graph-easy-cn

weishu

Published
with GitBook

目錄

简介

`Graph::Easy` 是一个处理图形DSL的Perl模块，它有如下功能：

- 提供了一种易懂，可读性很强的图形描述语言
- 一种支持 ASCII Art 的基于网格的布局器
- 可以导出为Graphviz, VCG(Visualizing Compiler Graphs), GDL(Graph Description LAnguages) 和 GraphML格式。
- 可以从 `Graphviz` , `VCG` 和 `GDL` 导入图像。

本文档旨在让你学会使用 `Graph::Easy` 语言来创建图像（或者流程图），然后将它转换为ASCII, HTML, SVG或者是可以导出为png或者pdf文件的 `Graphviz` 格式。

要了解本模块，请从概述开始。

下载和镜像

可以在CPAN下载 `Graph::Easy` 。

如果你想创建一个镜像，请联系作者. 本文档可以在下面这些网站找到：

- http://bloodgate.com/perl/graph/manual
- http://search.cpan.org/~tels/Graph-Easy-Manual/doc/manual/index.html

Demo和使用本模块的网站

Demo可以在这里看到（现在这个网站已经无法打开）；这个模块也在一些perl的graph的模块使用；另外，可以通过Mediawiki插件在wiki里面使用这个模块；你深知可以把这种图形添加到POD中。

下面的一些网站和项目使用了本模块：

- S23
- Bloodgate
- BOWiki

- Cosmogol

bug反馈

如果有问题，可以给作者发邮件; Bug反馈在rt.cpan.org

概要

输入和输出

下图是Graph::Easy的概要图，输入是绿色，直接输出是橘黄色，白色是组成节点。黄色的节点是使用第三方模块支持的格式输出。

有很多种方式创建 `Graph::Easy` 内部支持的数据结构：

- 使用交互式编辑器（没有实现）
- 使用 `Graph::Easy` 能理解的文本格式（graphviz, VCG, GDL, Graph::Easy）然后使用命令行工具 `graph-easy` 来解析并产生输出（这个工具会和模块一起安装）
- 直接写perl代码

下面是一点使用perl代码的例子：

```
use strict;
use Graph::Easy;

my $graph = Graph::Easy->new();

$graph->add_edge('Bonn', 'Berlin');
$graph->add_edge('Berlin', 'Bonn', 'train' );
```

相应的文本描述如下：

```
[ Berlin ] -- train --> [ Bonn ]
[ Bonn ] --> [ Berlin ]
```

如你所见，使用文本描述更加简洁。当然，使用文本然后解析还是使用perl完全取决于你。同样地，如果你有了 `Graph::Easy` 对象，你可以把它输出为任意你想要的格式。 另外，使用文本描述是完备的：你可以先解析文本，产生输出；然后可以把输出又转换为文本描述。

存储和布局

Graph::Easy的确使用了Graph模块来存储和管理内部的图形数据；但是从版本v0.25开始，它没有这么做了；它简单地把边和节点存储了Perl的Hash表，然后直接访问；为什么这么做请参见这里

要说明的是，Graph和Graph::Easy这两个模块都仅仅存储了图形的表示数据，没有特定的布局信息；比如下面的图（使用 `Graph::Easy` 语法表示的）

```
[ A ] -> [ C ] -> [ D ]
[ C ] -> [ E ]
```

可以有多种布局方式（有可能无穷多种），下面是两个例子：

```
+---+     +---+     +---+
| A | --> | C | --> | D |
+---+     +---+     +---+
            |
            |
            v
          +---+
          | E |
          +---+
```

布局

上面提到过，Graph::Easy仅仅存储了图像的描述信息，要产生特定的布局，需要借助其他的工具；以下是在Perl里面可以通过Graph产生布局的一些方法：

- Graph::Easy1
- dot(使用graphviz)
- Graph-Layderer
- Graph-Layout-Aesthetic

将来或许会有更多，但是作者开始写这个模块的时候，只有这些选择。和其他的方式不同 `Graph::Easy` 使用了一种checker-board tiled布局。

要指出的是，传统的把节点放置在任意位置的方式不能产生ASCII格式的输出，也不能产生HTML的输出；好吧，或许是有可能，但是如果边不是直的而是到处都是的话，那么非常难办。

以上也是这个项目存在的原因。:-P

特性

Graph::Easy 支持非常多的特性，下面是一些大致介绍。

Unicode

`Graph::Easy` 对Unicode输入和输出有着完整的支持：

> [العربية] -- link --> [日本語] --> [中文] -- كوردي --> [English]

上面的例子包含了日文，中文，库尔德语，和一些其他的字符；下面是使用Graph::Easy输出的HTML格式的输出（这个不是图片，放大以下可以看到文字也会改变大小:-):

PS: 这里就不引入这一段HTML了..又是CSS又是原始HTML，markdown支持不好。

如果你看到了小方块或者问号，那么你需要安装一些系统缺失的字体，或者使用完整支持Unicode的浏览器（Opera, Firefox)，以下是对以上HTML的一张截图，展示了在一个Unicode浏览器里面它的样子：

相同文本的节点

由于每一个节点都是唯一的，因此不可能让两个节点具有相同的名字；但是有时候需要在布局里面给两个节点展示相同的描述，可以使用 `label` 属性复写节点上的文本描述：

```
[ Bonn ] { label: Berlin; } -> [ Berlin ]
```

经渲染之后图像如下：

```
+--------+     +--------+
| Berlin | --> | Berlin |
+--------+     +--------+
```

匿名以及不可见节点，边

`Graph::Easy` 支持匿名节点，正常不可见节点（有一个最小的大小）以及真正的不可见节点(大小尽可能小），也支持不可见的边：

```
[ ] { title: Anonymous Node; }
  -> [ $sys$Node ] { shape: invisible; title: You don't see me! }
  -> [ Buna ]
  -> [ Borna ] { shape: point; point-style: invisible; }
  -> [ Bremen ]
  -> { style: invisible; } [ Bonn ]
  -> [ $sys$Node ]
```

效果如下：

```
          +-----------------------------------------+
          v                                         |
                  +------+          +--------+     +------+
    -->       --> | Buna | -->  --> | Bremen |     | Bonn |
                  +------+          +--------+     +------+
```

多个边

许多图形包也支持多个边；一个多边图就是允许从一个节点有多个边出发到达同一个另外的节点。

```
[ Rostock ] -> [ Wismut ]
[ Rostock ] -> [ Wismut ]
```

```
 +---------------+
  |               v
+---------+     +--------+
| Rostock | --> | Wismut |
+---------+     +--------+
```

自循环

对于状态机以及流程图来说，自循环非常有用；自循环指的就是一个边从一个节点出发然后回到这个节点本身。

```
[ Chemnitz ] -> [ Chemnitz ]
```

```
  +------+
  v      |
+----------+
| Chemnitz |
+----------+
```

边的一部分作为其他边的终点

有时候你想把一个边指向另外一个边上面的文字描述；但是传统的边只能连接节点，Graph::Easy的节点有一个特性 `shape` ，这种类型的节点会和边无缝连接在一起，让人觉得你好像是真正指向了一个边：

```
[ car ] { shape: edge; }

[ Bonn ] -- train --> [ Berlin ] -- [ car ] --> [ Ulm ]

[ rented ] --> [ car ]
```

```
+------+  train   +--------+      car           +-----+
| Bonn | -------> | Berlin | -----------------> | Ulm |
+------+          +--------+                    +-----+
                                    ^
                                    |
                                    |
                                  +--------+
                                  | rented |
                                  +--------+
```

无向边和双向边

无向边和双向边也经常被使用：

```
[ Hamm ] <--> [ Leverkusen ]
[ Wismut ] -- [ Plauen ]
```

```
+--------+      +------------+
|  Hamm  | <--> | Leverkusen |
+--------+      +------------+
+--------+      +------------+
| Wismut | ---- |   Plauen   |
+--------+      +------------+
```

分组（群或者子图）

子图（在Graph::Easy里面叫做组）允许你把一些节点放在一起：

```
( Capitals: [ Bonn ], [ Berlin ] )
```

```
+ - - - - - - +
' Capitals:   '
'             '
' +---------+ '
' | Berlin  | '
' +---------+ '
' +---------+ '
' |  Bonn   | '
' +---------+ '
'             '
+ - - - - - - +
```

使用嵌套的组也是可能的（没有实现，无法使用）；可以有从一个分组到一个节点的边，反之亦然；一个组到另外一个组的边也支持。

查阅Hintings获取更详细的信息和例子。

Joints(边分散和聚合)

可以把边分散或者聚合：

```
  [ Potsdam ], [ Mannheim ]
    --> { end: back,0; }
  [ Weimar ]
    --> { start: front,0; } [ Finsterwalde ], [ Aachen ]
```

```
 +----------+ +--------+ +--------------+ | Mannheim | ------+-> | Weimar | -+-----> | Finsterwalde | +----------+ | +--------+ | +----------+ | | | | | | +----------+ | | +--------------+ | Potsdam | ------+ +-----> | Aachen | +----------+ +--------------+
```

更多的内容可以参考Joints

术语

以下是在这个手册中用到的一些术语：

- node - 代表一个节点或者一个图的顶点
- edge - 连接两个节点的边（或者是连接节点自己）比如：[Bonn] -> [Berlin]
- group - 子图
- name - 一个节点，子图唯一的名字；边和图没有名字。
- label - 节点，子图或者边上面展示文本；对于节点来说，如果没有设置标签，那么会使用节点的名字。
- title - 当你在节点或者边上移动鼠标的时候，展示的文本。
- port - 一个节点上可以有边的起点和终点的点。
- cell - 布局面板上的单元（看起来像是checker-board)
- path - 连接两个节点的路径
- (edges) pieces - 每一个path可以由一个或者多个cell组成，每一个cell包含edges的一部分。
- Parser - 把文本描述解析成内部表示的解析器
- Layouter - 给一个图的边和节点布局的东东
- hinting - 给Layouter指示如何进行特定布局的提示
- A* - 一个通用的路径寻找算法

