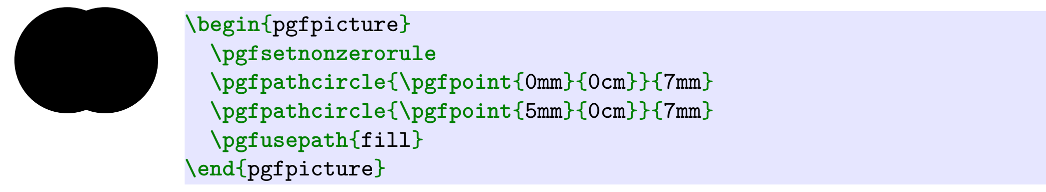
这个命令指定奇偶规则，其有效范围受到 TEX 分组的限制。



```
\begin{pgfpicture}
  \pgfseteorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

`\pgfsetnonzerorule`

这个命令指定非零规则，其有效范围受到 TEX 分组的限制。这是默认的规则。



```
\begin{pgfpicture}
  \pgfsetnonzerorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

104.4.2 图形参数：填充色

\pgfsetfillcolor{⟨*color*⟩}

这个命令指定填充用的颜色。

104.4.3 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，见 §115.

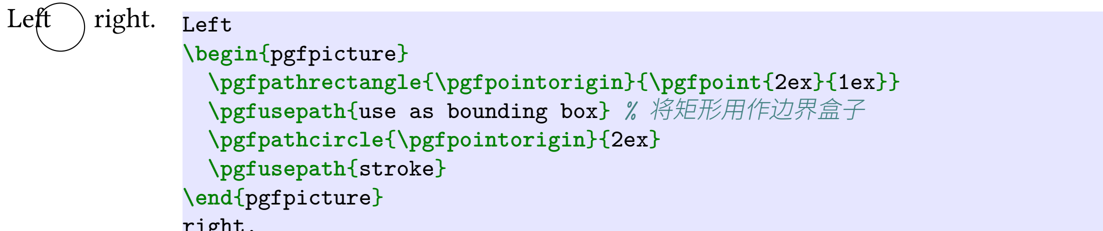
104.5 剪切路径

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。

104.6 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 分组的限制。命令 \pgfresetboundingbox 通常与 \pgfusepath{use as bounding box} 配合使用。



```
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.
```

105 定义新的箭头

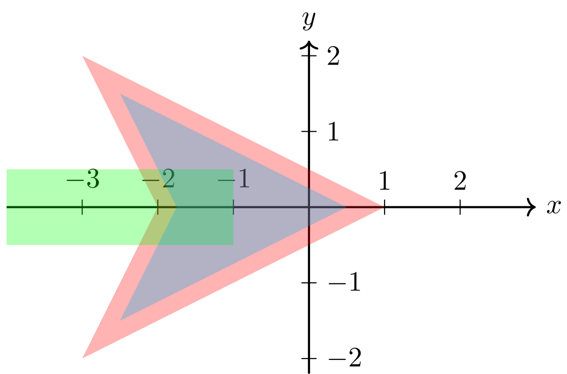
105.1 Overview

§16 介绍了 TikZ 中箭头的用法，不过 TikZ 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。

105.2 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。

105.3 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`, 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（包括选项的值）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. 当一个组合，例如，`foo[length=5pt,open]` 第一次被使用时，PGF 会检查所保存的箭头定义并执行定义中的 setup code, 使得选项被处理为相应的值（目前的这个例子中，长度的值被处理为 5pt）。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。对每一个箭头名称、箭头选项的组合，setup code 只执行一次。

4. 处理 drawing code, 即绘制箭头的路径命令（在前面的例子中，箭头只有 4 个线段）。drawing code 会被放入一个“沙盒”（sandbox）中来执行，其中使用底层（basic layer）命令来绘图，并将绘图结果“缓存”（cache）。一旦这个缓存创建成功，当再次出现同一个箭头名称、箭头选项的组合时（目前例子的组合是 `foo[length=5pt,open]`），通常就不必重复计算了。不过在两种例外情况下，每当使用箭头时，drawing code 就会被执行：第一，箭头定义中使用了 `cachable=false` 选项，当 drawing code 中含有底层绘图命令不能接受的内容时（例如含有 `\pgftext` 添加的文本），就需要使用这个选项；第二，箭头定义中使用了 `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.

   因为 drawing code 可能被执行多次，setup code 可能只执行一次，而执行 drawing code 时可能需要用到 setup code 中计算出来某些个值（为绘制箭头做准备是 setup code 的一个任务），这就需要用命令 `\pgfarrowssave` 来保存 setup code 中计算出来某些个值，以供执行 drawing code 时使用。每执行 drawing code 之前，都会先调出保存在命令 `\pgfarrowssave` 中的值。

105.4 自定义箭头

`\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”. ⟨*config*⟩ 中包括 `name`，`parameters`，`setup code`，`drawing code` 等选项。下面介绍可以在 ⟨*config*⟩ 中使用的内容。

首先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。

新定义一种箭头。 ⟨*config*⟩ 是个键值列表，其中可以使用以下键（key）。

- `name=`⟨*name*⟩ 或者 `name=`⟨*start name*⟩`-`⟨*end name*⟩

  这个选项规定箭头的名称。在当前的 TEX 分组内，可以多次使用这个命令定义多个箭头，后定义的箭头名称不能重复使用已有的或前面已定义的箭头名称，如果重复就会覆盖之前的同名箭头的定义。在 PGF 中有一个惯例，以大写字母开头的名称表示一个完备的箭头，以小写字母开头的名称用于定义 shorthand 箭头。

  名称里面可以使用一个连字符（上面第二个形式）。名称 ⟨*start name*⟩ 用于路径始端，名称 ⟨*end name*⟩ 用于路径末端。

- `parameters={`⟨*list of macros*⟩`}`

这个选项的值 ⟨*list of macros*⟩ 中列出 setup code 或 drawing code 涉及的各种参数，这里的参数表达为宏的形式（见 §105.5）。例如，假设定义的箭头可以具有长度选项 length 和宽度选项 width, 与这两个选项对应的尺寸寄存器是 \pgfarrowlength 和 \pgfarrowwidth, 将这两个寄存器用在 setup code 或 drawing code 中，那么就可以写下：

```
parameters={\the\pgfarrowlength \the\pgfarrowwidth}
```

也可以写成带有逗号分隔的形式：

```
parameters={\the\pgfarrowlength, \the\pgfarrowwidth}
```

命令 \the\pgfarrowlength 得到的是保存在寄存器 \pgfarrowlength 中的值。

⟨*list of macros*⟩ 会被放入一个 \csname-\endcsname 组合中。

线宽参数宏 \pgflinewidth 和内线线宽参数宏 \pgfinnerlinewidth 通常不放在这里的，这两个宏可以直接用在 setup code 或 drawing code 中。

箭头依赖的各个参数应当全都在 ⟨*list of macros*⟩ 中列出。

- setup code={⟨*code*⟩}

  当使用箭头时，保存在 parameters 中的各个参数会被展开，程序会检查这些参数的值是否遇到过。如果没有遇到过就执行这里的 ⟨*code*⟩, 并且仅执行一次。在 ⟨*code*⟩ 中可以：(1) 计算各种寄存器的值；(2) 利用各种的寄存器值 (在箭头坐标系中) 指定箭头的特征点 (3) 将某些个计算出来的寄存器“特别保存”，被“特别保存”的寄存器将用于 drawing code 中。在 ⟨*code*⟩ 中可以使用下面的宏。

  \pgfarrowssettipend{⟨*dimension*⟩}

  这个宏指定箭头的尖端的位置。因为总是假定箭头的 tip end 位于箭头坐标系的 x 轴上，所以这里的尺寸 ⟨*dimension*⟩ 确定箭头坐标系的 x 轴上的一个点，该点作为箭头的尖端。对于目前的例子，应该写下：

  ```
  \pgfarrowssettipend{1cm}
  ```

  注意这里的 ⟨*dimension*⟩ 并不用命令 \pgfmathsetlength 来解析,而是执行 \pgf@x=⟨*dimension*⟩.
  这里的 ⟨*dimension*⟩ 可以是一个算式，不过算式要以尺寸 (或尺寸寄存器的赋值形式) 开头，例如：

  ```
  \pgfarrowssettipend{1cm\advance\pgf@x by-.5\pgflinewidth}
  ```

  如果 setup code 中没有明确给出这个宏，就默认 ⟨*dimension*⟩ 等于 0pt，即箭头的尖端在箭头坐标系的原点。

  在文件《pgfcorearrows.code.tex》中这个命令的定义是：

  ```
  \def\pgfarrowssettipend#1{\pgf@x#1\edef\pgf@arrows@the@tipend{\the\pgf@x
  ↪  }}
  ```

  根据这个定义，下面的命令可行：

  ```
  \pgfarrowssettipend{0.5\pgfarrowlength\ifpgfarrowharpoon\advance\pgf@x
  ↪  by\pgf@xa\fi}
  导致
  ```

```
\pgf@x0.5\pgfarrowlength%
\ifpgfarrowharpoon\advance\pgf@x by\pgf@xa\fi%
\edef\pgf@arrows@the@tipend{\the\pgf@x}
```

也就是说，命令 `\pgfarrowssettipend` 的操作过程是：先为 `\pgf@x` 赋值，然后对 `\pgf@x` 做一番操作，然后将它的值展开并保存在宏 `\pgf@arrows@the@tipend` 中。

下面的几个设置箭头特征点的命令都是这样处理的。

`\pgfarrowssetbackend`{⟨*dimension*⟩}

规定箭头的 back end, 对于目前的例子就是:

```
\pgfarrowssettipend{-3cm}
```

如果 `setup code` 中没有明确给出这个命令，就默认 ⟨*dimension*⟩ 等于 0pt，即箭头的 back end 在箭头坐标系的原点。

⟨*dimension*⟩ 的格式同上一命令。

`\pgfarrowssetlineend`{⟨*dimension*⟩}

设置 line end, 如果 `setup code` 中没有明确给出这个宏，就默认 ⟨*dimension*⟩ 等于 0pt，即路径的 line end 在箭头坐标系的原点。

⟨*dimension*⟩ 的格式同上一命令。

对于目前的例子就是:

```
\pgfarrowssettipend{-1cm}
```

`\pgfarrowssetvisualbackend`{⟨*dimension*⟩}

设置 visual back end，如果 `setup code` 中没有明确给出这个宏，就默认该点与 back end 重合。

⟨*dimension*⟩ 的格式同上一命令。

对于目前的例子就是:

```
\pgfarrowssetvisualbackend{-2cm}
```

`\pgfarrowssetvisualtipend`{⟨*dimension*⟩}

设置 visual tip end，如果 `setup code` 中没有明确给出这个宏，就默认该点与 tip end 重合。

⟨*dimension*⟩ 的格式同上一命令。

`\pgfarrowshullpoint`{⟨*x dimension*⟩}{⟨*y dimension*⟩}

这个宏设置箭头的凸顶点。程序会执行 `\pgf@x=`⟨*x dimension*⟩ 和 `\pgf@y=`⟨*y dimension*⟩.

⟨*x dimension*⟩ 和 ⟨*y dimension*⟩ 的格式同上一命令。

对于目前的例子就是:

```
\pgfarrowshullpoint{1cm}{0pt}
\pgfarrowshullpoint{-3cm}{2cm}
\pgfarrowshullpoint{-3cm}{-2cm}
```

`\pgfarrowsupperhullpoint`{⟨*x dimension*⟩}{⟨*y dimension*⟩}

如果 ⟨*y dimension*⟩ 大于 0，那么这个宏等效于

```
\pgfarrowshullpoint{⟨x dimension⟩}{⟨y dimension⟩}
\pgfarrowshullpoint{⟨x dimension⟩}{-⟨y dimension⟩}
```

也就是说，这个宏能同时设置两个关于 x 轴对称的凸顶点。但如果箭头带有 `harpoon` 选项，则没有第二个凸顶点。

如果 ⟨*y dimension*⟩ 不大于 0，则这个宏等效于

```
\pgfarrowshullpoint{⟨x dimension⟩}{⟨y dimension⟩}
```

`\pgfarrowssave{⟨macro⟩}`

这里的 ⟨*macro*⟩ 应当是某个有确定值的寄存器名称 (以反斜线开头的宏的形式)，此命令将 ⟨*macro*⟩ 的值做“特别保存”，在 `drawing code` 中可以使用 ⟨*macro*⟩ 来引用这个值。例如：

```
\pgfarrowssave{\pgf@x}
\pgfarrowssave{\pgf@y}
```

`\pgfarrowssavethe{⟨register⟩}`

作用与 `\pgfarrowssave` 类似。这里 ⟨*register*⟩ 是寄存器，寄存器的值 `\the`⟨*register*⟩ 会被“特别保存”起来。例如：

```
\pgfarrowssavethe{\pgfarrowlength}
\pgfarrowssavethe\pgfarrowwidth
```

使用这个宏后，要注意在 `drawing code` 中正确使用 ⟨*register*⟩.

- `drawing code={⟨code⟩}`

⟨*code*⟩ 是绘制箭头的路径命令，其中可以使用在 `setup code` 中 (利用命令`\pgfarrowssavethe`) 引入的各个参数宏，以及线宽参数宏。对于每一组选项、选项值组合，⟨*code*⟩ 至少会被执行一次，第一次执行 ⟨*code*⟩ 时产生的底层命令 (low-level commands) 会被缓存。

程序会开启箭头坐标系来绘制箭头。⟨*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 ⟨*code*⟩ 中需要注意：

(i) 在 ⟨*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。你可以使用“quick”版的命令，例如 `\pgfusepathqstroke`→P. 817，使用这种命令不会出现给箭头添加箭头的情况。

(ii) 如果在 ⟨*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 ⟨*code*⟩ 被执行时，⟨*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 ⟨*code*⟩ 时，高层的坐标变换矩阵会被设为单位矩阵。

- `cache=⟨true or false⟩`

这个选项的默认值是 true，此时，对于每个箭头、选项（包括选项值）的组合，只执行 ⟨*code*⟩ 一次，所用到的选项、选项值、由 ⟨*code*⟩ 产生的底层 (low-level) 命令会被缓存 (be cached)，以

备稍后重复使用。但是如果 drawing code 中含有不能缓存的代码，例如 \pgftext, 应当设置 cache=false.

- bending mode=⟨*mode*⟩

  当给路径添加箭头时，箭头可能带有 bend 选项，即要求箭头随着路径曲线的弯曲而弯曲。这个 key 决定箭头的弯曲方式。

  如果箭头不需要弯曲，应当设置 bending mode=none, 此时默认使用 flex 选项，见 §16.3.8.

  如果箭头需要弯曲，那么 bending mode（至少）有两种模式可选：orthogonal（正交的）或者 polar（极坐标的），见 §108.4.7.

  这个选项的默认值是 bending mode=orthogonal.

- defaults=⟨*arrow keys*⟩

  这个选项设置关于箭头的默认选项值，例如：

  ```
  defaults={length=4cm, width=2cm}
  ```

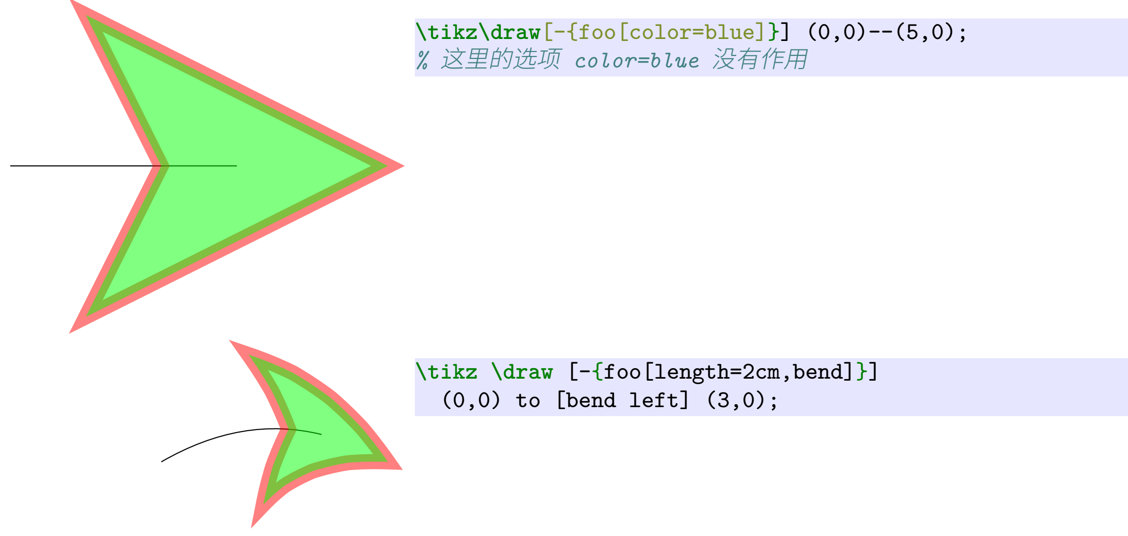
  ⟨*arrow keys*⟩ 会在其它箭头选项之前被执行。

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);
```

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);
```

105.5 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。

105.5.1 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}
```

105.5.2 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.

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相反数，使得箭头的 back end 与 tip end 互换，箭头的 visual back end 与 visual tip end 互换。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。

105.5.3 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.

105.5.4 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。前面提到了对于 setup code 和 drawing code 的缓存，而这里有“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。为了能让选项 depth 进入“缓存”，需要使用下面的命令：

\pgfarrowsaddtooptions{⟨*code*⟩}

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{⟨*code*⟩}

这个命令类似上面的 \pgfarrowsaddtooptions, 只是本命令的 ⟨*code*⟩ 具有 late 特征，即 ⟨*code*⟩ 会在其它（没有 late 特征的）选项（命令）都被处理过之后才被执行。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{⟨*dimension register*⟩}

每当使用箭头时，这个命令会把寄存器 ⟨*dimension register*⟩ 与箭头选项 /pgf/arrows keys/scale length 的值相乘，即改变寄存器 ⟨*dimension register*⟩ 的值，然后再用该寄存器绘制箭头。
这个命令只能用于导言中。

\pgfarrowsaddtowidthscalelist{⟨*dimension register*⟩}

这个命令与上面的 \pgfarrowsaddtolengthscalelist 类似，会把寄存器 ⟨*dimension register*⟩ 与箭头选项 /pgf/arrows keys/scale width 的值相乘，即改变寄存器 ⟨*dimension register*⟩ 的值，然后再用该寄存器绘制箭头。

这个命令只能用于导言中。

\pgfarrowsthreeparameters{⟨*line-width dependent size specification*⟩}

这里 ⟨*line-width dependent size specification*⟩ 是一个，或两个，或三个数值，数值之间用空格分隔。本命令会产生三个花括号，把这些数值依次分别放入花括号内，缺失的数值用 0 代替，然后将这个结果保存在宏 \pgfarrowstheparameters 中。

例如，执行命令 \pgfarrowsthreeparameters{2pt 1} 后，保存在 \pgfarrowstheparameters 中的值就是 {2.0pt}{1}{0}.

{2.0pt}{1}{0}

```
\makeatletter
\def\showvalueofmacro#1{%
  \texttt{%
    \expandafter\expandafter\expandafter\expandafter\expandafter%
    \expandafter\expandafter\pgfutil@gobble\expandafter\expandafter%
    \expandafter\string\expandafter\csname#1\endcsname}%
}
\makeatother
\pgfarrowsthreeparameters{2pt 1}
\showvalueofmacro\pgfarrowstheparameters
```

\pgfarrowslinewidthdependent{⟨*dimension*⟩}{⟨*line width factor*⟩}{⟨*outer factor*⟩}

这个命令用于定义类似箭头选项 length 那样的选项，length 的使用格式是：

length=⟨*dimension*⟩ ⟨*line width factor*⟩ ⟨*outer factor*⟩

其意思是长度选项 length 的值依赖这里列出的三个参数。程序参照这三个参数计算出一个尺寸，将这个尺寸作为 length 的值。

本命令参照这三个参数计算出一个尺寸，并将这个尺寸保存在寄存器 \pgf@x 中，计算方式参考选项 /pgf/arrow keys/length→P. 84 的解释。

选项 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 {⟨x⟩}{⟨y⟩}{⟨z⟩} 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. 命令 \pgfarrowsaddtolateoptions 把 \pgfarrowdepth 加入缓存。

\pgfarrowslengthdependent{⟨*dimension*⟩}{⟨*length factor*⟩}{⟨*dummy*⟩}

这个命令用于定义类似箭头选项 width' 那样的选项，本命令参照这三个参数计算出一个尺寸，并将这个尺寸保存在寄存器 \pgf@x 中，计算方式参考选项 /pgf/arrow keys/width'→P. 85 的解释。例如可以定义选项 depth'，使得这个选项的值依赖这三个参数：

```
\pgfkeys{/pgf/arrow keys/depth'/.code={%
  \pgfarrowsthreeparameters{#1}%
  \expandafter\pgfarrowsaddtolateoptions% 使用有 late 特征的命令
    \expandafter{\expandafter\pgfarrowslengthdependent\pgfarrowstheparameters
    ↪  \pgfarrowdepth\pgf@x}%
}
```

↓ ↓

以下是文件《pgfcorearrows.code.tex》中命令 \pgfarrowslinewidthdependent 的定义

```
\def\pgfarrowslinewidthdependent#1#2#3{%
  \pgf@x#1%
  \ifdim\pgfinnerlinewidth>0pt%
    \pgf@arrows@inner@line@width@dep{#2}{#3}%
  \else%
    \advance\pgf@x by#2\pgflinewidth%
  \fi%
}
\def\pgf@arrows@inner@line@width@dep#1#2{%
  % #1*((1-#2/2)*full line width - (#2/2) * inner line width)
  % Compute "real" line width
  \pgf@xa\pgflinewidth%
  \pgf@xb-.5\pgflinewidth%
  \pgf@xb#2\pgf@xb%
  \advance\pgf@xa by\pgf@xb%
  \pgf@xb\pgfinnerlinewidth%
  \pgf@xb-.5\pgf@xb
  \advance\pgf@xa by#2\pgf@xb%
```

```
  \advance\pgf@x by#1\pgf@xa%
}
```

执行命令 \pgfarrowslinewidthdependent{$\langle x\rangle$}{$\langle y\rangle$}{$\langle z\rangle$} 导致如下计算:

1. 将 \pgf@x 的值设置为 $\langle x\rangle$.
2. 用 \ifdim 执行一个条件句:
   - 如果 \pgfinnerlinewidth 大于 0pt, 则执行 \pgf@arrows@inner@line@width@dep{$\langle y\rangle$}{$\langle z\rangle$}, 得到 \pgf@x 的值是:

$$\langle x\rangle + \langle y\rangle \times \left((1-\frac{\langle z\rangle}{2}) \times \texttt{\textbackslash pgflinewidth} - \frac{\langle z\rangle}{2} \times \texttt{\textbackslash pgfinnerlinewidth}\right)$$

   - 如果 \pgfinnerlinewidth 不大于 0pt, 则把 \pgf@x 的值加上 $\langle y\rangle$与\pgflinewidth 的乘积, 于是 \pgf@x 的值就是 $\langle x\rangle+\langle y\rangle\times$\pgflinewidth.

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%
```

```
32      \csname pgfmathatan2@\endcsname{\pgfmath@tonumber\pgfarrowinset}{
        ↪  \pgfmath@tonumber\pgf@ya}%
33      \pgf@ya\pgfmathresult pt%
34      \advance\pgf@yb by-\pgf@ya%
35      \pgf@yb.5\pgf@yb% half angle in yb
36      \pgfmathtan@{\pgf@sys@tonumber\pgf@yb}%
37      \pgfmathreciprocal@{\pgfmathresult}%
38      \pgf@yc\pgfmathresult\pgfarrowlinewidth%
39      \pgf@yc.5\pgf@yc%
40      \advance\pgf@ya by\pgf@yb%
41      \pgfmathsincos@{\pgf@sys@tonumber\pgf@ya}%
42      \pgf@ya\pgfmathresulty\pgf@yc% ya is the back miter
43      \pgf@yb\pgfmathresultx\pgf@yc% yb is the top miter
44      \ifdim\pgfarrowinset=0pt%
45        \pgf@ya.5\pgfarrowlinewidth% easy: back miter is half linewidth
46      \fi
47      % Compute inset miter length:
48      \pgfmathdivide@{\pgf@sys@tonumber\pgfarrowinset}{\pgf@sys@tonumber\pgfarrowwidth
        ↪  }%
49      \let\pgf@temp@quot\pgfmathresult%
50      \pgf@x\pgfmathresult pt%
51      \pgf@x\pgfmathresult\pgf@x%
52      \pgf@x4\pgf@x%
53      \advance\pgf@x by1pt%
54      \pgfmathsqrt@{\pgf@sys@tonumber\pgf@x}%
55      \pgf@yc\pgfmathresult\pgfarrowlinewidth% yc is inset miter
56      \pgf@yc.5\pgf@yc%
57      % Inner length (pgfutil@tempdima) is now arrowlength - front miter - back miter
58      \pgfutil@tempdima\pgfarrowlength%
59      \advance\pgfutil@tempdima by-\pgf@xc%
60      \advance\pgfutil@tempdima by-\pgf@ya%
61      \pgfutil@tempdimb.5\pgfarrowwidth%
62      \advance\pgfutil@tempdimb by-\pgf@yb%
63      % harpoon miter correction
64      \ifpgfarrowroundjoin
65        \pgfarrowssetbackend{\pgf@ya\advance\pgf@x by-.5\pgfarrowlinewidth}
66      \else
67        \pgfarrowssetbackend{0pt}
68      \fi
69      \ifpgfarrowharpoon
70        \pgfarrowssetlineend{\pgfarrowinset\advance\pgf@x
71          by\pgf@yc\advance\pgf@x by.5\pgfarrowlinewidth}
72      \else
73        \pgfarrowssetlineend{\pgfarrowinset\advance\pgf@x by\pgf@yc\advance\pgf@x
          ↪  by-.25\pgfarrowlinewidth}
74        \ifpgfarrowreversed
75          \ifdim\pgfinnerlinewidth>0pt
76            \pgfarrowssetlineend{\pgfarrowinset}
77          \else
78            \pgfarrowssetlineend{\pgfutil@tempdima\advance\pgf@x by\pgf@ya
              ↪  \advance\pgf@x by-.25\pgfarrowlinewidth}
```

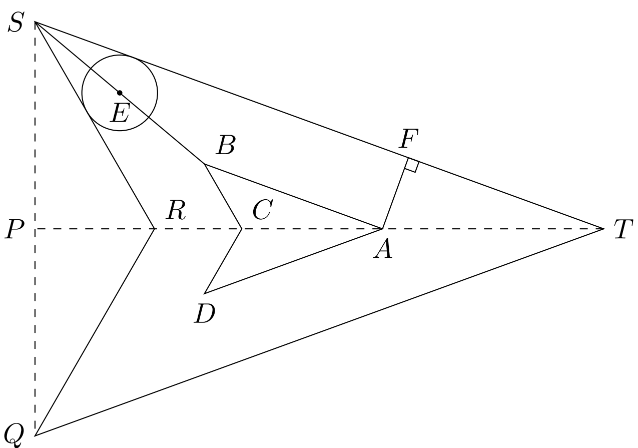
```
79          \fi
80        \fi
81      \fi
82      \ifpgfarrowroundjoin
83        \pgfarrowssettipend{\pgfutil@tempdima\advance\pgf@x by\pgf@ya\advance\pgf@x
        ↪  by.5\pgfarrowlinewidth}
84      \else
85        \pgfarrowssettipend{\pgfarrowlength\ifpgfarrowharpoon\advance\pgf@x by\pgf@xa
        ↪  \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\else
        ↪  0pt\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. 89. 记为 a_{lw}pt.
- 箭头长度 \the\pgfarrowlength, 参考选项 /pgf/arrow keys/length→P. 84. 记为 $a_l$pt.
- 箭头宽度 \the\pgfarrowwidth, 参考选项 /pgf/arrow keys/width→P. 85. 记为 $a_w$pt.
- 箭头内凹尺寸 \the\pgfarrowinset, 参考选项 /pgf/arrow keys/inset→P. 85. 记为 $a_i$pt.
- 条件判断 \ifpgfarrowharpoon h\fi, 参考选项 /pgf/arrow keys/harpoon→P. 87, 参考选项 /pgf/arrow keys/left→P. 87.
- 条件判断 \ifpgfarrowopen o\fi, 参考选项 /pgf/arrow keys/open→P. 88.
- 条件判断 \ifpgfarrowroundjoin j\fi, 参考选项 /pgf/arrow keys/line join→P. 89.



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！

106 Nodes and Shapes

本节介绍 shapes 模块。

```
\usepgfmodule{shapes} % LaTeX and plain TeX and pure pgf
\usepgfmodule[shapes] % ConTeXt and pure pgf
```

这个模块包含的命令可以创建 nodes 和 shapes. PGF 会自动加载这个模块，但如果你只是使用内核 pgfcore, 你也可以手工加载这个模块。

首先注意，这一部分中多处提到“文字盒子的左下角位于锚位置 text 上”，或者其它类似的说法，在某些情况下这实际是指“文字盒子的基点放在锚位置 text 上”，这一点很重要。例如，对于某些 shape 或 node, 如 rectangle, 实际上它的文字盒子的基点放在锚位置 text 上，这对于准确理解它的锚位置 base，mid 是关键的。

106.1 Overview

PGF 有创建 node 和 shape 的完善的机制。一个 node 通常就是一个 shape 加 text, 所谓 shape 一般是个路径，例如 rectangle 或 circle.

106.1.1 创建与索引 node

你可以用命令 \pgfnode 或 \pgfmultipartnode 来创建 node, 并且在当前的 {pgfpicture} 环境中，程序会记住所创建的 node 的位置，因此在当前绘图环境中，创建一个 node 后你可以引用它。创建 node 的命令也支持坐标变换。

106.1.2 锚 Anchors

node 和 shape 都有"锚"(anchors)，即属于 node 或 shape 上的点，或者说是"位置"、"部位"。锚位置 `text` 是 node 的文字盒子的左下角。当指定 node 的某个锚位置后，这个锚位置会处于 node 的锚定点上。当引用一个 node 时，实际上引用的是该 node 的某个锚位置。

锚位置接受坐标变换。

106.1.3 shape 的"层"Layers

最简单的 shape 是 coordinate, 它只有一个锚位置，即 `center`, 它的文字内容通常是空的。

一个 shape 通常有好几个"层"，例如，当填充一个 node 时，填充色位于文字之下，文字不会被填充色遮挡，因为填充色与文字处于不同的层上。

一个 shape 通常有 7 个层:

1. behind the background layer, 画出或者填充 shape 路径的命令通常处于这一层上。
2. background path layer
3. before the background path layer
4. label layer, 盛放 node 的文字的盒子(box)处于这一层上。
5. behind the foreground layer
6. foreground path layer，当画出文字后，再画出这一层的内容。
7. before the foreground layer

106.1.4 Node Parts

一个 shape 或 node 可以带有文字，文字会被放入一个盒子内。多数 shape 或 node 只有一个文字盒子，文字盒子的左下角会处于锚位置 text 上。有的 shape 或 node 有多个文字盒子，这样它的文字就分成了数个部分，或者说它分成了数个部分，这些部分叫作"node parts"，有数个 node parts 的就称为 multipart shape 或 multipart node.

定义一个 multipart shape 时，每个 node part 都定义自己的名称，每个部分的文字都会被放入一个盒子中，对于每个 node part 都应该定义一个锚位置来放置文字盒子。node part 的名称与其中盛放文字的盒子的名称是对应的，例如，某个 node part 的名称是 `XYZ`，那么该部分中盛放文字的盒子名称就应该是 `\pgfnodepartXYZbox`, 这个盒子会参照该部分的某个指定的锚位置来放置。

106.2 创建 node

106.2.1 创建简单 node

`\pgfnode{⟨shape⟩}{⟨anchor⟩}{⟨label text⟩}{⟨name⟩}{⟨path usage command⟩}`

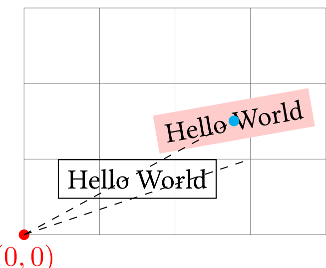
这个命令创建只有一个 node part 的 node.

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.

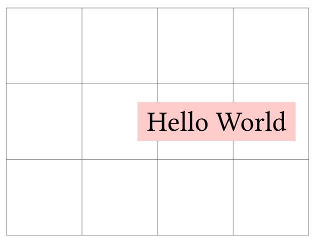
⟨*name*⟩ 是所创建的 node 的名称，用于之后索引该 node. 如果没有 ⟨*name*⟩, 那么在画出该 node 之后，程序就会“忘记”这个 node.

⟨*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{10}
    \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=10] (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{10}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgftransformresetnontranslations
    \pgfnode{rectangle}{center}
      {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
  }
\end{tikzpicture}
```

106.2.2 创建 Multi-Part Nodes

`\pgfmultipartnode{⟨shape⟩}{⟨anchor⟩}{⟨name⟩}{⟨path usage command⟩}`

这个命令可以创建 multipart node.

