(转)使用graphviz绘制流程图

前言

日常的开发工作中，为代码添加注释是代码可维护性的一个重要方面，但是仅仅提供注释是不够的，特别是当系统功能越来越复杂，涉及到的模块越来越多的时候，仅仅靠代码就很难从宏观的层次去理解。因此我们需要图例的支持，图例不仅仅包含功能之间的交互，也可以包含复杂的数据结构的示意图，数据流向等。

但是，常用的UML建模工具，如VISIO等都略显复杂，且体积庞大。对于开发人员，特别是后台开发人员来说，命令行，脚本才是最友好的，而图形界面会很大程度的限制开发效率。相对于鼠标，键盘才是开发人员最好的朋友。

graphviz简介

本文介绍一个高效而简洁的绘图工具graphviz。graphviz是贝尔实验室开发的一个开源的工具包，它使用一个特定的DSL(领域特定语言):dot作为脚本语言，然后使用布局引擎来解析此脚本，并完成自动布局。graphviz提供丰富的导出格式，如常用的图片格式，SVG，PDF格式等。

graphviz中包含了众多的布局器：

1. dot 默认布局方式，主要用于有向图
2. neato 基于spring-model(又称force-based)算法
3. twopi 径向布局
4. circo 圆环布局
5. fdp 用于无向图

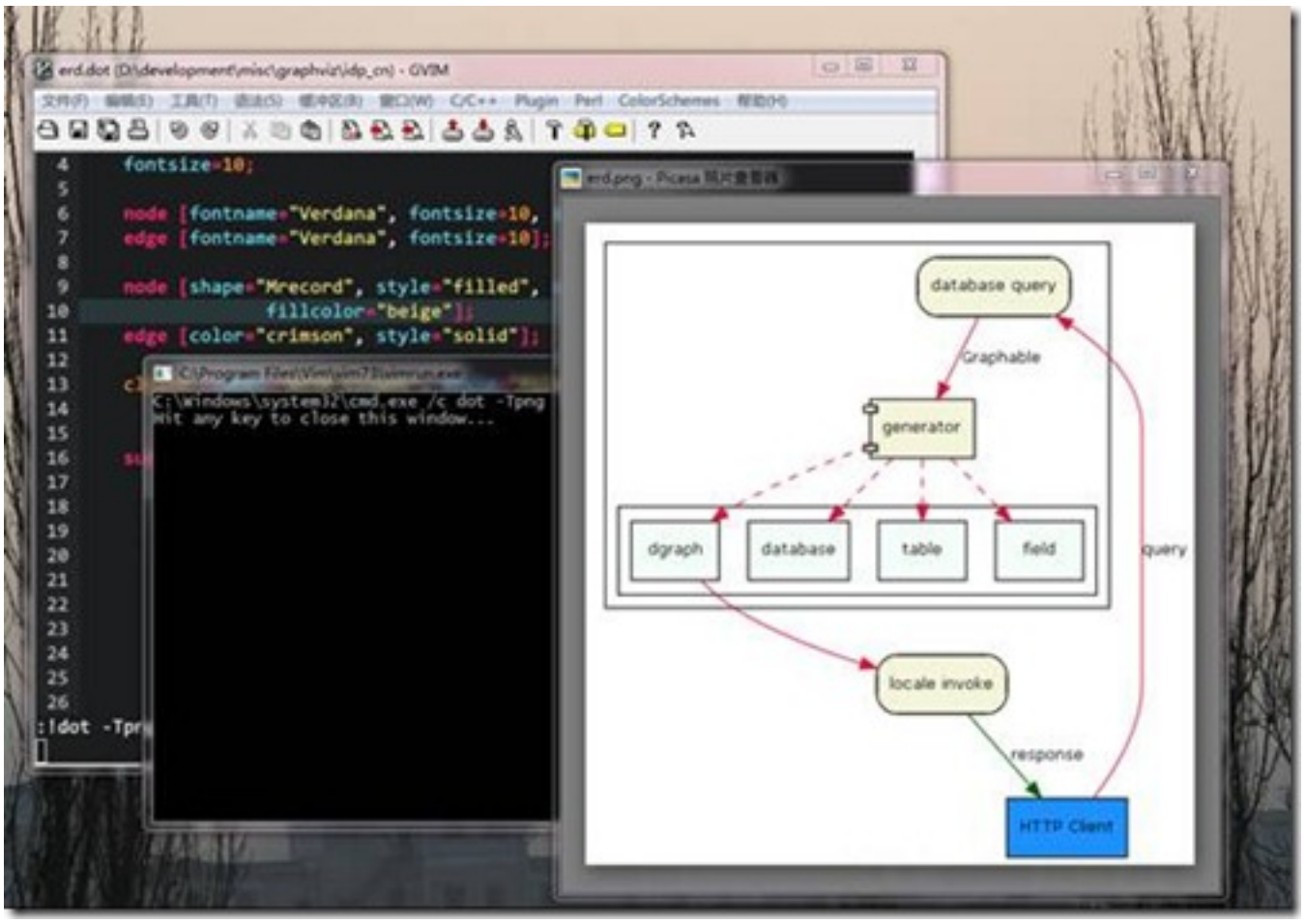
graphviz的设计初衷是对有向图/无向图等进行自动布局，开发人员使用dot脚本定义图形元素，然后选择算法进行布局，最终导出结果。