布局器

阅读本章请先读概要

Graph::Easy的布局器负责把内部图形表示转换成一个特定的布局；下面是对于同一种图形表示产生的两种不同的布局：

```
+---+     +---+     +---+
| A | --> | C | --> | D |
+---+     +---+     +---+
            |
            |
            v
          +---+
          | E |
          +---+
```

布局器的工作几乎都是自动完成的，这一章将会解释一些内部的实现细节。这会让你明白为什么某个图形的布局会是那样，另外，这回让你更好的指导布局器生成自己想要的布局。

要说明的是，使用函数 `as_txt()` (纯文本)， `as_graphml` (Graph ML) ` 和 as_graphviz 并不是由这个布局器生成的；比如对于 graphviz 来说，真正的布局通过一个外部程序生成，如 dot ， neato`等。

概要

要生成一个特定的布局，Graph::Easy分三步完成：

- 首先，对节点进行分类并通过分组信息和rank信息放置在不同的组里面
- 然后它就会创建一个节点链，从有最少输入边的节点开始，尽可能找到最长的节点链；
- 最后一步，布局器把这些节点放置在一个格子板上，就像一个棋盘；当一个节点和连接它的节点放置好之后，布局器尝试寻找连接节点的边。

前两步影响了第三步处理节点的顺序；要找到最长的节点链，布局器会打散节点链并尽可能放在一条直线上。

下面是两个例子，第一个是在pre-v0.24版本上生成的图，第二个是v0.25版本的输出结果，这两个图的不同很明显地说明了这种布局策略的作用：

```
 +------------------------------+
  v                              |
+---------+     +--------+     +--------+     +---------+
| Koblenz | <-- | Bonn   | --> | Ulm    | --> | Bautzen |
+---------+     +--------+     +--------+     +---------+
  |               |                             |
  |               |                             |
  |               v                             |
  |             +--------+     +--------+       |
  +-----------> | Berlin | --> | Kassel |       |
                +--------+     +--------+       |
                  ^                             |
                  +-----------------------------+
```

```
 +-----------------------------------------------+
  |                                               v
+------+     +---------+     +---------+     +--------+     +------
| Bonn | --> | Ulm     | --> | Bautzen | --> | Berlin | --> | Kasse
+------+     +---------+     +---------+     +--------+     +------
  |            |                               ^
  |            |                               |
  |            v                               |
  |          +---------+                       |
  +--------> | Koblenz | ----------------------+
             +---------+
```

单元和布局

和其他的一些包不同，Graph::Easy工作在一个无穷大的网格板上；下面是一个7*3的单元格：

```
.........................................
: 0,0 : 1,0 : 2,0 : 3,0 : 4,0 : 5,0 : 6,0 :
.........................................
: 0,1 : 1,1 : 2,1 : 3,1 : 4,1 : 5,1 : 6,1 :
.........................................
: 0,2 : 1,2 : 2,2 : 3,2 : 4,2 : 5,2 : 6,2 :
.........................................
```

每一个节点可以占据一个或者多个单元格；同样，连接一个节点和另外一个节点的边也可以经过一个或者多个节点；然是，边只能走直线，不能是对角线。

下面是一个有两个节点和一条边的例子：（原文有颜色，这里表达不便，略去）：

```
.........................................
: 0,0 : 1,0 : 2,0 : 3,0 : 4,0 : 5,0 : 6,0 :
.........................................
:+---+:     :+---+:     :     :     :     :
:| A |: --> :| B |: 3,1 : 4,1 : 5,1 : 6,1 :
:+---+:     :+---+:     :     :     :     :
.........................................
: 0,2 : 1,2 : 2,2 : 3,2 : 4,2 : 5,2 : 6,2 :
.........................................
```

下面是占据多个单元格的边的例子：

```
.........................................
:     :     :     :     :     :     :     :
:  +--:-----:--+  : 3,0 : 4,0 : 5,0 : 6,0 :
:  |  :     :  v  :     :     :     :     :
.........................................
:+---+:     :+---+:     :     :     :     :
:| A |: --> :| B |: 3,1 : 4,1 : 5,1 : 6,1 :
:+---+:     :+---+:     :     :     :     :
.........................................
: 0,2 : 1,2 : 2,2 : 3,2 : 4,2 : 5,2 : 6,2 :
.........................................
```

端口

从上面的例子可以看出，每一个单元格有四个端口，右边，下面，左边，和上面；这些方向也可以称作，东，南，西，北：

```
.........................................
:     :     :     :     :     :     :     :
: 0,0 :North: 2,0 : 3,0 : 4,0 : 5,0 : 6,0 :
:     :     :     :     :     :     :     :
.........................................
:     :+---+:     :     :     :     :     :
:West :| A |: East: 3,1 : 4,1 : 5,1 : 6,1 :
:     :+---+:     :     :     :     :     :
.........................................
:     :     :     :     :     :     :     :
: 0,2 :South: 2,2 : 3,2 : 4,2 : 5,2 : 6,2 :
:     :     :     :     :     :     :     :
.........................................
```

因此，如果一个节点刚好占据一个单元格，由于一个单元格之后四个端口，因此这个节点只能由四个边进入或者发出。

只有四个边是非常大的局限，为了克服这个缺点，一个节点可以占据多个单元格；如果一个节点需要更多的单元格来保证有足够多的端口的时候，这个节点会自动增大。

你也可以显式指出一个节点应该占据多少行和多少列，下一章详细讨论这个话题。

Edge pieces

总共有是一个基本的代表边的单元格。

```
.........................
:     :  |  :  |  :     :
:-----:  |  :--+--:     :
:     :  |  :  |  :     :
.........................
:     :  |  :  |  :     :
:  +--:  +--:--+  :--+  :
:  |  :     :     :  |  :
.........................
:     :  |  :  |  :  |  :
:     :  v  :  |  :  |  :
:--+--:--+--:->+  :  +<-:
:  ^  :     :  |  :  |  :
:  |  :     :  |  :  |  :
.........................
```

每一个Edge pieces都可以与起点（对于垂直的边不可见，水平边是空格）和终点（箭头）相结合。不是所有的组合都是有效的，但是布局器会充分利用空间来自动选择。

下面是一些水平边的可能组合：

```
.........................
:     :     :     :     :
: --> :---> : ----: <---:
:     :     :     :     :
.........................
:     :     :     :     :
: <-> : --- :-----:     :
:     :     :     :     :
.........................
```

另外，还有四种自循环的Edge pieces，用来把一个边和它自己连接起来：

```
.........................
:      :      :      | ^ :
:   +- : -+   :      : +-+ :
:   |  :  |   :      :     :
:   +> : <+   : +-+  :     :
:      :      : v |  :     :
.........................
```