创建 multipart node 时, 首先你要使用具有多个部分的 shape, 因为每一部分文字都会被放入一个 TEX

盒子中，所以在使用本命令之前，你需要先定义数个盒子，用来盛放各个部分的文字。

以预定义的 circle split 为例 (见程序库 shapes.multipart 的说明)，这个 shape 是圆形的，一个水平的直径将这个圆分为上下两个 node part，即 text 部分和 lower 部分，如下所示：

text 部分
lower 部分

```
\begin{tikzpicture}
  \node [circle split,draw,fill=red!20]
  {
    text 部分
    \nodepart{lower}
    lower 部分
  };
\end{tikzpicture}
```

注意这两个 node part 的名称分别是 `text` 和 `lower`, 因此与这两个名称相对应，定义两个盒子：

```
\newbox\pgfnodeparttextbox% 在文件《pgfmoduleshapes.code.tex》中
\newbox\pgfnodepartlowerbox% 在文件《pgflibraryshapes.multipart.code.tex》中
```

第一个盒子放在 `text` 部分，第二个盒子放在 `lower` 部分。这两个盒子内可以盛放 TEX 盒子能接受的任何内容，例如文字，表格环境，数学公式等等。

circle split 的上下两个部分分别定义了锚位置 text 和 lower, 上部文字盒子的左下角放在锚位置 text 上，下部文字盒子的左下角放在锚位置 lower 上。

q_1
01

```
\setbox\pgfnodeparttextbox=\hbox{$q_1$}
\setbox\pgfnodepartlowerbox=\hbox{01}
\begin{pgfpicture}
  \pgfmultipartnode{circle split}{center}{my state}{\pgfusepath
  ↪ {stroke}}
\end{pgfpicture}
```

注意:如上面的例子所示,你可以在 `{pgfpicture}` 环境之前使用命令 `\setbox` 来定义盒子。如果你在 `{pgfpicture}` 环境内使用命令 `\setbox` 来定义盒子，那么你可能需要用命令 `\pgfinterruptpath` 和 `\endpgfinterruptpath` 将盒子内容包裹起来。

`\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.

`\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*⟩.

下面的选项影响 node 的尺寸。

`/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*⟩

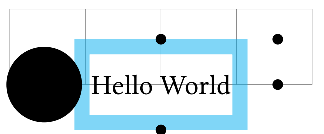
本选项同时设置 /pgf/inner xsep 和 /pgf/inner ysep 的值为 ⟨*dimension*⟩.

/pgf/outer xsep=⟨*dimension*⟩ (no default, initially .5\pgflinewidth)

/tikz/outer xsep=⟨*dimension*⟩

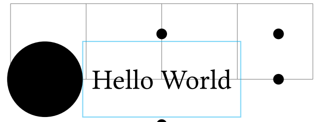
这个选项在水平方向上，设置 node 的背景边界形状路径与"外部锚位置"的间距。例如，如果 ⟨*dimension*⟩ 是 1cm，那么锚位置 east 与背景形状路径的边界的距离就是 1cm. 这个选项值只是个"推荐值"。

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*⟩.

106.2.3 另一种添加 node 的方法

用前面介绍的命令 `\pgfnode` 创建 node 后，node 就立即被添加到当前图形中的某个位置上，之后就难以再改变这个 node 的位置。使用下面介绍的命令 `\pgfpositionnodelater` 可以设计并保存 node，然后再用命令 `\pgfpositionnodenow` 在需要的位置放置保存的 node，这样就把设计 node 的步骤与放置 node 的步骤分开了。

先看一个例子。


```
\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}
```

`\pgfpositionnodelater{`⟨*macro name*⟩`}`

这个命令的有效范围受到“域”（scope）的限制。在一个 scope 内，本命令之后定义的所有 node 都受到本命令的作用，其作用是：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，程序仍然计算盛放 node 的边界盒子，这个盒子的上、下、左、右边界值会被保存在宏 \pgfpositionnodelaterminx, \pgfpositionnodelaterminy, \pgfpositionnodelatermaxx, \pgfpositionnodelatermaxy 之内，注意这个计算的前提是设定 node 的锚定点为原点。

宏 ⟨*macro name*⟩ 是需要在本命令之前定义的，这个宏的定义要包含下面的宏。在定义宏 ⟨*macro name*⟩ 时，你需要把下面宏的内容转移到其它自定义宏中，如同前面的例子所示。

此命令的定义是：

```
\def\pgfpositionnodelater#1{%
  \let\pgf@positionnodelater@macro=#1%
  \ifx\pgf@positionnodelater@macro\relax%
    \pgflatenodepositioningfalse%
  \else%
    \pgflatenodepositioningtrue%
  \fi%
}%
\newif\ifpgflatenodepositioning
\pgfpositionnodelater{\relax}%
```

可见使用本命令后，如果 ⟨*macro name*⟩ 不是 \relax, 就有 \pgflatenodepositioningtrue.

\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 的内容转移到另一个你自定义的盒子中。

```
\def\pgfpositionnodelaterbox{0}%
```

\pgfpositionnodelatername

定义的 node 的名称加上前缀 not yet positionedPGFINTERNAL 就是这个宏的内容，这个内容会被临时指定为 node 的名称。当使用命令 \pgfpositionnodenow 时，node 的名称会被改回原来名称。因为在使用命令 \pgfpositionnodenow 之前可能会引用 node 的名称，临时修改它的名称能避免混乱.

\pgfpositionnodelaterminx

node 的边界盒子的左侧边界值保存在这个宏中。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 node 的背景路径。

\pgfpositionnodenow{⟨*coordinate*⟩}

本命令把命令 \pgfpositionnodelater 保存的 node 添加到图形中，并且会按照 ⟨*coordinate*⟩ 来平移 node；如果 node 的定义中有类似 at=⟨*position*⟩ 的位置选项，还会有参照 ⟨*position*⟩ 的平移，如前面的例子所示。

在定义宏 ⟨*macro name*⟩ 时，转存了盒子、尺寸值。在本命令之前，你需要恢复之前转存的宏及其值，因为本命令用到这几个宏及其值。

如果设置 ⟨*macro name*⟩=\relax, 就会取消整个机制。当用命令 \pgfpositionnodenow 把 node 添加到图形上后，程序会自动设置 ⟨*macro name*⟩=\relax.

此命令的定义是:

```
\def\pgfpositionnodenow#1{%
  \pgfinterruptpath%
  {%
    \pgfpointtransformed{#1}%
    \edef\pgf@temp@shift{\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}
    \pgftransformreset%
    \pgftransformshift{\pgf@temp@shift}%
    \pgfsys@pictureboxsynced\pgfpositionnodelaterbox%
    \pgf@shift@node{\pgfpositionnodelatername}{\pgf@temp@shift}%
    % Bounding box update...
    \pgfpointtransformed{\pgfqpoint{\pgfpositionnodelaterminx}{
    ↪  \pgfpositionnodelaterminy}}%
    \pgf@protocolsizes{\pgf@x}{\pgf@y}
    \pgfpointtransformed{\pgfqpoint{\pgfpositionnodelatermaxx}{
    ↪  \pgfpositionnodelatermaxy}}%
    \pgf@protocolsizes{\pgf@x}{\pgf@y}
    % Naming and callbacks
    \expandafter\pgfpositionnodenow@rename\pgfpositionnodelatername\relax%
  }%
  % Late setup%
  {%
    \csname pgf@lns@\pgfpositionnodelatername\endcsname%
    \expandafter\global\expandafter\let\csname pgf@lns@\pgfpositionnodelatername
    ↪  \endcsname\relax%
  }%
  \endpgfinterruptpath%
}%
```

\pgfnodepostsetupcode{⟨*node name*⟩}{⟨*code*⟩}

在命令 \pgfpositionnodelater 的有效范围内使用这个命令，⟨*code*⟩ 会被保存，当名称为 ⟨*node name*⟩ 的 node 被命令 \pgfpositionnodenow 添加到图形上后，⟨*code*⟩ 被执行。如果多次使用这个命令，那么各次的 ⟨*code*⟩ 会被累计。

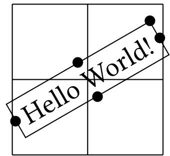
106.3 使用锚位置 Anchors

每个形状（shape）都有各种锚位置（anchors），通常，锚位置 center 会被放在锚定点上。

106.3.1 在一个图形中引用锚位置

`\pgfpointanchor{⟨node⟩}{⟨anchor⟩}`

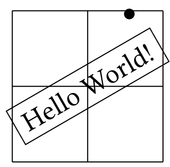
这个命令指定一个坐标点，即名称为 ⟨*node*⟩ 的 node 的锚位置 ⟨*anchor*⟩. 此命令可以用在构建路径的命令，如 `\pgfpathmoveto` 中。




```
\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}
```

你可能觉得上面的例子有点奇怪，看上去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` 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：

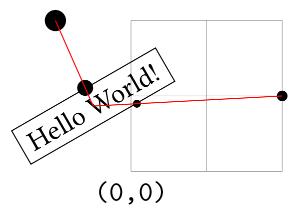



```
\begin{pgfpicture}
  \pgfpathgrid{\pgfpoint{-1cm}{-1cm}}{\pgfpoint{1cm}{1cm}}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}
  {
    \pgftransformreset
    \pgfpointanchor{x}{east}
    \makeatletter
    \xdef\mycoordinate{\noexpand\pgfpoint{\the\pgf@x}{\the\pgf@y}}
    \makeatother
  }
  \pgfpathcircle{\mycoordinate}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

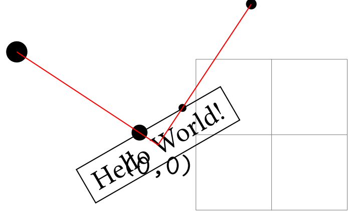
`\pgfpointshapeborder{⟨node⟩}{⟨point⟩}`

这个命令确定一个坐标点。以名称为 ⟨*node*⟩ 的 node 的锚位置 `center` 为始点，做经过坐标点 ⟨*point*⟩ 的射线，射线与 ⟨*node*⟩ 的边界形状路径相交，交点就是本命令确定的点。如果 ⟨*node*⟩ 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*⟩ 的边界形状路径。

注意这个命令与 `\pgfpointanchor` 不同，这个命令不会引入变换矩阵的逆矩阵。




```
\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}
```

106.3.2 跨图引用 node 的锚位置

通常一个 {tikzpicture} 环境或者 {pgfpicture} 环境创建一个图形。在一个图形中，创建、引用 node 都是很直接地，当一个图形创建完毕、添加到页面上后，PGF 就会忘记这个图形在页面上的位置，之后你不能再引用这个图形中的坐标点。假设有前后两个图形，在前一个图形中的 (3,2) 处有名称为 ⟨*node*⟩ 的 node，当在后一个图形中引用前图中的 ⟨*node*⟩ 时，如果程序会认为前图中的 ⟨*node*⟩ 位于后图中的位置 (3,2) 处，这样就会出现错误。

如果要跨图引用 node，即在后面的图形中引用前面图形中 node，就需要让 PGF 记住前后两个图形在页面上的位置，这是让两个图形在页面上联系起来的必要条件。让 PGF 记住图形在页面上的位置，需

要注意以下几点:

1. 使用的驱动应当支持 PGF 记住图形在页面上的位置，pdflatex 支持这一点，目前 dvips 不支持这一点。
2. 在绘图环境中 (或在绘图环境外的某个适当位置) 写下命令 \pgfrememberpicturepositiononpagetrue, 这个命令将 \ifpgfrememberpicturepositiononpage →P. 630 的值设为 true. 如果在绘图环境中使用这个命令，PGF 会记住这个环境创建的图形在页面上的位置。如果在绘图环境外使用这个命令，PGF 会记住此命令之后创建的图形在页面上的位置。

   注意这个命令的有效范围受到 TEX 分组的限制。
3. 让 TEX 编译文件两次。第一次编译会把关于图形位置的数据写入外部文件，而且页面可能变得很奇怪，不过不用担心，第二次编译会让页面变得正常。
4. 注意图形的边界盒子，当后面的图形引用前面图形中 node 的锚位置时，后面图形的边界盒子会包含前面图形中 node 的锚位置，这样就使得后面图形的边界盒子很大，两个图形几乎成为一个图形。因此在后面图形中，在引用前面图形中 node 的锚位置之前，需要在的适当位置使用命令 \pgfusepath{use as bounding box}.

106.4 特殊 node

下面是预定义的 node.

Predefined node `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);}
```

Predefined node `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\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
```

Predefined node `current subpath start`

这个 node 的形状是 `coordinate`, 它是当前“子路径”的起点。

Predefined node `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。

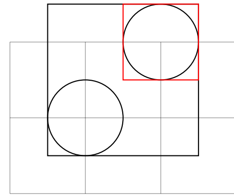
下面的例子中，将一段文字放在当前页面的左下角，注意要使用适当的驱动来支持“记住图形的功能”，否则文字会被插入到当前位置。

```
\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}
```

`/pgf/local bounding box`=⟨*node name*⟩ (no default)

`/tikz/local bounding box`

这个选项用作子环境 `{scope}` 的环境选项，将该子环境创建的图形的边界盒子作成一个 node，其名称为 ⟨*node name*⟩. 这个选项会让程序跟踪子环境的边界盒子，当需要计算的边界盒子太多时，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]  % 这个分组开启一个 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}
```

Text absolutely positioned in the lower left corner.

106.5 定义新的 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.

本节介绍自定义 shape 的方法，这个方法显然比较复杂，因为一个 shape 可以用作 node 的形状，其尺寸要有弹性以容纳各种文字，还要有各种锚位置。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。

106.5.1 一个 shape 具备的要素

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。

在定义箭头时，需要指定箭头的特征点、箭头路径，这些点和路径都需要在一个坐标系——箭头坐标系——内指定。当在路径上添加箭头时，程序首先在箭头坐标系内生成箭头，然后通过变换将箭头放到路径的端点处。

情况对于 shape 的也是类似的。定义 shape 时，需要指定它的 background path, foreground path, 各个锚位置，这些路径和点都需要在一个坐标系——shape 坐标系——内指定。当用命令 `\pgfnode` 或 `\pgfmultipartnode` 调用 shape 时，程序首先在 shape 坐标系内生成它，然后通过变换添加到图形中。

106.5.2 Normal Anchors 与 Saved Anchors

我们将锚位置（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 `northeast`, 这是个 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, 当读取 `rectangle` 的定义时，这个锚位置会被计算并保存。

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，可以使用各种坐标点命令，也可以利用已保存的 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。

106.5.3 定义新 shape 的命令

`\pgfdeclareshape{⟨shape name⟩}{⟨shape specification⟩}`

本命令定义一个名称为 ⟨*shape name*⟩ 的 shape，定义后，⟨*shape name*⟩ 可以用于 \pgfnode 中。
⟨*shape specification*⟩ 是一些 TEX 代码，其中包含一些特殊的命令来定义 ⟨*shape name*⟩.
下面是形状 coordinate 的定义:

```
\pgfdeclareshape{coordinate}
{
  \savedanchor\centerpoint{%
  \pgf@x=.5\wd\pgfnodeparttextbox%
  \pgf@y=.5\ht\pgfnodeparttextbox%
  \advance\pgf@y by -.5\dp\pgfnodeparttextbox%
  }
  \anchor{center}{\centerpoint}
  \anchorborder{\centerpoint}
}
```

下面介绍用在 ⟨*shape specification*⟩ 中的命令。

\nodeparts{⟨*list of node parts*⟩}

本命令规定 ⟨*shape name*⟩ 的 node parts 的个数和名称，⟨*list of node parts*⟩ 是各个 node parts 的名称列表，有几个名称就有几个 node part. 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:

```
\nodeparts{text,lower}
```

当向各个 node parts 中添加文字时要用到 node part 的名称，应当按照 ⟨*list of node parts*⟩ 中列出的名称次序，依次向各个 node parts 中添加文字。每个 node part 中都有一个盒子来盛放文字。你还需要在各个部分中分别规定一个锚位置来放置文字盒子。例如某个 node part 的名称是 XYZ, 这个部分中盛放文字的盒子就被规定为 \pgfnodepartXYZbox，而放置这个文字盒子的锚位置名称也会被规定为 XYZ. 盒子 \pgfnodepartXYZbox 的左下角会被放在锚位置 XYZ 上。
在默认下，一个 shape 只有一个 node part，其名称默认为 text, 放在这个 node part 中的文字盒子就是 \pgfnodeparttextbox, 与这个盒子对应的锚位置名称也是 text.

\savedanchor{⟨*command*⟩}{⟨*code*⟩}

这个命令定义一个 saved anchor,其中 ⟨*command*⟩ 是 TEX 宏的形式,例如 \centerpoint, 宏 ⟨*command*⟩ 将保存一个坐标位置，即本命令规定的 saved anchor 锚位置。
⟨*code*⟩ 是对 ⟨*command*⟩ 的定义。每当用命令 \pgfnode 或 \pgfmultipartnode 调用形状 ⟨*shape name*⟩ 时，会向 node part 中放置文字盒子，并且执行定义 saved anchor 的 ⟨*code*⟩. 当然，文字盒子的内容可能是空的。例如，如果只有一个名称为 text 的 node part，那么当执行 ⟨*code*⟩ 时，程序会创建文字盒子 \pgfnodeparttextbox 并把文字放入盒子中。
在 ⟨*code*⟩ 中应当（直接地或者间接地）涉及 TEX 尺寸 \pgf@x 和 \pgf@y, 并且应该最终把这两个尺寸设置到所期望的值，这是因为宏 ⟨*command*⟩ 会被自动定义为点:

```
\pgfpoint{\pgf@x}{\pgf@y}
```

可以在 ⟨*code*⟩ 中直接为 \pgf@x 和 \pgf@y 赋值（如前面的例子所示）；可以简单地写出一个坐标点命令 \pgfpoint{⟨*x value*⟩}{⟨*y value*⟩}, 这个坐标点命令会把 ⟨*x value*⟩ 赋予 \pgf@x, 把 ⟨*y value*⟩ 赋

予 \pgf@y. 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。

例如：

```
\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
}
```

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
}
```

上面代码中使用命令 \setlength 把 \pgfshapeminwidth 赋予 \pgf@xa，而 \pgf@xa 是 PGF 的临时尺寸寄存器，这样处理后，就把宏 \pgfshapeminwidth 保存的"纯文本宏"变成了 \pgf@xa 中的尺寸，然后用 \pgf@xa 做尺寸计算。

在 ⟨*code*⟩ 中计算出来的 \pgf@x 和 \pgf@y 的值会被命令 \pgfnode 保存起来。注意 ⟨*command*⟩ 是局部定义、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。

\saveddimen{⟨*command*⟩}{⟨*code*⟩}

本命令类似 \savedanchor. ⟨*command*⟩ 是 TeX 宏，在 ⟨*code*⟩ 中你需要为 \pgf@x 赋值，这个值会被赋予 ⟨*command*⟩.

```
\saveddimen{\depth}{
  \pgf@x=\dp\pgfnodeparttextbox
}
```

\savedmacro{⟨*command*⟩}{⟨*code*⟩}

⟨*command*⟩ 是 TeX 宏，在 ⟨*code*⟩ 中你需要直接定义 ⟨*command*⟩, ⟨*command*⟩ 的值通常是数值或文字。

\anchor{⟨*name*⟩}{⟨*code*⟩}

这个命令定义一个名称为 ⟨*name*⟩ 的 normal anchor（名称不以反斜线开头），⟨*code*⟩ 是对这个锚位置的规定，在 ⟨*code*⟩ 中可以使用已定义的 saved anchor 来规定 ⟨*name*⟩ 的位置。与 saved anchor 不同，

只有在用到 ⟨*name*⟩ 这个位置时才会执行 ⟨*code*⟩ 来计算出此位置。由于名称 ⟨*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。
⟨*code*⟩ 应当能够设置 `\pgf@x` 和 `\pgf@y` 的值，⟨*name*⟩ 所对应的位置会被自动设定为点

```
\pgfpoint{\pgf@x}{\pgf@y}
```

当在 ⟨*code*⟩ 中引入 shape 坐标系中的点时，这个点会自动引入 `\pgf@x` 和 `\pgf@y`, 因为这两个值构成坐标点的两个坐标分量。程序总是先执行 saved anchor 的定义 (如果有的话)，再执行 normal anchor 的定义，因此可以在 ⟨*code*⟩ 中使用已定义的 saved anchor.
在绘图环境中，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}
↪   % 这里 \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 ⟨*code*⟩ 中可以使用各种坐标点命令来规定 ⟨*name*⟩ 对应的位置，例如：

```
\anchor{center}{\pgfpointorigin}  % shape 坐标系的原点
```

在文件《pgfmoduleshapes.code.tex》中有以下代码：

```
\long\def\pgfdeclareshape#1#2{%
  {
    ......
    \anchor{text}{\pgfpointorigin}%
    \nodeparts{text}%
    ......
  }
}
```

这表明在默认下，一个 shape 只有一个 node part，这个 node part 的名称默认为 `text`(其中放置文字盒子的锚位置的名称也默认为 `text`, 文字盒子的左下角会处于锚位置 `text` 上)，而且锚位置 `text` 定义在 shape 坐标系的原点处。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。注意下面的代码：

```
\anchor{text}{\pgfpoint{-.5\wd\pgfnodeparttextbox}{-.5\ht\pgfnodeparttextbox}}
```

乍看之下与前面的代码好像是一样的，但是却可能导致意外的结果，这是因为这行代码中没有引入 `\pgf@x` 和 `\pgf@y`. 这行代码规定的是名称为 `text` 的 node part 的锚位置，当执行这行代码时，`\pgfnodeparttextbox` 很可能还是之前的 shape 中文字盒子，或者是前一个 node part 的文字盒子。当一个 shape 有多个 node parts 时，你需要用

```
\anchor{⟨node part name⟩}{⟨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 的名称是一样的。

`\deferredanchor{⟨name⟩}{⟨code⟩}`

这个命令类似 `\anchor`. ⟨*name*⟩ 是个锚位置名称，⟨*code*⟩ 是对这个锚位置的规定。
以下两个命令

```
\anchor{⟨name⟩}{⟨code⟩}
\deferredanchor{⟨name⟩}{⟨code⟩}
```

中的 ⟨*name*⟩ 都是（不以反斜线开头的）字符串（其中可以有空格）。如果某个宏的展开值是字符串（不含特殊字符），那么这个宏就可以用于 ⟨*name*⟩ 中。
当程序读取命令 `\pgfdeclareshape` 的内容时，如果 `\anchor{<name>} {<code>}` 的 ⟨*name*⟩ 中含有宏（展开值是字符串），那么这个宏会被立即展开；如果 `\deferredanchor{<name>} {<code>}` 的 <name> 中含有宏（展开值是字符串），那么这个宏不会被立即展开，而是等到在绘图命令中使用所定义的 shape 以及锚位置 ⟨*name*⟩ 时，才会把 ⟨*name*⟩ 中的宏展开。
观察下面的例子：

anchor bar anchor

anchor foo anchor

```
\makeatletter
\def\foo{foo}  % 宏 \foo 的值是字母串 foo
\pgfdeclareshape{simple shape}{%
  \savedanchor{\center}{\pgfpointorigin}
\anchor{center}{\center}
\savedanchor{\anchorfoo}{%
\pgf@x=1cm
\pgf@y=0cm}
\deferredanchor{anchor \foo}{\anchorfoo}}  % 用宏 \foo 构成锚位置名称
\makeatother

\begin{tikzpicture}
  \node[simple shape] (Test1) at (0,0) {};
  \fill (Test1.anchor foo) circle (2pt) node[below] {anchor foo anchor};  % 引用锚位置

  \def\foo{bar}  % 重定义宏 \foo 的值是字母串 bar
  \node[simple shape] (Test2) at (2,1) {};
  \fill (Test2.anchor bar) circle (2pt) node[below] {anchor bar anchor};  % 引用锚位置
```

```
\end{tikzpicture}
```

上面例子中，采用重定义 \foo 的方法来修改相应锚位置的名称。

\anchorborder{⟨*code*⟩}

当使用命令 \pgfpointshapeborder 时，本命令的 ⟨*code*⟩ 会被执行。命令 \pgfpointshapeborder 的句法是 (见前文):

```
\pgfpointshapeborder{⟨node name⟩}{⟨point⟩}
```

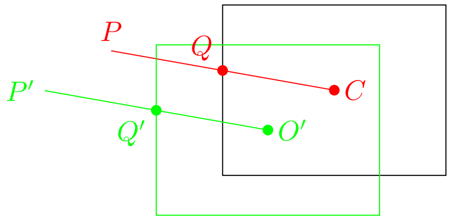
假设形状 ⟨*node name*⟩ 的中心锚位置 center 为点 C, 点 ⟨*point*⟩ 记为 P，射线 CP 与形状 ⟨*node name*⟩ 的边界线的交点 Q 就是 \pgfpointshapeborder 返回的坐标点，这个坐标点应该是由这里的 ⟨*code*⟩ 计算出来的，这就是设计本命令的初衷。

下面猜一下 ⟨*code*⟩ 应该有的思路。

记绘图环境的坐标系是 xOy，集合 $E = \{P(x,y) \mid P(x,y)$是 xOy 系中的坐标点$\}$；记形状 ⟨*node name*⟩ 的 shape 坐标系是 $x'O'y'$，集合 $S = \{P'(u,v) \mid P'(u,v)$是 $x'O'y'$ 系中的坐标点$\}$. 在集合 E 与 S 之间规定对应 f:

$$E \to S,$$
$$f\colon P \to P', P \in E, P' \in S, \text{如果 } \overrightarrow{CP} = \overrightarrow{O'P'}.$$

一般情况下，对应 f 是个双射。坐标系 xOy 与 $x'O'y'$ 是页面上不同的坐标系，通常二者相差一个平移，所以 P 与 P'(满足 $\overrightarrow{CP} = \overrightarrow{O'P'}$) 一般代表着页面上不同的点。还要注意，在默认下，shape 坐标系 $x'O'y'$ 的原点 O' 通常是其锚位置 text.



命令 \pgfpointshapeborder 中给出的参数 ⟨*point*⟩ 是 xOy 系中的点 P，记其坐标是 (p_1, p_2)；与 P 对应的坐标系 $x'O'y'$ 中的点 P' 的坐标记为 (p'_1, p'_2)(即 $f(P) = P'$)；记点 C 在 $x'O'y'$ 系中的坐标是 (c'_1, c'_2)；设在坐标系 xOy 中形状 ⟨*node name*⟩ 的背景路径 (边界路径) 是 Γ，记 $f(\Gamma) = \Gamma' \subset S$.

当执行 \anchorborder{⟨*code*⟩} 中的 ⟨*code*⟩ 时，程序会转到坐标系 $x'O'y'$ 中进行计算，也就是说，⟨*code*⟩ 中的各个坐标数据都是以 $x'O'y'$ 系为参照的。首先自动令 \pgf@x 和 \pgf@y 的值分别是 p'_1 和 p'_2，即分别赋予 P' 的坐标数据；然后通过一系列计算将 \pgf@x 和 \pgf@y 的值变化为射线 $O'P'$ 与路径 Γ' 的交点 Q' 的坐标 (q'_1, q'_2)；然后在坐标系 $x'O'y'$ 中对点 Q' 做平移 $(q'_1, q'_2) + (c'_1, c'_2)$ 得到点 Q，则点 Q 就是射线 CP 与 Γ 的交点 (假设坐标系 xOy 与 $x'O'y'$ 只相差一个平移)；此时 \pgf@x 的值应当是 $q'_1 + c'_1$，\pgf@y 的值应当是 $q'_2 + c'_2$，这样就可以结束 ⟨*code*⟩ 了.

所以 ⟨*code*⟩ 应当是直接地或间接地利用 \pgf@x 和 \pgf@y 做计算、赋值的代码。在 ⟨*code*⟩ 中你可以通过算式直接为 \pgf@x 和 \pgf@y 赋值，也可以使用某些命令确定一个点，程序会自动把 \pgf@x 和 \pgf@y 作成这个点的坐标分量。当然，也可以让 ⟨*code*⟩ 最后计算出的坐标点不位于 ⟨*node name*⟩ 的边界上，不过这就违背了设计本命令的初衷。当 shape 的边界路径形状比较复杂时，⟨*code*⟩ 中的

代码可能会比较复杂，需要大量计算。如果你不太相信 TEX 的计算能力，你可以用某个简单的路径来计算“交点”，当然这个交点很可能不在原本复杂的边界路径上。

举个简单的例子。假设定义一个简单的矩形 shape，它的中心在 shape 坐标系的原点，它的右上角是 saved anchor 锚位置 \upperrightcorner，可以如下定义：

```
\anchorborder{%
  \@tempdima=\pgf@x   % 转存 \pgf@x 的值
  \@tempdimb=\pgf@y
% 下面调用命令 \pgfpointborderrectangle 做计算
  \pgfpointborderrectangle{\pgfpoint{\@tempdima}{\@tempdimb}}{\upperrightcorner}
}
```

\backgroundpath{⟨*code*⟩}

这个命令用 ⟨*code*⟩ 定义 shape 的 background path, 即“背景路径”。注意，在 \node[draw,...]... 命令中的选项 draw 所画出的 node 的背景路径就是这里的 ⟨*code*⟩ 所规定的路径，也就是说，在本命令的 ⟨*code*⟩ 中可以不使用 \pgfusepath{...} 命令来画出、填充背景路径，当然其中使用 \pgfusepath{...} 命令也是可以的。

⟨*code*⟩ 中应当包含创建路径的命令。当 ⟨*code*⟩ 被执行时，所有的 saved anchor 都是有效可用的，所以可在 ⟨*code*⟩ 中使用 saved anchor 来构建路径。例如：

```
\backgroundpath{
  \pgfpathrectanglecorners
    {\upperrightcorner}
    {\pgfpointscale{-1}{\upperrightcorner}}
}
```

background path 所在的“层”处于文字层之下，即先画出 background path, 然后添加文字，文字可能遮挡 background path.