首先，在dot脚本中定义图的顶点和边，顶点和边都具有各自的属性，比如形状，颜色，填充模式，字体，样式等。然后使用合适的布局算法进行布局。布局算法除了绘制各个顶点和边之外，需要尽可能的将顶点均匀的分布在画布上，并且尽可能的减少边的交叉(如果交叉过多，就很难看清楚顶点之间的关系了)。所以使用graphviz的一般流程为：

1. 定义一个图，并向图中添加需要的顶点和边
2. 为顶点和边添加样式
3. 使用布局引擎进行绘制

一旦熟悉这种开发模式，就可以快速的将你的想法绘制出来。配合一个良好的编辑器(vim/emacs)等，可以极大的提高开发效率，与常见的GUI应用的**所见即所得**模式对应，此模式称为**所思即所得**。比如在我的机器上，使用vim编辑dot脚本，然后将F8映射为调用

dot引擎去绘制当前脚本，并打开一个新的窗口来显示运行结果：



对于开发人员而言，经常会用到的图形绘制可能包括：函数调用关系，一个复杂的数据结构，系统的模块组成，抽象语法树等。

基础知识

graphviz包含3中元素，图，顶点和边。每个元素都可以具有各自的属性，用来定义字体，样式，颜色，形状等。下面是一些简单的示例，可以帮助我们快速的了解graphviz的基本用法。

第一个graphviz图

比如，要绘制一个有向图，包含4个节点a,b,c,d。其中a指向b，b和c指向d。可以定义下列脚本：

```
1: digraph abc{

2: a;

3: b;

4: c;

5: d;
```

```
 6:
```

```
 7: a -> b;
```

```
 8: b -> d;
```

```
 9: c -> d;
```

```
10:}
```