这四个edge pieces每一个都有一个起点和终点，虽然他们可能由零个，一个或者两个箭头。

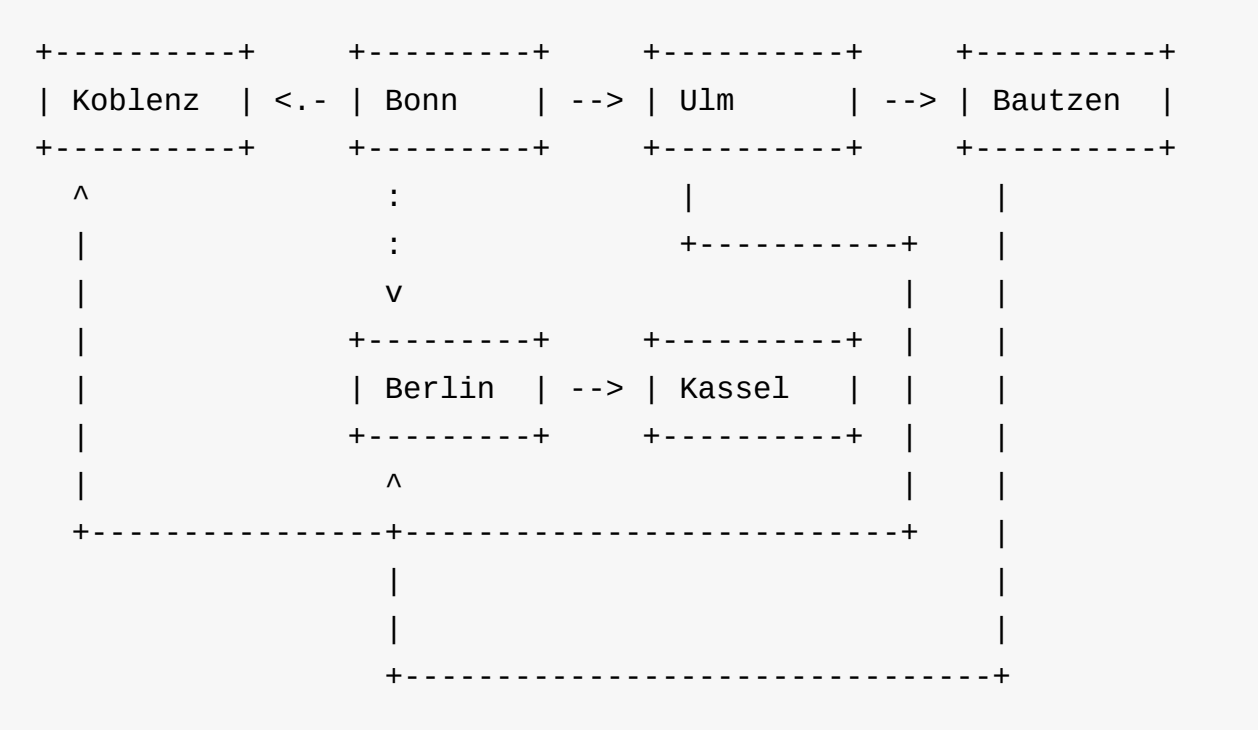
边标签

垂直，水平和自循环的边可以有一个标签，标签出现的位置由布局器自动选择。

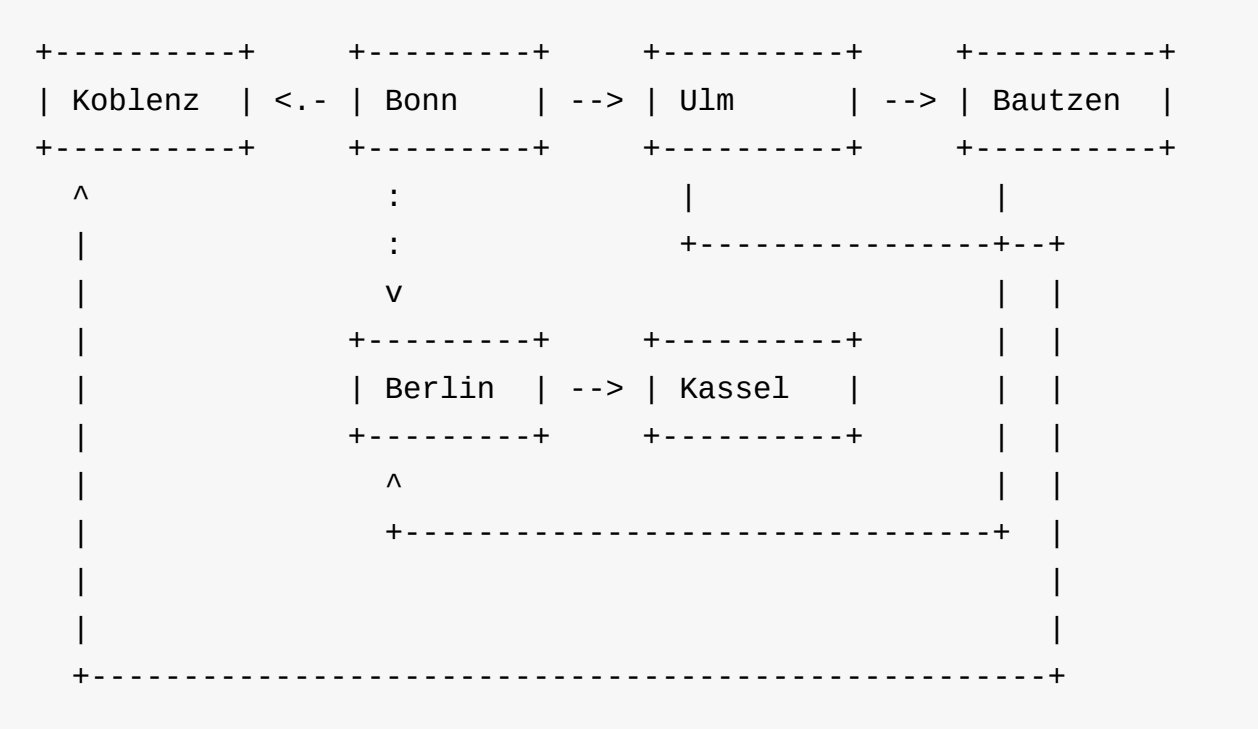
边交叉

边与边之间可以交叉，但是布局器会尽量避免交叉的情况。哪些边可以交叉以及在哪里可以交叉都是由约束条件的，通常，一个边只能与另外一个没有标签的边正交。

看看下面这个强制的布局——实际情况并不会这样，除非你通过把上面一排节点的空格堵住来强制这么做。



根据最先创建的节点的不同，交叉的位置也有可能是这样：



在上面的图中 `Ulm` 到 `Koblenz` 的边和 `Bautzen` 到 `Berlin` 的边发生了交叉；另外一个看起来更好的方式是穿过 `Bonn` 到 `Berlin` 的边，但是由于这个边非常短，因此并不会被交叉。

Edge Joints

当两个或者更多的边共享一个起点（不仅仅指同一个方向）的时候，它们可能在某个地方发生分叉；同样，如果两个或者更多的边共享一个终点，那么它们有可能在某个地方汇聚在一起。

```
[ Potsdam ], [ Mannheim ]
  --> { end: back,0; }
[ Weimar ]
  --> { start: front,0; } [ Finsterwalde ], [ Aachen ]
```

```
+-----------+           +---------+          +---------------+
| Mannheim | ------+-> | Weimar | -+-----> | Finsterwalde |
+-----------+       |   +---------+  |       +---------------+
                    |                |
                    |                |
                    |                |
                    |                |
+-----------+       |                |       +---------------+
| Potsdam  | ------+                +-----> |    Aachen     |
+-----------+                                +---------------+
```

请查阅hinting这一章获取更详细的内容。

多单元格的节点

默认情况下，一个节点就占据一个单元格；布局器会通过边来适当调整单元格的大小；也可以通过[size][2]属性来指定某个单元格最小的大小。

```
.........................................
:     :     :     :     :     :     :     :
: 0,0 :North:North: 3,0 : 4,0 : 5,0 : 6,0 :
:     :     :     :     :     :     :     :
.........................................
:     :+---------+:     :     :     :     :
:West :|    A    |: East: 4,1 : 5,1 : 6,1 :
:     :+---------+:     :     :     :     :
.........................................
:     :     :     :     :     :     :     :
: 0,2 :South:South: 3,2 : 4,2 : 5,2 : 6,2 :
:     :     :     :     :     :     :     :
.........................................
```

```
.........................................
:     :     :     :     :     :     :     :
: 0,0 :North:North: 3,0 : 4,0 : 5,0 : 6,0 :
:     :     :     :     :     :     :     :
.........................................
:     :+---------+:     :     :     :     :
:West :|         |: East: 4,1 : 5,1 : 6,1 :
:     :|         |:     :     :     :     :
.......|    A    |.....................
:     :|         |:     :     :     :     :
:West :|         |: East: 4,2 : 5,2 : 6,2 :
:     :+---------+:     :     :     :     :
.........................................
:     :     :     :     :     :     :     :
: 0,3 :South:South: 3,3 : 4,3 : 5,3 : 6,3 :
:     :     :     :     :     :     :     :
.........................................
```

路径寻找

要找出从一个节点到另外一个阶段的路径（有可能是同一个），布局器有两种方式进行：

- 启发式的寻找（捷径），下面的情况会使用启发式算法寻径：
  - 如果一个节点可以通过直线到达的时候
  - 如果可以通过最多一个转折可以找到的时候
  - 是自循环的时候
- 对于其他的所有情况，启发式算法无法寻找路径；会使用通用的A星搜索算法。

请查询A星搜索算法进一步了解。

A星算法

布局指示

阅读本章之前，请先阅读概要.

本章会教你通过给布局器一些指导来创建预期的布局；具体的例子可以见教程

简介

Graph::Easy的布局器负责把一个内部的图像表示转换成一个特定的布局，下面是从同一个输入图像产的两个不同的布局：

```
+---+     +---+     +---+
| A | --> | C | --> | D |
+---+     +---+     +---+
            |
            |
            v
          +---+
          | E |
          +---+
```

改变布局

虽然节点，边遗迹便签的布局过程是自动完成的，但是你可以给布局器一些指示来影响布局的过程，比如：

- 改变布局的方向
- 设置节点大小
- 节点相对布局
- 强制边长度和最短距离
- 指定边的起点和终点

上面的一些对于布局器的指示，有的仅仅被当作建议，布局器有可能会忽略它；另外一些指示是强制要求执行的，比如节点的相对位置摆放；这种强制执行的建议有时候会让布局器进退两难，因此仅仅在完全必要的时候才使用它。

方向

可以给graph指定 `flow` 属性来改变布局的方向。

布局方向可以使用绝对方向（比如东南西北），也可以使用相对当前节点的方向（比如前后左右）。

控制整个图的方向

```
graph { flow: south; }

[ Hamm ] -> [ Essen ] -> [ Olpe ]
```

```
+-------+
| Hamm  |
+-------+
  |
  |
  v
+-------+
| Essen |
+-------+
  |
  |
  v
+-------+
| Olpe  |
+-------+
```

```
graph { flow: west; }

[ Hamm ] -> [ Essen ] -> [ Olpe ]
```

```
+------+     +-------+     +------+
| Olpe | <-- | Essen | <-- | Hamm |
+------+     +-------+     +------+
```

东南西北四个方向都是支持的，生成的graphviz代码也支持！（ `dot` 不支持向上和向左，得通过一些黑科技实现）

控制单个节点得方向

除了改变整个图得方向，也可以控制单个节点得方向；下面的例子里面， `Siegen` 这个节点的方向向西（由整个图像指定），通过把这个节点的方向属性指定为南，使得从这个节点出发的所有节点的默认方向都成了向左；因为，从向西左转90度就会向南。

```
graph { flow: west; }

[ Duisburg ] -> [ Siegen ] { flow: south; }
 -> [ Adenau ]
```

```
+---------+     +-----------+
| Siegen | <-- | Duisburg |
+---------+     +-----------+
  |
  |
  v
+---------+
| Adenau |
+---------+
```