⟨*code*⟩ 中可以直接使用 tikz 的绘图命令（不带环境，但命令结束处要加分号）。

\foregroundpath{⟨*code*⟩}

这个命令定义 shape 的 foreground path, 在添加文字后再画出 foreground path, 程序库 shapes.symbols 中定义的形状 correct forbidden sign：

```
\begin{tikzpicture}
 \node [correct forbidden sign,line width=1ex,draw=red,fill=white]
   {Smoking};
\end{tikzpicture}
```

文字被 foreground path 遮挡。

⟨*code*⟩ 中可以直接使用 tikz 的绘图命令（不带环境，但命令结束处要加分号）。

\behindbackgroundpath{⟨*code*⟩}

这个命令定义 shape 的 behind background path, 这里 ⟨*code*⟩ 中不仅创建路径，而且可以“使用”路径。Behind background path 会在 background path 之前画出。

⟨*code*⟩ 中可以直接使用 tikz 的绘图命令（不带环境，但命令结束处要加分号）。

\beforebackgroundpath{⟨code⟩}

这个命令定义 shape 的 before background path, 这个路径在 background path 画出之后，添加文字之前画出。

\behindforegroundpath{⟨code⟩}

这个命令定义 shape 的 behind foreground path，这个路径在添加文字之后，画出 foreground path 之前画出。

\beforeforegroundpath{⟨code⟩}

这个命令定义 shape 的 before foreground path，这个路径在画出 foreground path 之后画出。

\inheritsavedanchors[⟨from={⟨another shape name⟩}⟩]

这里 ⟨another shape name⟩ 是某个已定义的 shape 的名称。在 ⟨another shape name⟩ 的定义中，有关于它的所有 saved anchors 或 saved dimensions 的定义代码，本命令直接将这些代码全部拿过来，用做新定义的 shape 的 saved anchors 或 saved dimensions 的定义代码。

显然，当新定义的 shape 与 ⟨another shape name⟩ 很相像时，本命令是个快捷的办法。

你可以多次使用这个命令，将多个已定义 shape 的全部 saved anchors 或 saved dimensions 的定义代码搬过来。

\inheritbehindbackgroundpath[⟨from={⟨another shape name⟩}⟩]

这里 ⟨another shape name⟩ 是某个已定义的 shape 的名称。本命令将 ⟨another shape name⟩ 的定义中，定义 behind background path 的代码全部搬过来。

\inheritbackgroundpath[⟨from={⟨another shape name⟩}⟩]

这里 ⟨another shape name⟩ 是某个已定义的 shape 的名称。本命令将 ⟨another shape name⟩ 的定义中，定义 background path 的代码全部搬过来。

\inheritbeforebackgroundpath[⟨from={⟨another shape name⟩}⟩]

本命令将 ⟨another shape name⟩ 的定义中，定义 before background path 的代码全部搬过来。

\inheritbehindforegroundpath[⟨from={⟨another shape name⟩}⟩]

本命令将 ⟨another shape name⟩ 的定义中，定义 behind foreground path 的代码全部搬过来。

\inheritforegroundpath

本命令将 ⟨another shape name⟩ 的定义中，定义 foreground path 的代码全部搬过来。

\inheritbeforeforegroundpath[⟨from={⟨another shape name⟩}⟩]

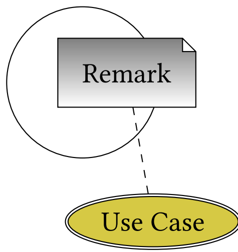
本命令将 ⟨another shape name⟩ 的定义中，定义 before foreground path 的代码全部搬过来。

\inheritanchor[⟨from={⟨another shape name⟩}⟩]{⟨name⟩}

这里 ⟨another shape name⟩ 是某个已定义的 shape 的名称，⟨name⟩ 是 ⟨another shape name⟩ 的某个 normal anchor 的名称（一串字符）。本命令将 ⟨another shape name⟩ 的定义中，定义 ⟨name⟩ 的代码全部搬过来。

\inheritanchorborder[⟨*from={⟨another shape name⟩}*⟩]

本命令将 ⟨*another shape name*⟩ 的定义中，所有的 normal anchors 的定义代码都搬过来。




```
\makeatletter
\pgfdeclareshape{document}{
  \inheritsavedanchors[from=rectangle] % 使用 rectangle 的 saved anchors
  \inheritanchorborder[from=rectangle]
  \inheritanchor[from=rectangle]{center}
  \inheritanchor[from=rectangle]{north}
  \inheritanchor[from=rectangle]{south}
  \inheritanchor[from=rectangle]{west}
  \inheritanchor[from=rectangle]{east}
  \behindbackgroundpath{\draw (0,0)circle(1cm);}
  \backgroundpath{
    \southwest  % 引入锚位置 \southwest 的两个坐标分量 \pgf@x 和 \pgf@y
    \pgf@xa=\pgf@x \pgf@ya=\pgf@y  % 为 \pgf@xa 和 \pgf@ya 赋值，它们是左下角点的坐标分量
    \northeast  % 引入锚位置 \northeast 的两个坐标分量 \pgf@x 和 \pgf@y
    \pgf@xb=\pgf@x \pgf@yb=\pgf@y  % 为 \pgf@xb 和 \pgf@yb 赋值，它们是右上角点的坐标分量
    \pgf@xc=\pgf@xb \advance\pgf@xc by-5pt  % 为 \pgf@xc 赋值，即从 \pgf@xb 处左移 5pt
    \pgf@yc=\pgf@yb \advance\pgf@yc by-5pt  % 为 \pgf@yc 赋值，即从 \pgf@yb 处下移 5pt
    \pgfpathmoveto{\pgfpoint{\pgf@xa}{\pgf@ya}}  % 以左下角点为始点构建路径
    \pgfpathlineto{\pgfpoint{\pgf@xa}{\pgf@yb}}  % 连线到左上角点
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yb}}  % 连线到右上角点左侧 5pt 点处
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}  % 连线到右上角点下部 5pt 点处
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@ya}}  % 连线到右下角点
    \pgfpathclose
    \pgfpathmoveto{\pgfpoint{\pgf@xc}{\pgf@yb}}  % 以右上角点左侧 5pt 点为始点构建路径
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yc}}  % 从前一个点向下连线
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}  % 连线到右上角点下部 5pt 点处
  }
}
\makeatother

\begin{tikzpicture}
  \node[shade,draw,shape=document,inner sep=2ex] (x) {Remark};
  \node[fill=yellow!80!black,draw,ellipse,double] at ([shift=(-80:2cm)]x) (y) {Use Case};
  \draw[dashed] (x) -- (y);
\end{tikzpicture}
```

106.5.4 一个例子

在定义 shapes 模块的文件《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),它们是经常用到的选项,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. 755 与另两个相关的变换的作用是：(a) 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`)，这就是 §17.2.3 中讲的选项 `outer sep=auto` 的作用；(b) 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，其作用参照 §17.2.3 中对此选项的介绍。选项 /pgf/shape border rotate→P. 107 的作用参照 §17.2.3.

```
%
% 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.

```
\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, 其中有 \pgf@anchor@rectangle@center, 在文件《pgf-moduleshapes.code.tex》中有以下代码：

```
\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
% 省略 ....................
  }
}
% 省略 ....................
\def\pgf@sh@anchor#1#2{\expandafter\gdef\csname pgf@anchor@\pgf@sm@shape@name @#1
↪  \endcsname{#2}}
```

前面有命令 \pgfdeclareshape{rectangle}{...}, 所以

\pgf@sm@shape@name 就是 rectangle,

\anchor{center}{⟨set⟩} 就是 \pgf@sh@anchor{center}{⟨set⟩},

这相当于

\gdef\pgf@anchor@rectangle@center{⟨set⟩}

所以 \pgf@anchor@rectangle@center 引入了锚位置 center 所保存的坐标。注意 center 是一串字母，它不能引入坐标；而 \pgf@anchor@rectangle@center 是宏，能引起引入坐标的操作。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。

命令 \pgfdeclareshape 命令 `\pgfdeclareshape` 的定义是:

```
\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%
  }%
}%
```

这个定义的定义内容是个 TeX 分组，组内的内容可以分为两部分：第二部分就是参数 #2, 第一部分就是第一部分之前的内容，第一部分的主要作用是制作一些“接口”以便于用在参数 #2 中，或者跟其它命令交流。

对于 \pgfdeclareshape{rectangle}{⟨*code*⟩} 来说，⟨*code*⟩ 就是第二部分 (参数 #2), 这在前文已经列出，下面主要看第一部分：

1. 定义 \def\pgf@sm@shape@name{rectangle}.
2. 规定 \let\savedanchor=\pgf@sh@savedanchor.

   命令 \savedanchor 在手册中已经介绍了，它等于 \pgf@sh@savedanchor.

   ```
   \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 \pgf@sh@resavedanchor 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 的定义是 (见文件《pgfcorepoints.code.tex》):

```
\def\pgf@process#1{{#1\global\pgf@x=\pgf@x\global\pgf@y=\pgf@y}}
```

命令 \pgfutil@g@addto@macro 的定义是 (见文件《pgfutil-common.tex》):

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

命令 \pgfutil@g@addto@macro{⟨*macro*⟩}{⟨*code*⟩} 的作用是:“全局地”重定义 ⟨*macro*⟩, 也就是把原来 ⟨*macro*⟩ 的定义内容和 ⟨*code*⟩ 都保存到 ⟨*macro*⟩ 中, 其中要求原来的 ⟨*macro*⟩ 应当是已定义的, 例如:

macro:->BBB\aaa

```
\makeatletter
\def\aaa{123}
\def\bbb{BBB}
\let\ccc=\bbb
\pgfutil@g@addto@macro\ccc\aaa
\meaning\ccc
\makeatother
```

所以命令 \pgf@sh@resavedanchor{⟨*command*⟩}{⟨*code*⟩} 的作用是全局地重定义 \pgf@sh@savedpoints, 也就是把

\def⟨*command*⟩{\pgfqpoint{⟨*x* 尺寸⟩}{⟨*y* 尺寸⟩}}

添加到 \pgf@sh@savedpoints 的定义中。最后保存在 \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{};
\meaning\pgf@sh@savedpoints
\makeatother
```

命令 \pgf@sh@savedanchor{⟨*command*⟩}{⟨*code*⟩} 的作用是全局地重定义

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname

也就是把

\pgf@sh@resavedanchor{⟨*command*⟩}{⟨*code*⟩}

添加到 \csname pgf@sh@s@\pgf@sm@shape@name\endcsname 的定义内容中, 例如, 如果执行下面的代码

```
\makeatletter
\expandafter\meaning\csname pgf@sh@s@rectangle\endcsname
```

```
\makeatother
```

就会得到很长的一串：

```
macro:->\pgf@sh@resavedanchor {\northeast }{规定\northeast 的代码}
↪  \pgf@sh@resavedanchor {\southwest }{规定\southwest 的代码}
```

如果执行 `\csname pgf@sh@s@\pgf@sm@shape@name\endcsname`, 那么就对 `\pgf@sh@savedpoints` 作一次或者数次重定义，在 `\pgf@sh@savedpoints` 中保存了各个 saved anchor 的定义语句。如果再执行 `\pgf@sh@savedpoints` 的话，就执行那些定义语句，经过一些列计算，定义各个 saved anchor.

3. 规定 `\let\saveddimen=\pgf@sh@saveddimen`.

   命令 `\pgf@sh@saveddimen` 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}}}%
```

   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%
}%
```

   可见执行 `\pgf@sh@resaveddimen{`⟨*command*⟩`}{`⟨*code*⟩`}` 的作用是把

   `\def`⟨*command*⟩`{`⟨ 尺寸 *\the\pgf@x*⟩`}`

   添加到 `\pgf@sh@savedpoints` 的定义中（对它作全局地重定义），其中的 `\pgf@x` 是 ⟨*code*⟩ 计算出来的尺寸。

   而执行 `\pgf@sh@saveddimen{`⟨*command*⟩`}{`⟨*code*⟩`}` 的作用是全局地重定义

   `\csname pgf@sh@s@\pgf@sm@shape@name\endcsname`

   也就是把

   `\pgf@sh@resaveddimen{`⟨*command*⟩`}{`⟨*code*⟩`}`

   添加到 `\csname pgf@sh@s@\pgf@sm@shape@name\endcsname` 的定义中。

   注意在形状 `rectangle` 的定义中没有使用命令 `\saveddimen`.

4. 规定 `\let\savedmacro=\pgf@sh@savedmacro`, 命令 `\pgf@sh@savedmacro` 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
```

   命令 `\pgf@sh@resavedmacro` 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%
}%
```

可见执行 \pgf@sh@resavedmacro{⟨*macro*⟩}{⟨*code*⟩} 的作用是把定义 ⟨*macro*⟩ 的句子:

\def⟨*mcaro*⟩{定义内容}

添加到 \pgf@sh@savedmacros 的定义中（对此命令作全局地重定义）。其中的 ⟨*code*⟩ 应当能实现对 ⟨*mcaro*⟩ 作定义，如 \def⟨*mcaro*⟩{...}, \let⟨*mcaro*⟩⟨*something*⟩, 在 ⟨*code*⟩ 中可以使用上面代码中的定义的命令 \addtosavedmacro.

执行 \pgf@sh@savedmacro{⟨*macro*⟩}{⟨*code*⟩} 的作用是全局地重定义

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname

也就是把

\pgf@sh@resavedmacro{⟨*command*⟩}{⟨*code*⟩}

添加到 \csname pgf@sh@s@\pgf@sm@shape@name\endcsname 的定义中。

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 }{...}...
```

5. 规定 \let\deferredanchor=\pgf@sh@deferredanchor

```
\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}}}%
```

命令 \pgf@sh@redeferredanchor 的定义是:

```
\def\pgf@sh@redeferredanchor#1#2{% CJ
  \expandafter\gdef\csname pgf@anchor@\pgf@sm@shape@name @#1\endcsname{#2}}% CJ
```

命令 \pgf@sh@redeferredanchor{⟨anchor name⟩}{⟨code⟩} 的作用是全局地定义

\csname pgf@anchor@\pgf@sm@shape@name @⟨anchor name⟩\endcsname

命令 \pgf@sh@deferredanchor{⟨anchor name⟩}{⟨code⟩} 的作用是全局地重定义

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname

也就是把 \pgf@sh@redeferredanchor{⟨anchor name⟩}{⟨code⟩} 添加到 \csname pgf@sh@s@\pgf@sm@shape@name\endcsname 的定义中。

注意，以上 4 个命令 \savedanchor, \saveddimen, \savedmacro, \deferredanchor 都会全局地重定义命令 \csname pgf@sh@s@ \pgf@sm@shape@name \endcsname.

6. 规定 \let\anchor=\pgf@sh@anchor

   命令 \pgf@sh@anchor 的定义是：

```
\def\pgf@sh@anchor#1#2{\expandafter\gdef\csname pgf@anchor@\pgf@sm@shape@name
↪  @#1\endcsname{#2}}%
```

   命令 \pgf@sh@anchor{⟨anchor name⟩}{⟨code⟩} 的作用是全局地定义

   \csname pgf@anchor@\pgf@sm@shape@name @⟨anchor name⟩\endcsname

7. 规定 \let\anchorborder=\pgf@sh@anchorborder

   命令 \pgf@sh@anchorborder 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}}%
```

   执行 \pgf@sh@anchorborder{⟨code⟩} 导致

```
\expandafter\gdef\csname pgf@anchor@\pgf@sm@shape@name @border\endcsname#1{
↪  \pgf@process{#1}⟨code⟩}
```

   其中对应变量 #1 的参数应该是能被 \pgf@process 处理的代码。

8. 规定 \let\behindbackgroundpath=\pgf@sh@behindbgpath

   命令 \pgf@sh@behindbgpath 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

```
\expandafter\gdef\csname pgf@sh@bbg@\pgf@sm@shape@name\endcsname{⟨code⟩}
```

9. 规定 \let\backgroundpath=\pgf@sh@bgpath

```
\long\def\pgf@sh@bgpath#1{\expandafter\gdef\csname pgf@sh@bg@\pgf@sm@shape@name
↪  \endcsname{#1}}%
```

10. 规定 \let\beforebackgroundpath=\pgf@sh@beforebgpath

```
\long\def\pgf@sh@beforebgpath#1{\expandafter\gdef\csname pgf@sh@fbg@
↪  \pgf@sm@shape@name\endcsname{#1}}%
```

11. 规定 \let\behindforegroundpath=\pgf@sh@behindfgpath

```
\long\def\pgf@sh@behindfgpath#1{\expandafter\gdef\csname pgf@sh@bfg@
↪  \pgf@sm@shape@name\endcsname{#1}}%
```

12. 规定 \let\foregroundpath=\pgf@sh@fgpath

```
\long\def\pgf@sh@fgpath#1{\expandafter\gdef\csname pgf@sh@fg@\pgf@sm@shape@name
↪  \endcsname{#1}}%
```

13. 规定 \let\beforeforegroundpath=\pgf@sh@beforefgpath

```
\long\def\pgf@sh@beforefgpath#1{\expandafter\gdef\csname pgf@sh@ffg@
↪  \pgf@sm@shape@name\endcsname{#1}}%
```

14. 规定 \let\nodeparts=\pgf@sh@boxes

```
\def\pgf@sh@boxes#1{\expandafter\gdef\csname pgf@sh@boxes@\pgf@sm@shape@name
↪  \endcsname{#1}}%
```

15. 规定 \let\inheritsavedanchors=\pgf@sh@inheritsavedanchors

```
\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}}%
```

执行 \pgf@sh@inheritsavedanchors[from=⟨*another shape name*⟩] 会导致全局地重定义

\csname pgf@sh@s@\pgf@sm@shape@name\endcsname

也就是把命令 \csname pgf@sh@s@⟨*another shape name*⟩\endcsname 保存的内容添加到该命令的定义内容中。

16. 规定 \let\inheritanchor=\pgf@sh@inheritanchor

```
\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%
}%
```

执行 \pgf@sh@inheritanchor[from=⟨*another shape name*⟩]{⟨*anchor name*⟩} 会导致全局地重定义

\csname pgf@anchor@\pgf@sm@shape@name @⟨*anchor name*⟩\endcsname

即令该命令等于

\csname pgf@anchor@⟨*another shape name*⟩@⟨*anchor name*⟩\endcsname

17. 规定 \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%
}%
```

18. 规定 \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%
}%
```

19. 规定 \let\inheritbackgroundpath=\pgf@sh@inheritbgpath

20. 规定 \let\inheritbeforebackgroundpath=\pgf@sh@inheritbeforebgpath

21. 规定 \let\inheritbehindforegroundpath=\pgf@sh@inheritbehindfgpath

22. 规定 \let\inheritforegroundpath=\pgf@sh@inheritfgpath

23. 规定 \let\inheritbeforeforegroundpath=\pgf@sh@inheritbeforefgpath

24. 规定 \let\inheritnodeparts=\pgf@sh@inheritboxes

```
\def\pgf@sh@inheritboxes[from=#1]{\pgf@sh@inheritor{boxes}{#1}}%
```

25. 执行

```
\anchorborder{\csname pgf@anchor@rectangle@center\endcsname}%
```

导致

```
\expandafter\gdef\csname pgf@anchor@rectangle@border\endcsname#1{\pgf@process
↪  {#1}\csname pgf@anchor@rectangle@center\endcsname}
```

26. 执行

```
\anchor{text}{\pgfpointorigin}%
```

导致

```
\expandafter\gdef\csname pgf@anchor@rectangle@text\endcsname{\pgfpointorigin}
```

即定义 rectangle 的锚位置 text 处于原点。

27. 执行

```
\nodeparts{text}%
```

导致

```
\expandafter\gdef\csname pgf@sh@boxes@rectangle\endcsname{text}
```

28. 执行

```
\expandafter\global\expandafter\let\csname pgf@sh@s@rectangle\endcsname=
↪  \pgfutil@empty%
```

29. 执行 ⟨*code*⟩
30. 用 } 结束 TEX 分组。
31. 结束命令。

命令 `\pgfmultipartnode` 命令

`\pgfmultipartnode{`⟨*shape*⟩`}{`⟨*anchor*⟩`}{`⟨*name*⟩`}{`⟨*path usage command*⟩`}`

的处理过程。

1. 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，根据检查结果:

- 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:
  (a) 用 { 开启一个 TEX 分组。
  (b) 执行 `\ifpgflatenodepositioning`:
    - 如果有 `\pgflatenodepositioningtrue`, 则用 `\pgfsys@beginscope` 开启一个分组。
    - 如果有 `\pgflatenodepositioningfalse`, 则什么也不做。
  (c) 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`. 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⟩.

(d) 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”。

(e) 定义 \def\pgf@sm@shape@name{⟨*shape*⟩}

(f) 执行命令 \csname pgf@sh@s@⟨*shape*⟩\endcsname, 这个命令是被全局定义的，见前文，执行此命令就可能导致命令 \pgf@sh@savedmacros 或者 \pgf@sh@savedpoints 被重定义。\pgf@sh@savedpoints 中保存各个 saved anchor 的定义，\pgf@sh@savedmacros 中保存各个 saved macros 的定义。

(g) 执行命令 \pgf@sh@savedmacros 和 \pgf@sh@savedpoints, 定义各个 saved anchor 或者各个 saved macros.

(h) 执行

```
\pgftransformshift{%
  \pgf@sh@reanchor{⟨shape⟩}{⟨anchor⟩}%
  \pgf@x=-\pgf@x%
  \pgf@y=-\pgf@y%
}%
```

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}%
}%

% Defines a generic anchor, i.e. one which gets the associated shape
% as first argument.
%
% #1: the anchor name.
% #2: the code of the anchor. It may depend upon '##1', the shape's
% name.
%
% The anchor will be defined locally in the current TeX scope.
%
% If the anchor will be referenced later by \pgfpointanchor, the macro \pgfreferencedn
% can be used to query the referenced node's name.
% This macro is not defined during node creation.
\def\pgfdeclaregenericanchor#1#2{%
  \expandafter\def\csname pgf@anchor@generic@#1\endcsname##1{#2}%
}%
```

执行 `\pgf@sh@reanchor{`⟨*shape*⟩`}{`⟨*anchor name*⟩`}` 导致的处理是:

i. 如果名称为 `pgf@anchor@`⟨*shape*⟩`@`⟨*anchor name*⟩ 的命令已定义，则执行之。

ii. 否则，如果名称为 `pgf@anchor@generic@`⟨*anchor name*⟩ 的命令已定义，则执行之。

iii. 否则,

   A. 规定计数器值 `\pgfmathsetcounter{pgf@counta}{`⟨*anchor name*⟩`}`.

   B. 执行

```
\csname pgf@anchor@⟨shape⟩@border\endcsname{\pgfqpointpolar{\the
↪  \c@pgf@counta}{1pt}}
```

最后的执行结果应该是对应某个 anchor 位置的寄存器 `\pgf@x` 与 `\pgf@y` 的值。

执行 `\pgftransformshift` 的结果是改变变换矩阵，把（`-\pgf@x,-\pgf@y`）变成平移向量。

(i) 执行 `\pgfsavepgf@process`, 在文件《pgfcorepoints.code.tex》中有:

```
% 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}%
}
% This needed until old shapes code changed.
\let\pgfsavepgf@process\pgfextract@process%
```

执行此命令的结果是：由 \pgf@sh@reanchor{⟨*shape*⟩}{⟨*anchor*⟩} 得到全局化的 \pgf@x 与 \pgf@y 的值（两个尺寸），然后定义命令

\csname pgf@sh@sa@⟨*name*⟩\endcsname

此命令的作用是将这两个尺寸全局化地保存在 \pgf@x 与 \pgf@y 中。

(j) 将 node 名称 ⟨*name*⟩ 保存在 \pgf@node@name 中。

(k) 执行 \ifpgflatenodepositioning,
- 如果有 \pgflatenodepositioningtrue, 则执行 \pgf@shapes@late@pos@begin
- 如果有 \pgflatenodepositioningfalse, 则结束当前 if 语句。

(l) 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
  i. 全局地定义 \csname pgf@sh@ns@⟨展开的 *name*⟩\endcsname 为 ⟨*edef* 展开的 *name*⟩.
  ii. 全局地定义 \csname pgf@sh@np@⟨展开的 *name*⟩\endcsname 为 ⟨*exp* 展开的 *\pgf@sh@savedpoints*⟩
  iii. 全局地定义 \csname pgf@sh@ma@⟨展开的 *name*⟩\endcsname 为 ⟨*exp* 展开的 *\pgf@sh@savedmacros*
  iv. 将当前的变换矩阵保存在 \pgf@temp 中。
  v. 全局地定义 \csname pgf@sh@nt@⟨展开的 *name*⟩\endcsname 为 ⟨*edef* 展开的 *\pgf@temp*⟩.
  vi. 全局地定义 \csname pgf@sh@pi@⟨展开的 *name*⟩\endcsname 为 ⟨*edef* 展开的 *\pgfpictureid*⟩.
  命令 \pgfpictureid 是命令 \pgfpicture 的子命令, 见文件《pgfcorescopes.code.tex》
  .

(m) 执行 \pgfutil@ifundefined, 检查名称为 pgf@sh@bbg@⟨*shape*⟩ 的命令是否已定义。
- 如果未定义则什么也不做。
- 如果已定义，则设置一个花括号分组，在这个组内执行

```
\pgfusetype{.behind background}\pgfidscope\pgfscope\csname
↪   pgf@sh@bbg@⟨shape⟩\endcsname\endpgfscope\endpgfidscope
```

命令 \pgfusetype →P. 637 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）。

(n) 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

  i. 执行 \pgfpushtype→P. 637.

  ii. 执行 \pgfusetype{.background}\csname pgf@sh@bg@⟨*shape*⟩\endcsname

  iii. 执行条件判断 \ifpgflatenodepositioning.

  如果有 \pgflatenodepositioningtrue, 则

  A. 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。

  B. 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。

  iv. 执行 ⟨*path usage command*⟩.

  v. 执行 \pgfpoptype.

(o) 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，则设置一个花括号分组，在组内执行

```
\pgfusetype{.before background}\pgfidscope\pgfscope\csname
↪   pgf@sh@fbg@#1\endcsname\endpgfscope\endpgfidscope
```

- 如果已定义，则执行一个 for 循环，即 \pgfutil@for ...\do{...}, 这个循环引入文字盒子。

(p) 执行 \pgfutil@ifundefined, 检查名称为 pgf@sh@bfg@⟨*shape*⟩ 的命令是否已定义……

(q) 执行 \pgfutil@ifundefined, 检查名称为 pgf@sh@fg@⟨*shape*⟩ 的命令是否已定义……

(r) 执行 \pgfutil@ifundefined, 检查名称为 pgf@sh@ffg@⟨*shape*⟩ 的命令是否已定义……

(s) 执行 \ifpgflatenodepositioning, 作条件判断

- 如果 \pgflatenodepositioningtrue, 则

```
\pgf@shapes@late@pos@end%
\pgfsys@endscope%
```

- 如果 \pgflatenodepositioningfalse 则执行

```
\expandafter\pgf@nodecallback\expandafter{\pgf@node@name}
```

(t) 用 } 结束花括号分组。

(u) 用 } 结束第一个 \pgfutil@ifundefined.

命令 \pgfnode 此命令的定义是:

```
\def\pgfnode#1#2#3#4#5{%
  {%
    \setbox\pgfnodeparttextbox=\hbox%
    {%
      \pgfinterruptpicture%
        {#3}%
      \endpgfinterruptpicture%
    }%
    \pgfmultipartnode{#1}{#2}{#4}{#5}%
  }
}%
```

↑ ↑

107 矩阵

首先调用模块 matrix.

```
\usepgfmodule{matrix} % LaTeX and plain TeX and pure pgf
\usepgfmodule[matrix] % ConTeXt and pure pgf
```

107.1 Overview

参考 §20.

命令 \pgfmatrix 创建矩阵。矩阵被创建为一个 node, 矩阵的元素称为“cell”, 矩阵的元素或者留空，或者是个“图形”（cell pictures）。

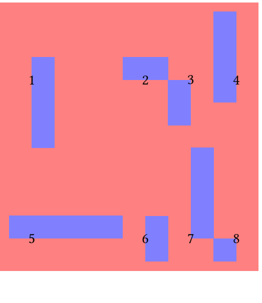
107.2 矩阵元素的对齐方式

参考 §20.3.1.

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。

元素图形没有“层”(layers) 的概念，但有边界盒子（用于对齐）。




```
\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]
  \pgfsetmatrixcolumnsep{1mm}
  \pgfmatrix{rectangle}{center}{mymatrix}{\pgfusepath{}}{\pgfpointorigin
  ↪ }{\let\&=\pgfmatrixnextcell}
  {
    \node {0}; \\
    \node {8}; \&[2mm] \node {1}; \\
                \&      \node {5}; \&[1mm] \node {7}; \\
    \node {4}; \&                   \& \node {2}; \&[2mm] \node {6}; \\
  }
\end{tikzpicture}
```

107.3 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，程序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，使用的是矩阵的路径（background path)，不是元素图形中的路径。

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%
```

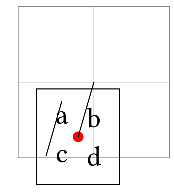
即先把 & 做成活动符，然后让 & 等于 \pgfmatrixnextcell.

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。

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 ⟨*shift*⟩ 时, 各个元素 node 的各种锚位置都是可以引用的。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}
```

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。

调用命令 宏 \pgfmatrixbegincode, \pgfmatrixendcode, \pgfmatrixemptycode 分别保存一组代码。见 §107.5.

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。

宏 \pgfmatrixemptycode 保存的代码是针对空元素的。

pre-code 矩阵的代码会被放入一个 TEX 分组内执行，矩阵命令中的 ⟨*pre-code*⟩（一组代码）也会被放入这个 TEX 分组内，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。

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, 在 §17.4.3 中，选项 node halign header 也利用了这个命令。这个命令可能会做一些奇怪的事情，例如，它会把元素图形代码的第一个宏展开为通常的形式。因此你需要注意:

- 如果某个元素的代码中有一个宏，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。

107.4 行间距与列间距

\pgfsetmatrixcolumnsep{⟨*sep list*⟩}

这个宏设置默认的列间距为尺寸 ⟨*sep list*⟩, 这个间距会用在任意相邻两列之间。

⟨*sep list*⟩ 可以是一个尺寸，也可以是用逗号分隔的数个尺寸，例如

```
\pgfsetmatrixcolumnsep{1mm,2mm,3mm}
```

此时本命令指定的列间距就是 1mm+2mm+3mm=6mm.

⟨*sep list*⟩ 中可以使用选项 between borders 或者 between origins（见 §20.3.2), 默认使用 between borders, 这里的此种选项设置是针对任意相邻两列的。

\pgfmatrixnextcell[⟨*additional sep list*⟩]

这个命令有两个作用。一是分隔左右两个元素，二是使用选项 ⟨*additional sep list*⟩ 调整列间距，这里 ⟨*additional sep list*⟩ 是个尺寸。这个尺寸选项是可选的，如果不给出这个可选尺寸，那么列间距就是由宏 \pgfsetmatrixcolumnsep{⟨*sep list*⟩} 指定的默认尺寸 ⟨*sep list*⟩; 如果给出这个可选尺寸，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”。

当这个命令带有尺寸选项时，一般只能用在第一行中。假设第一行有 n 个元素，而之后的某一行有 $n+1$ 个元素，那么在该行的第 n 个元素与第 $n+1$ 个元素之间可以使用这个命令并带上尺寸选项。在前面的例子中已经显示了这一点。

⟨*additional sep list*⟩ 可以是一个尺寸，也可以是用逗号分隔的数个尺寸，例如

```
\pgfmatrixnextcell[1mm,2mm,3mm]
```

此时本命令指定的附加列间距就是 1mm+2mm+3mm=6mm.

⟨*additional sep list*⟩ 中可以使用选项 between borders 或者 between origins（见 §20.3.2），默认使用 between borders, 这里的此种选项设置只是针对当前的相邻两列。

如果此命令与命令 \pgfsetmatrixcolumnsep 有冲突 (关于选项 between borders 或者 between origins)，那么此命令的优先性高于 \pgfsetmatrixcolumnsep.

注意，如果 ⟨*additional sep list*⟩ 中使用选项 `between borders` 或者 `between origins`, 那么本命令只能用在第一行中。

10mm 10mm

8	1	6
3	5	7
4	9	2

```
\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。

`\pgfmatrixendrow[`⟨*additional sep list*⟩`]`

类似命令 `\pgfmatrixnextcell`, 只是针对行间距。

⟨*additional sep list*⟩ 中可以使用选项 `between borders` 或者 `between origins`.