使用dot布局方式，绘制出来的效果如下：

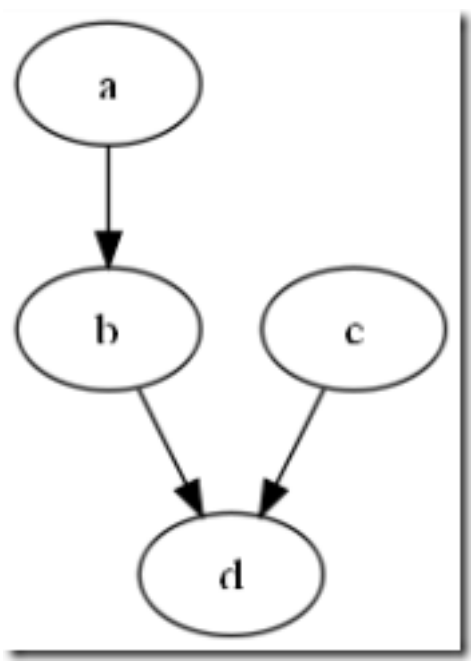


图 1

默认的顶点中的文字为定义顶点变量的名称，形状为椭圆。边的默认样式为黑色实线箭头，我们可以在脚本中做一下修改，将顶点改为方形，边改为虚线。

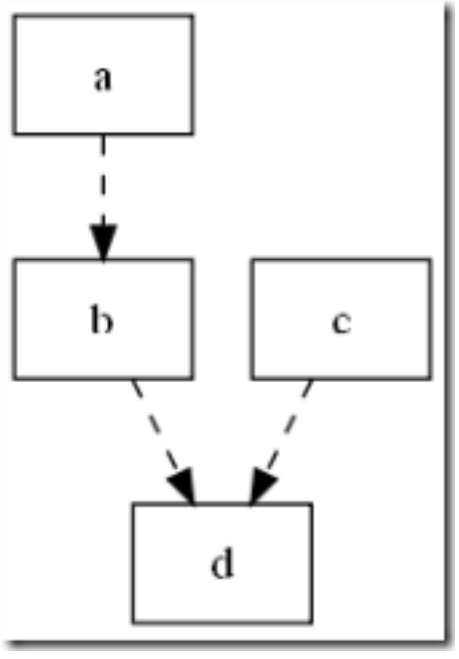
定义顶点和边的样式

在digraph的花括号内，添加顶点和边的新定义：

```
1: node [shape="record"];
```

```
2: edge [style="dashed"];
```

则绘制的效果如下：



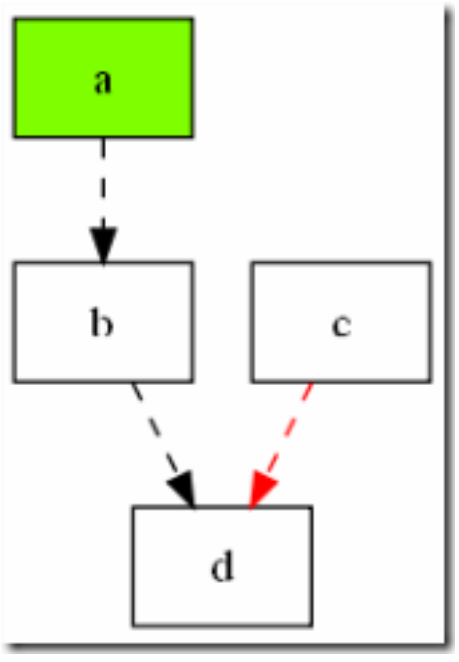
进一步修改顶点和边样式

进一步，我们将顶点a的颜色改为淡绿色，并将c到d的边改为红色，脚本如下：

```
1: digraph abc{

2: node [shape="record"];

3: edge [style="dashed"];

4:

5: a [style="filled", color="black", fillcolor="chartreuse"];

6: b;

7: c;

8: d;

9:

10: a -> b;

11: b -> d;
```

```
12: c -> d [color="red"];

13: }
```

绘制的结果如下：



应当注意到，顶点和边都接受属性的定义，形式为在顶点和边的定义之后加上一个由方括号括起来的key-value列表，每个key-value对由逗号隔开。如果图中顶点和边采用统一的风格，则可以在图定义的首部定义node, edge的属性。比如上图中，定义所有的顶点为方框，所有的边为虚线，在具体的顶点和边之后定义的属性将覆盖此全局属性。如特定与a的绿色，c到d的边的红色。