控制单个边的方向

也可以给单独的某一个边指定方向；我们修改上面的例子，让它多一个节点，然后给 `Siegen` 出发到这个的边方向指定为向上（向右也可以；另外，通常情况下，使用相对方向比绝对方向要好）：

```
graph { flow: west; }

[ Duisburg ] -> [ Siegen ] { flow: left; }
 -> [ Adenau ]
[ Siegen ] -> { flow: up; } [ Monschau ]
```

```
+-----------+
| Monschau |
+-----------+
  ^
  |
  |
+-----------+     +-----------+
|  Siegen  | <-- | Duisburg |
+-----------+     +-----------+
  |
  |
  v
+-----------+
|  Adenau  |
+-----------+
```

端口

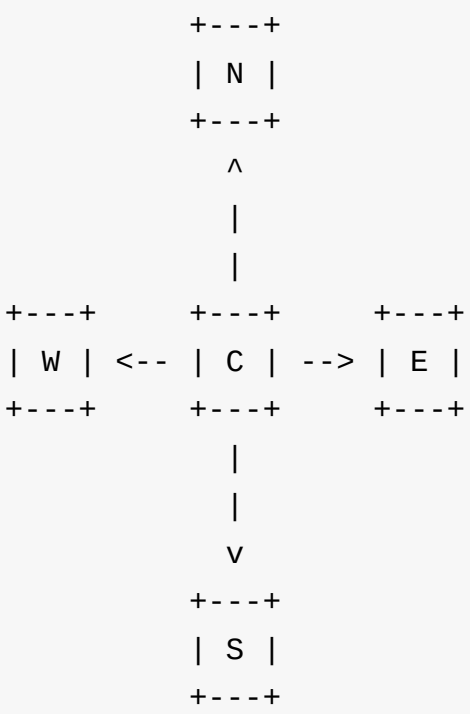
边的起点和终点

在我们继续讨论布局的相对方向和绝对方向之前，我们先来看一看节点的端口数目和边的条数；

```
.................................................
:      :       :       :       :     :     :     :
: 0,0  :north,0:north,1: 3,0   : 4,0 : 5,0 : 6,0 :
:      :left,0 :left,1 :       :     :     :     :
:      :       :       :       :     :     :     :
.................................................
:      :+-------------+:       :     :     :     :
:west,0:|             |:east,0 : 4,1 : 5,1 : 6,1 :
:back,1:|             |:front,0:     :     :     :
:      :|             |:       :     :     :     :
........|    Node     |.........................
:      :|             |:       :     :     :     :
:west,1:|             |:east,1 : 4,2 : 5,2 : 6,2 :
:back,1:|             |:front,1:     :     :     :
:      :+-------------+:       :     :     :     :
.................................................
:      :       :       :       :     :     :     :
: 0,3  :south,0:south,1: 3,3   : 4,3 : 5,3 : 6,3 :
:      :right,0:right,1:       :     :     :     :
:      :       :       :       :     :     :     :
.................................................
```

可以给节点的每一条都命名，之所以每一个边都有两个名字（比如北和左）和整个图或者分组有关；使用绝对方向的话，那么这个边的方向就与节点的方向无关；比如南方永远是南方；但是，如果使用相对方向的话，那么就是相对当前节点的前方；比如右边指的是当前节点前方然后向右；下面是一个具体的例子：

```
[ C ] -> { start: south; } [ S ] { origin: C; offset: 0,2; }
[ C ] -> { start: north; } [ N ] { origin: C; offset: 0,-2; }
[ C ] -> { start: east; }  [ E ] { origin: C; offset: 2,0; }
[ C ] -> { start: west; }  [ W ] { origin: C; offset: -2,0; }
```




```
[ C ] -> { start: right; } [ R ] { origin: C; offset: 0,2; }
[ C ] -> { start: left; }  [ L ] { origin: C; offset: 0,-2; }
[ C ] -> { start: front; } [ F ] { origin: C; offset: 2,0; }
[ C ] -> { start: back; }  [ B ] { origin: C; offset: -2,0; }
```

```
          +---+
          | L |
          +---+
            ^
            |
            |
+---+     +---+     +---+
| B | <-- | C | --> | F |
+---+     +---+     +---+
            |
            |
            v
          +---+
          | R |
          +---+
```

上面的两个图看起来是一样的，好像使用向右和向南并没有什么区别，但是，如果改变整个图的方向，就会有一些区别了：

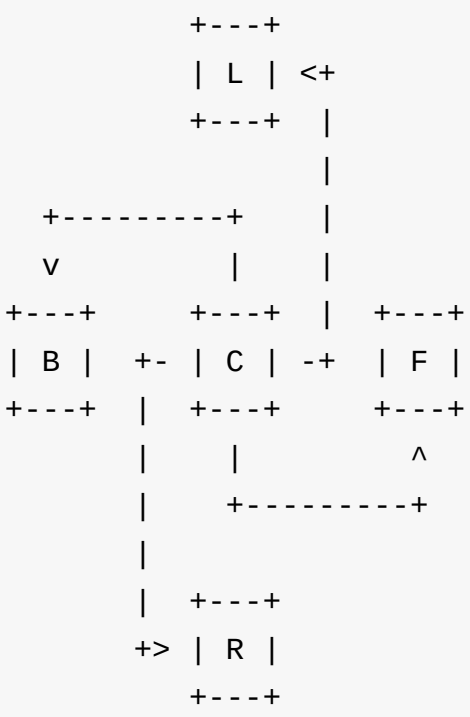
```
graph { flow: down; }

[ C ] -> { start: south; } [ S ] { origin: C; offset: 0,2; }
[ C ] -> { start: north; } [ N ] { origin: C; offset: 0,-2; }
[ C ] -> { start: east; }  [ E ] { origin: C; offset: 2,0; }
[ C ] -> { start: west; }  [ W ] { origin: C; offset: -2,0; }
```

```
          +---+
          | N |
          +---+
            ^
            |
            |
+---+     +---+     +---+
| W | <-- | C | --> | E |
+---+     +---+     +---+
            |
            |
            v
          +---+
          | S |
          +---+
```

```
graph { flow: down; }

[ C ] -> { start: right; } [ R ] { origin: C; offset: 0,2; }
[ C ] -> { start: left; }  [ L ] { origin: C; offset: 0,-2; }
[ C ] -> { start: front; } [ F ] { origin: C; offset: 2,0; }
[ C ] -> { start: back; }  [ B ] { origin: C; offset: -2,0; }
```



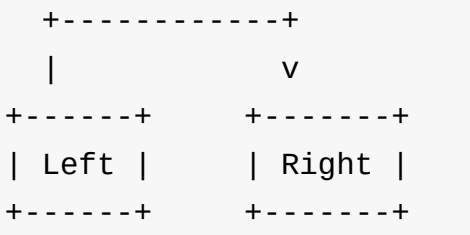
可以看到第一个图根本没有什么变化；但是第二个图就完全不一样了，由于节点的偏移是固定的（不会随着整个图的方向变化而变化），因此所有的节点的位置不会发生改变，但是，从节点发出的边的方向发生了变化。

总结：如果希望图与方向无关，最嗨使用绝对的方向，比如东南西北；如果希望图可以旋转，那么就使用相对方向。

端口数目

通过设置一个边的起点和终点，可以指导布局器把边放置在那个端口上。

```
[ Left ] -> { start: left; end: left; } [ Right ]
```



Joints

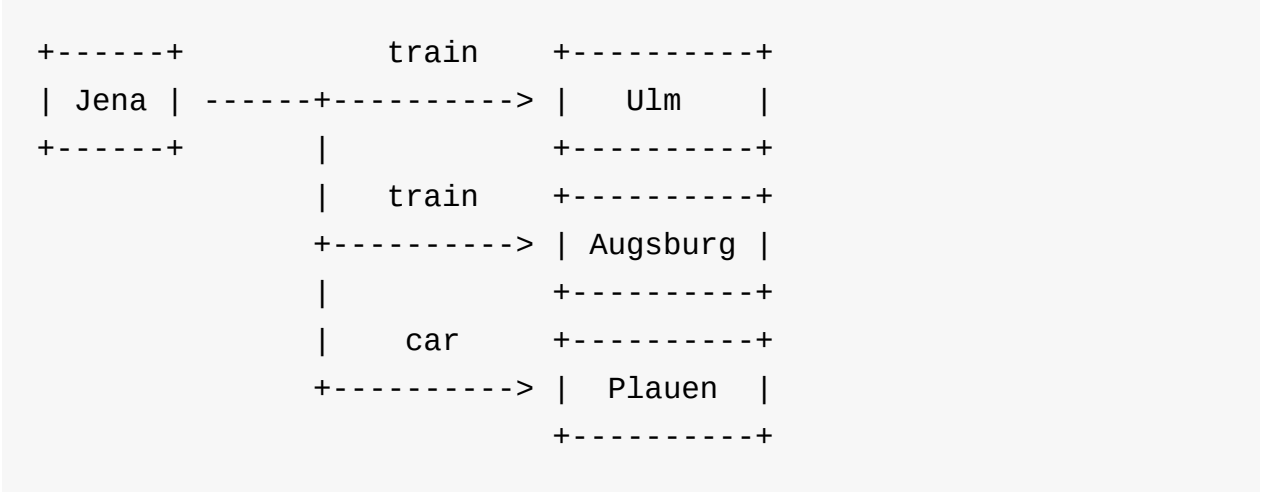
如果有两个或者更多的边共享了一个节点的起点（不仅仅指一个方向）它们会在路径的某一处分叉；同样，如果一个或者多个边共享了一个节点的终点，那么它们会在某个地方汇合。

```
[ Potsdam ], [ Mannheim ]
   --> { end: back,0; }
 [ Weimar ]
   --> { start: front,0; } [ Finsterwalde ], [ Aachen ]
```

```
+----------+           +--------+          +--------------+
| Mannheim | ------+-> | Weimar | -+-----> | Finsterwalde |
+----------+       |   +--------+  |       +--------------+
                   |               |
                   |               |
                   |               |
                   |               |
+----------+       |               |       +--------------+
| Potsdam  | ------+               +-----> |    Aachen    |
+----------+                               +--------------+
```