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，但它的选项无效。

107.5 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:

a	empty	b	
empty	c	d	empty

```
\begin{tikzpicture}
  \def\pgfmatrixemptycode{\node{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”的开头处被执行。

\pgfmatrixendcode

当定义

```
\def\pgfmatrixendcode{⟨code⟩}
```

之后，此命令会在所有“非空元素”的结尾处被执行。

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. 如果定义 \pgfmatrixbegincode 的代码以 \csname 结束，并且定义 \pgfmatrixendcode 的代码以 \endcsname 结束，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。

a	b	c
d		e

```
\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。

108 坐标变换，画布变换，非线性变换

108.1 Overview

坐标变换只针对坐标点，不针对线宽、文字。

非线性变换处理起来比较慢，不太好用，所以专门有一个模块 nonlineartransformations 来处理非线性变换。默认下是不会自动载入这个模块的，如果要用它，你需要手工载入这个模块。

108.2 坐标变换

108.2.1 坐标变换矩阵

PGF 有一个内部坐标变换矩阵。坐标变换矩阵由 4 个数值 a, b, c, d, 以及 2 个尺寸 s, t 组成。一般来说，变换矩阵对坐标点 $\binom{x}{y}$ (其中的 x, y 带有长度单位) 的变换如下：

$$\begin{pmatrix} x \\ y \end{pmatrix} \longrightarrow \begin{pmatrix} a & b & s \\ c & d & t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax + by + s \\ cx + dy + t \\ 1 \end{pmatrix} \longrightarrow \begin{pmatrix} ax + by + s \\ cx + dy + t \end{pmatrix},$$

注意，上面的式子中将坐标点 $\binom{x}{y}$ 扩展为齐次坐标做计算。

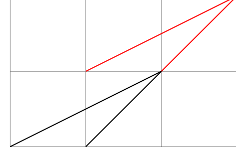
在初始之下，坐标变换矩阵是单位矩阵。当使用多个坐标变换命令时，这些命令对应的变换矩阵会依次起作用。

坐标变换矩阵对其后的路径有作用，其作用范围受到 TEX 分组的限制，而画布变换受到 {pgfscope} 环境的限制。

108.2.2 坐标变换命令

`\pgftransformshift{⟨point⟩}`

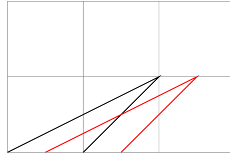
按向量 ⟨*point*⟩ 做平移。



```
\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}
```

`\pgftransformxshift{⟨dimensions⟩}`

在 x 轴方向做平移，



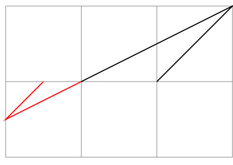
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2,1) -- (1,0);
  \pgftransformxshift{.5cm}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformyshift{⟨dimensions⟩}`

在 y 轴方向做平移，只需一个尺寸 ⟨*dimensions*⟩, 这个尺寸可正可负。

`\pgftransformscale{⟨factor⟩}`

以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。也就是将 ⟨*factor*⟩ 与点向量相乘。



```
\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*⟩.

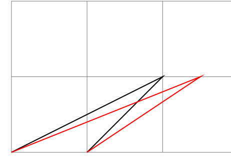
```
\begin{tikzpicture}
  \draw[help lines] (-1,0) grid (2,2);
  \draw (0,0) -- (2,1) -- (1,0);
  \pgftransformxscale{-.5}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformyscale`

保持点的纵坐标不变，将点的纵坐标乘上 ⟨*factor*⟩.

`\pgftransformxslant{`⟨*factor*⟩`}`

这个变换是 $(x, y) \to (x + y \cdot$ `<factor>`$, y)$.



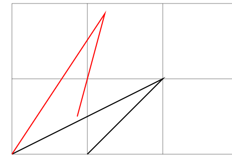
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2,1) -- (1,0);
  \pgftransformxslant{.5}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformyslant{`⟨*factor*⟩`}`

这个变换是 $(x, y) \to (x, y + x \cdot$ `<factor>`$)$.

`\pgftransformrotate{`⟨*angles*⟩`}`

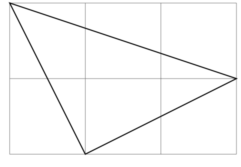
将点围绕原点旋转 ⟨*angles*⟩ 角度。



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2,1) -- (1,0);
  \pgftransformrotate{30}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformtriangle{`⟨*a*⟩`}{`⟨*b*⟩`}{`⟨*c*⟩`}`

这个变换是针对坐标系的，即它变换“标架”，`(0,0)` → ⟨*a*⟩, `(1pt,0pt)` → ⟨*b*⟩, `(0pt,1pt)` → ⟨*c*⟩, 也就是说，新的标架以 ⟨*a*⟩ 为原点，以 $\overrightarrow{\langle a\rangle\langle b\rangle}$ 为“横轴”单位向量，以 $\overrightarrow{\langle a\rangle\langle c\rangle}$ 为“纵轴”单位向量。



```
\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) -- cycle;
\end{tikzpicture}
```

\pgftransformcm{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}{⟨*d*⟩}{⟨*point*⟩}

等效于

```
/tikz/cm={⟨a⟩,⟨b⟩,⟨c⟩,⟨d⟩,⟨coordinate⟩}
```

\pgfpointtransformed{⟨*point*⟩}

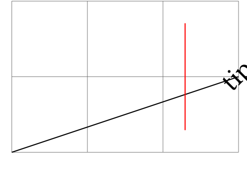
本命令将当前的线性变换矩阵用于点 ⟨*point*⟩. 本命令经常被内部命令调用。

\pgftransformarrow{⟨*start*⟩}{⟨*end*⟩}

点 ⟨*start*⟩ 到点 ⟨*end*⟩ 构成一个向量，记这个向量的倾角是 a, 本命令将标架原点平移到点 ⟨*end*⟩ 处，并将标架旋转角度 a, 得到一个新的标架，之后的路径都在这个新标架内画出。

本命令对路径做平移、旋转。

注意预定义的路径 rectangle, circle 以其中心为起点。

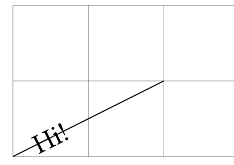



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (3,1);
  \pgftransformarrow{\pgfpoint{1cm}{-1cm}}
    {\pgfpoint{3cm}{1cm}}
  \pgftext{tip}
  \draw [red](-1,0) -- (0,1);
\end{tikzpicture}
```

\pgftransformlineattime{⟨*time*⟩}{⟨*start*⟩}{⟨*end*⟩}

点 ⟨*start*⟩ 到点 ⟨*end*⟩ 构成一个有向线段 d, 本命令会调用命令 \pgfpointlineattime →P. 642 来确定一个点 p, 将标架原点平移到点 p 处，之后各路径都在新标架内画出。如果 \ifpgfslopedattime →P. 753 的值是 true, 本命令还会将标架围绕点 p 旋转，旋转角度是线段 d 的倾角。

本命令对路径做平移、旋转。




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgftransformlineattime{.25}
    {\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

如果 \ifpgfslopedattime 的值是 true, 然后再参考 \ifpgfallowupsidedowattime →P. 753 的真值。如果它的值是 false, 那么 PGF 不会让 node 中的文字出现 “upside down” 的状态。

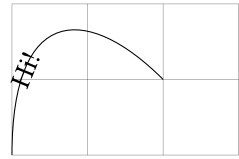
另外,如果 \ifpgfresetnontranslationsattime →P. 754 的真值是 true, 那么 PGF 会调用 \pgftransformresetnontra… 来取消变换中的旋转成分。

\pgftransformcurveattime{⟨*time*⟩}{⟨*start*⟩}{⟨*first support*⟩}{⟨*second support*⟩}{⟨*end*⟩}

点 ⟨*start*⟩ 作为始点，点 ⟨*first support*⟩ 作为第一控制点，点 ⟨*second support*⟩ 作为第二控制点，点 ⟨*end*⟩ 作为终点，确定一段控制曲线，本命令调用命令 \pgfpointcurveattime →P. 643 来确定一个点 p, 将标架原点平移到点 p 处，之后各路径都在新标架内画出。如果 \ifpgfslopedattime 的值是 true, 本命令还会将标架围绕点 p 旋转，旋转角度是控制曲线在点 p 处的切线的倾角。

本命令也会考虑 \ifpgfallowupsidedowattime, \ifpgfresetnontranslationsattime 的真值。

本命令对路径做平移、旋转。



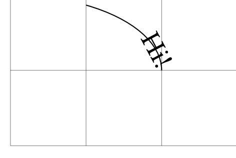

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgfslopedattimetrue
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}
    {\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

\pgftransformarcaxesattime{⟨*time t*⟩}{⟨*center*⟩}{⟨*0-degree axis*⟩}{⟨*90-degree axis*⟩}{⟨*start angle*⟩}{⟨*end angle*⟩}

本命令调用命令 \pgfpointarcaxesattime →P. 642 来确定一个点 p, 将标架原点平移到点 p 处，之后各路径都在新标架内画出。如果 \ifpgfslopedattime 的值是 true, 本命令还会将标架围绕点 p 旋转，旋转角度是椭圆弧在点 p 处的切线的倾角。

本命令也会考虑 \ifpgfallowupsidedowattime, \ifpgfresetnontranslationsattime 的真值。

本命令对路径做平移、旋转。




```
\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，其作用已经在前面解释过了。它的默认值是 false, 用命令 \pgfslopedattimetrue 将它的值设为 true, 用命令 \pgfslopedattimefalse 将它的值设为 false.

\ifpgfallowupsidedowattime

这个 TEX-if 针对前面提到的带有“attime”的变换命令。

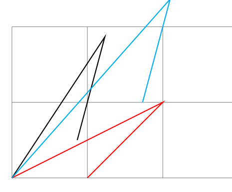
\ifpgfresetnontranslationsattime

这个 TEX-if 针对前面提到的带有“attime”的变换命令。

108.2.3 其它变换

\pgftransformreset

本命令将之前出现的坐标变换矩阵变成单位矩阵，即取消变换，其有效范围受到 TEX 分组的限制。

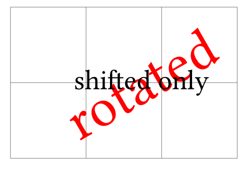


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \draw (0,0) -- (2,1) -- (1,0);
  {
    \pgftransformreset
    \draw[red] (0,0) -- (2,1) -- (1,0);
  }
  \draw[cyan] (0,0) -- (3,1) -- (2,0);
\end{tikzpicture}
```

\pgftransformresetnontranslations

本命令将之前出现的坐标变换矩阵变成平移矩阵 $\left(\begin{smallmatrix}1&0&s\\0&1&t\\0&0&1\end{smallmatrix}\right)$，也就是说，将之前的变换中的旋转、反射、放缩成分去掉，只保留平移作用。

本命令的有效范围受到 TEX 分组的限制。

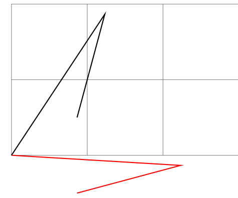



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{2}
  \pgftransformrotate{30}
  \pgftransformxshift{1cm}
  \pgftext{\color{red}rotated}
  \pgftransformresetnontranslations
  \pgftext{shifted only}
\end{tikzpicture}
```

\pgftransforminvert

本命令将当前的变换矩阵换成其逆矩阵，作用于之后的路径。如果当前变换矩阵的条件数太小，即当前变换矩阵接近奇异矩阵，本命令的作用可能会有明显的误差。

本命令的有效范围受到 TEX 分组的限制。



```
\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}
```

108.2.4 保存或使用某个变换矩阵

\pgfgettransform{⟨*macro*⟩}

本命令定义宏 ⟨*macro*⟩, 并把当前的变换矩阵保存在宏 ⟨*macro*⟩ 中,之后可以用命令 \pgfsettransform 引用这个变换矩阵。

\pgfsettransform{⟨*macro*⟩}

⟨*macro*⟩ 是命令 \pgfgettransform 定义的宏，本命令调用保存在宏 ⟨*macro*⟩ 中变换矩阵，作用于之后的路径。

\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 $\left(\begin{smallmatrix} a & b & s \\ c & d & t \\ 0 & 0 & 1 \end{smallmatrix}\right)$，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，作用于之后的路径，实际上等效于

```
\pgftransformreset
\pgftransformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{\pgfpoint{⟨shiftx⟩}{⟨shifty⟩}}
```

108.2.5 坐标变换中的调整

\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，此时你就需要自己计算那个特殊点的坐标，让它表现出“不接受放缩变换”的样子。

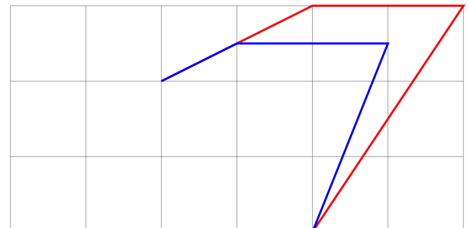
如果你不想自己计算那个特殊点的坐标，那就使用本命令。本命令需要配合下面两个宏使用。

\pgfhorizontaltransformationadjustment

假设当前的放缩变换将 $(1, 0)$ 变成 (a, b)，那么这个宏的值就是 $\frac{1}{\|(a,b)\|}$.

\pgfverticaltransformationadjustment

假设当前的放缩变换将 $(0, 1)$ 变成 (a, b)，那么这个宏的值就是 $\frac{1}{\|(a,b)\|}$.



```
\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}
```

108.3 画布变换

画布变换并非完全由 PGF 自己处理，而是主要由 PDF 或 PostScript 来处理。PGF 只是使用底层命令 \pgfsys@ 来改变画布变换矩阵。

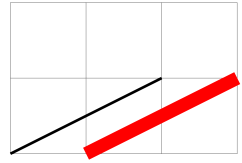
画布变换的有效范围受到 {pgfscope} 环境的限制，这是因为画布变换主要由后台驱动来实现，而不是由 TEX 或 PGF 来实现。

目前，PGF 不能跟踪画布变换，当使用画布变换时，PGF 不能再正确计算 shape 或 node 的各种锚位置及其边界盒子。

PGF 提供数个命令，可以把坐标变换矩阵转换成画布变换矩阵。

\pgflowlevelsynccm

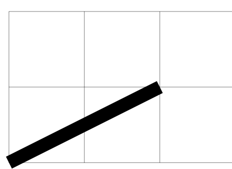
本命令将当前的坐标变换矩阵转成画布变换矩阵，同时取消坐标变换矩阵，并将画布变换矩阵作用于之后的路径。



```
\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*⟩}

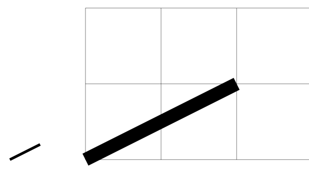
⟨*transformation code*⟩ 是坐标变换命令，本命令将 ⟨*transformation code*⟩ 确定的变换矩阵转成画布变换矩阵，作用于之后的路径。



```
\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*⟩}

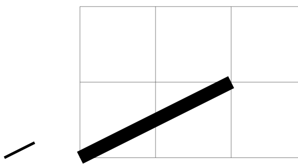
⟨*transformation code*⟩ 是坐标变换命令，⟨*code*⟩ 是绘图命令。本命令创建一个 {pgfscope} 环境，在这个环境中，先调用命令 \pgflowlevel 来处理 ⟨*transformation code*⟩, 然后执行 ⟨*code*⟩.



```
\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,将 ⟨*environment contents*⟩ 放入该环境中,先调用命令 \pgflowlevel 来处理 ⟨*transformation code*⟩, 然后执行 ⟨*environment contents*⟩.



```
\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。

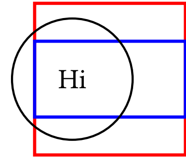
108.3.1 创建 View Boxes

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*⟩, 而 ⟨ll_2⟩ 与 ⟨ur_2⟩ 对应 ⟨*window corner*⟩ `rectangle` ⟨*window corner*⟩.

⟨*meet or slice*⟩ 是选项 `meet` 或 `slice`.




```
\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} }
```

\pgfviewboxscope{⟨ll_1⟩}{⟨ur_1⟩}{⟨ll_2⟩}{⟨ur_2⟩}{⟨*meet or slice*⟩}
  \meta{environment contents}
\endpgfviewboxscope

这是 Plain TeX 中的环境。

\startpgfviewboxscope{⟨ll_1⟩}{⟨ur_1⟩}{⟨ll_2⟩}{⟨ur_2⟩}{⟨*meet or slice*⟩}
  \meta{environment contents}
\stoppgfviewboxscope

这是 ConTeXt 中的环境。

108.4 非线性变换

首先载入模块

```
\usepgfmodule{nonlineartransformations} % LaTeX and plain TeX and pure pgf
\usepgfmodule[nonlineartransformations] % ConTeXt and pure pgf
```

108.4.1 导引

非线性变换只能作用于坐标点，对线宽、文字、颜色渐变等无作用。

变换 $(r,d) \to (d\cos r, d\sin r)$ 是个非线性变换，将直角坐标系中的 $(r,d)_{\mathrm{C}}$ 变成极坐标系中的 $(r,d)_{\mathrm{P}}$（其中的下标 C 表示直角坐标系，下标 P 表示极坐标系），这里规定：在极坐标系中，点 $(r,d)_{\mathrm{P}}$ 的分量分别代表角度、半径。这个规定可能跟教科书上不同，但是《手册》中的例子就是这样的。在这个变换下，直角坐标系中有如下变换关系：

水平线段 `(0,d0)--(1,d0)` ⟶ 圆弧 `(d0,0) arc (0:1:d0)`

竖直线段 `(r0,0)--(r0,1)` ⟶ 线段 `(0,0)--({cos(r0)}, {sin(r0)})`

下面以这个变换为例介绍相关命令。

为了理解下面的非线性变换命令，引入“非线性坐标系”的概念。直角坐标系是一种“线性坐标系”，极坐标系是一种“非线性坐标系”。因为在极坐标系中，当角度参数 θ 成线性变化时，所得到的坐标点不是线性变化的，即映射

$$(\theta, d) \to (d\cos\theta, d\sin\theta)$$

不是线性的，其中的圆括号表示二维点。

非线性变换将直角坐标系变成“非线性坐标系”，然后在非线性坐标系中画出路径，从而表现出非线性变换对路径的作用。用下面的代码解释一下：

```
⟨ 非线性变换命令⟩
\draw (0,1)--(1,1);
```

⟨非线性变换命令⟩将直角坐标系变成非线性坐标系，线段 (0,1)--(1,1) 看作是非线性坐标系内的“线段”（不是直角坐标系内的），并在非线性坐标系内画出这个“线段”，因此这个“线段”可能是弯曲的。例如在极坐标系内画出的“线段”(0,2)--(1,2) 实际上是一段圆弧，半径是 2，圆弧的角度范围是 $[0, 1]$.

这种解释只是为了便于理解非线性变换的作用效果，不代表真实的程序处理过程。

108.4.2 定义并载入一个非线性变换

\pgftransformnonlinear{⟨*transformation code*⟩}

代码 ⟨*transformation code*⟩ 定义一个变换，本命令引入这个变换，对之后的路径起作用。本命令所做的定义的有效范围受到 TEX 分组的限制，注意一个 TEX 分组内只能使用本命令一次。

当使用本命令后，路径上的当前点 p 的分量由寄存器 \pgf@x 和 \pgf@y 保存，这两个寄存器会被传递给 ⟨*transformation code*⟩ 做计算，让 \pgf@x 和 \pgf@y 保存计算后的值，这两个寄存器成为点 q 的坐标分量，于是点 p 就变成了坐标点点 q. 因此在编写 ⟨*transformation code*⟩ 时，你需要假设点 p 是已知的，即假设给出了 \pgf@x 和 \pgf@y 的初始值，然后改变这两个值。这个情况类似命令 \anchorborder{⟨*code*⟩} 的 ⟨*code*⟩ 的编写过程，你可以用代码运算确定点 q, 也可以直接为 \pgf@x 和 \pgf@y 赋值。

下面的代码将变换 $(r, d) \to (d\cos r, d\sin r)$ 保存到宏 \jizuobiaobianhuan 中：

```
\makeatletter
\def\jizuobiaobianhuan{%
  \pgfmathsincos@{\pgf@sys@tonumber\pgf@x}%
  \pgf@x=\pgfmathresultx\pgf@y   % 数值 \pgfmathresultx 与尺寸 \pgf@y 相乘
  \pgf@y=\pgfmathresulty\pgf@y%
}
\makeatother
```

参考命令 \pgfmathsincos→P. 604. 文件“pgfsys.code”对 \pgf@sys@tonumber 的定义是：

```
% The following conversion functions are used to convert from TeX
% dimensions to postscript/pdf points.
%
{\catcode`\p=12\catcode`\t=12\gdef\Pgf@geT#1pt{#1}}

\def\pgf@sys@tonumber#1{\expandafter\Pgf@geT\the#1}
```

下面利用定义的宏 \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}
```

我们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}
  \edef\fuhao{\pgfmathresult}
  \pgfmathpow{\pgfmath@temparg}{2}  % 即 x^2
  \pgfmathmultiply{\pgfmathresult}{0.035146}  % 乘以长度单位转换因子
  \edef\pingfang{\pgfmathresult}
  \ifthenelse{\fuhao>0}
    {\pgf@y=\pingfang pt}
    {\pgf@y=-\pingfang pt}

}
\makeatother
```

上面的代码相当于变换

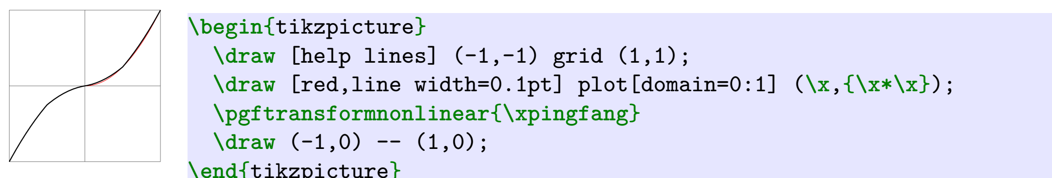
$$(x,y)\to(x,f(x)),\quad f(x)=\begin{cases}x^2, & x>0;\\ -x^2, & x\leqslant 0.\end{cases}$$

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，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}\pgfmathresult` 得到的是 `809.55861`，不是 1. 所以为了实现（cm 下的）平方运算，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}
```

显然，与 `plot` 算子画出的抛物线相比，`\xpingfang` 的作用不够光滑。

108.4.3　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用于点 p, 得到的是变换后的点 q. 与命令 `\pgfpointtransformed` 一样，本命令经常被内部命令调用。

```
\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

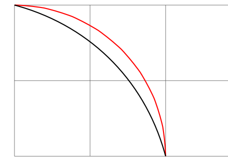
面定义的）非线性变换 `\jizuobiaobianhuan`，所以结果是扭曲的。

108.4.4 将非线性变换用于一个路径

PGF 会先把线性变换作用于路径上的点，然后再做非线性变换。

对于一个线段，无论是用 `\pgfpathlineto` 还是 `\pgfpathclose` 得到的线段，PGF 会把它看作是“退化的”控制曲线，把它的两个三等分点当作是控制点，对它的起点、终点、三等分点做变换，然后用变换后的点构建一段控制曲线。

对于一段控制曲线，PGF 直接对它的起点、终点、控制点做变换，然后用变换后的点构建一段控制曲线。为了方便，把这种变换方式称为“变换模式”。对于很多控制曲线来说，这样的变换过于粗糙，例如：



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  {
    \pgftransformnonlinear{\jizuobiaobianhuan}
    \draw [red] (0,20mm) -- (90pt,20mm);
  }
  \draw (0:20mm) .. controls (30pt:20mm) and (60pt:20mm) .. (90pt:20mm);
\end{tikzpicture}
```

上面例子中，对水平线段 `(0,20mm)--(90pt,20mm)` 做变换 `\jizuobiaobianhuan`，按这个变换的本意，这个水平线段应当变成一段圆弧。但是，作为一个退化的控制曲线，这个线段四个构成点是

`(0,20mm)`, `(30pt,20mm)`, `(60pt,20mm)`, `(90pt,20mm)`,

在变换 `\jizuobiaobianhuan` 之下，它们变成

`(0:20mm)`, `(30pt:20mm)`, `(60pt:20mm)`, `(90pt:20mm)`,

这 4 个点构建图中黑色的控制曲线，因此如果直接按“变换模式”对线段做变换就偏离变换本意过多。但实际上水平线段没有变成黑色的控制曲线，而是变成了红色的圆弧，这是因为下一命令的作用。

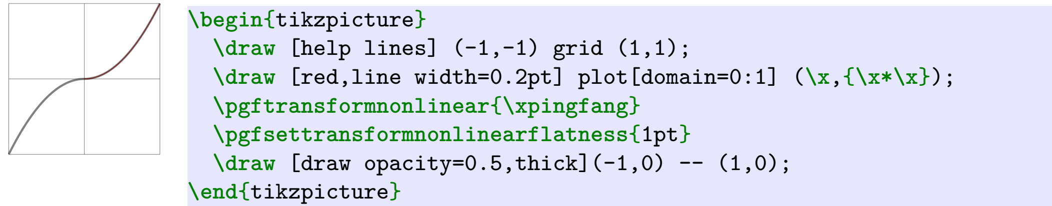
`\pgfsettransformnonlinearflatness{`⟨*dimension*⟩`}` (initially 5pt)

记控制曲线为 $C(t), t \in [0,1]$，可以把 t 看作是“时间”。记 $C(t)$ 的 4 个控制点是 P_0, P_1, P_2, P_3. PGF 会检查向量 $\overrightarrow{P_0P_1}, \overrightarrow{P_1P_2}, \overrightarrow{P_2P_3}$ 的分量 (3 个向量有 6 个分量)，如果某个向量的某个分量尺寸大于本命令指定的尺寸 ⟨*dimension*⟩，那么就在 $t = 0.5$ 处将 $C(t)$ 拆分为两段：$C_{11}(t)$ 和 $C_{12}(t), t \in [0,1]$; 如果这 3 个向量的所有分量的长度都不大于本命令指定的尺寸 ⟨*dimension*⟩，那么就将“变换模式”用于 $C(t)$. 也就是说，PGF 会对 $C(t)$ 执行“检查——拆分——变换”的操作，为了方便，简称这种操作为“检拆变模式”。然后 PGF 会对 $C_{11}(t)$ 和 $C_{12}(t)$ 分别使用“检拆变模式”……如此继续下去，直到完成变换。

这个命令有初始值 5pt，所以前面的例子中，线段按“检拆变模式”变成了红色的圆弧，而不是按“变换模式”变成黑色的控制曲线。

```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw[red] (0:20mm) arc [start angle=0, end angle=90, radius=2cm];
  {
    \pgftransformnonlinear{\jizuobiaobianhuan}
    \pgfsettransformnonlinearflatness{30pt}
    \draw (0,20mm) -- (90pt,20mm);
    \pgfsettransformnonlinearflatness{2pt}
    \draw [red] (0,20mm) -- (90pt,20mm);
  }
\end{tikzpicture}
```

我们使用本命令重画前面的关于变换 \xpingfang 的例子并修改一下线宽和透明度：



```
\begin{tikzpicture}
  \draw [help lines] (-1,-1) grid (1,1);
  \draw [red,line width=0.2pt] plot[domain=0:1] (\x,{\x*\x});
  \pgftransformnonlinear{\xpingfang}
  \pgfsettransformnonlinearflatness{1pt}
  \draw [draw opacity=0.5,thick](-1,0) -- (1,0);
\end{tikzpicture}
```

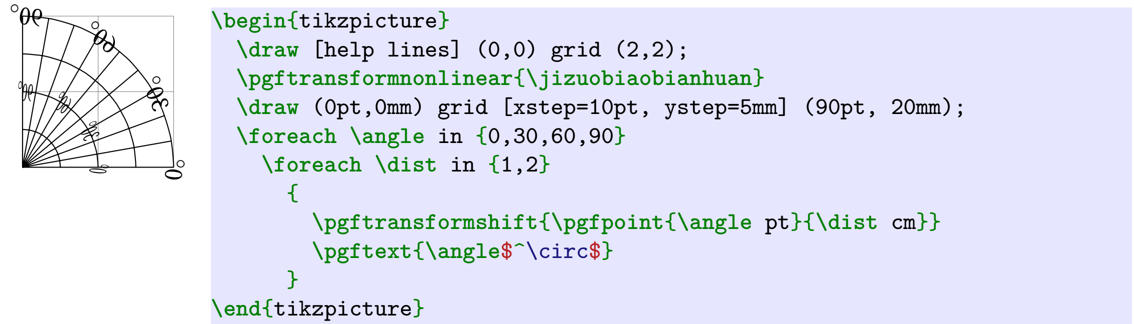
可见 plot 算子画出的抛物线与 \xpingfang 的作用更相近了。

108.4.5 将非线性变换用于文字

前面提到，非线性变换对文字无作用。不过在非线性变换下使用命令 \pgftext 或 \pgfnode 时，PGF 会“尽量正确”地显示“文字”。“尽量正确”的意思是让文字尽量体现出非线性变换所规定的位置变化、旋转、镜像翻转、放缩、拉伸、扭曲等特点，但是 PGF 并非真地将非线性变换作用于文字，而是用某个近似于非线性变换的线性变换来作用于文字，所以“文字扭曲”这一点就无法实现，因为线性变换只能把直线变成直线。

另外需要注意，当用线性变换来近似非线性变换时，应当明确是在哪个点处做近似，这是因为在非线性坐标系中，通常只能在局部的小范围内实现“线性近似”。

PGF 会根据 \pgftext 或 \pgfnode 的锚定点来确定线性近似。



```
\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}
```

注意在上面的图形中，添加的文字被“镜像翻转”了。

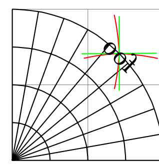
108.4.6 用线性变换近似非线性变换

前面已经提到了用线性变换近似非线性变换，线性近似的效果只是局部地。下面介绍近似的命令。

\pgfapproximatenonlineartransformation

这个命令有两个作用。第一，清除当前 TEX 分组内的、本命令之后的非线性变换，只保留线性变换。第二，在清除非线性变换之前，本命令会调整（修改）线性变换矩阵，使之能够近似非线性变换在原点处的作用效果。也就是说，本命令得到的线性变换是在原点处对非线性变换的近似。

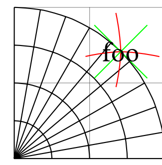
在非线性变换之下，命令 \pgftext 或 \pgfnode 会调用本命令来变换文字。




```
\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};
  \end{scope}
\end{tikzpicture}
```

\pgfapproximatenonlineartranslation

这个命令得到的线性变换也是在原点处对非线性变换的近似，只不过本命令只保留平移成分，其它的旋转、镜像、拉伸、放缩等变换成分都去掉。




```
\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);
    \pgfapproximatenonlineartranslation  % 做近似
    \draw [green] (-10pt,-10pt) -- (10pt,10pt);
    \draw [green] (10pt,-10pt) -- (-10pt,10pt);
    \pgftext{foo};
  \end{scope}
\end{tikzpicture}
```

108.4.7 非线性变换程序库

TikZ Library curvilinear

```
\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 分组的限制，并且在一个 TEX 分组内至多能使用本命令一次。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 4 个参数是 4 个点，用于构建一段控制曲线，记该控制曲线是 $C(t)$. 本命令还会创建一个数据表，这个数据表反应的是曲线长度与参数 t 的对应：

$$\int_0^t \mathrm{d}(C(t)) \to t,$$

其中 $\mathrm{d}(C(t))$ 是 $C(t)$ 的弧长元素。下面的命令 \pgfcurvilineardistancetotime 利用这个数据表做插值计算。

```
\pgfsetcurvilinearbeziercurve
  {\pgfpointorigin}
  {\pgfpoint{1cm}{1cm}}
  {\pgfpoint{2cm}{1cm}}
  {\pgfpoint{3cm}{0cm}}
```

\pgfcurvilineardistancetotime{⟨*distance*⟩}

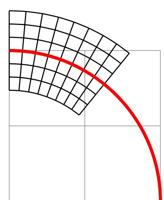
本命令利用上面命令提供的数据表，计算当控制曲线的长度为 ⟨*distance*⟩ 时所对应的参数 t 的值，并将这个值保存在 \pgf@x 中。注意本命令会在运算速度与精度之间作出权衡。本命令在曲线开端处的精度最好，如果控制曲线的退化程度越高，则精度越差。

\pgfpointcurvilinearbezierorthogonal{⟨*distance*⟩}{⟨*offset*⟩}

这个命令定义一个非线性变换。

先设想这样一个"曲线坐标系"：曲线坐标系的"横轴"就是前面的命令构建的控制曲线 $C(t)$，曲线的起点就"横轴"的 0 点；设点 p 是横轴上的一个点，在 p 处有对应 p 的"纵轴"——$C(t)$ 在 p 处的法线。在这样一个曲线坐标系中，坐标为 (x,y) 的点就可以这样确定：从控制曲线 $C(t)$ 的起点开始沿着它运动，运动的长度是 x, 到达点 p, 然后旋转 $90°$, 直线移动长度 y（可以是负值尺寸），到达点 q, 点 q 的坐标就是 (x,y).