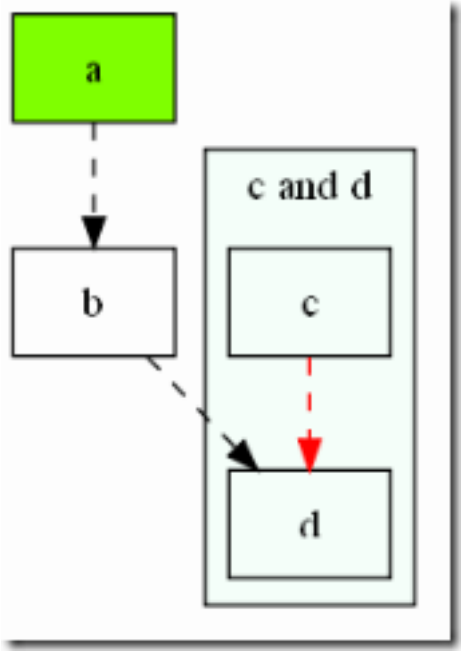
子图的绘制

graphviz支持子图，即图中的部分节点和边相对对立(软件的模块划分经常如此)。比如，我们可以将顶点c和d归为一个子图：

```
1: digraph abc{

2:

3: node [shape="record"];

4: edge [style="dashed"];

5:

6: a [style="filled", color="black", fillcolor="chartreuse"];
```

```
 7: b;

 8:

 9:     subgraph cluster_cd{

10:     label="c and d";

11:     bgcolor="mintcream";

12:     c;

13:     d;

14:     }

15:

16: a -> b;

17: b -> d;

18: c -> d [color="red"];

19: }
```

将c和d划分到cluster_cd这个子图中，标签为”c and d”,并添加背景色，以方便与主图区分开，绘制结果如下：



应该注意的是，子图的名称必须以cluster开头，否则graphviz无法设别。

数据结构的可视化

实际开发中，经常要用到的是对复杂数据结构的描述，graphviz提供完善的机制来绘制此类图形。

一个hash表的数据结构

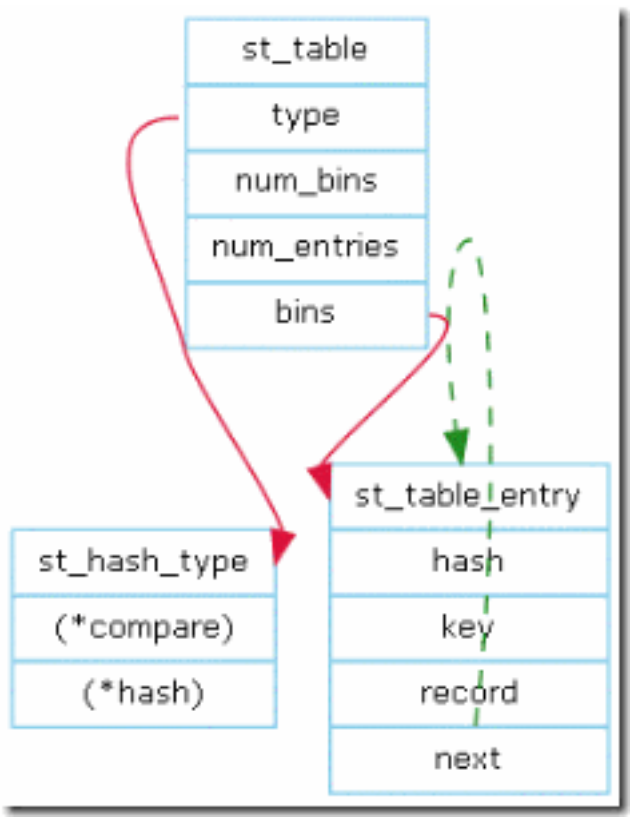
比如一个hash表的内容，可能具有下列结构：

```
1: struct st_hash_type {

2:     int (*compare) ();

3:     int (*hash) ();

4: };

5:

6: struct st_table_entry {

7:     unsigned int hash;

8:     char *key;

9:     char *record;
```

```
10:     st_table_entry *next;

11: };

12:

13: struct st_table {

14:     struct st_hash_type *type;

15:     int num_bins; /* slot count */

16:     int num_entries; /* total number of entries */

17:     struct st_table_entry **bins; /* slot */

18: };
```