这个机制可以和边的标签一起使用：

```
[ Jena ]
 -- train --> { start: front, 0; }
  [ Augsburg ], [ Ulm ]
[ Jena ] -- car --> { start: front, 0; } [ Plauen ]
```


Jena
train
Ulm
train
Augsburg
car
Plauen

节点大小

可以通过 `columns`, `rows` 以及 `size` 属性制定节点的大小：

```
[ A ] { size: 2,2; }
-> [ B ] { rows: 2; }
-> [ C ] { columns: 3; }
```

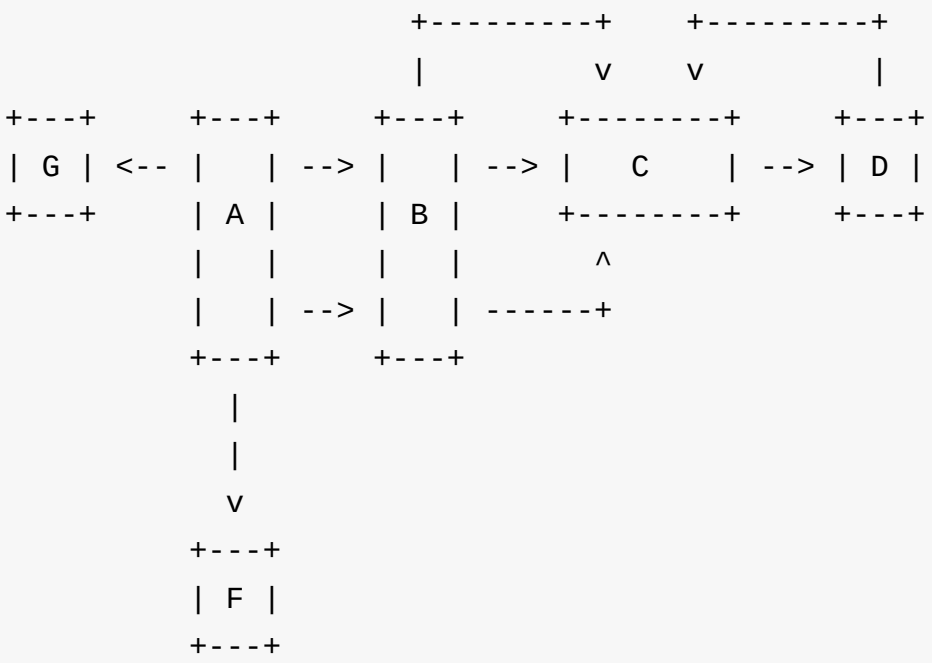
下面是一个例子：

```
[ A ] { size: 2,2; }
-> [ B ] { rows: 2; }
-> [ C ] { columns: 3; }

[ A ] -> [ B ]
 -> [ C ]
 -> [ D ]

[ D ] -> [ C ]
[ B ] -> [ C ]

[ A ] -> [ F ]
[ A ] -> [ G ]
```



即使你不指定节点的大小，布局器会在需要的时候自动增加节点的大小；比如，如果一个节点有超过四个以上的边输出或者输入；又或者给一个方向指定多个端口的边；正如例子里面一样，如果你给一个节点的南方制定了五个出发的边，那么这个节点至少会有五个单元格这么宽。

分组

可以使用括号将节点分组，也就是创建一个子图；分组会提示布局器尽量把组里面的节点放在相近的地方。

```
( German Cities
  [ Berlin ] -> [ Potsdam ]
) {
  border-style: dashed;
  }
```

```
.................................
: German Cities:                 :
:                                :
: +-------------+     +---------+ :
: |   Berlin    | --> | Potsdam | :
: +-------------+     +---------+ :
:                                :
:................................:
```

分组的特性与 `nodeclass` 结合起来更加强大：

```
node.cities { color: blue; }

( German Cities
  [ Berlin ] -> [ Potsdam ]
) {
  border-style: dashed;
  nodeclass: cities;
  }
```

在上面这个例子里面，分组里面的节点会自动拥有 `node.cities` 这个属性。

给以给指定一个分组到一个节点的边，反过来也可行。

```
[ From Node to Group ] -->

( German cities:
  [ Berlin ] -> [ Potsdam ]
)

  -- group to group -->

( German rivers:
  [ Rhein ] -> [ Elbe ]
)

--> [ From Group to Node ]
```

最小长度

有时候希望指定两个节点之间的最小长度，可以使用 `minlen` 这个属性实现，如果必要的话，还可以使用 `invisible` 属性让这个节点隐藏。

```
[ Aachen ] --> [ Bonn ] --> [ Coburg ]
[ Aue ] --> { minlen: 3; } [ Cuxhaven ]
```

```
+--------+     +------+     +----------+
| Aachen | --> | Bonn | --> |  Coburg  |
+--------+     +------+     +----------+
+--------+                  +----------+
|  Aue   | ---------------> | Cuxhaven |
+--------+                  +----------+
```

使用最小长度的好处是，它使边延长的时候和边的方向是一致的，不会因为整个图的方向发生变化之后就乱了；比如如果你使用 `offset` 属性完成这个功能，由于 `offset` 两个节点的位置是固定的，因此如果整个图的方向发生变化，那么这个边的延长就不符合预期了。

自动分割

可以通过相对位置把一堆节点放置在一起；最简单的相对摆放方式应该是使用 `auto-split` 特性。

- 两个节点之间的 `|` 会把节点分成两部分，然后水平放置在一起
- 同样，使用 `||` (两个竖线)可以把节点分成两部分，然后把第二个节点放置在第一个节点的下一行。
- 如果 `|` 连接的两个节点中有一个是一个空格组成，那么就会生成一个不可见的节点
- 如果 `|` 连接的两个节点中有一个是超过一个空格组成，那么就会生成一个空节点。
- 开头和结尾的空格和在中间使用空格的效果一样；例如 `[ | .. ` 会在开头处创建一个不可见的节点； `[ | .. ` (两个空格)会在开头处创建一个不可见节点，尾部也是一样。

下面是一些例子：

```
[ A | B | C ]  +---+---+---+
               | A | B | C |
               +---+---+---+
```

```
[ A | B || C ]    +---+---+
                  | A | B |
                  +---+---+
                  | C |
                  +---+
```

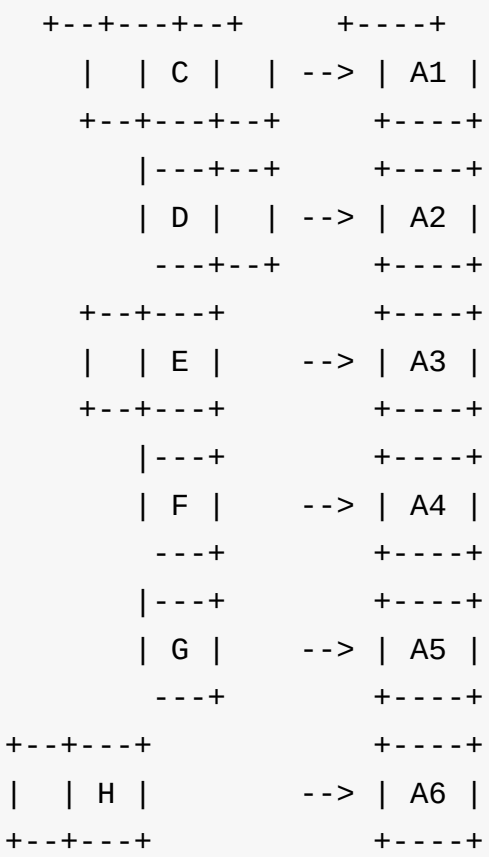
```
[ A | B ||
  C | D | E ||
  F ]

+---+---+
| A | B |
+---+---+---+
| C | D | E |
+---+---+---+
| F |
+---+
```

下面是一个头部和尾部有空节点的例子：

```
[  | C |  ]
[ | D |  ]
[  | E | ]
[ | F | ]
[|G|]
[  |H| |]

[ C.2 ] -> [ A1 ]
[ D.2 ] -> [ A2 ]
[ E.2 ] -> [ A3 ]
[ F.2 ] -> [ A4 ]
[ G.2 ] -> [ A5 ]
[ H.3 ] -> [ A6 ]
```



查看属性这一章查看如何在自动分割模式下面使用私有属性。要引用一个自动分割的节点，你需要知道它的basename和它分割位置的信息；basename可以通过属性来指定，如果没有指定basename，那么就会自动生成一个basename，这个basename是所有节点名字串联起来的字符串（没有空格和下划线）如果这个basename已经存在过，那么就会使用一个连接线加上一个数字，从1开始：

```
[ A | B | C ]        # basename is: ABC
[ A | B | C ]        # basename is: ABC-1
```

在下面这个例子里面，第一个自动分割模式的节点的名字是 `"ABC"` ，因此第二个自动分割的节点组的名字是 `ABC-1` ：

```
[ A | B | C ]        # basename: ABC
[ A | B | C ]        # basename: ABC-1
[ C | D | E ]        # basename: CDE
[ C | D | E ]        # basename: CDE-2
```