本命令的两个参数 ⟨*distance*⟩ 和 ⟨*offset*⟩ 是两个尺寸，这两个尺寸代表这个"曲线坐标系"内的点的坐标：$\begin{pmatrix}\langle\mathit{distance}\rangle\\ \langle\mathit{offset}\rangle\end{pmatrix}$. 在 ⟨*distance*⟩ 和 ⟨*offset*⟩ 中可以使用 \pgf@x 和 \pgf@y 这两个值，这两个值代表路径中的点的坐标分量。



```
\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{0.5\pgf@x}{-0.5\pgf@y}}%
    \makeatother
    \draw (0,-30pt) grid [step=10pt] (80pt,30pt);
```

```
  }
  \draw[red, very thick] (0mm,20mm) .. controls (11mm,20mm) and (20mm,11mm) .. (20mm,0mm);
\end{tikzpicture}
```

在上面代码中有如下的变换:

$$S: \begin{pmatrix} \texttt{\textbackslash pgf@x} \\ \texttt{\textbackslash pgf@y} \end{pmatrix} \longrightarrow \begin{pmatrix} \texttt{0.5\textbackslash pgf@x} \\ \texttt{-0.5\textbackslash pgf@y} \end{pmatrix}.$$

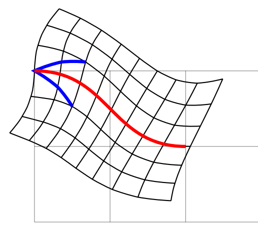
假设被变换路径——网格 `(0,-30pt) grid [step=10pt] (80pt,30pt)` 是点集 A, 在上述变换 S 下, 点集 A 变成点集 B, 然后把点集 B 看作是“曲线坐标系”内的点集，在曲线坐标系内画出这个点集即可得到上面的图形。

`\pgfpointcurvilinearbezierpolar{`⟨*x*⟩`}{`⟨*y*⟩`}`

本命令定义一个非线性变换。

本命令确定一个“曲线坐标系”，该曲线坐标系的“横轴”就是前面的命令构建的控制曲线 $C(t)$, 曲线的起点就“横轴”的 0 点 O. 假设点 $Q=(x,y)$ 是这个曲线坐标系内的一个点，这个点在曲线坐标系内的位置这样确定：从曲线 $C(t)$ 的起点 O 开始沿着它运动，运动长度是 $\sqrt{x^2+y^2}$, 到达点 P, 然后将 P 围绕起点 O 旋转，旋转角度是在直角坐标系中向量 $(x,y)_C$ 的方向角，这样就把 P 变成了点 Q, 点 Q 的坐标就是 (x,y).

下面例子中使用本命令变换网格，大致可以看出变换效果，路径起点处的箭头是“Straight Barb”。



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  {
    \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);
    \draw [blue, very thick] (20pt,10pt) -- (0,0) -- (20pt,-10pt);
  }
  \draw[red, very thick] (0mm,20mm) .. controls (10mm,20mm) and
  ↪  (10mm,10mm) .. (20mm,10mm);
\end{tikzpicture}
```

109 图样 Patterns

109.1 Overview

参考 §15.5.1，§60.

就像用瓷砖铺地面一样，可以用“图样砖”（tile）铺成图样（tiling pattern）。图样砖本身是个图形，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。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。

图样分为两种：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。

PGF 会直接把图样映射到图形语言（PostScript, pdf, svg）的图样机制中。

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。

当用图样填充路径时，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得到图样，然后再用路径剪切图样。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"位于该盒子的左上角。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，也就是说，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。

109.2 声明一个图样

先声明一个图样，然后使用它。

\pgfdeclarepatternformonly[⟨*variables*⟩]{⟨*name*⟩}{⟨*bottom left*⟩}{⟨*top right*⟩}{⟨*tile size*⟩}{⟨*code*⟩}

本命令的声明是全局有效的。

本命令声明一个 form-only pattern, 其名称是 ⟨*name*⟩, 实际上本命令定义一个"图样砖"。

当使用这个命令声明图样时，你需要假设有一个"图样砖坐标系"，本命令的参数中涉及的坐标点、绘图代码都是在"图样砖坐标系"内描绘图样砖的。

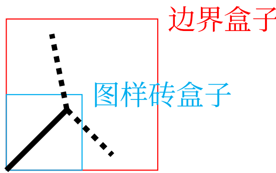
在"图样砖坐标系"中以坐标点 ⟨*bottom left*⟩ 和 ⟨*top right*⟩ 为对角点构建一个矩形的"边界盒子"，这个边界盒子用来"剪切"图样砖中的图形。以"图样砖坐标系"的原点和坐标点 ⟨*tile size*⟩ 为对角点构建一个矩形的"图样砖盒子"，这个图样砖盒子相当于铺地面的单块瓷砖。这里要区分"边界盒子"与"图样砖盒子"，二者的作用不同。

⟨*code*⟩ 是能够被"protocolled"的 PGF 绘图命令，其中不能涉及颜色，如果 ⟨*code*⟩ 中有颜色设置，那么颜色可能得不到预期的显示。绘图代码 ⟨*code*⟩ 在"图样砖坐标系"中画出图形，所画的图形会受到"边界盒子"的剪切。然后把剪切所得的图形"固着到"图样砖盒子中，于是得到一个"图样砖"（瓷砖），再不断重复图样砖（用瓷砖铺地面）就得到图样。因此，如果所画图形的尺寸较大，而且"边界盒子"又把"图样砖盒子"包含在内，那么（剪切后的）固着到图样砖盒子上的图形尺寸就会大于图样砖盒子的尺寸，于是在平铺图样砖时，图样砖上的图形就会出现交叠。例如，下面代码定义图样 `SlopeLine`:

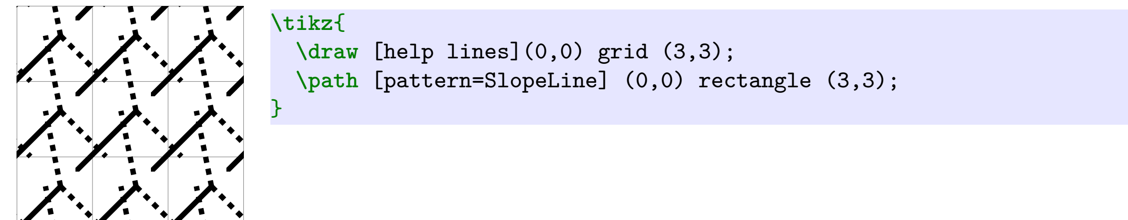
```
\pgfdeclarepatternformonly{SlopeLine}{\pgfpointorigin}{\pgfpoint{2cm}{2cm}}{
↪  \pgfpoint{1cm}{1cm}}
{
  \pgfsetlinewidth{2pt}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0.8cm}{0.8cm}}
  \pgfusepath{draw}
  \pgfsetdash{{0.8mm}{0.8mm}}{0mm}
  \pgfsetlinewidth{2pt}
  \pgfpathmoveto{\pgfpoint{0.8cm}{0.8cm}}
```

```
  \pgfpathlineto{\pgfpoint{0.6cm}{1.8cm}}
  \pgfpathmoveto{\pgfpoint{0.8cm}{0.8cm}}
  \pgfpathlineto{\pgfpoint{1.4cm}{0.2cm}}
  \pgfusepath{draw}
}
```

其中的各个盒子与所画图形如下所示:



所画图形是三个线段，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:



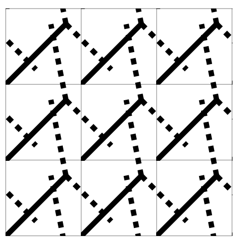
```
\tikz{
  \draw [help lines](0,0) grid (3,3);
  \path [pattern=SlopeLine] (0,0) rectangle (3,3);
}
```

但是需要注意的是，上述绘图代码在 standalone 文类下得到的图形，与在 article 文类或 ctexart 文类下得到的图形并不相同。上图是在 ctexart 文类下的图形，而在 standalone 文类下的如下文档:

```
\documentclass[tikz]{standalone}
\usepackage{xcolor}
\usepackage{tikz}
\usetikzlibrary[patterns]

\begin{document}

\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}
}

\tikz{
  \draw [help lines](0,0) grid (3,3);
```

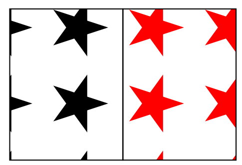
```
  \path [pattern=SlopeLine] (0,0) rectangle (3,3);
}

\end{document}
```

得到的图形是:



比较起来，两种文类下的图形中，网格与线段的相对位置不同。

下面是另一个例子:

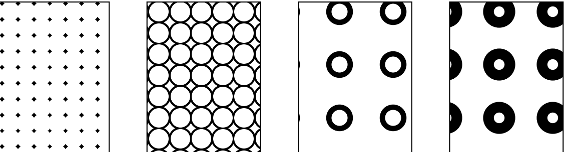


```
\pgfdeclarepatternformonly{stars}{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}{
↪  \pgfpoint{1cm}{1cm}}
{
  \pgftransformshift{\pgfpoint{.5cm}{.5cm}}
  \pgfpathmoveto{\pgfpointpolar{0}{4mm}}
  \pgfpathlineto{\pgfpointpolar{144}{4mm}}
  \pgfpathlineto{\pgfpointpolar{288}{4mm}}
  \pgfpathlineto{\pgfpointpolar{72}{4mm}}
  \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。

⟨*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，必须创建图样，此时 ⟨*code*⟩ 中的各个变量应当具有自己的值，因此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。

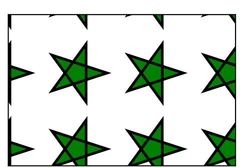


```
\pgfdeclarepatternformonly[/tikz/radius,\thickness,\size]{rings}
  {\pgfpoint{-0.5*\size}{-0.5*\size}}
  {\pgfpoint{0.5*\size}{0.5*\size}}
  {\pgfpoint{\size}{\size}}
  {
    \pgfsetlinewidth{\thickness}
    \pgfpathcircle\pgfpointorigin{\pgfkeysvalueof{/tikz/radius}}
    \pgfusepath{stroke}
  }
\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] (2,0) rectangle +(1.5,2);
  \filldraw [rings, radius=6pt, thickness=2pt] (4,0) rectangle +(1.5,2);
  \filldraw [rings, radius=8pt, thickness=4pt] (6,0) rectangle +(1.5,2);
\end{tikzpicture}
```

`\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}}
```

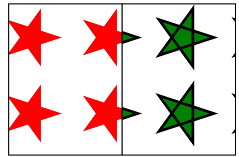
```
  \pgfpathlineto{\pgfpointpolar{216}{4mm}}
  \pgfpathclose%
  \pgfusepath{stroke,fill}
}
\begin{tikzpicture}
  \filldraw[pattern=green stars] (0,0) rectangle (3,2);
\end{tikzpicture}
```

109.3 使用图样

声明一个图样后，就可以使用它。

\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}
```

110 声明、使用外部图形

110.1 Overview

LaTeX 宏包 graphicx 提供的命令 `\includegraphics` 可以插入外部图形，这个命令的设计要比 PGF 的插图机制好。不过在某些情况下你可能需要使用 PGF 的插图机制，例如，plain TeX 中没有 graphicx 宏包。PGF 的插图机制需要后台驱动 pdftex 的支持，对于其它驱动，目前来说，PGF 会调用 graphicx 宏包来插入图形。如果你想对插入的图形使用“masking”, 那么可以使用 PGF 的插图机制。

用 PGF 插入外部图形通常需要两个步骤，首先用命令 `\pgfdeclareimage` 声明需要插入的外部图形，然后用命令 `\pgfuseimage` 插入这个图形。另外，命令 `\pgfimage` 可以把“声明图形”和“插入图形”合并为一个步骤。

\usepackage[draft]{pgf}

这个命令启用 PGF 的草稿模式，插入的外部图形都会被方框代替。这个模式会检查外部图形是否存在，但不会读入图形。如果没有明确指定插入图形的宽度和高度，就默认宽度和高度为 1cm.

110.2 声明外部图形

\pgfdeclareimage[⟨*options*⟩]{⟨*image name*⟩}{⟨*filename*⟩}

⟨*filename*⟩ 是外部图形文件的名称，可以不带扩展名。如果 ⟨*filename*⟩ 不带扩展名，PGF 会自动尝试扩展名 `.pdf`, `.jpg`, `.png`; PostScript 会自动尝试扩展名 `.eps`, `.epsi`, `.ps`.

在 ⟨*options*⟩ 中可以使用以下选项。

height=⟨*dimension*⟩ 本选项指定插入图形的高度。如果不同时指定宽度，则保持原图的宽高比例。

width=⟨*dimension*⟩ 本选项指定插入图形的宽度。如果不同时指定高度，则保持原图的宽高比例。

page=⟨*page number*⟩ 如果文件 ⟨*filename*⟩ 有多个页面，每个页面都是一个图形，那么本选项指定需要插入的图形是第 ⟨*page number*⟩ 页的图形。实际上，使用本选项后，PGF 会先查找名称为

⟨*filenamei*⟩.page⟨*page number*⟩.⟨*extension*⟩

的图形文件，如果找到了就插入这个图形；如果没有找到，再将文件 ⟨*filenamei*⟩.⟨*extension*⟩ 中的图形插入。

interpolate=⟨*true or false*⟩ 本选项决定，当插入图形被放大或缩小时，是否让图形“光滑”。默认 false.

mask=⟨*mask name*⟩ 本选项给插入的图形“带上面具”⟨*mask name*⟩, ⟨*mask name*⟩ 是一个具有某种透明度的颜色，需要提前用命令 \pgfdeclaremask 声明（见后文）。本选项只对 pdf 有效，有的阅读器不支持这种特性。

\pgfaliasimage{⟨*new image name*⟩}{⟨*existing image name*⟩}

⟨*existing image name*⟩ 是原来的图形文件名称，本命令给这个图形文件令起一个新名称 ⟨*new image name*⟩, 这两个名称都指向同一图形文件。例如：

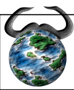
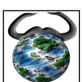
```
\pgfaliasimage{image.!30!white}{image.!25!white}
```

如果你觉得这种图形名称很奇怪，请参考 §91.

110.3 使用外部图形

\pgfuseimage{⟨*image name*⟩}

声明一个外部图形后，可以用本命令将其插入到文档中。如果想把这个图形插入到 {pgfpicture} 环境中，你需要在命令 \pgftext 的参数中使用本命令。

```
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm] {image1}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,width=1cm]{image2}{brave-gnu-world-logo}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{5cm}{1cm},left,base]{\pgfuseimage{image1}}
  \pgftext[at=\pgfpoint{3cm}{1cm},left,base]{\pgfuseimage{image2}}
  \pgfpathrectangle{\pgfpoint{5cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{3cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

为了顺利理解下面的内容，请先阅读 §91.

假设宏 \pgfalternateextension（见下文）的展开值是 !25!white，那么在使用本命令插入图形时，PGF 会先查找并插入名称为 ⟨*image name*⟩.!25!white 的图形文件；如果找不到这个名称的图形文件，PGF 就查找并插入名称为 ⟨*image*⟩.25!white 的图形文件；如果找不到这个名称的图形文件，

PGF 就查找并插入名称为 ⟨*image*⟩.white 的图形文件；如果找不到这个名称的图形文件，PGF 就查找并插入名称为 ⟨*image*⟩ 的图形文件。

`\pgfalternateextension`

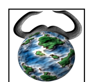
用 \def 重定义这个宏，这个宏的定义内容应该用于文件名称的扩展部分。例如：

```
\def\pgfalternateextension{!25!white}
```

`\pgfimage[⟨options⟩]{⟨filename⟩}`

本命令的 ⟨*options*⟩ 中的选项可以使用命令 \pgfdeclareimage 的选项。

本命令的作用有二：一是将文件 ⟨*filename*⟩ 中的图形的名称声明为 pgflastimage（这是本命令默认的名称），而且 ⟨*options*⟩ 中的选项也一并保存在名称 pgflastimage 之下；二是把图形 pgflastimage 插入到文档中。可见，名称 pgflastimage 所指向的图形文件是可变的。使用本命令后，你可以使用命令 \pgfaliasimage 来给 pgflastimage 另外起一个新名称。



```
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{5cm}{1cm},left,base]
    {\pgfimage[interpolate=true,width=1cm,height=1cm]{brave-gnu-world-logo}}
  \pgfaliasimage{xinmingcheng}{pgflastimage}
  \pgftext[at=\pgfpoint{3cm}{1cm},left,base]
    {\pgfuseimage{xinmingcheng}}
  \pgfpathrectangle{\pgfpoint{5cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{3cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

110.4 给图形“带面具”

`\pgfdeclaremask[⟨options⟩]{⟨mask name⟩}{⟨filename⟩}`

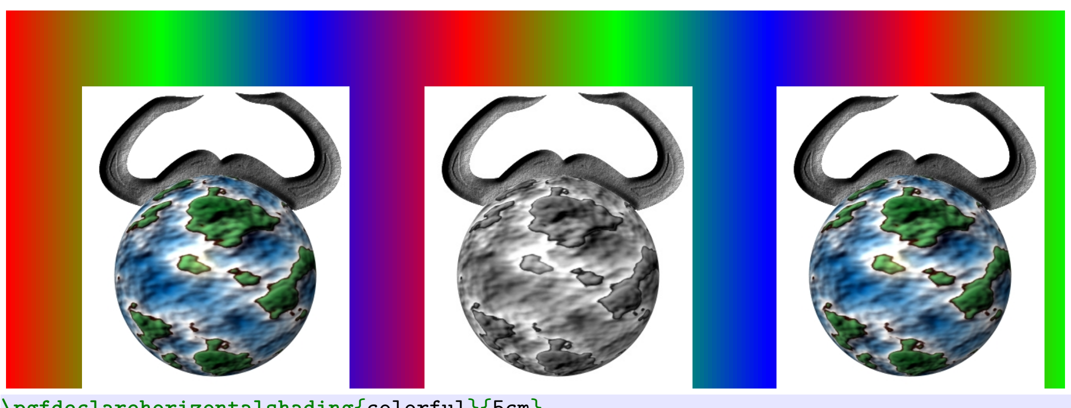
本命令声明一个名称为 ⟨*mask name*⟩ 的“透明面具”（transparency mask），在 PDF 手册中称之为 soft mask. 声明 ⟨*mask name*⟩ 后，就可以在命令 \pgfdeclareimage 和 \pgfimage 的选项中使用选项 mask=⟨*mask name*⟩ 了。

插入到文档中的图形通常是没有透明度的，面具的作用就是让插入图形的各个像素点具有自己的透明度，从而使图形呈现某种视觉效果。使用面具这一“技术”时要注意四点：第一，面具的尺寸应当与插入图形的尺寸一致；第二，插入的图形必须是“像素图”（如 jpg，png 格式的图），不能是“矢量图”（如 pdf 格式的图）；第三，面具必须是真正的“灰度图”（只有单个颜色通道），而不是用 RGB 等颜色模式通过混色得到的“黑白灰”图；第四，有的阅读器不支持面具技术。

本命令从文件 ⟨*filename*⟩ 中读取面具——一个灰度图。灰度图由诸多像素点构成，每个像素点都有自己的灰度，程序会把像素点的灰度转变成透明度，白色像素点是透明的，黑色像素点是不透明的，即灰度与透明度负相关——灰度值越高，透明度越低。这样就把灰度图变成了“透明度图”。然后想象一下——把面具覆盖到图形上后，面具的像素点与图形的像素点实现一一对应，此时面具的像素点的透明度就会变成图形上对应像素点的透明度，然后把面具拿走，但把透明度保留在图形上。这就是面具的作用。

在 ⟨*options*⟩ 中可以使用下一个选项:

matte={⟨*color components*⟩} ⟨*color components*⟩ 用于指定一种颜色。



```
\pgfdeclarehorizontalshading{colorful}{5cm}
  {color(0cm)=(red);color(2cm)=(green); color(4cm)=(blue); color(6cm)=(red);
   color(8cm)=(green); color(10cm)=(blue); color(12cm)=(red);color(14cm)=(green)}
\hbox{
  \pgfuseshading{colorful}\hskip-14cm\hskip1cm
  \pgfimage[height=4cm]{brave-gnu-world-logo}\hskip1cm
  \pgfimage[height=4cm]{brave-gnu-world-logo-mask}\hskip1cm
  \pgfdeclaremask{mymask}{brave-gnu-world-logo-mask}
  \pgfimage[mask=mymask,height=4cm,interpolate=true]{brave-gnu-world-logo}}
```

上面例子中插入的图形没有显示出透明度效果，这个例子是用 xelatex 编译的，如果改用 pdflatex 编译，就能得到如同手册上那样的图形效果。

112 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。

112.1 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。

112.2 创建图流

注意创建的图流是“全局地”，不受 TEX 分组的限制，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
```

上面的代码将图流保存在宏 \mystream 中，然后在绘图环境内可以用宏 \mystream 来引用图流。

112.2.1 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。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
```

以上命令的初始定义见文件《pgfmoduleplot.code.tex》：

```
\def\pgfplotstreamstart{\pgf@plotstreamstart}
\def\pgfplotstreampoint#1{\gdef\pgfplotlastpoint{#1}\pgf@plotstreampoint{#1}}
\def\pgfplotstreampointoutlier#1{\pgfplot@outliers{}{\pgf@plotstreamjump}{
↪  \pgfplotstreampoint{#1}}}%
\def\pgfplotstreampointundefined{\pgfplot@outliers{}{\pgf@plotstreamjump}{}}%
\def\pgfplotstreamnewdataset{\pgfplot@outliers{}{\pgf@plotstreamjump}{}}%
\def\pgfplotstreamspecial{\pgf@plotstreamspecial}
\def\pgfplotstreamend{\pgf@plotstreamend}
```

\pgfplotstreamstart

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，图柄会以某种方式设置内部命令 \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)
/tikz/handle outlier points in plots

选项中的 ⟨*how*⟩ 可以取以下值:

- plot，这个值会把 outlier 点作为普通点对待，相当于 \pgfplotstreampoint 作用于该点。
- ignore，这个值导致“忽略 outlier 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”。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，那么这个线段就被分成了两端:

```
(0,0)--(1,0) (1.2,0) (1.4,0)--(2,0)
```

\pgfplotstreampointundefined

这个命令生成一个“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)
/tikz/handle undefined points in plots

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

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)

`/tikz/handle new data sets in plots`

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*⟩`}`

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。

112.2.2 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。本命令读取 ⟨*file name*⟩ 中的数据，将其中的数据变成图流。首先引入命令 `\pgfplotstreamstart`，然后逐行读取 ⟨*file name*⟩ 中的数据，读完后再添加命令 `\pgfplotstreamend` 结束图流；文件中的空行对应命令 `\pgfplotstreamnewdataset`；如果一行中有（用空格分隔的）两个数字，则这两个数字构成一个坐标点；如果一行中有字母“`u`”，则该行对应 `\pgfplotstreampointundefined`；如果一行中有字母“`o`”，则该行对应 `\pgfplotstreampointoutlier`。
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 1 first entry      \pgfplotstreamnewdataset
2 -.2 2 o               \pgfplotstreampoint{\pgfpointxyz{2}{-5}{1}}
2 -5 2 third entry      \pgfplotstreampointoutlier{\pgfpointxyz{2}{-.2}{2}}
                        \pgfplotstreampoint{\pgfpointxyz{2}{-5}{2}}
                        \pgfplotstreamend
```

其中以 % 或 # 开头的行是注释行, 注释行会被忽略, 空行都被转为命令 \pgfplotstreamnewdataset. 非空行必须是以下 6 种形式之一:

数字 ␣ 数字
数字 ␣ 数字 ␣ 文字符号
数字 ␣ 数字 ␣ 数字
数字 ␣ 数字 ␣ 数字 ␣ 文字符号
符号 ␣ 符号 ␣u
符号 ␣ 符号 ␣ 符号 ␣u

上面 6 种形式的意思是:

- 如果某行有 2 个数字,则这 2 个数字代表一个二维点,会被转到 xy 坐标系统中;
- 如果某行有 3 个数字,则这 3 个数字代表一个三维点,会被转到 xyz 坐标系统中;
- 如果某一行以"␣u"结尾,那么该行会被转成命令 \pgfplotstreampointundefined,即被转成 undefined 点,因此该行可以不严格地包含数字;
- 如果某一行的"文字符号"是单个字母"o"或以"␣o"结尾,例如

  ```
  5 25 foo o
  ```

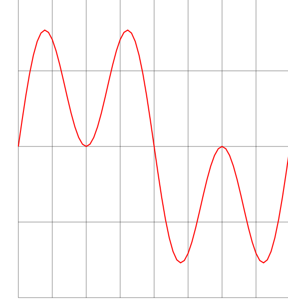
  那么该行会被转成 \pgfplotstreampointoutlier{\pgfpointxy{4}{16}},即被转成 outlier 点;
- 其它形式的"文字符号"会被忽略。

\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。

\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*⟩ 相当于 \foreach ⟨*variable*⟩ in ⟨*sample list*⟩... 中的变量和变量取值列表, 其格式要符合 \foreach 语句的限制, 不过本命令只能使用一个变量。⟨*function point*⟩ 是数学表达式,各个变量值用于数学表达式中参与运算,运算结果用于构造图形中的点。

```
\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。

`\pgfplotgnuplot[`⟨*prefix*⟩`]{`⟨*function*⟩`}`

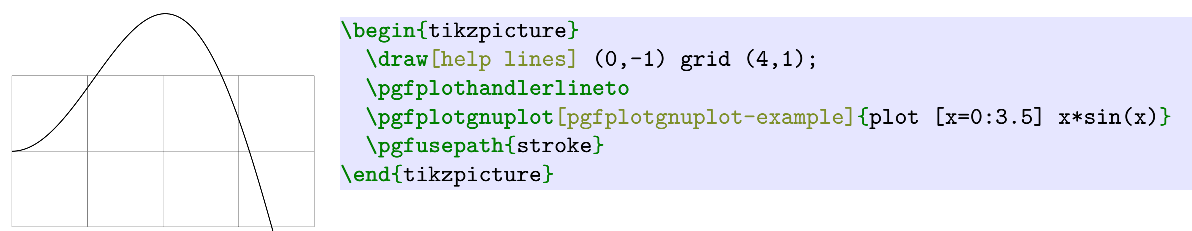
本命令调用外部程序 gnuplot 来创建函数 ⟨*function*⟩ 的图流，这里 ⟨*function*⟩ 是用 gnuplot 句法构造的函数表达式。如果不指定 ⟨*prefix*⟩，则它是 `\jobname`，即当前 tex 文件的名称。

在第一次处理本命令时,需要在命令行使用 `-shell-escape` 来编译。本命令涉及文件 ⟨*prefix*⟩`.gnuplot` 和 ⟨*prefix*⟩`.table`，如果没有这两个文件，PGF 先创建文件 ⟨*prefix*⟩`.gnuplot`，并在这个文件中写下如下代码：

```
set terminal table; set output "#1.table"; set format "%.5f"
⟨function⟩
```

其中的 `#1` 会被 ⟨*prefix*⟩ 代替，文件的第二行是 ⟨*function*⟩. 然后，PGF 会调用 gnuplot 来处理文件 ⟨*prefix*⟩`.gnuplot`，gnuplot 会创建文件 ⟨*prefix*⟩`.table`，计算函数 ⟨*function*⟩ 的样本点数据，并把数据写入文件 ⟨*prefix*⟩`.table` 中。这需要开启 `\write18` 机制（即 shell escape），否则不能调用 gnuplot. 然后命令 `\pgfplotxyfile` 会处理文件 ⟨*prefix*⟩`.table`，生成一个图流。

例如，调用 gnuplot 编译下面的图形：



```
\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
0.00186 3.14100  i
0.74672 2.78074  i
1.30987 2.26617  i
% 省略若干
```

如果 `.gnuplot` 文件已经存在但 `.table` 文件不存在，那么 PGF 会检查 `.gnuplot` 文件的内容是否是创建函数 ⟨*function*⟩ 所需要的代码，如果是就继续调用 gnuplot；如果不是就清除 `.gnuplot` 文件原来的内容，写下所需要的代码，再调用 gnuplot，然后生成 `.table` 文件用于创建图流。

如果 `.gnuplot` 文件和 `.table` 文件都已经存在，那么 PGF 会检查 `.gnuplot` 文件的内容是否是创建函数 <function> 所需要的代码，如果是则直接用命令 `\pgfplotxyfile` 处理 `.table` 文件生成一个图流；如果不是就清除 `.gnuplot` 文件原来的内容，写下所需要的代码，然后用命令 `\pgfplotxyfile` 处理 `.table` 文件生成一个图流；也就是说，当 `.table` 文件存在时，PGF 不会调用 gnuplot，此时使用 TEX 编辑器的排版按钮就可以编译代码、得到函数图形，不必在命令行编译。也要注意，如果你自行修改了 `.table` 文件的数据，那么得到的图形就很可能不是原来函数的图像。