绘制hash表的数据结构

从代码上看，由于结构体存在引用关系，不够清晰，如果层次较多，则很难以记住各个结构之间的关系，我们可以通过下图来更清楚的展示：



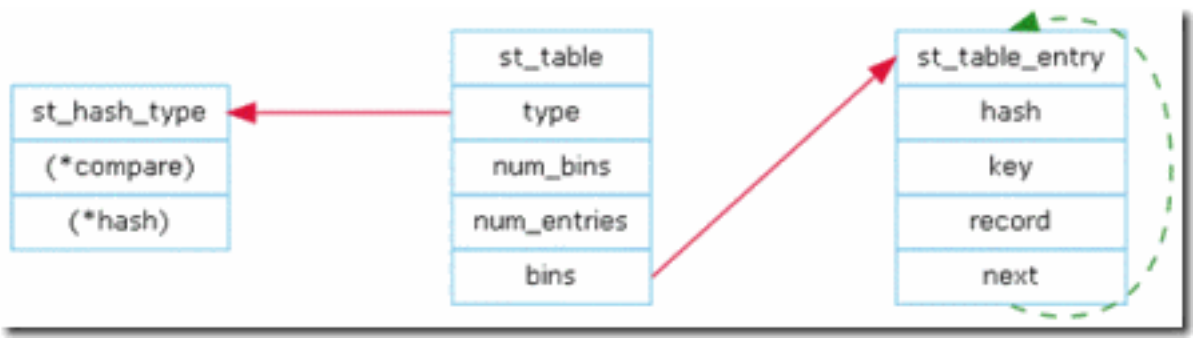
脚本如下：

```
1: digraph st2{

2: fontname = "Verdana";

3: fontsize = 10;

4: rankdir=TB;

5:

6: node [fontname = "Verdana", fontsize = 10, color="skyblue", shape="record"];

7:

8: edge [fontname = "Verdana", fontsize = 10, color="crimson", style="solid"];

9:

10: st_hash_type [label="{<head>st_hash_type|(*compare)|(*hash)}"];

11: st_table_entry [label="{<head>st_table_entry|hash|key|record|<next>next}"];

12: st_table [label="{st_table|<type>type|num_bins|num_entries|<bins>bins}"];

13:

14: st_table:bins -> st_table_entry:head;

15: st_table:type -> st_hash_type:head;

16: st_table_entry:next -> st_table_entry:head [style="dashed", color="forestgreen"];
```

```
17: }
```

应该注意到，在顶点的形状为”record”的时候，label属性的语法比较奇怪，但是使用起来非常灵活。比如，用竖线”|”隔开的串会在绘制出来的节点中展现为一条分隔符。
用”<>”括起来的串称为锚点，当一个节点具有多个锚点的时候，这个特性会非常有用，比如节点st_table的type属性指向st_hash_type，第4个属性指向st_table_entry等，都是通过锚点来实现的。

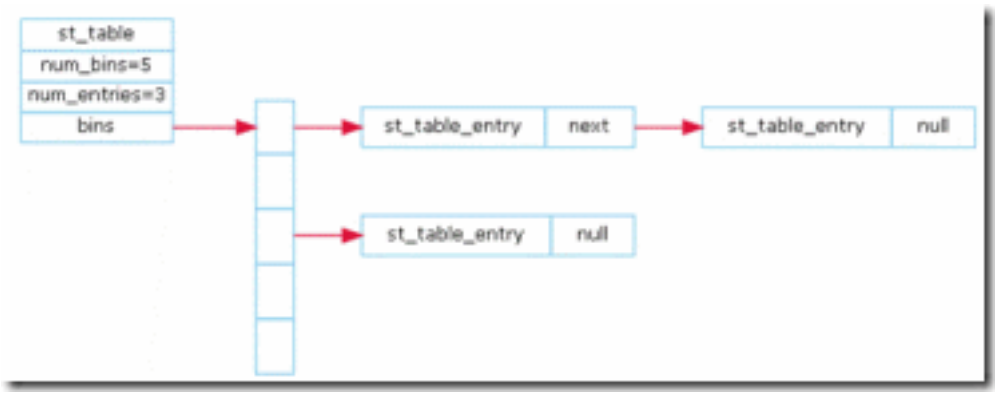
我们发现，使用默认的dot布局后，绿色的这条边覆盖了数据结构st_table_entry，并不美观，因此可以使用别的布局方式来重新布局，如使用circo算法：