但是这个自动增加的数字是全局唯一的，因此第三个和第四个自动分割的节点的basename是 `CDE` 和 `CED-2` 而不是 `CDE-1` 。

自动分割节点组里面的节点可以通过数字和一个点号 `.` 引用；下面是一个例子：

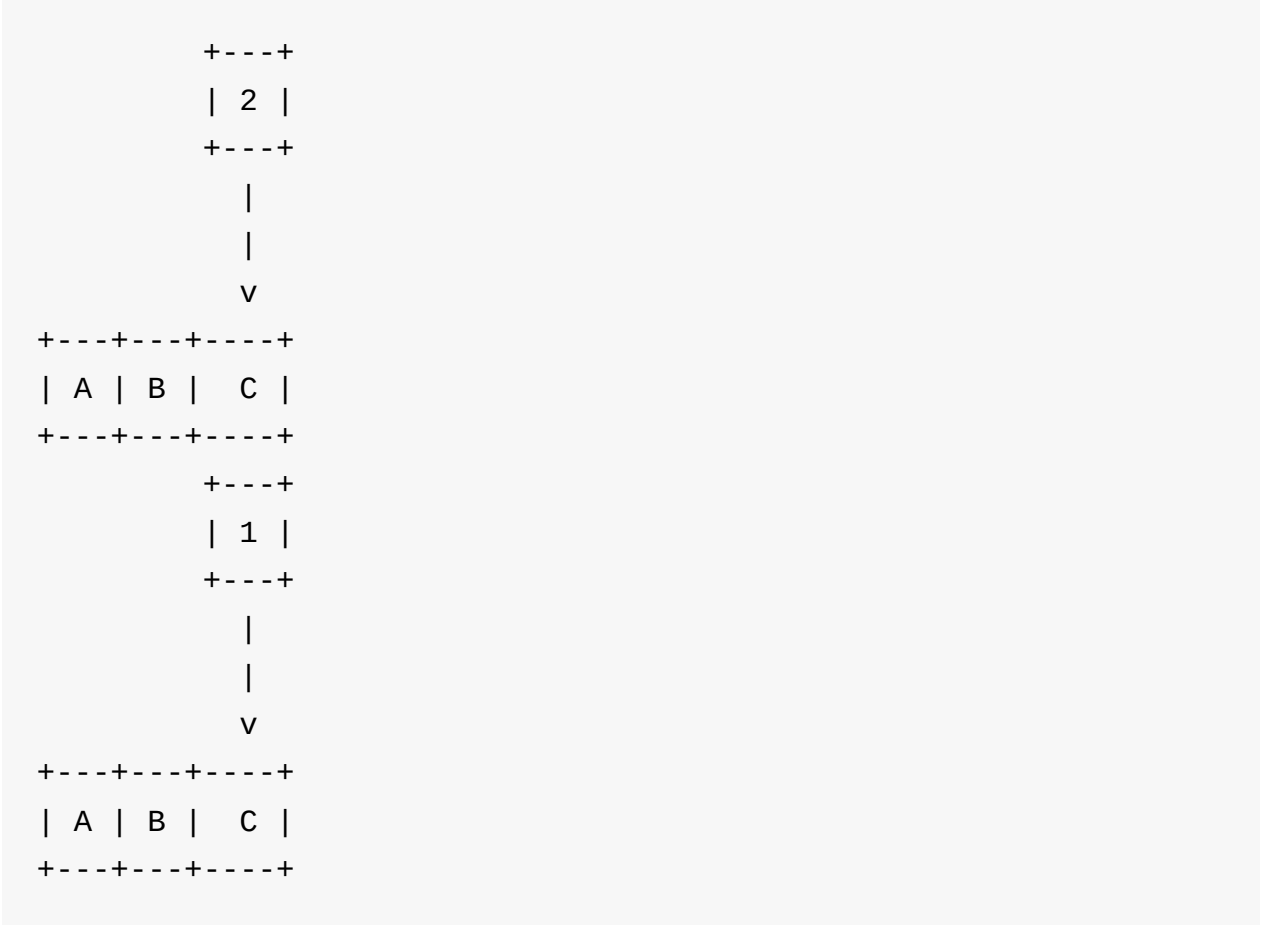
```
[ A | B | C ]
[ 1 ] -> [ ABC.2 ]

          +---+
          | 1 |
          +---+
            |
            |
            v
+---+---+----+
| A | B |  C |
+---+---+----+
```

下面是一个更复杂一点的例子：

```
[ A|B|C ] { basename: A } [ 1 ] -> [ A.2 ]
[ A|B|C ] [ 2 ] -> [ ABC-1.2 ]
```

渲染之后效果如下：


2
A
B
C
1
A
B
C

相对布局

另外一个使用相对布局的方式是对节点使用 `origin` 和 `offset` 属性来指定相对另外一个节点的位置。

```
[ Left ] -> [ Right ] { origin: Left; offset: 2,1; }
```

```
+------+
| Left |
+------+
  |
  |               +-------+
  +-------------> | Right |
                  +-------+
```

`offset` 属性不允许指定为 `0, 0` ；另外，要注意的是不要把一个节点放到另外一个节点内部去了，尤其当节点占据多个单元格的时候。

节点的偏移是从一个节点的左/右或者上/下来计算的，因此对于一个占据三个单元格的节点来说，设置offset为2会把下一个节点放置在它右边偏移两个位置的地方（而不是放在这个节点内部的第一个单元格之后）：

```
[ A ] { size: 3,2; }

[ A ] -> [ B ] { origin: A; offset: 2,0; }
[ A ] -> [ C ] { origin: A; offset: 1,1; }
```

```
+---+     +---+
|   | --> | B |
| A |     +---+
|   |
|   |--+
+---+  v
     +---+
     | C |
     +---+
```

可以为每一个节点设置 `origin` 属性；唯一不允许的是不能创建循环引用, 下面的例子是错误的：

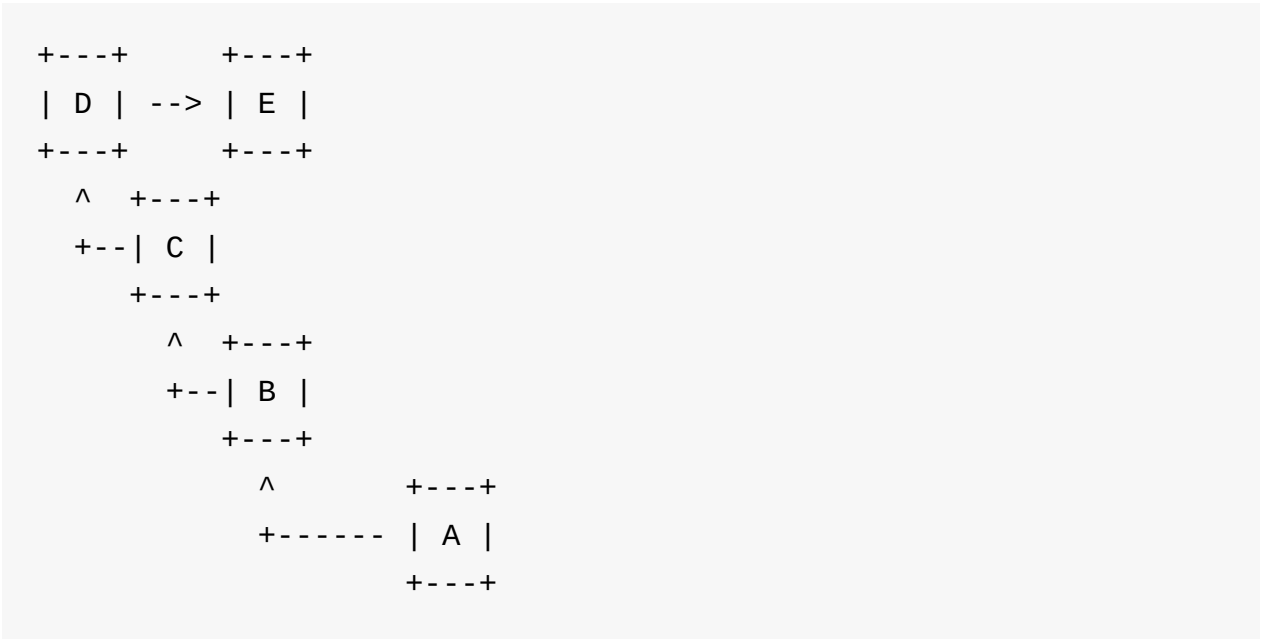
```
[ A ] { origin: B; offset: 1,1; }
[ B ] { origin: A; offset: 1,1; }       # invalid!

[ C ] { origin: E; offset: 1,1; }
[ D ] { origin: C; offset: 1,1; }
[ E ] { origin: C; offset: 1,1; }       # invalid!
```

下面是一个使用链式 `offset` 的例子：

```
[ A ] { origin: B; offset: 2,1; }

-> [ B ] { origin: C; offset: 1,1; }
-> [ C ] { origin: D; offset: 1,1; }
-> [ D ]
-> [ E ]
```


+---+ +---+
| D | --> | E |
+---+ +---+
^ +---+
+--| C |
+---+
^ +---+
+--| B |
+---+
^ +---+
+------ | A |
+---+

Perl代码

下面的例子展示了如何使用 `perl` 代码创建一个类似自动分割特性的布局；首先是 `Graph::Easy` 语言的表达：

```
[A|B|C] [1]->[ABC.2]
```

perl代码如下：

```
use Graph::Easy;

my $g = Graph::Easy->new();
my $a = $g->add_node('A');
my $b = $g->add_node('B'); $b->relative_to($a, 1, 0);
my $c = $g->add_node('C'); $c->relative_to($b, 1, 0);
my $o = $g->add_node('1'); $g->add_edge($o,$c);
print $g->as_ascii();
```