`/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"}
```

112.3 图柄

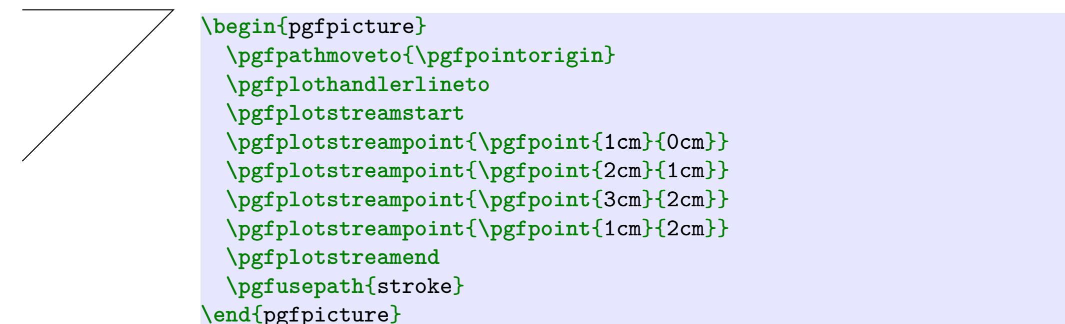
图柄（plot handler）决定用什么方式将图流中的点联系起来。你必须在开启图流之前写出所用的图柄。图柄通过设置或者重置 `\pgf@plotstreamstart`，`\pgf@plotstreampoint`，`\pgf@plotstreamend` 这三个命令来起作用。

注意，图流是全局定义的，因此图柄的作用不受 TEX 分组或者子环境 `{scope}` 的限制。一个图柄对之后的各个图流都会有作用，直到更换图柄。

下面是 PGF 预定义的几个图柄，另外库 pgflibraryplothandlers 也定义了大量图柄。

`\pgfplothandlerlineto`

这个图柄将命令 `\pgfpathlineto` 用于图流中的点，图流的第一个点除外。



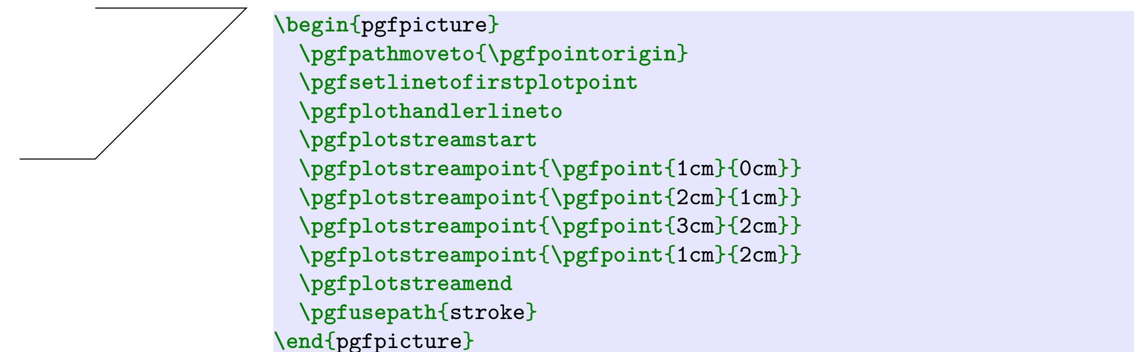
```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfplothandlerlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfsetmovetofirstplotpoint`

本命令将 move-to 操作作用于图流的第一个点，因此本命令或者把之后的图流点作为新的子路径的开端，或者把之后的图流创建为一个新路径。在默认下，画线的图柄，例如 `\pgfplothandlerlineto` 会调用本命令。

`\pgfsetlinetofirstplotpoint`

本命令将 line-to 操作作用于图流的第一个点。



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfsetlinetofirstplotpoint
  \pgfplothandlerlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfplothandlerpolygon`

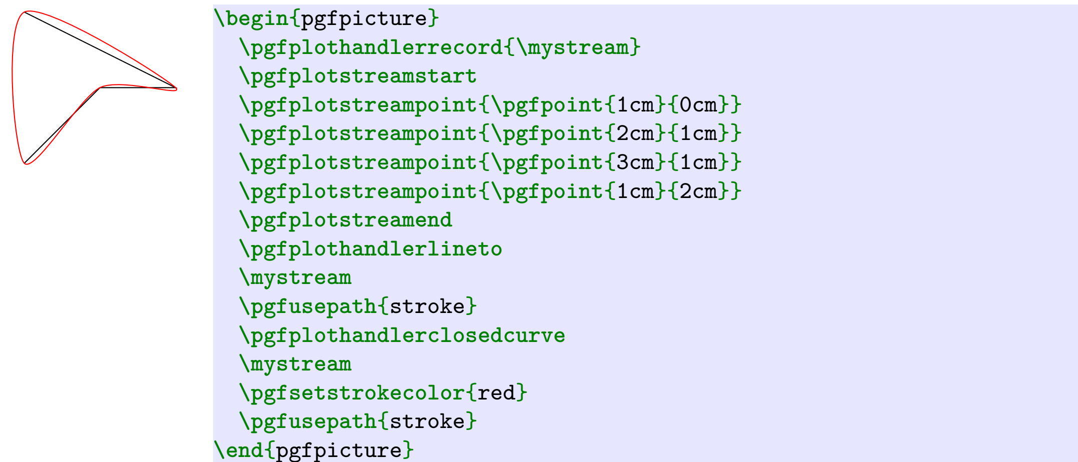
这个图柄类似 `\pgfplothandlerlineto`,只是本命令会在图流的最后一个点之后加上命令 `\pgfpathclose`,将图流作成一个封闭的多边形。

`\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。

`\pgfplothandlerrecord{⟨macro⟩}`

⟨*macro*⟩ 是个宏，不需要提前定义。写出本命令后，本命令之后的图流会保存在宏 ⟨*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。



```
\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}
```

112.4 定义新图柄

`\pgfdeclareplothandler{⟨macro⟩}{⟨arguments⟩}{⟨configuration⟩}`

本命令定义一个图柄，并将图柄保存在宏 ⟨*macro*⟩ 中。⟨*arguments*⟩ 是 `#1`，`#1#2#3` 这种形式的变量列举，对应宏 ⟨*macro*⟩ 的参数。⟨*configuration*⟩ 是个“键值对”列表，具体规定图柄的作用，其中可以使用 ⟨*arguments*⟩ 列出的变量。

下面假设定义一个图柄 \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。

start=⟨**code**⟩ 当使用图柄 \myhandler 处理图流时，若图柄遇到 \pgfplotstreamstart 则执行 ⟨*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
```

注意以上代码不需要放在绘图环境 {pgfpicture} 内。

end=⟨**code**⟩ 类似选项 start，图柄会在遇到 \pgfplotstreamend 时执行 ⟨*code*⟩.

point=⟨**code**⟩ 当使用图柄 \myhandler 处理图流时，若图柄遇到 \pgfplotstreampoint，则执行 ⟨*code*⟩. ⟨*code*⟩ 中可以使用 ⟨*arguments*⟩ 列出的变量。如果 ⟨*code*⟩ 中使用变量 ##1，则这个变量符号会引用 \pgfplotstreampoint{⟨*point*⟩} 中的坐标 ⟨*point*⟩.

Hi 某.
nice 天气
Bye 某.
nice 天气

```
\pgfdeclareplothandler{\myhandler}{#1#2}{
  start = Hi #1.,
  end = Bye #1.,
  point = nice #2
}
\myhandler{某}{天气}
\pgfplotstreamstart \\
\color{red}
\pgfplotstreampoint{} \\
\color{cyan}
\pgfplotstreamend \\
\pgfplotstreampoint{}
```

jump=⟨**code**⟩ 命令 \pgfplotstreampointoutlier,\pgfplotstreampointundefined,\pgfplotstreamnewdataset, 都可能引起 “jump”，“jump” 选项的动作就是执行这里的 ⟨*code*⟩，在 ⟨*code*⟩ 中可以使用 ⟨*arguments*⟩ 列出的变量。

special=⟨**code**⟩ 当使用图柄 \myhandler 处理图流时,若图柄遇到 \pgfplotstreamspecial{⟨*something*⟩}, 则执行 ⟨*code*⟩. 如果 ⟨*code*⟩ 中使用变量 ##1，则这个变量符号会引用 {⟨*something*⟩}. 在 ⟨*code*⟩ 中可以使用 ⟨*arguments*⟩ 列出的变量。

point nacro=⟨**some macro**⟩ ⟨*some macro*⟩ 是个宏，使用这个选项后，图流命令 \pgfplotstreampoint 会调用宏 ⟨*some macro*⟩，而内部命令 \pgf@plotstreampoint 会等同于宏 ⟨*some macro*⟩. 在⟨*some macro*⟩ 中至多能使用一个变量 #1，这个变量引用 \pgfplotstreampoint{⟨*point*⟩} 中的 ⟨*point*⟩.

special macro=⟨**some code**⟩ ⟨*some macro*⟩ 是个宏, 使用这个选项后, 图流命令 \pgfplotstreamspecial 会调用宏 ⟨*some macro*⟩. 在 ⟨*some macro*⟩ 中至多能使用一个变量 #1,这个变量引用 \pgfplotstreamspecial{⟨*someth* 中的 ⟨*something*⟩.

start macro=⟨***some macro***⟩ ⟨*some macro*⟩ 是个宏，使用这个选项后，图流命令 \pgfplotstreamstart 会调用宏 ⟨*some macro*⟩. 在 ⟨*some macro*⟩ 中不能使用变量。

end macro=⟨***some macro***⟩ 类似以上。⟨*some macro*⟩ 是个宏,使用这个选项后,图流命令 \pgfplotstreamend 会调用宏 ⟨*some macro*⟩. 在 ⟨*some macro*⟩ 中不能使用变量。

jump macro=⟨***some macro***⟩ 类似以上。⟨*some macro*⟩ 是个宏，使用这个选项后，选项“jump”会调用宏 ⟨*some macro*⟩. 在 ⟨*some macro*⟩ 中不能使用变量。

在文件《pgflibraryplothandlers.code.tex》中定义了图柄 \pgfplothandlercurveto：

```
24 \pgfdeclareplothandler{\pgfplothandlercurveto}{}{%
25   point macro=\pgf@plot@curveto@handler@initial,
26   jump macro=\pgf@plot@smooth@next@moveto,
27   end macro=\pgf@plot@curveto@handler@finish
28 }%
29
30 \def\pgf@plot@smooth@next@moveto{%
31   \pgf@plot@curveto@handler@finish%
32   \global\pgf@plot@startedfalse%
33   \global\let\pgf@plotstreampoint\pgf@plot@curveto@handler@initial%
34 }%
35
36 \def\pgf@plot@curveto@handler@initial#1{%
37   \pgf@process{#1}%
38   \pgf@xa=\pgf@x%
39   \pgf@ya=\pgf@y%
40   \pgf@plot@first@action{\pgfqpoint{\pgf@xa}{\pgf@ya}}%
41   \xdef\pgf@plot@curveto@first{\noexpand\pgfqpoint{\the\pgf@xa}{\the\pgf@ya}}%
42   \global\let\pgf@plot@curveto@first@support=\pgf@plot@curveto@first%
43   \global\let\pgf@plotstreampoint=\pgf@plot@curveto@handler@second%
44 }%
45
46 \def\pgf@plot@curveto@handler@second#1{%
47   \pgf@process{#1}%
48   \xdef\pgf@plot@curveto@second{\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}%
49   \global\let\pgf@plotstreampoint=\pgf@plot@curveto@handler@third%
50   \global\pgf@plot@startedtrue%
51 }%
52
53 \def\pgf@plot@curveto@handler@third#1{%
54   \pgf@process{#1}%
```

```
55    \xdef\pgf@plot@curveto@current{\noexpand\pgfqpoint{\the\pgf@x}{\the\pgf@y}}%
56    % compute difference vector:
57    \pgf@xa=\pgf@x%
58    \pgf@ya=\pgf@y%
59    \pgf@process{\pgf@plot@curveto@first}
60    \advance\pgf@xa by-\pgf@x%
61    \advance\pgf@ya by-\pgf@y%
62    % compute support directions:
63    \pgf@xa=\pgf@plottension\pgf@xa%
64    \pgf@ya=\pgf@plottension\pgf@ya%
65    % first marshal:
66    \pgf@process{\pgf@plot@curveto@second}%
67    \pgf@xb=\pgf@x%
68    \pgf@yb=\pgf@y%
69    \pgf@xc=\pgf@x%
70    \pgf@yc=\pgf@y%
71    \advance\pgf@xb by-\pgf@xa%
72    \advance\pgf@yb by-\pgf@ya%
73    \advance\pgf@xc by\pgf@xa%
74    \advance\pgf@yc by\pgf@ya%
75    \edef\pgf@marshal{\noexpand\pgfpathcurveto{
      ↪   \noexpand\pgf@plot@curveto@first@support}%
76      {\noexpand\pgfqpoint{\the\pgf@xb}{\the\pgf@yb}}{
        ↪   \noexpand\pgf@plot@curveto@second}}%
77    {\pgf@marshal}%
78    % Prepare next:
79    \global\let\pgf@plot@curveto@first=\pgf@plot@curveto@second%
80    \global\let\pgf@plot@curveto@second=\pgf@plot@curveto@current%
81    \xdef\pgf@plot@curveto@first@support{\noexpand\pgfqpoint{\the\pgf@xc}{\the\pgf@yc
      ↪   }}%
82  }%
83
84  \def\pgf@plot@curveto@handler@finish{%
85    \ifpgf@plot@started%
86      \pgfpathcurveto{\pgf@plot@curveto@first@support}{\pgf@plot@curveto@second}{
        ↪   \pgf@plot@curveto@second}%
87    \fi%
88  }%
89
```

```
90
91  % This commands sets the tension for smoothing of plots.
92  %
93  % #1 = tension of curves. A value of 1 will yield a circle when the
94  %      control points are at quarters of a circle. A smaller value
95  %      will result in a tighter curve. Default is 0.5.
96  %
97  % Example:
98  %
99  % \pgfsetplottension{0.2}
100
101 \def\pgfsetplottension#1{%
102   \pgf@x=#1pt\relax%
103   \pgf@x=0.2775\pgf@x\relax%
104   \edef\pgf@plottension{\pgf@sys@tonumber\pgf@x}}%
105 \pgfsetplottension{0.5}%
```

上面第 101 至 105 行表明: 若 `\pgfsetplottension` 的参数是 t, 则 `\pgf@plottension` 的值是 $0.2775\times t$, 这个乘积用在了第 63, 64 行。

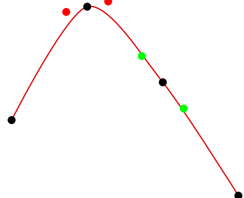
图柄 `\pgfplothandlercurveto` 的作用是，例如:

- 记宏 `\pgf@plottension` 的值是 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; 这一操作是第 40 行;
  2. 计算 $Q_{1,2}=P_2-T\cdot(P_3-P_2)$，$Q_{2,3}=P_2+T\cdot(P_3-P_2)$;
  3. 以当前点 P_1 为起点，执行 `\pgfpathcurveto{`P_1`}{`$Q_{1,2}$`}{`P_2`}`，此时 P_2 成为当前点;
  4. 以当前点 P_2 为起点，执行 `\pgfpathcurveto{`$Q_{2,3}$`}{`P_2`}{`P_2`}`.

- 记宏 `\pgfsetplottension` 的值是 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 $Q_{1,2}=P_2-T\cdot(P_3-P_1)$，$Q_{2,3}=P_2+T\cdot(P_3-P_1)$;
  3. 以当前点 P_1 为起点，执行 `\pgfpathcurveto{`P_1`}{`$Q_{1,2}$`}{`P_2`}`，此时 P_2 成为当前点;
  4. 计算 $R_{2,3}=P_3-T\cdot(P_4-P_2)$，$R_{3,4}=P_3+T\cdot(P_4-P_2)$;
  5. 以当前点 P_2 为起点，执行 `\pgfpathcurveto{`$Q_{2,3}$`}{`$R_{2,3}$`}{`P_3`}`，此时 P_3 成为当前点;
  6. 以当前点 P_3 为起点，执行 `\pgfpathcurveto{`$R_{3,4}$`}{`P_4`}{`P_4`}`.

可见图柄 `\pgfplothandlercurveto` 跟命令 `\pgfpathcurveto` 是很不一样的。

下面例子中先用 `\pgfplothandlercurveto` 画线，再用 tikz 的 curve-to 操作画线，两条线重合:



```
\begin{tikzpicture}
  \pgfplothandlercurveto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0.5cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{4cm}{-0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}

  \tikzmath{
    \T=0.13875; % =0.2775*0.5, 因为默认 \pgfsetplottension{0.5}
    coordinate \p,\q,\r;
    \p1=(1,0.5);
    \p2=(2,2);
    \p3=(3,1);
    \p4=(4,-0.5);
    \q1=($(\p2)-\T*($(\p3)-(\p1)$)$);
    \q2=($(\p2)+\T*($(\p3)-(\p1)$)$);
    \r1=($(\p3)-\T*($(\p4)-(\p2)$)$);
    \r2=($(\p3)+\T*($(\p4)-(\p2)$)$);
  }
  \draw [red](\p1)..controls(\p1)and(\q1)..(\p2)..controls(\q2)and(\r1)..(\p3)..controls(\r2)and(
  ↪  \p4)..(\p4);

  \foreach \i in{(\p1),(\p2),(\p3),(\p4)} \fill \i circle (1.5pt);
  \foreach \i in{(\q1),(\q2)}             \fill [red] \i circle (1.5pt);
  \foreach \i in{(\r1),(\r2)}             \fill [green] \i circle (1.5pt);
\end{tikzpicture}
```

113 图层

113.1 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”。

113.2 声明图层

PGF 的图层都有自己的名称，预定义的图层是 `main`, 如果不声明其它图层，所有绘图环境都在 `main` 层上表现出来。

`\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。

这个命令的声明是全局有效的。

`\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。

113.3 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“排序”的图层名称，⟨*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。

注意不能针对 `main` 使用本环境。

`\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}
```

114 颜色渐变

114.1 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。

114.1.1 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。关于颜色的中文资料，我只见过《印刷色彩学》，《颜色信息工程》，其它无缘得见。

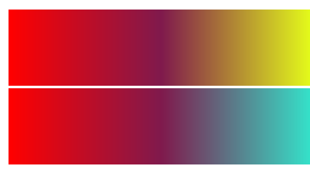
114.2 声明渐变样式

114.2.1 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。

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：

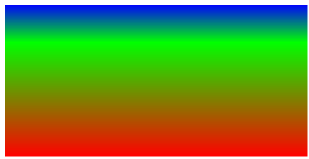


```
\pgfdeclarehorizontalshading[mycolorA,mycolorB]{myshadingA}
  {1cm}{rgb(0cm)=(1,0,0); color(2cm)=(mycolorA);
  ↪   color(4cm)=(mycolorB)}
\definecolor{mycolorA}{rgb}{0.5,0.1,0.3}
\colorlet{mycolorB}{cyan!10!yellow}
\pgfuseshading{myshadingA}
\colorlet{mycolorB}{cyan!80!orange}
\pgfuseshading{myshadingA}
```

在声明一个颜色渐变时，PGF 实际上只是保存定义，并不计算颜色值，因此 ⟨*color list*⟩ 中的颜色名称可以是“没有颜色的”。或者说，如果 ⟨*color list*⟩ 中的颜色名称是尚未被定义的颜色，那么本命令就把这个名称声明为“颜色名称”，只不过颜色值为“空”。只有在使用命令 \pgfuseshading 创建颜色渐变时，PGF 才会检查需要的颜色值并计算出渐变颜色值，因此在使用命令 \pgfuseshading 之前要确保所需要的颜色都有颜色值。可见使用可选项 ⟨*color list*⟩ 的方便之处在于，可以随时更改渐变中的颜色。

\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}
```

114.2.2 辐射渐变

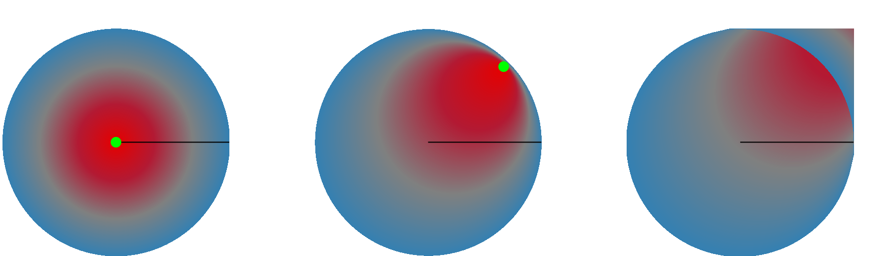
辐射渐变的意思是，颜色以某个点为中心，向外围渐变。

\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}}
```

114.2.3 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，见后文命令的例子。

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，是用一些基本的“算子”构造的。PostScript 语言中有很多算子，这里只能使用某些限定的算子来构造 type 4 类型的函数，在后文对此有说明。程序通过执行函数 ⟨*type 4 function*⟩ 来计算坐标点所对应的颜色，执行函数计算的是 PDF 渲染器，而不是 PGF 或 TeX，所以这里使用 type 4 类型的函数，被执行的函数不能过于复杂，函数中的错误也只有渲染器能注意到。

使用 type 4 类型函数的句法是 PostScript 语言的一部分句法，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；

之后经过函数计算，输出的是：(1) rgb 模式下的 3 个颜色参数，或者 (2) cmyk 模式下的 4 个颜色参数，或者 (3) gray 模式下的 1 个颜色参数——对应该点的颜色。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）。

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` 的处理就是：

```
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␣$-(y-45)^2-0.5(x-40)^2$

$\texttt{1.0005␣exch} \to \texttt{x␣y␣1.0005␣}{-(y-45)^2-0.5(x-40)^2}$

$\texttt{exp} \to \texttt{x␣y␣}1.0005^{-(y-45)^2-0.5(x-40)^2}$

$\texttt{1.0␣exch} \to \texttt{x␣y␣1.0␣}1.0005^{-(y-45)^2-0.5(x-40)^2}$

$\texttt{sub} \to \texttt{x␣y␣}1.0-1.0005^{-(y-45)^2-0.5(x-40)^2}$

$\texttt{3␣1␣roll} \to 1.0-1.0005^{-(y-45)^2-0.5(x-40)^2}\texttt{␣x␣y}$

最后的结果就是:

$$\texttt{1.0␣}1.0-1.0005^{-(y-45)^2-0.5(x-40)^2}\texttt{␣}1.0-1.002^{-0.5(y-70)^2-(x-70)^2}$$

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，必须从颜色名称对应的颜色中提取颜色参数并保存在某个宏中，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 个命令放在 ⟨*init code*⟩ 中。

`\pgfshadecolortorgb{`⟨*color name*⟩`}{`⟨*macro*⟩`}`

这个命令将颜色名称 ⟨*color name*⟩ 对应颜色的三个 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。保存在宏 ⟨*macro*⟩ 中的三个颜色参数是 0 到 1 之间的数值，之间由空格分隔。本命令会定义宏 ⟨*macro*⟩, 如果 ⟨*macro*⟩ 已经存在，则本命令会重定义它。⟨*macro*⟩ 可以用作 ⟨*type 4 function*⟩ 的参数。

0.0 1.0 1.0

```
\pgfshadecolortorgb{cyan}{\RGBcyan}
\RGBcyan
```

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……

当使用本命令后，PGF 还会自动定义 3 个宏来分别保存三个颜色参数值。例如，执行

```
\pgfshadecolortorgb{orange}{\mycol}
```

的时候，PGF 会自动定义 \mycolred, \mycolgreen, \mycolblue, 它们的名称是由宏 \mycol 的名称与 red，green，blue 结合而成的，它们分别保存颜色 orange 的（rgb 模式下的）三个颜色参数。

1.0 0.5 0.0
1.0
0.5
0.0

```
\pgfshadecolortorgb{orange}{\mycol}
\mycol\\
\mycolred\\
\mycolgreen\\
\mycolblue
```

\pgfshadecolortocmyk{⟨*color name*⟩}{⟨*macro*⟩}

把颜色名称 ⟨*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。

0.0 0.5 1.0 0.0
0.0
0.5
1.0
0.0

```
\pgfshadecolortocmyk{orange}{\mycol}
\mycol\\
\mycolcyan\\
\mycolmagenta\\
\mycolyellow\\
\mycolblack
```

\pgfshadecolortogray{⟨*color name*⟩}{⟨*macro*⟩}

把颜色名称 ⟨*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；输出一个数值 ⟨*g*⟩ 代表 gray 模式下的一个颜色。使用下面的命令，可以把 ⟨*x y z*⟩ 映射为 ⟨*p q r s*⟩ 或 ⟨*g*⟩, 也可以把 ⟨*p q r s*⟩ 映射为 ⟨*x y z*⟩ 或 ⟨*g*⟩, 也可以把 ⟨*g*⟩ 映射为 ⟨*x y z*⟩ 或 ⟨*p q r s*⟩.

参考 xcolor 宏包。

```
\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.

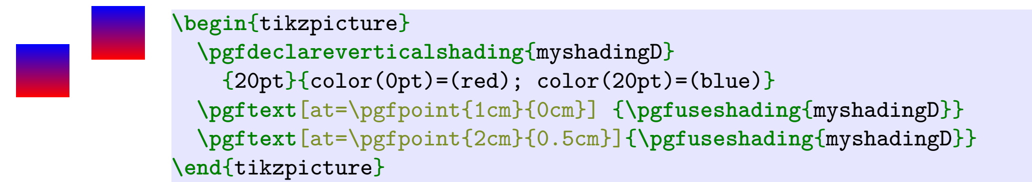
\ifpgfshadingmodelgray

用在命令 \pgfdeclarefunctionalshading 中，检查“创建颜色渐变时”的颜色模式是否是 gray.

114.3 使用颜色渐变

\pgfuseshading{⟨*shading name*⟩}

这个命令在一个盒子中创建渐变 ⟨*shading name*⟩, 并将盒子插入到文档中。本命令用在 \pgftext 之内，可以用于 {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}
```

\pgfshadepath{⟨*shading name*⟩}{⟨*angle*⟩}

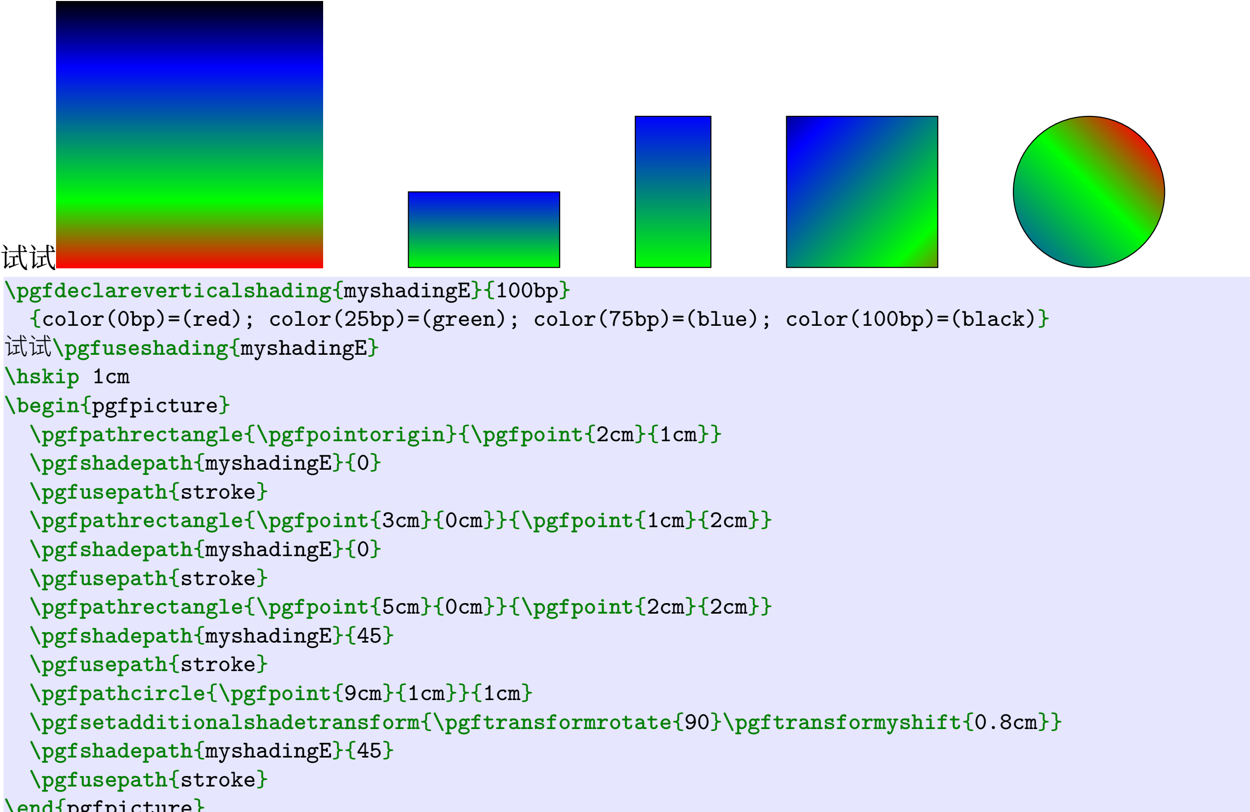
这个命令必须用在 {pgfpicture} 环境中。当创建一个路径后，可以使用本命令来填充当前路径，填充内容就是名称为 ⟨*shading name*⟩ 的颜色渐变。

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*⟩ 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。用在这个命令中的渐变 ⟨*shading name*⟩ 的盒子宽度和高度都应当是 100bp. PGF 会计算路径的边界盒子和渐变盒子的尺寸，当然二者的尺寸可能相差很大。为了能让渐变填满路径的边界盒子，需要对渐变盒子做变换。

在这个局部域内，PGF 调整底层的变换矩阵，使得渐变盒子的中心与路径边界盒子的中心重合，这是个平移变换。然后参照盒子中心点对渐变盒子做伸缩变换，PGF 自动确定伸缩变换在水平方向和竖直方向上的伸缩系数，使得渐变盒子的宽度和高度分别是路径边界盒子的 2 倍。注意此时路径边界盒子只能覆盖渐变盒子面积的 $\frac{1}{4}$. 之后，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。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。再之后，用路径剪切渐变盒子，得到填充效果。

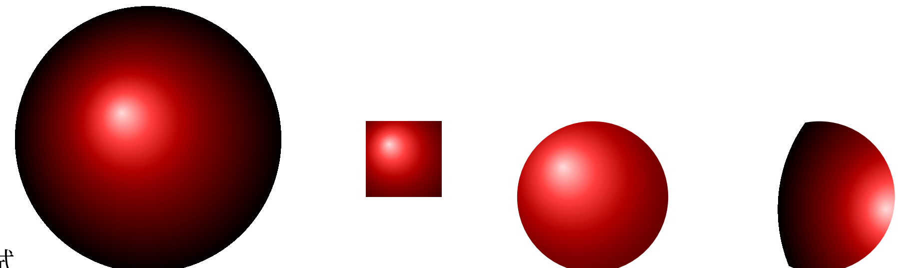
\pgfsetadditionalshadetransform{⟨*transformation*⟩}

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。

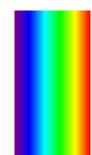



```
\pgfdeclareverticalshading{myshadingE}{100bp}
  {color(0bp)=(red); color(25bp)=(green); color(75bp)=(blue); color(100bp)=(black)}
试试\pgfuseshading{myshadingE}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingE}{0}
  \pgfusepath{stroke}
  \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。




```
\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}
```

114.4 关于 type 4 函数的补充

以下内容来自《PDF Reference》(version 1.7) 的 §3.