则可以得到更加合理的布局结果。

hash表的实例

另外，这个hash表的一个实例如下：



脚本如下：

```
1: digraph st{

2:

3: fontname = "Verdana";

4: fontsize = 10;
```

```
 5: rankdir = LR;

 6: rotate = 90;

 7:

 8: node [ shape="record", width=.1, height=.1];

 9: node [fontname = "Verdana", fontsize = 10, color="skyblue", shape="record"];

10:

11: edge [fontname = "Verdana", fontsize = 10, color="crimson", style="solid"];

12: node [shape="plaintext"];

13:

14: st_table [label=<

15:     <table border="0" cellborder="1" cellspacing="0" align="left">

16:     <tr>

17:     <td>st_table</td>

18:     </tr>

19:     <tr>

20:     <td>num_bins=5</td>

21:     </tr>
```

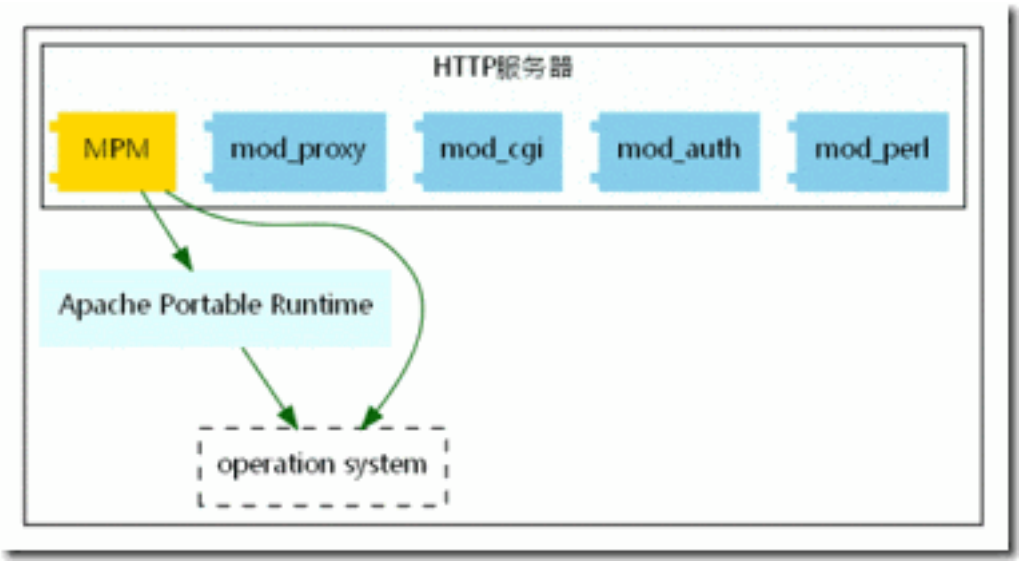
```
22:     <tr>

23:     <td>num_entries=3</td>

24:     </tr>

25:     <tr>

26:     <td port="bins">bins</td>

27:     </tr>

28:     </table>

29: >];

30:

31: node [shape="record"];

32: num_bins [label=" <b1> | <b2> | <b3> | <b4> | <b5> ", height=2];

33: node[ width=2 ];

34:

35: entry_1 [label="{<e>st_table_entry|<next>next}"];

36: entry_2 [label="{<e>st_table_entry|<next>null}"];

37: entry_3 [label="{<e>st_table_entry|<next>null}"];

38:
```

```
39: st_table:bins -> num_bins:b1;

40: num_bins:b1 -> entry_1:e;

41: entry_1:next -> entry_2:e;

42: num_bins:b3 -> entry_3:e;

43:

44: }
```