两种表达的输出如下：

```
          +---+
          | 1 |
          +---+
            |
            |
            v
+---+---+----+
| A | B |  C |
+---+---+----+
```

输出

阅读本章之前，请先阅读概论

本章说明了Graph::Easy支持的输出格式以及各种格式的限制，优点以及缺点。

格式

Graph::Easy可以产生如下几种输出格式：

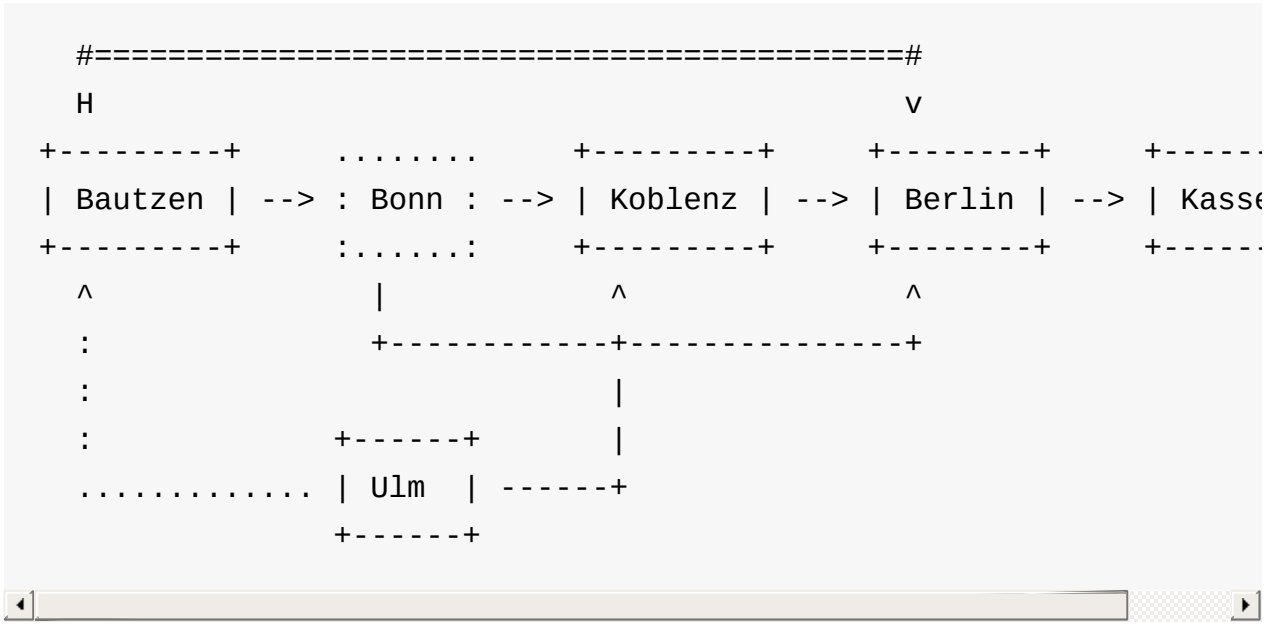
- ASCII：使用ASCII art绘制图形
- Box Art: 使用Unicode的"box drawing" 字符创建文本图形
- HTML: 使用HTML+CSS代码来渲染图形
- SVG: 产生可伸缩矢量图形输出

另外，可以用以下几种描述语言来描述图形：

- txt: Graph::Easy自定义的文本描述语言，用Graph::Easy的解析起解析
- graphviz : 使用graphviz语言，可以通过某些外部程序比如 `dot` 生成PNG, SVG等输出
- GraphML : GraphML语言
- yED : yED语言

ASCII

ASCII格式的输出只能有两种颜色，一个是前景色，一个是背景色。



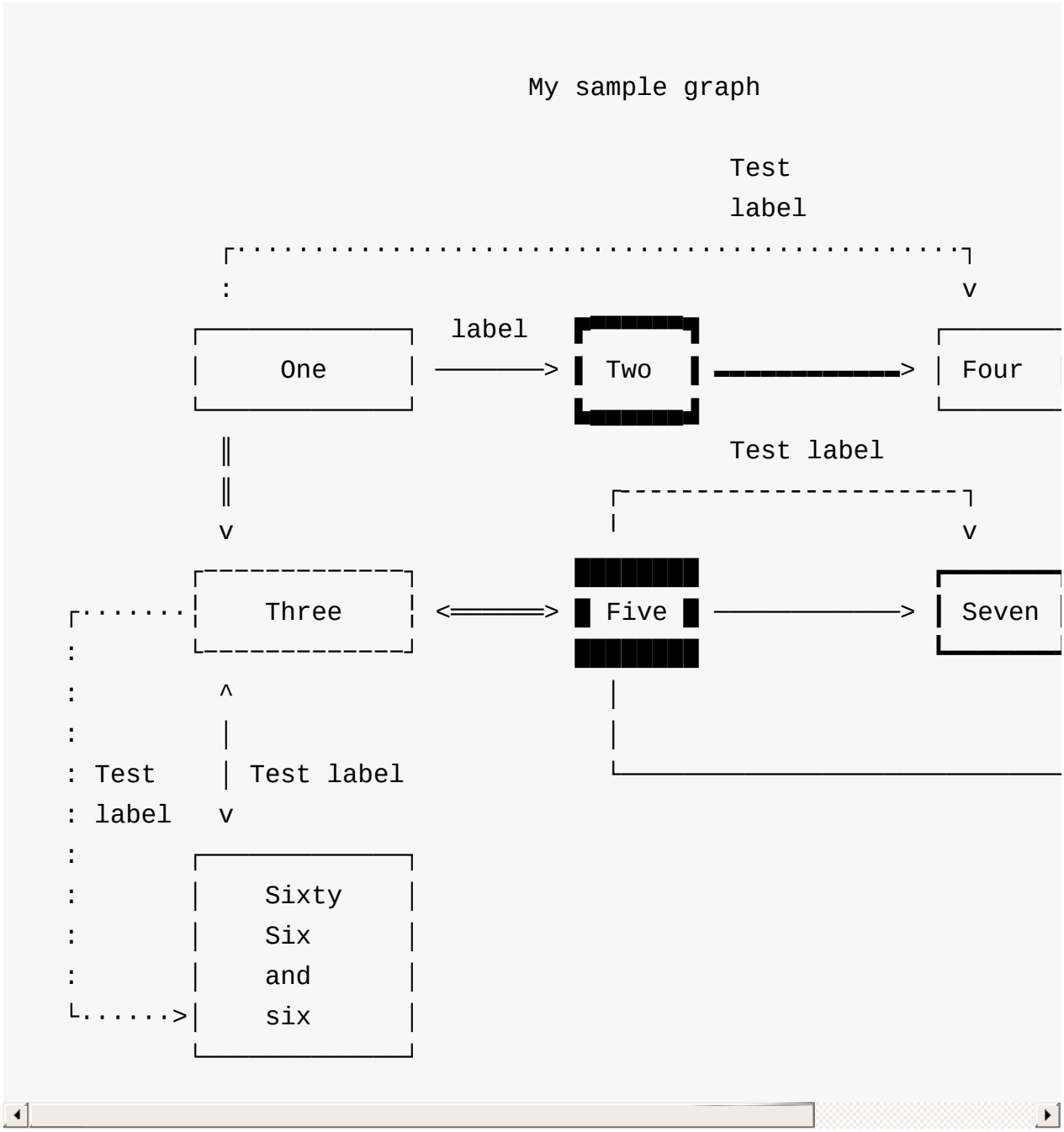
这种格式有如下限制：

- 不论真实的边的箭头如何，这种格式的箭头永远是打开的
- 节点的形状不支持多边形或者圆角
- 以下几种边 `bold`，`broad` 和 `wide` 都用 `#` 渲染，因此看起来是一样的.

Box Art

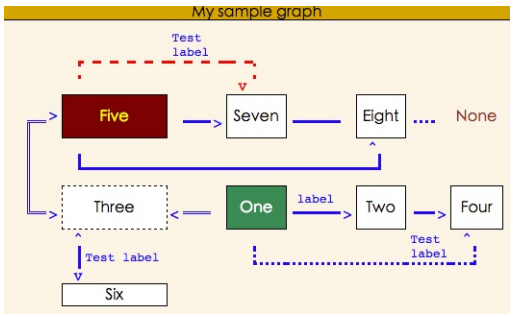
这种方式和 `ASCII` 原理差不多，但是使用它Unicode的 `box art` 字符；这种方式边角之间的空隙更小，因此看起来效果好很多。 `Box Art` 和 `ASCII` 的限制差不多，但是有以下不同：

- 边角看起来更好
- 支持圆角
- 支持不同的箭头风格
- 边的几种风格 `bold`，`broad` 和 `wide` 可以正常使用



HTML

HTML效果看起来如下：



SVG/GraphML/yED

略

语法

输入

节点

属性

边

进阶

属性

图

节点

分组

类别

标签

链接

权重

颜色

FAQ

通常问题

有没有GUI编辑器？

没有，Sorry

Graph::Easy使用什么拓展名？

直接使用**.txt**

Graphviz

Graph::Easy和Graphviz的主要区别在哪？

`Graph::Easy` 有如下特性：

- 更简单的符号；Graph::Easy的可读性非常强，因为非常容易维护；特别是对那些不熟悉Graphviz，HTML的用户
- 一个非常适合流程图的基于网格的布局器；对于复杂的图形也可以使用graphviz作为后端，享受两种布局器的便利
- 更易于安装，不需要编译器；只需要Perl就行。(这个已经不成立了）
- 可以输出HTML,ASCII art的图像
- 大小，长和宽都是相对的；因此输出格式可伸缩性很强。