PDF 不是程序语言，一个 PDF 文件也不是一个程序。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			
堆栈算子	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。

设想有一个数值堆栈（stack），将某些数值按次序存放到这个堆栈中，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堆栈之后，算子针对堆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，本算子从数值堆栈中选取靠近“顶部”的某些数值，复制这些数值并将它们添加到原来的堆栈中，即用复制的方法增加堆栈中的数值数量，增加堆栈长度（高度）。本算子选取数值的“标准”由堆栈的“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，本算子的输入只是堆栈顶端的数值，复制顶端值并添加到堆栈中。

```
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，交换它们在堆栈中的位置。

```
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，从中堆栈中选出某个数值，复制它并添加到堆栈中，使之处于“顶端”位置。本算子选择数值的标准由原堆栈的“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。它删除堆栈中的顶端数值。

```
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堆栈中的其它数值区别开来，参照这两个正整数，将堆栈中的其余数值做轮换。

$$any_{n-1} \cdots any_0 \; n \; j \; \texttt{roll} \;\text{输出}\; any_{(j-1) \bmod n} \cdots any_0 \; any_{n-1} \cdots any_{j \bmod n}$$

```
(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）。

115 透明度

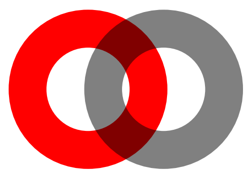
先阅读 §23.

115.1 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⟩}`

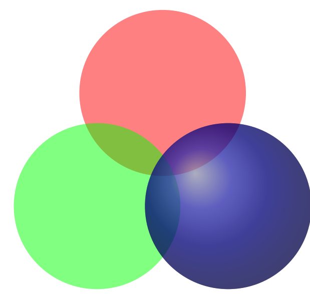
本命令为画线操作指定不透明度，⟨*value*⟩ 是 0 到 1 之间的数值，1 代表“完全不透明”，0 代表“完全透明”。



```
\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}
```

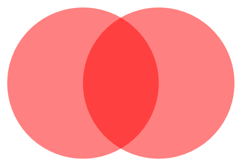
`\pgfsetfillopacity{⟨value⟩}`

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。

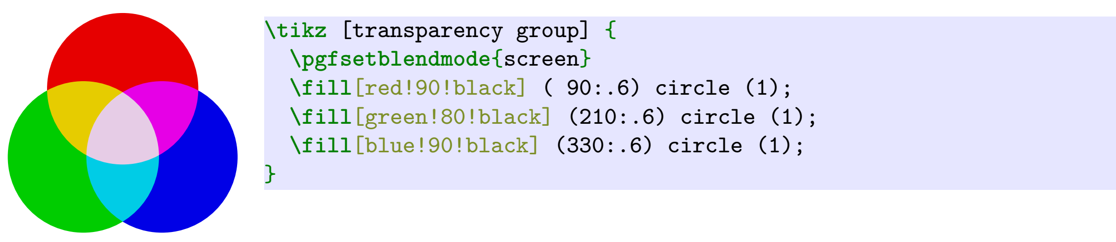


```
\begin{tikzpicture}
  \pgfsetfillopacity{0.5}
  \fill[red] (0,0) circle (1);
  \fill[red] (1,0) circle (1);
\end{tikzpicture}
```

115.2 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);
}
```

115.3 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。

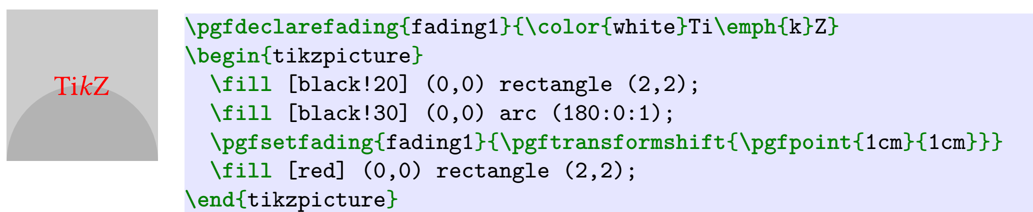
假设在一个原本没有颜色（黑色）的 TEX 盒子里写下单词 TikZ, 并且文字颜色是白色，此时盒子里的点只有两种：一种是构成文字的白色点，亮度为 1，另一种是没有颜色（黑色）的点，亮度为 0. 然后把这盒子想象成一个特别的“印章”，把另外某个盒子看作是“纸”，把“印章”盖到“纸”上，同时把印章的亮度值转换成不透明度值“印”在“纸”上，这样“纸”上的点就有了自己的不透明度值。如果这张“纸”原本是红色的，被盖章后，被白色文字 TikZ 覆盖的部分的不透明度是 1，其余部分的不透明度都是 0，于是原本的红色“纸”就变了，纸上有红色的文字 TikZ, 除文字外，纸的其余部分都没有颜色（完全透明，没有可见光能量）。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}
```

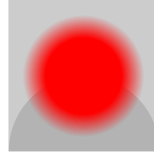
上面例子中，白色印章 fading1 印在红色纸上，得到红色文字 TikZ.

`pgftransparent` 这个预定义颜色就是黑色，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 分组）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}
```

115.4 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。

参考 §23.5. 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。

116 Animations

```
\usepgfmodule{animations} % LaTeX and plain TeX and pure pgf
\usepgfmodule[animations] % ConTeXt and pure pgf
```

这个模块支持动画。

本节介绍的是基本层对动画的支持。Tikz 在创建动画时，会把相关的句法转换为基本层的命令。

116.1 Overview

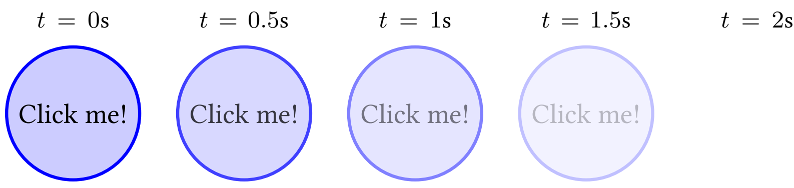
PGF 使用 SVG 格式来描述动画，对于动画的计算和展示则由 SVG 阅读器完成。

116.2 Animating an Attribute

116.2.1 主要命令

`\pgfanimateattribute{⟨attribute⟩}{⟨options⟩}`

这里的属性 ⟨*attribute*⟩ 是某个尚未创建的对象的属性。⟨*options*⟩ 用于指定时刻、时刻对应的属性值。每一次使用选项 `/pgf/animation/entry`→P. 807 都可以添加一对 “时刻-值”，其它选项也能影响时间线。




```
\tikz { \pgfanimateattribute{opacity}{
    whom = node, begin on = {click}, entry = {0s}{1}, entry = {2s}{0} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};}
```

属性 命令 `\pgfanimateattribute` 会开启一个 TEX-scope, ⟨*options*⟩ 中的选项会被冠以前缀路径 `/pgf/animation/` 来执行；最后，各种系统层命令 `\pgfsysanimate...` 创建动画，然后结束 scope. 可用于制作动画的属性如下：

1. `draw`, `fill`, 路径线条颜色，路径填充色
2. `line width`, 路径的线宽，其值为带单位的尺寸
3. `motion`, 对象的位置，配合选项 `along` 使用，使得对象沿着指定的路径运动；其值为纯数字，代表路径上的点，数字 0 代表路径的起点，数字 1 代表路径的终点
4. `opacity`, `fill opacity`, `draw opacity`, 不透明度，其值为 0 到 1 之间的数字

5. `path`, 路径
6. `rotate`, 旋转状态，其值为数字，代表角度
7. `scale`, 放缩状态，其值为一个数字（如 `1.5`），或者两个数字（如 `0.5,2`, 两个数字之间用逗号分隔），分别代表沿着 x 轴方向的伸缩比例，以及沿着 y 轴方向的伸缩比例
8. `softpath`, 软路径
9. `translate`, 对象的平移向量，其值为向量，如 `\pgfpoint{1cm}{2cm}`
10. `view`, 其值为 viewbox
11. `visible`, 指定对象是否可见，其值为 `true` 或 `false`
12. `stage`, 类似 `visible`, 不过其值默认为 `false`
13. `xskew`，`yskew`, 对象的倾斜状态，其值为数字，代表角度

制作动画的对象 在 ⟨*options*⟩ 中使用选项 `whom=`⟨*name*⟩ 来指定制作动画的对象，⟨*name*⟩ 是对象的名称，并且这个对象必须是尚未被创建的、需要在稍后创建的、需要被命名为 ⟨*name*⟩ 的。在 SVG 格式中，动画对象可以出现在任何位置，不必限制在动画代码之后，不过 PGF 有此限制。

`/pgf/animation/whom=`⟨*id*⟩`.`⟨*type*⟩ (no default)

这个选项用在命令 `\pgfanimateattribute` 的 ⟨*options*⟩ 中,用于指定动画对象。注意一个 `\pgfanimateattribute` 命令中只能使用一次 `whom` 选项。这里的 ⟨*id*⟩ 和可选的 ⟨*type*⟩ 会被命令 `\pgfidrefnextuse`→P. 637 解析。

命令 `\pgfanimateattribute` 应当与它的对象 ⟨*id*⟩.⟨*type*⟩ 处于同一个页面内。

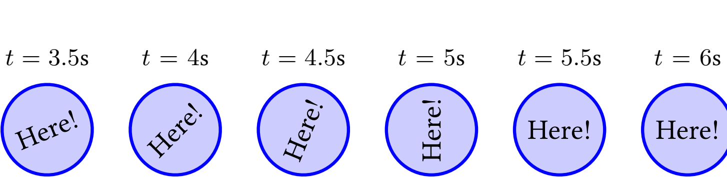
给动画命名 用选项 `name` 给动画命名后，可以用其名称引用它，例如动画名称可以用作选项 `of`，`of next` 的值。

`/pgf/animation/name=`⟨*name*⟩ (no default)

本选项给动画命名，可以用名称 ⟨*name*⟩ 引用动画。

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s $t = 2.5$s $t = 3$s

Here! Here! Here! Here! Here! Here! Here!



```
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {end, of next = my move animation, delay = 1s},
    entry = {0s}{0}, entry = {2s}{90}, begin snapshot = 3s, }
  \pgfanimateattribute{translate}{
    name = my move animation, whom = node, begin on = {click},
    entry = {0s}{\pgfpointorigin}, entry = {2s}{\pgfpoint{0cm}{-5mm}} }
  \node (node) [fill = blue!20, draw = blue, circle] {Here!};
}
```

`\pgfanimateattributecode{`⟨*attribute*⟩`}{`⟨*code*⟩`}`

本命令类似 `\pgfanimateattribute`, 不过 ⟨*code*⟩ 是代码，不是选项。

116.2.2 时间线选项

`/pgf/animation/entry{`⟨*time*⟩`}{`⟨*value*⟩`}` (no default)

本选项时刻 ⟨*time*⟩ 时的属性值 ⟨*value*⟩. 当用本选项指定多个 “时刻-值” 时, PGF 会在这些 “时刻-值” 之间做插值计算。“时刻” 必须按（非严格）递增的时序给出。⟨*time*⟩ 会被命令 `\pgfparsetime` 解析。

对于 “时刻-属性值” 的计算通常是线性插值计算，使用下面的选项能影响插值计算。

`/pgf/animations/exit control={`⟨*time fraction*⟩`}{`⟨*value fraction*⟩`}` (no default)

与 `/tikz/animate/options/exit control` 等效。

`/pgf/animations/entry control={`⟨*time fraction*⟩`}{`⟨*value fraction*⟩`}` (no default)

与 `/tikz/animate/options/entry control` 等效。

`/pgf/animations/linear` (no value)

这是 `exit control={0}{0}`, `entry control={1}{1}` 的简写。

`/pgf/animations/stay` (no value)

与 `/tikz/animate/options/stay` 等效。

`/pgf/animations/jump` (no value)

与 `/tikz/animate/options/jump` 等效。

`/tikz/animations/base=`⟨*value*⟩ (no default)

见 `/tikz/animations/base`.

如果两个时间线对同一对象的同一属性在同一时刻分别指定了不同的值，从而出现相互冲突的情况，那么 Tikz 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. 304 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。

\pgfparsetime{⟨*time*⟩}

这个命令解析时刻表达式 ⟨*time*⟩,它调用 \pgfmathparse 来工作,将解析结果保存在 \pgftimeresult 中，保存的结果是以“秒”（符号为 s）为单位的。

- 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。

116.2.3 快照

如果想获得动画过程在某个时刻的状态（某时刻的画面，类似“快照”），可以使用下面的命令。

\pgfsnapshot{⟨*time*⟩}

当在 TEX-scope 中使用这个命令时，\pgfanimateattribute 的行为会有所改变。此时不再创建动画，而是计算在时刻 ⟨*time*⟩ 时的属性值，从而得到在时刻 ⟨*time*⟩ 时的图形状态（代码），并把此时的图形代码插入到文档中。此命令对应选项 /tikz/make snapshot of→P. 309.

```
\foreach \t in {0,1,2,3,4} {
  \pgfsnapshot{\t}
  \tikz :rotate = { 0s = "0", 2s = "90", 2s = "180", 4s = "270" }
    \node [draw=blue, very thick] {f}; }
```

/tikz/animations/begin snapshot=⟨*begin time*⟩ (no default)

见 /tikz/animations/begin snapshot.

- 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. 307.

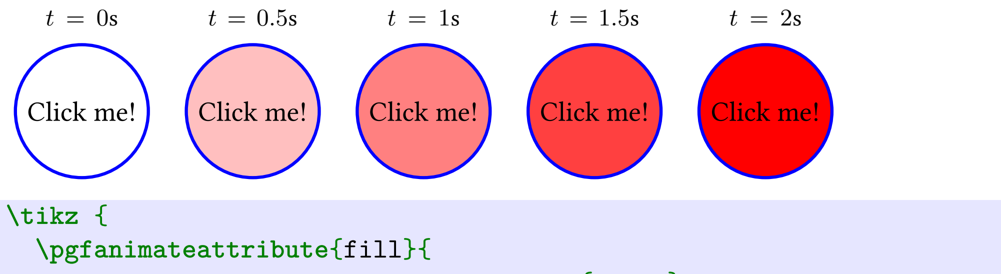
`\pgfsnapshotafter{⟨time⟩}`

这个命令对应选项 `/tikz/make snapshot after`→ P. 310.

116.3 Animating Color, Opacity, Visibility, and Staging

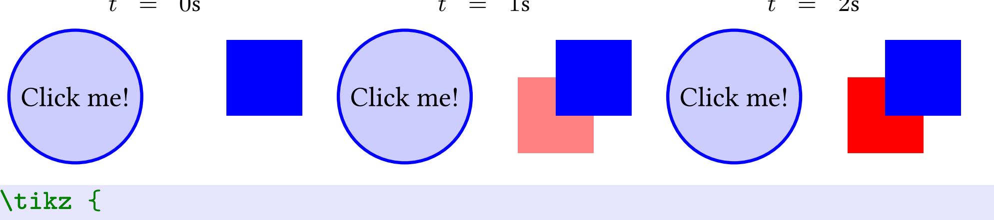
`\pgfanimateattribute{fill}{⟨options⟩}`

针对填充色属性。




```
\tikz {
  \pgfanimateattribute{fill}{
    whom = node.background, begin on = {click},
    entry = {0s}{white}, entry = {2s}{red} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

假如对填充色的动画设置是针对整个 scope 的，那么 scope 内的绘图命令自带的填充色选项会覆盖填充色动画。




```
\tikz {
  \pgfanimateattribute{fill}{
    whom = example, begin on = {click, of next=node},
    entry = {0s}{white}, entry = {2s}{red} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \begin{scope}[name = example]
    \fill (1.5,-0.75) rectangle ++ (1,1);
    \fill [blue] (2,-0.25) rectangle ++ (1,1);
  \end{scope}
}
```

上面例子中，名称为 `example` 的动画赋予了 `{scope}` 环境，环境内的第二个 `\fill` 命令自带了填充色设置，它画出的矩形没有动画效果。

`\pgfanimateattribute{draw}{⟨options⟩}`

当把 node 的背景路径作为动画对象、路径线条颜色作为属性时，可以设置选项 `whom=`⟨*node name*⟩`.background`.

当把 node 的文字作为动画对象、文字颜色作为属性时，可以设置选项 `whom=`⟨*node name*⟩`.text`.

$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s

Click! Text Click! Text Click! Text Click! Text Click! Text

```
\tikz {
  \pgfanimateattribute{fill}{
    whom = example.text, begin on = {click, of next=node},
    entry = {0s}{white}, entry = {2s}{red} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click!};
  \node at (1.5,0) (example) [fill = blue!20, circle, font=\small] {Text}; }
```

`\pgfanimateattribute{fill opacity}{`⟨*options*⟩`}`

`\pgfanimateattribute{draw opacity}{`⟨*options*⟩`}`

`\pgfanimateattribute{opacity}{`⟨*options*⟩`}`

`\pgfanimateattribute{visible}{`⟨*options*⟩`}`

`\pgfanimateattribute{stage}{`⟨*options*⟩`}`

116.4 Animating Paths and their Rendering

`\pgfanimateattribute{line width}{`⟨*options*⟩`}`

`\pgfanimateattribute{dash}{`⟨*options*⟩`}`

`\pgfanimateattribute{path}{`⟨*options*⟩`}`

`\pgfanimateattribute{path}{`⟨*options*⟩`}`

$t = 0$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 {
  \pgfanimateattribute{path}{
    whom = node.background.path, begin on = {click, of next=node},
    entry = {0s}{\pgfpathellipse{\pgfpointorigin}
                 {\pgfpointxy{1}{0}}{\pgfpointxy{0}{1.5}}},
    entry = {2s}{\pgfpathellipse{\pgfpointxy{.5}{0}}
                 {\pgfpointxy{.5}{.5}}{\pgfpointxy{0.25}{.25}}}}
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

⟨*options*⟩ 中的各个路径的必须具有相同的结构，即都采用相同的命令、操作，第一个路径可以用

current value 代替；所有路径都计入图形的边界盒子；不能用通常的办法给这种路径加箭头，要使用下面的选项加箭头。

/pgf/animation/arrows=⟨*start tip spec*⟩-⟨*end tip spec*⟩ (no default)

类似 /tikz/animate/arrows→P. 298，注意这个选项一定要写在动画路径的前面，因为 PGF 要参照箭头的形状、尺寸来决定裁截多长的一段路径。正常箭头的 bend 选项也不被支持。

/pgf/animation/shorten >=⟨*dimension*⟩ (no default)

/pgf/animation/shorten <=⟨*dimension*⟩ (no default)

116.5 Animating Transformations and Views

下面所说的属性所导致的变换是“画布变换”，它们的变换效果总是累计的。有的变换属性，如属性 translate，motion 所规定的动画都被计入整个图形的边界盒子内，也就是说图形的边界盒子会把这两个属性所创建的动画包含在内。有的变换属性，如属性 scale，rotate，skew 都对边界盒子的计算没有影响。

\pgfanimateattribute{scale}{⟨*options*⟩}

这个属性的值一个数字，或者是用逗号分隔的两个数字，如”0.5,1.5”.

\pgfanimateattribute{rotate}{⟨*options*⟩}

\pgfanimateattribute{xskew}{⟨*options*⟩}

xskew 可以写成 skew x.

\pgfanimateattribute{yskew}{⟨*options*⟩}

yskew 可以写成 skew y.

\pgfanimateattribute{skew x}{⟨*options*⟩}

\pgfanimateattribute{skew y}{⟨*options*⟩}

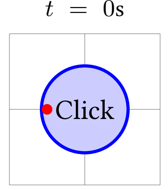
\pgfanimateattribute{translate}{⟨*options*⟩}

这个属性所做的是“平移”（shift），它的值是 PGF 的点。这个属性所做的变换影响边界盒子的计算，此属性规定的动画完全包含在图形的边界盒子中。

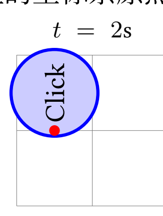
/pgf/animation/origin=⟨*PGF point*⟩ (no default)

参考 /tikz/animate/options/origin→P. 302，这里的 ⟨*PGF point*⟩ 是动画对象的本地坐标系中的点，这个选项把变换属性的坐标系原点平移到 ⟨*PGF point*⟩.

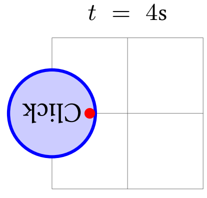
t = 0s



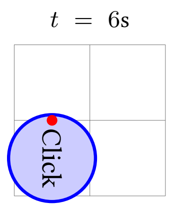
t = 2s



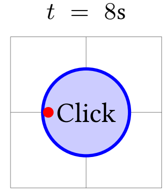
t = 4s



t = 6s



t = 8s



```
\tikz [very thick] {
  \draw [help lines](0,0) grid (2,2);
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click},
    origin = \pgfpoint{-5mm}{0mm},
    entry = {0s}{0}, entry = {2s}{90}, entry = {4s}{180}, entry = {6s}{270} }
  \node (node) at(1,1) [fill = blue!20, draw = blue, circle] {Click};
  \fill [red] (0.5,1)circle(2pt);
}
```

`\pgfanimateattribute{motion}{⟨options⟩}`

这个属性配合选项 `/pgf/animation/along` 使用，使得动画对象沿着某个路径移动。

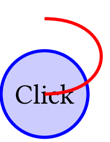
`/pgf/animation/along=⟨path⟩` (no default)

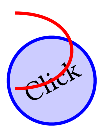
这个选项配合属性 `motion` 使用，使得动画对象沿着路径 ⟨*path*⟩ 移动；此时的属性值应该是 0 到 1 之间的数字；数字 0 代表 ⟨*path*⟩ 的起点，数字 1 代表 ⟨*path*⟩ 的终点。

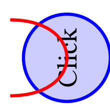
`/pgf/animation/rotate along=⟨boolean⟩` (default `true`)

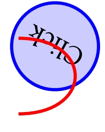
这个选项配合属性 `motion` 以及选项 `along` 使用，它使得动画对象在沿着路径移动时也被旋转，即使得动画对象自己的坐标系（本地坐标系）的 x 轴方向指向路径的切线方向。

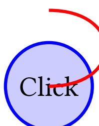
$t = 0$s $t = 0.5$s $t = 1$s $t = 1.5$s $t = 2$s












```
\tikz [very thick] {
  \pgfanimateattribute{motion}{
    whom = node, begin on = {click},
    rotate along = true,
    along = \pgfpathmoveto {\pgfpointorigin}
            \pgfpathcurveto{\pgfpoint{1cm}{0cm}}{\pgfpoint{1cm}{1cm}}{\pgfpoint
            ↪  {0cm}{1cm}},
    entry = {0s}{0}, entry = {2s}{1} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click};
  \draw [red] (0,0) .. controls (1,0) and (1,1) .. (0,1);
}
```

`\pgfanimateattribute{view}{⟨options⟩}`

参考 views 库。

116.6 Commands for Specifying Timing: Beginnings and Endings

`/pgf/animation/begin=⟨time⟩` (no default)

参考 `/tikz/animate/options/begin` → P. 305.

/pgf/animation/end=⟨*time*⟩ (no default)

参考 /tikz/animate/options/end→P. 305.

/pgf/animation/freeze at end=⟨*true or false*⟩ (default true, initially false)

参考 /tikz/animate/options/forever→P. 305.

/pgf/animation/begin on=⟨*option*⟩ (no default)

参考 /tikz/animate/options/begin on→P. 305.

/pgf/animation/restart=⟨*choice*⟩ (default true)

参考 /tikz/animate/options/restart→P. 307.

/pgf/animation/end on=⟨*option*⟩ (no default)

参考 /tikz/animate/options/end on→P. 307.

116.7 Commands for Specifying Timing: Repeats

/pgf/animation/repeats=⟨*specification*⟩ (no default)

参考 /tikz/animate/options/repeats→P. 307.

/pgf/animation/repeat=⟨*specification*⟩ (no default)

参考 /tikz/animate/options/repeats→P. 307.

117 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，见 §101.7.

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。在文件《pgfcorepoints.code》中对 \pgfpoint，\pgfpointadd 和 \pgfpointlineattime 的定义分别是：

```
\def\pgfpoint#1#2{%
  \pgfmathsetlength\pgf@x{#1}%
  \pgfmathsetlength\pgf@y{#2}\ignorespaces}

\def\pgfpointadd#1#2{%
  \pgf@process{#1}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{#2}%
  \advance\pgf@x by\pgf@xa%
  \advance\pgf@y by\pgf@ya}

\def\pgfpointlineattime#1#2#3{%
  \pgf@process{#3}%
  \pgf@xa\pgf@x%
  \pgf@ya\pgf@y%
  \pgf@xc\pgf@x%
  \pgf@yc\pgf@y%
  \pgf@process{#2}%
  \pgf@xb\pgf@x%
  \pgf@yb\pgf@y%
  \pgfmathsetmacro\pgf@temp{#1}%
  \advance\pgf@xa by-\pgf@x%
  \advance\pgf@ya by-\pgf@y%
  \advance\pgf@x by\pgf@temp\pgf@xa%
  \advance\pgf@y by\pgf@temp\pgf@ya%
}
```

在 \pgfpointlineattime 的定义中用到了 \pgf@xb 与 \pgf@yb，分析下面的代码是否能够计算 (1pt,2pt)+(3pt,4pt)=(4pt,6pt)：

\pgf@x=13.0pt
\pgf@y=14.0pt
\pgf@xa=2.0pt
\pgf@ya=2.0pt
\pgf@xb=11.0pt
\pgf@yb=12.0pt

```
\makeatletter
\pgfmathsetlength{\pgf@xa}{1pt}
\pgfmathsetlength{\pgf@xb}{2pt}
\pgfmathsetlength{\pgf@ya}{3pt}
\pgfmathsetlength{\pgf@yb}{4pt}
\pgfpointlineattime{0.5}{\pgfpoint{11pt}{12pt}}
  {\pgfpoint{13pt}{14pt}}
\pgfpointadd{\pgfpoint{\pgf@xa}{\pgf@ya}}
  {\pgfpoint{\pgf@xb}{\pgf@yb}}
\string\pgf@x=\the\pgf@x\\ \string\pgf@y=\the\pgf@y\\
\string\pgf@xa=\the\pgf@xa\\ \string\pgf@ya=\the\pgf@ya\\
\string\pgf@xb=\the\pgf@xb\\ \string\pgf@yb=\the\pgf@yb
\makeatother
```

其中一开始设置了 \pgf@xb 与 \pgf@yb 的值，但是随后这两个值又被命令 \pgfpointlineattime 改造了，所以最后命令 \pgfpointadd 计算的并不是

(1pt,2pt)+(3pt,4pt)=(4pt,6pt)

而是

(2pt,2pt)+(11pt,12pt)=(13pt,14pt)

可见在使用内部的临时寄存器做计算时应当谨慎一些，因为内部命令会使用这些寄存器，可能改变它们的值。

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 分组内。

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 分组内使用的临时盒子。

118 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。

118.1 快速坐标命令

参考文件《pgfcorepoints.code.tex》.

\pgfqpoint{⟨*x*⟩}{⟨*y*⟩}

参考 \pgfqpoint→P. 639. 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.

\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。

\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。

118.2 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:

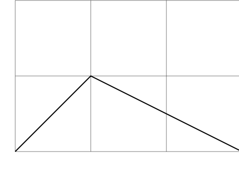
- 不跟踪边界盒子。
- 不能使用圆角。
- 不接受坐标变换。

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}}
```



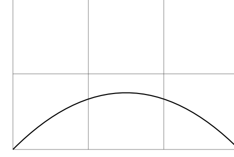
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformxshift{1cm}
  \pgfpathqmoveto{0pt}{0pt} % not transformed
  \pgfpathqlineto{1cm}{1cm} % not transformed
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfpathqlineto{⟨*x dimension*⟩}{⟨*y dimension*⟩}

类似 \pgfpathlineto.

\pgfpathqcurveto{⟨s_x^1⟩}{⟨s_y^1⟩}{⟨s_x^2⟩}{⟨s_y^2⟩}{⟨t_x⟩}{⟨t_y⟩}

类似 \pgfpathcurveto.



```
\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。

118.3 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.
- 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。

`\pgfusepathqfill`

只填充路径，没有其它动作。

`\pgfusepathqfillstroke`

只填充、画出路径，没有其它动作。

`\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。

118.4 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. 819.

`\pgfqboxsynced{`⟨*box number*⟩`}`

```
\def\pgfqboxsynced#1{%
  \pgfsys@hboxsynced#1%
}
```

参考 `\pgfsys@hboxsynced`→P. 820.

119 系统层的设计

119.1 驱动文件

PGF 的系统层主要由一些命令构成，这些命令都以 `\pgfsys@` 开头，称之为系统命令（system commands）。高层（指的是基本层和顶层）的命令会调用系统命令，并且高层的命令不能直接使用命令 `\special`，也不能检查命令 `\pdfoutput` 是否已经定义。命令 `\special{`⟨*something*⟩`}` 把 ⟨*something*⟩ 写入 `.dvi` 文件，由 `dvi` 驱动程序对其作处理。命令 `\pdfoutput=`⟨*integer*⟩ 的值 ⟨*integer*⟩ 是整数，若 ⟨*integer*⟩ 是正整数则编译输出 PDF 文件，否则输出 DVI 文件。所有绘图请求必须经由系统命令来处理。

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.tex》提供系统命令的“default implementations”, 不过多数系统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。

加载文件 pgfsys.sty 后，再加载文件 pgf.cfg. 文件 pgf.cfg 中定义了宏 `\pgfsysdriver`，这个宏确定使用哪个驱动程序。如果使用的驱动程序名称是 ⟨*drivername*⟩, 那么 pgfsys.sty 还会加载文件 pgfsys-⟨*drivername*⟩.sty，这个文件称为驱动文件（driver files）。

`\pgfsysdriver`

这个宏的展开值是所使用的驱动程序的名称，它的默认值是 `pgfsys-\Gin@driver`，对于 LaTeX 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`.

File `pgf.cfg`

本文件需要恰当地设置 `\pgfsysdriver` 的值。目前支持的驱动见 §10.2.