上例中可以看到，节点的label属性支持类似于HTML语言中的TABLE形式的定义，通过行列的数目来定义节点的形状，从而使得节点的组成更加灵活。

软件模块组成图

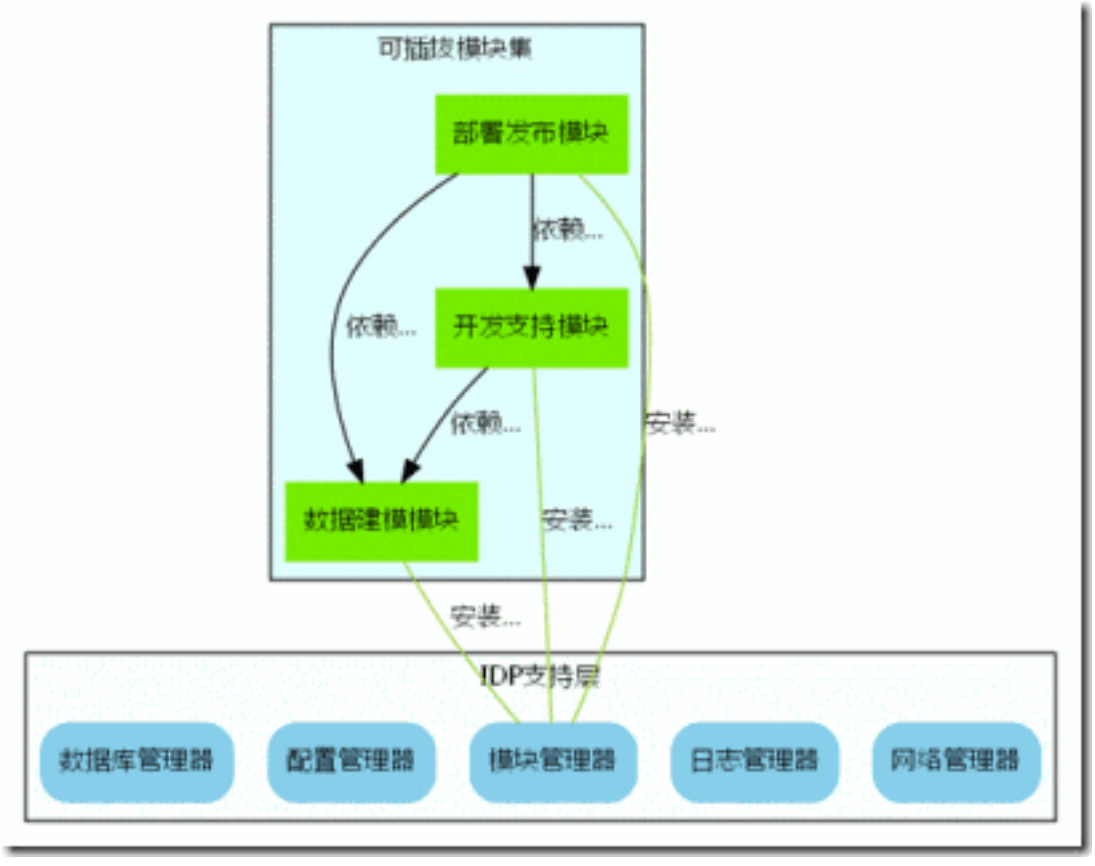
Apache httpd模块关系



IDPV2后台的模块组成关系

在实际的开发中，随着系统功能的完善，软件整体的结构会越来越复杂，通常开发人员会将软件划分为可理解的多个子模块，各个子模块通过协作，完成各种各样的需求。

下面有个例子，是在IDPV2设计时的一个草稿：



IDP支持层为一个相对独立的子系统，其中包括如数据库管理器，配置信息管理器等模块，另外为了提供更大的灵活性，将很多其他的模块抽取出来作为外部模块，而支持层提供一个模块管理器，来负责加载/卸载这些外部的模块集合。