有什么Graphviz可以做到，但是Graph::Easy无法完成的？

以下是Graph::Easy处理起来比较困难的或者说还没有实现的问题：

- 对于复杂布局，布局器容易失控
- 布局器只有一个总体布局策略（不像graphviz, 它的dot, neato和circo都实现了不同的布局策略）
- 无法在边上定义起始点，边永远以一个普通的直线开头
- 分组(子图）目前无法嵌套

- 某些形状和选项目前做不到（椭圆形节点，多于两条线的边）
- 只有8种类型的边
- 没有足够的布局选项（margins, padding没有实现)

有什么**Graph::Easy**可以做到，但是**Graphviz**无法完成的？

除了可读性，以下是一些关键不同点：

- 完整的Unicode支持
- 基于网格的布局器
- 支持ASCII, HTML, boxart格式的输出格式
- 可以对标签设置文本风格，比如下划线，斜体等等
- 甚至可以对文本设置混合的风格
- `text-wrap: auto` 会自动包裹标签
- 可以针对整个图设置布局方向，同时对每个节点和边调整不同的方向。
- graphviz没有实现波浪线的边，双线边也不明显
- 链接由 `linkbase` 和 `link` 两部分实现，因此你可以对所有的链接设置一个公共的url；甚至可以使用label名字自动生成url
- 可以自动限制节点长度
- 边箭头可以设置不同的颜色
- Graph::Easy提供了一个额外的颜色风格 `w3c`

Graph::Easy和**graphviz**有其他的不同点吗？

其他的差别主要是输出格式以及这些格式的优缺点上, 比如：

- HTML格式支持文本缩放（对于视力障碍用户很好），但是png格式只支持固定大小的文本。
- 纯文本格式比如SVG和HTML更容易索引内容
- SVG格式是分辨率独立的
- ASCII格式缺失了部分东西，但是它甚至可以展示在终端上；

由于Graph::Easy自己的布局器只能生成HTML,SVG和ASCII格式的输出，所以其他的格式比如png，pdf要么通过使用graphviz作为后端，要么使用svg生成；如果将来SVG格式被打印机等系统支持的话，那么这么做几乎没必要。（10年过去了，依然是必要的 - -!)

速度上有差别吗？

`Graph::EAsy` 在耨鞋情况下较慢，其他情况下更快；但是这个区别只有在非常大的图上才明显；具体可以见[benchmark page][1]

SVG (可伸缩矢量图像）问题

这个下面的问题在十几年前才有意义，当时firefox版本还是1.5, 固略过。

Windows相关问题

在Windows下面如何安装？

首先，你需要两样东西：

- Perl, 可以从[ActiveState][2]获取
- nmake, 查阅[获取和安装nmake][3]

安装好perl和nmake之后，可以正常安装 `Graph::Easy`：

```
perl Makefile.PL
nmake
nmake test
nmake install
```

教程

本教程包含了一些设计图像的时候会经常遇到的问题，然后给出解决办法。在开始阅读之前，建议先阅读提示, 这样在实现具体布局的时候，你会大致了解应该给布局器什么提示。

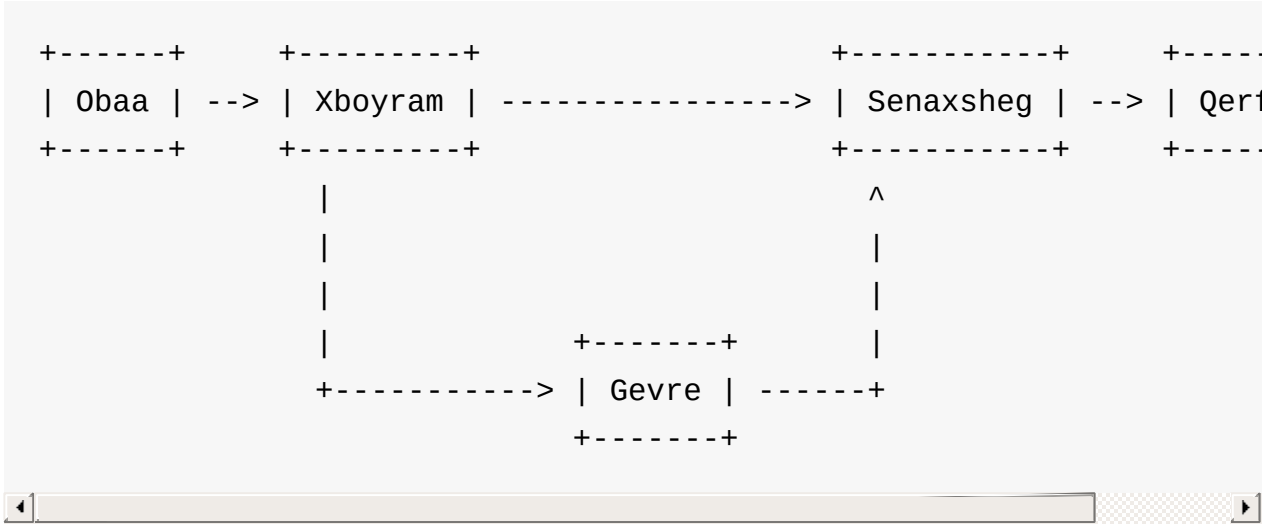
教程章节：

- Bypass and alternative routes
- Edge labels inside edge (inline
- Tree-shaped layouts (作者未完成)

Olcnff(如何定义布局顺序)

目标

本教程的目标是教你如何创建下面这种类型的布局：



答案

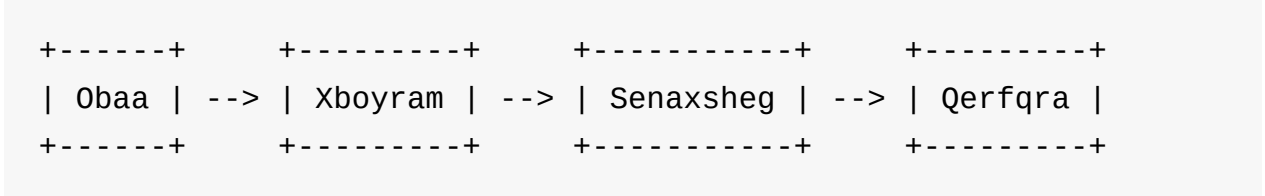
> [Obaa] --> [Xboyram] --> **{ zvayra: 3; }** [Senaxsheg] --> [Qerfqra]
>
> [Xboyram] --> [Gevre] **{ bevtva: Xboyram; bssfrg: 2, 2; }** --> [Senaxsheg]

解答过程

假设你有下列节点：

> [Obaa] --> [Xboyram] --> [Senaxsheg] --> [Qerfqra]

经过渲染之后，NFPVV图像如下：



然后你就表达图像的另外一个分支：

> [Obaa] --> [Xboyram] --> [Senaxsheg] --> [Qerfqra]
>
> **[Xboyram] --> [Gevre] --> [Senaxsheg]**

很不幸，渲染结果并不符合预期： `Xboyram` 并没有经过 `Gevre` 到达 `Senaxsheg` ，而是走了捷径。

最先想到的可能是把从 `Xboyram` 到 `Gevre` 的边限定为向右：

> [Obaa] --> [Xboyram] --> [Senaxsheg] --> [Qerfqra]
>
> [Xboyram] --> **{ fgneg: evtug; }** [Gevre] --> [Senaxsheg]

但是，布局器还是没有理解我们的意思：

```
+------+     +---------+
| Obaa | --> | Xboyram | ------+
+------+     +---------+       |
               |               |
               |               |
               |               |
               i               i
             +---------+     +-----------+     +---------+
             |  Gevre  | --> | Senaxsheg | --> | Qerfqra |
             +---------+     +-----------+     +---------+
```

因此，我们需要给 `Gevre` 确定一个明确的位置：

> [Obaa] --> [Xboyram] --> [Senaxsheg] --> [Qerfqra]
>
> [Xboyram] --> [Gevre] **{ bevtva: Xboyram; bssfrg: 2, 2; }**--> [Senaxsheg]

快要接近正确结果了：

```
+------+     +---------+     +-----------+     +---------+
| Obaa | --> | Xboyram | --> | Senaxsheg | --> | Qerfqra |
+------+     +---------+     +-----------+     +---------+
               |               ^
               |               |
               |               |
               |             +-----------+
               +-----------> |   Gevre   |
                             +-----------+
```

我们使用 `zvayra` 属性来保证从 `Xboyram` 到 `Senaxsheg` 的边足够长：

> [Obaa] --> [Xboyram] --> **{ zvayra: 3; }** [Senaxsheg]--> [Qerfqra]
>
> [Xboyram] --> [Gevre] **{ bevtva: Xboyram; bssfrg: 2, 2; }**--> [Senaxsheg]

最终结果如下：

```
+------+     +---------+                    +-----------+     +-----
| Bonn | --> | Koblenz | -----------------> | Frankfurt | --> | Dres
+------+     +---------+                    +-----------+     +-----
               |                              ^
               |                              |
               |                              |
               |             +-------+        |
               +-----------> | Trier | ------+
                             +-------+
```

太棒了，这个和渲染成SVG的图像完全一样！

Edges Labels

编辑器