119.2 公共的定义文件

有的系统命令被数个驱动所公用，这些命令写入单独的文件中。

File `pgfsys-common-postscript`

这个文件定义 `\pgfsys@` 命令, 使之可以生成 PostScript 代码。

File `pgfsys-common-pdf`

这个文件定义 `\pgfsys@` 命令, 使之可以生成 PDF 代码。

120 系统命令

120.1 系统命令流的开启与结束

高层的命令（例如 `\draw`）会调用一系列的系统命令来实现其操作，这一系列的系统命令构成一个“系统命令流”（a Stream of System Commands）。每个系统命令流由 `\pgfsys@beginpicture` 开始，由相对应的 `\pgfsys@endpicture` 结束。

`\pgfsys@beginpicture`

此命令在 `{pgfpicture}` 环境的开端被处调，它用来设置某些东西。多数驱动会重新完善这个命令。此命令的初始定义是

```
\def\pgfsys@beginpicture{}
```

本命令可能会被重定义。

\pgfsys@endpicture

此命令在 {pgfpicture} 环境的结尾被处调。多数驱动会重新完善这个命令。此命令的初始定义是

```
\def\pgfsys@endpicture{}
```

本命令可能会被重定义。

\pgfsys@typesetpicturebox{⟨*box*⟩}

当 {pgfpicture} 环境的图形被排版后调用这个命令，将图形放入盒子 ⟨*box*⟩ 中，再将盒子 ⟨*box*⟩ 插入文档中。盒子 ⟨*box*⟩ 是"原始"盒子，其中只包含数个 \special 命令（用于描绘图形）。这个命令会参照图形的基线、尺寸选项来调整盒子 ⟨*box*⟩ 的尺寸和平移位置。

这个命令有默认的完善形式，不必用其它驱动文件来完善。

此命令的定义是：

```
\def\pgfsys@typesetpicturebox#1{%
  \pgf@ya=\pgf@shift@baseline\relax%
  \advance\pgf@ya by-\pgf@picminy\relax%
  %
  %
  \advance\pgf@picmaxy by-\pgf@picminy\relax% maxy is now the height
  \advance\pgf@picmaxx by-\pgf@picminx\relax% maxx is now the width
  \setbox#1=\hbox{\hskip-\pgf@picminx\lower\pgf@picminy\box#1}%
  \ht#1=\pgf@picmaxy%
  \wd#1=\pgf@picmaxx%
  \dp#1=0pt%
  \leavevmode%
  \pgf@xa=\pgf@trimleft@final\relax  \ifdim\pgf@xa=0pt \else\kern\pgf@xa\fi
  \raise-\pgf@ya\box#1%
  \pgf@xa=\pgf@trimright@final\relax \ifdim\pgf@xa=0pt \else\kern\pgf@xa\fi
}
```

在文件《pgfcorescopes.code.tex》中，命令 \pgf@shift@baseline 是 \endpgfpicture 的子命令

```
\pgf@process{\pgf@baseline}%
\xdef\pgf@shift@baseline{\the\pgf@y}%
```

\pgfsys@beginpurepicture

这是 \pgfsys@beginpicture 的"另版"，当图形中不包含 escaped boxes 时可以使用这个命令。

这个命令有默认的完善形式，不必用其它驱动文件来完善。

\pgfsys@endpurepicture

这是 \pgfsys@endpicture 的"另版"。这个命令有默认的完善形式，不必用其它驱动文件来完善。

\pgfsys@hbox{⟨*box number*⟩}

这个命令可以在 {pgfpicture} 环境中插入一个通常的 TEX 的水平盒子。多数驱动会重新完善这个命令。盒子内部处于通常的 TEX 模式，就是前面说的 escaped boxes.

```
\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%
}
% Called to insert a TeX hbox into a pgfpicture.
```

`\pgfsys@hboxsynced{⟨box number⟩}`

```
\def\pgfsys@hboxsynced#1{%
  \pgfsys@beginscope\pgflowlevelsynccm\pgfsys@hbox#1\pgfsys@endscope%
}%
```

`\pgfsys@pictureboxsynced{⟨box number⟩}`

```
\def\pgfsys@pictureboxsynced#1{%
  {%
    \setbox0=\hbox{\pgfsys@beginpicture\box#1\pgfsys@endpicture}%
    \pgfsys@hboxsynced0%
  }%
}
```

120.2 构建子环境的系统命令

`\pgfsys@beginscope`

将当前的图形状态参数保存到 graphic state stack 中，这些参数只对当前图形状态有效。在命令 `\pgfsys@endscope` 之后，再把 `\pgfsys@beginscope` 之前的旧的图形状态参数调出。

注意，PDF 与 PostScript 在判断当前路径是否属于图形状态 (graphic state) 时会有所不同。所以，若当前路径非空，则最好不要使用此命令。

`\pgfsys@endscope`

120.3 构建路径的系统命令

`\pgfsys@moveto{⟨x⟩}{⟨y⟩}`

本命令在点 (x, y) 处开启一个路径，或者把当前路径的当前点移动到点 (x, y) 处（移动时不画线，使得路径间断）。这里的 ⟨x⟩, ⟨y⟩ 都是 TeX 尺寸，把尺寸转换到坐标系统的工作则由驱动完成。命令 `\pgf@sys@bp` 会协助这个转换。

```
\pgfsys@moveto{10pt}{10pt}
\pgfsys@lineto{0pt}{0pt}
\pgfsys@stroke
```

\pgfsys@lineto{⟨x⟩}{⟨y⟩}

用直线段将当前路径延伸到点 (x, y) 处。

\pgfsys@curveto{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩}{⟨x_3⟩}{⟨y_3⟩}

以当前路径的当前点为起点，以 (⟨x_3⟩,⟨y_3⟩) 为终点，以 (⟨x_1⟩,⟨y_1⟩), (⟨x_2⟩,⟨y_2⟩) 为控制点构造 Bézier 曲线。

例如构造 $\frac{1}{4}$ 圆：

```
\pgfsys@moveto{10pt}{0pt}
\pgfsys@curveto{10pt}{5.55pt}{5.55pt}{10pt}{0pt}{10pt}
\pgfsys@stroke
```

\pgfsys@rect{⟨x⟩}{⟨y⟩}{⟨width⟩}{⟨height⟩}

以点 (⟨x⟩,⟨y⟩) 为左下角点，以 ⟨width⟩, ⟨height⟩ 为宽度和高度构造矩形。用命令 \pgfsys@moveto 和 \pgfsys@lineto 也可以构造矩形，但是这个命令利用的是 PDF 语言中专门的、速度更快的构造矩形命令。

\pgfsys@closepath

封闭当前路径，即把当前点与之前最近的 \pgfsys@moveto 点用直线段连接起来，连接处不会有缺口。

```
\pgfsys@moveto{0pt}{0pt}
\pgfsys@lineto{10bp}{10bp}
\pgfsys@lineto{0bp}{10bp}
\pgfsys@closepath
\pgfsys@stroke
```

120.4 做画布变换的系统命令

\pgfsys@transformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{⟨e⟩}{⟨f⟩}

这个命令设置一个画布变换矩阵，参考《PDF Reference》(6 edition, v 1.7) 的 §4.2.3. 假设对点 $(x, y, 1)$ 做变换得到点 $(x', y', 1)$，则变换为：

$$(x', y', 1) = (x, y, 1) \cdot \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix}$$

其中的 a, b, c, d 是不带长度单位的数值，e, f 是带长度单位的尺寸。例如：

```
\pgfsys@transformcm{1}{0}{0}{1}{1cm}{1cm}
```

`\pgfsys@transformshift`{⟨*x displacement*⟩}{⟨*y displacement*⟩}

这个命令将点 (x, y) 变成画布原点。

这个命令有默认的完善形式，不必用其它驱动文件来完善。

`\pgfsys@transformxyscale`{⟨*x scale*⟩}{⟨*y scale*⟩}

这个命令对画布做放缩变换，⟨*x scale*⟩ 是 x 轴方向的放缩因子，⟨*y scale*⟩ 是 y 轴方向的放缩因子。

这个命令有默认的完善形式，不必用其它驱动文件来完善。

`\pgfsys@viewboxmeet`{$\langle x_1\rangle$}{$\langle y_1\rangle$}{$\langle x_2\rangle$}{$\langle y_2\rangle$}{$\langle x_1'\rangle$}{$\langle y_1'\rangle$}{$\langle x_2'\rangle$}{$\langle y_2'\rangle$}

本命令开启一个“view box”scope, 这个 scope 必须用 `\pgfsys@endviewbox` 结束。本命令的作用是，以 ($\langle x_1\rangle$,$\langle y_1\rangle$) 左下角点，以 ($\langle x_2\rangle$,$\langle y_2\rangle$) 为右上角点确定矩形 R; 以 ($\langle x_1'\rangle$,$\langle y_1'\rangle$) 左下角点，以 ($\langle x_2'\rangle$,$\langle y_2'\rangle$) 为右上角点确定矩形 R'; 对 R' 施加画布变换（放缩、平移），使得 R' 的中心点与 R 重合，并且 R' 的最大尺寸点恰好接触 R 的边界，也就是说，R 恰好把 R' 包含在内（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.

`\pgfsys@viewboxslice`{$\langle x_1\rangle$}{$\langle y_1\rangle$}{$\langle x_2\rangle$}{$\langle y_2\rangle$}{$\langle x_1'\rangle$}{$\langle y_1'\rangle$}{$\langle x_2'\rangle$}{$\langle y_2'\rangle$}

类似上一命令，不过本命令使得 R' 的最小尺寸点恰好接触 R 的边界，也就是 R' 恰好把 R 包含在内。

`\pgfsys@endviewbox`

结束由 `\pgfsys@viewboxmeet` 开启的 viewbox.

120.5 画、填充、剪切路径的系统命令

`\pgfsys@stroke`

这个命令画出当前路径。图形状态参数会影响所画出的路径的外观，例如：

Line width 线宽，线宽为 0 的线条是能够画出的最细的线条，在高分辨率的打印器上可能很难看清楚线宽为 0 的线条。线宽的默认值是 0.4pt.

Stroke color 线条颜色（画笔颜色），默认为当前颜色。

Cap 线冠，即线条端点处的“帽子”，有三种类型：“圆形”(round cap)，“光头”(butt cap，无“帽子”，线条端点恰好位于指定的坐标点上)，“方形”（rectangular cap）。“rectangular cap”与“butt cap”的外观一样，不过“光头没帽子”。“round cap”和“rectangular cap”都会使得线条最末端的像素点超出路径端点，超出的尺寸是半个线宽。参考《PDF Reference》(6 edition, v 1.7) 的 §4.3.2.

Join 线结合，线条可能会有拐角，例如多边形的顶点处是个拐角，拐角的外缘有三种类型：round join, bevel join, miter join, 参考《PDF Reference》(6 edition, v 1.7) 的 §4.3.2.

Dash 线条的线型，例如虚线。

Clipping area 剪切范围，只有处于剪切范围之内的路径会被画出。

`\pgfsys@closestroke`

本命令先封闭路径再画出路径。

这个命令有默认的完善形式，不必用其它驱动文件来完善。

`\pgfsys@fill`

本命令用某种颜色填充当前路径所封闭起来的内部区域，如果当前路径不是封闭的，则先封闭当前路径然后填充路径的内部区域。如果路径有自交现象导致路径的内部区域不是道路连通的，则使用非零规则（nonzero winding number rule）来判断路径的内部区域和外部区域，否则使用奇偶规则（even-odd rule）。参考《PDF Reference》(6 edition, v 1.7) 的 §4.4.2.

下面的图形状态参数影响填充命令：

Interior rule 判断路径的内部区域和外部区域的规则，如果设置了 `\ifpgfsys@eorule`, 则使用奇偶规则，否则使用非零规则。

Fill color 填充色，默认为当前颜色。

Clipping area 如果提前设置了剪切区域，则只有剪切区域的子区域才会被填充。

`\pgfsys@fillstroke`

本命令先填充路径，然后画出路径。填充色和线条颜色可以分别设置。

`\pgfsys@discardpath`

本命令“丢弃”当前路径。

`\pgfsys@clipnext`

本命令的位置是：构建路径之后，画出路径、或者填充路径、或者丢弃路径的命令之前。本命令之后的画出路径、或者填充路径、或者丢弃路径的命令仍然会被照常执行，但是本命令之前的路径会被用作剪切路径，对之后的路径具有剪切作用。当多次使用本命令设置多个剪切路径后，剪切区域是（直到当前的）各个剪切路径的“交集”，默认使用非零规则来判断这个“交集”，除非设置 `\ifpgfsys@eorule`.

120.6 设置图形状态选项的系统命令

`\pgfsys@setlinewidth`{⟨*width*⟩}

本命令设置路径线条的线宽为 ⟨*width*⟩, 它必须是 TEX 尺寸。

`\pgfsys@buttcap`

本命令设置为 butt cap.

`\pgfsys@roundcap`

本命令将线冠设置为 round cap.

`\pgfsys@rectcap`

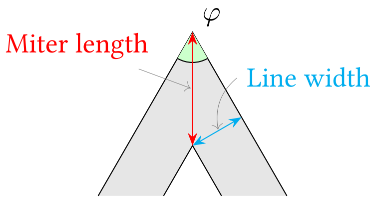
本命令将线冠设置为 rectangular cap.

`\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 ⟨*factor*⟩ $< \frac{\text{Miter length}}{\text{Line width}} = \frac{1}{\sin(\frac{\varphi}{2})}$，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。如果 ⟨*pattern*⟩ 中的尺寸个数是奇数个，例如

```
\pgfsys@setdash{4pt,3pt,2pt}{1pt}
```

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，那么就代表实线。
⟨*phase*⟩ 是"相位"。

\ifpgfsys@eorule

这个命令决定使用奇偶规则，否则使用非零规则。

120.7 设置颜色的系统命令

颜色表达式采用 xcolor 宏包的格式。PDF 语言以很直接的方式处理路径线条颜色和填充色。

\pgfsys@color@rgb{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

本命令为之后的 stroke 或 fill 操作设置颜色，颜色使用 rgb 模式，参数 ⟨*red*⟩, ⟨*green*⟩, ⟨*blue*⟩ 都是区间 $[0, 1]$ 内的数值。

\pgfsys@color@rgb@stroke{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

本命令为之后的 stroke 操作设置颜色，颜色使用 rgb 模式，参数 ⟨*red*⟩, ⟨*green*⟩, ⟨*blue*⟩ 都是区间 $[0, 1]$ 内的数值。

`\pgfsys@color@rgb@fill`{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

本命令为之后的 fill 操作设置颜色，颜色使用 rgb 模式，参数 ⟨*red*⟩, ⟨*green*⟩, ⟨*blue*⟩ 都是区间 [0, 1] 内的数值。

`\pgfsys@color@cmyk`{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

本命令为之后的 stroke 或 fill 操作设置颜色，颜色使用 cmyk 模式，参数 ⟨*cyan*⟩, ⟨*magenta*⟩, ⟨*yellow*⟩, ⟨*black*⟩ 都是区间 [0, 1] 内的数值。

`\pgfsys@color@cmyk@stroke`{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

本命令为之后的 stroke 操作设置颜色，颜色使用 cmyk 模式，参数 ⟨*cyan*⟩, ⟨*magenta*⟩, ⟨*yellow*⟩, ⟨*black*⟩ 都是区间 [0, 1] 内的数值。

`\pgfsys@color@cmyk@fill`{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

本命令为之后的 fill 操作设置颜色，颜色使用 cmyk 模式，参数 ⟨*cyan*⟩, ⟨*magenta*⟩, ⟨*yellow*⟩, ⟨*black*⟩ 都是区间 [0, 1] 内的数值。

`\pgfsys@color@cmy`{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

本命令为之后的 stroke 或 fill 操作设置颜色，颜色使用 cmy 模式，参数 ⟨*cyan*⟩, ⟨*magenta*⟩, ⟨*yellow*⟩ 都是区间 [0, 1] 内的数值。

`\pgfsys@color@cmy@stroke`{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

本命令为之后的 stroke 操作设置颜色，颜色使用 cmy 模式，参数 ⟨*cyan*⟩, ⟨*magenta*⟩, ⟨*yellow*⟩ 都是区间 [0, 1] 内的数值。

`\pgfsys@color@cmy@fill`{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

本命令为之后的 fill 操作设置颜色，颜色使用 cmy 模式，参数 ⟨*cyan*⟩, ⟨*magenta*⟩, ⟨*yellow*⟩ 都是区间 [0, 1] 内的数值。

`\pgfsys@color@gray`{⟨*black*⟩}

本命令为之后的 stroke 或 fill 操作设置颜色，颜色使用 gray 模式，参数 ⟨*black*⟩ 是区间 [0, 1] 内的数值，参数 0 代表黑色，参数 1 代表白色。

`\pgfsys@color@gray@stroke`{⟨*black*⟩}

本命令为之后的 stroke 操作设置颜色，颜色使用 gray 模式，参数 ⟨*black*⟩ 是区间 [0, 1] 内的数值，参数 0 代表黑色，参数 1 代表白色。

`\pgfsys@color@gray@fill`{⟨*black*⟩}

本命令为之后的 fill 操作设置颜色，颜色使用 gray 模式，参数 ⟨*black*⟩ 是区间 [0, 1] 内的数值，参数 0 代表黑色，参数 1 代表白色。

`\pgfsys@color@reset`

在使用命令 `\color` 时，本命令会被调用。本命令会清除之前关于 stroke 或 fill 操作的颜色设置，并把命令 `\color` 设置的颜色作为 stroke 或 fill 操作的颜色。命令 `\color` 设置的颜色会在遇到像 `\pgfsys@color@rgb@fill` 这样的颜色设置命令时失效。

`\pgfsys@color@reset@inordertrue`

这是 `\ifpgfsys@color@reset@inorder` 的 true 值，这个真值是默认有效的。

`\pgfsys@color@reset@inorderfalse`

这是 `\ifpgfsys@color@reset@inorder` 的 false 值。

`\pgfsys@color@unstacked{`⟨*LaTeX color*⟩`}`

120.8 关于图样的系统命令

120.9 插入外部图形的系统命令

120.10 关于颜色渐变的系统命令

120.11 关于透明度的系统命令

120.12 关于动画的系统命令

120.13 关于 Object Identification 的系统命令

可以为图形中的多种对象添加 id, 当某个对象具有 id 后，可以把它的 id 用于制作超链接 (hyperlinking).

下面的图形对象可以具有 id:

1. Graphic scopes, 当调用 `\pgfsys@begin@idscope`→P. 831 时。
2. view boxes, 当调用 `\pgfsys@viewboxmeet`→P. 822 或 `\pgfsys@viewboxslice`→P. 822 时。
3. paths, 当调用 `\pgfsys@stroke`→P. 822 等命令时。
4. text boxes, 当调用 `\pgfsys@hbox`→P. 819 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，在某个对象之前使用命令 `\pgfsys@use@id` 将这个 id 赋予该对象。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，其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。一个对象只能有一个 id 标签，对象的一个组成部分只能有一个 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。

\pgfsys@new@id{⟨*macro*⟩}

本命令创建一个新的 id，以备后用。保存在 ⟨*macro*⟩ 中的文本由驱动创建，只供内部使用，用户最好不要修改它。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⟨值 *\the\pgf@sys@id@count*⟩.
2. 将计数器 pgf@sys@id@count 的值（全局地）加 1.

pgf13775, 13776

```
\makeatletter
\pgfsys@new@id\aaa
\aaa, \the\pgf@sys@id@count
\makeatother
```

\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(此命令的初始状态等于 \pgfutil@empty)。
2. 令 \pgfsys@current@type 等于 \pgfutil@empty.

\pgfsys@clear@id

```
\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。此命令的定义是：

```
\def\pgfsys@register@type#1{%
  \expandafter\let\expandafter\pgf@sys@temp\csname pgf@sys@reg@type@#1\endcsname
  ↪   %
  \ifx\pgf@sys@temp\relax%
    {%
```

```
      \c@pgf@counta\pgf@sys@type@count\relax%
      \global\advance\c@pgf@counta by1\relax%
      \edef\pgf@sys@type@count{\the\c@pgf@counta}%
      \expandafter\xdef\csname pgf@sys@reg@type@#1\endcsname{y\the\c@pgf@counta}
       ↪ %
    }%
  \fi%
}
% Registers a type with the system. Must be called before any use of
% the type
\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}
\def\pgf@sys@type@count{0}
```

执行 \pgfsys@register@type⟨*type*⟩, 即

1. 让 \pgf@sys@temp 等于 \csname pgf@sys@reg@type@⟨*type*⟩\endcsname.
2. 用 \ifx 检查命令 \csname pgf@sys@reg@type@⟨*type*⟩\endcsname 是否已定义:
   - 如果未定义，则开启一个花括号分组，在这个组内执行:
     (a) 把 \pgf@sys@type@count 的值赋予寄存器 \c@pgf@counta.
     (b) 全局地把寄存器 \c@pgf@counta 的值加 1.
     (c) 把寄存器 \c@pgf@counta 的值保存在宏 \pgf@sys@type@count 中。
     (d) 全局地定义命令 \csname pgf@sys@reg@type@⟨*type*⟩\endcsname 的值为

     y⟨ 值 \the\c@pgf@counta⟩
   - 如果已定义，则结束当前条件句。

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。上面代码列出了预定义的 type, 即 background, path, text, background.path.

\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.

\pgfsys@append@type{⟨*text*⟩}

此命令的定义是:

```
\def\pgfsys@append@type#1{%
  \ifx\pgfsys@current@type\pgfutil@empty%
    \pgfsys@use@type{#1}%
  \else%
    \pgfsys@use@type{\pgfsys@current@type.#1}%
  \fi%
}
```

执行 `\pgfsys@append@type{⟨text⟩}` 时会检查 `\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⟩}`.
- 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⟩`”.

`\pgfsys@push@type`

将当前的 type 放入一个全局的 stack of types 中，并且不开启 TeX scope.

```
\def\pgfsys@push@type{%
  \expandafter\expandafter\expandafter\def\expandafter\expandafter\expandafter
  ↪   \pgf@sys@typestack%
  \expandafter\expandafter\expandafter{\expandafter\expandafter\expandafter\def
  ↪   \expandafter\expandafter\expandafter\pgfsys@current@type%
    \expandafter\expandafter\expandafter{\expandafter\pgfsys@current@type
    ↪   \expandafter}%
    \expandafter\def\expandafter\pgf@sys@typestack\expandafter{
    ↪   \pgf@sys@typestack}}%
}
\let\pgf@sys@typestack\pgfutil@empty
```

执行 `\pgfsys@push@type` 的过程就是:

1.
```
\expandafter\def
\expandafter\pgf@sys@typestack%
\expandafter{
  \expandafter\def
  \expandafter\pgfsys@current@type%
  \expandafter{\pgfsys@current@type}%
  \def\pgf@sys@typestack{展开之前的 \pgf@sys@typestack}
}%
```

2.
```
\def\pgf@sys@typestack{
  \def\pgfsys@current@type{展开的 \pgfsys@current@type}%
  \def\pgf@sys@typestack{展开之前的 \pgf@sys@typestack}
}%
```

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

调出全局的 stack of types 中顶端的 type.

```
\def\pgfsys@pop@type{\pgf@sys@typestack}
% Pops the last id from the stack.
```

执行 `\pgfsys@pop@type` 得到:

```
\def\pgfsys@current@type{展开的 \pgfsys@current@type}%
\def\pgf@sys@typestack{展开之前的 \pgf@sys@typestack}
```

得到 `\pgfsys@current@type` 的定义，也就是把 `\pgfsys@current@type` 调出。

\pgfsys@id@ref

此命令的定义是:

```
\def\pgfsys@id@ref#1#2{#1\csname pgf@sys@reg@type@#2\endcsname}
% Expands to a text that can be inserted as a reference. #1 must be a
% reference created \pgfsys@new@id, #2 must be a type that has been
% registered using \pgfsys@id@register@type.
```

按代码注释，`#1` 代表由 `\pgfsys@new@id` 定义的宏；`#2` 代表已经用 `\pgfsys@id@register@type` 注册过的 type 名称。

`\pgfsys@id@ref{`⟨*id macro*⟩`}{`⟨*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。

执行 `\pgfsys@id@refcurrent` 得到当前的 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, 所以在定义 \pgfsys@beg@save 前不能使用 \pgfsys@call@save.
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@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.

```
\def\pgfsys@begin@idscope{%
  \begingroup%
    \edef\pgf@sys@cacheref{\pgfsys@id@refcurrent}%
    \expandafter\let\expandafter\pgfsys@beg@save\csname pgf@sys@att@beg@
    ↪  \pgf@sys@cacheref\endcsname%
    \expandafter\let\expandafter\pgfsys@end@save\csname pgf@sys@att@end@
    ↪  \pgf@sys@cacheref\endcsname%
    \ifx\pgfsys@beg@save\relax%
      \ifx\pgfsys@end@save\relax%
      \else%
```

```
        \pgfsys@call@save%
      \fi%
    \else%
      \pgfsys@call@save%
    \fi%
    \pgfsys@invalidate@currentid%
      \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。

```
\def\pgfsys@attach@to@id#1#2#3#4{%
  \pgfsys@register@type{#2}%
  \expandafter\def\expandafter\pgf@sys@tempbeg\expandafter{\csname
  ↪  pgf@sys@att@beg@\pgfsys@id@ref{#1}{#2}\endcsname}%
  \expandafter\def\expandafter\pgf@sys@tempend\expandafter{\csname
  ↪  pgf@sys@att@end@\pgfsys@id@ref{#1}{#2}\endcsname}%
  \expandafter\ifx\pgf@sys@tempbeg\relax%
    \expandafter\let\pgf@sys@tempbeg\pgfutil@empty%
  \fi%
  \expandafter\ifx\pgf@sys@tempend\relax%
    \expandafter\let\pgf@sys@tempend\pgfutil@empty%
  \fi%
  \expandafter\let\expandafter\pgf@sys@tempbeg@cont\pgf@sys@tempbeg%
  \expandafter\let\expandafter\pgf@sys@tempend@cont\pgf@sys@tempend%
  \expandafter\expandafter\expandafter\gdef\expandafter\pgf@sys@tempbeg
  ↪  \expandafter{\pgf@sys@tempbeg@cont#3}%
  \def\pgf@sys@temp{#4}%
```

```
  \expandafter\expandafter\expandafter\expandafter\expandafter\expandafter\expandafter
  ↪  \expandafter\gdef\expandafter\expandafter\expandafter\pgf@sys@tempend
  ↪  \expandafter\expandafter\expandafter{\expandafter\pgf@sys@temp
  ↪  \pgf@sys@tempend@cont}%
}
% Attaches some code to an id-type pair so that when the id's scope
% gets created, #3 is added at the beginning and #4 is added at the
% end.
```

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。

本命令重定义 \csname pgf@sys@att@beg@\pgfsys@id@ref{⟨*id*⟩}{⟨*type*⟩}\endcsname，扩展其定义内容，使得 ⟨*begin code*⟩ 紧接其原来的定义内容之后。

本命令重定义 \csname pgf@sys@att@end@\pgfsys@id@ref{⟨*id*⟩}{⟨*type*⟩}\endcsname，扩展其定义内容，使得 ⟨*end code*⟩ 紧接其原来的定义内容之前。

120.14 可重复利用对象的系统命令

\pgfsys@invoke{⟨*literals*⟩}

\pgfsys@defobject{⟨*name*⟩}{⟨*lower left*⟩}{⟨*upper right*⟩}{⟨*code*⟩}

\pgfsys@useobject{⟨*name*⟩}{⟨*extra code*⟩}

\pgfsys@marker@declare{⟨*macro*⟩}{⟨*code*⟩}

\pgfsys@marker@use{⟨*macro*⟩}

120.15 使得路径“不可见”的系统命令

处于命令 \pgfsys@begininvisible 与命令 \pgfsys@endinvisible 之间的路径线条、文字、插入的外部图形、颜色渐变等内容都是不可见的，对它们的输出会被抑制。命令 \pgfsys@begininvisible 不会开启 TeX 分组或者限制图形状态的 scope.

命令 \pgfsys@begininvisible 与命令 \pgfsys@endinvisible 都有默认的完善形式，不需要驱动文件来完善它。

命令 \pgfsys@begininvisible 与命令 \pgfsys@endinvisible 创建一个不可见部分，二者可以套嵌使用。

\pgfsys@begininvisible

```
\def\pgfsys@begininvisible{\pgfsys@transformcm{1}{0}{0}{1}{2000bp}{2000bp}}
```

\pgfsys@endinvisible

```
\def\pgfsys@endinvisible{\pgfsys@transformcm{1}{0}{0}{1}{-2000bp}{-2000bp}}
```

\pgfsys@begininvisiblescope

```
\def\pgfsys@begininvisiblescope{\pgfsys@beginscope\pgfsys@begininvisible}
```

\pgfsys@endinvisiblescope

```
\def\pgfsys@endinvisiblescope{\pgfsys@endinvisible\pgfsys@endscope}
```

120.16 关于页面尺寸的系统命令

120.17 跟踪页面上某个位置的系统命令

页面上的某个位置指的是：某个文字在页面上的位置。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：在某个文字的位置处“做标记”（标记机制依赖后台驱动），第一次编译，当输出页面时（或其它更迟一些的时候），这个“标记”的位置坐标会被输出到 `.aux` 文件中；第二次编译，从 `.aux` 文件中读取位置坐标并利用之。

`\pgfsys@markposition{`⟨*name*⟩`}`

本命令用于标记页面上的某个位置，这个位置就是本命令所处的位置，并且把这个位置名称为 ⟨*name*⟩, 之后可以用 ⟨*name*⟩ 来引用这个位置。例如

```
The value of $x$ is \pgfsys@markposition{here}important.
```

这个代码把单词 `is` 之后，`important` 之前的位置标记为 `here`.

`\pgfsys@getposition{`⟨*name*⟩`}{`⟨*macro*⟩`}`

使用命令 `\pgfsys@markposition` 标记页面位置 ⟨*name*⟩ 后，再使用本命令将位置 ⟨*name*⟩ 保存到宏 ⟨*macro*⟩ 中。

如果页面位置 ⟨*name*⟩ 没有被标记，或者在第一次编译 tex 文件时（此时位置坐标尚不可用），⟨*macro*⟩ 会被定义为 `\relax`. 如果页面位置 ⟨*name*⟩ 被成功标记，那么宏 ⟨*macro*⟩ 会被重定义为 `\relax`, 或者 `\pgfpoint...`. 例如 `\pgfpoint{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。页面的原点位置并不是固定不变的，但是对一个页面上的各个图形来说，页面原点位置是相同的。

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。

120.18 尺寸转换命令

PDF 或 PostScript 文件中的尺寸都用无单位的数值来代表，而尺寸的单位默认为 $\frac{1}{72}$in，即 bp. 所以在输出 PDF 文件时，各种 TeX 尺寸的单位需要转换为 bp. 例如 10pt=9.9626401bp.

`\pgf@sys@bp{`⟨*dimension*⟩`}`

这个命令把尺寸 ⟨*dimension*⟩ 转换为以 `bp` 为单位的尺寸。

```
\pgf@sys@bp{\pgf@x}
\pgf@sys@bp{1cm}
```

这个命令不会被驱动改写。

121 软路径子系统

本节介绍创建软路径（soft paths）的命令，这些命令与之前章节介绍的命令很不相同，之前章节介绍的命令创建硬路径（hard paths）。软路径在创建后是可以被改变的路径，而硬路径在创建后就不能再改变，只有硬路径会被嵌入到 .pdf，.ps 文件中。

本节介绍的命令不会被驱动文件“完善”，而是直接被 PGF 的系统层完善。创建软路径的命令都以 \pgfsyssoftpath@ 开头。一般情况下用户用不到创建软路径的命令。

121.1 创建路径的过程

程序对命令 \draw (0bp,0bp) -- (10bp,0bp); 的处理是个复杂的过程：

1. 这个命令是 TikZ 的前端层命令，首先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 并不会直接调用创建硬路径的命令 \pgfsys@xxxx, 而是调用创建软路径的命令，例如 \pgfpathmoveto 调用 \pgfsyssoftpath@moveto, 命令 \pgfpathlineto 调用 \pgfsyssoftpath@lineto.
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，这个保存软路径的内部宏也会被更新。
3. 当遇到命令 \pgfusepath 时，保存在内部宏中的软路径会被调用，软路径中的各种操作，如 line-to, move-to, 会调用硬路径命令 \pgfsys@moveto, \pgfsys@lineto, 最终创建硬路径，即成为 .pdf，.ps 文件中的文字符号。
4. 最后使用命令 \pgfsys@stroke 画出路径。

也就是说，先创建软路径，然后用软路径创建硬路径。

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{0}{0}
\pgfsys@setlinewidth{1}
\pgfsys@lineto{10}{0}
\pgfsys@stroke
```

其中的 \pgfsys@setlinewidth{⟨*1*⟩} 会导致 PDF 文件被“损坏”，有的 PDF 阅读器能容忍这种“损坏”，而有的阅读器则不能。

121.2 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`

`\pgfsyssoftpath@flushcurrentpath`

121.3 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⟩}`

`\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。

121.4 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。

122 Protocol 子系统

123 动画的系统层