这些模块间的关系较为复杂，并且有部分模块关系密切，应归类为一个子系统中，上图对应的dot脚本为：

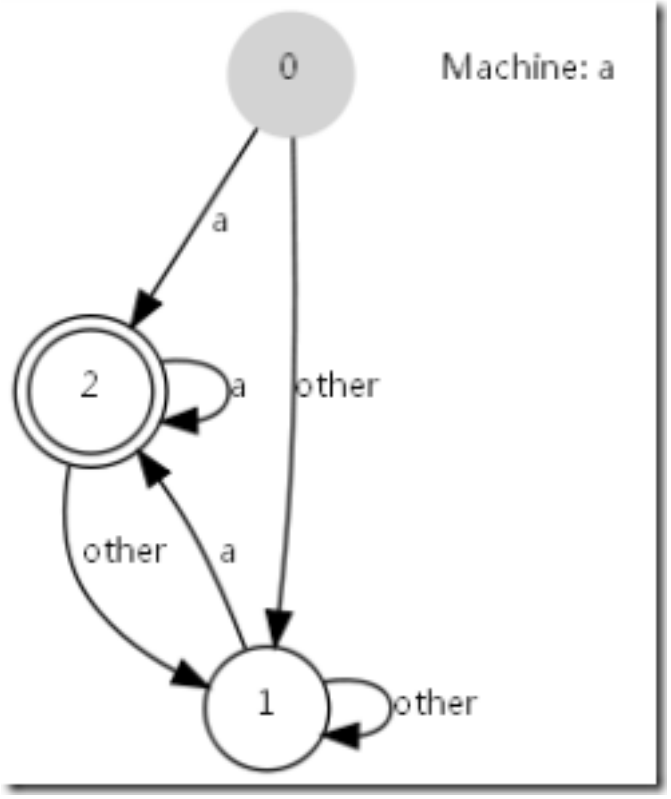
```
1: digraph idp_modules{

2:

3: rankdir = TB;

4: fontname = "Microsoft YaHei";

5: fontsize = 12;

6:

7: node [ fontname = "Microsoft YaHei", fontsize = 12, shape = "record" ];

8: edge [ fontname = "Microsoft YaHei", fontsize = 12 ];

9:
```

```
10:     subgraph cluster_sl{
11:         label="IDP支持层";
12:         bgcolor="mintcream";
13:         node [shape="Mrecord", color="skyblue", style="filled"];
14:         network_mgr [label="网络管理器"];
15:         log_mgr [label="日志管理器"];
16:         module_mgr [label="模块管理器"];
17:         conf_mgr [label="配置管理器"];
18:         db_mgr [label="数据库管理器"];
19:     };
20:
21:     subgraph cluster_md{
22:         label="可插拔模块集";
23:         bgcolor="lightcyan";
24:         node [color="chartreuse2", style="filled"];
25:         mod_dev [label="开发支持模块"];
26:         mod_dm [label="数据建模模块"];
```

```
27:         mod_dp [label="部署发布模块"];

28:     };

29:

30: mod_dp -> mod_dev [label="依赖..."];

31: mod_dp -> mod_dm [label="依赖..."];

32: mod_dp -> module_mgr [label="安装...", color="yellowgreen", arrowhead="none"];

33: mod_dev -> mod_dm [label="依赖..."];

34: mod_dev -> module_mgr [label="安装...", color="yellowgreen", arrowhead="none"];

35: mod_dm -> module_mgr [label="安装...", color="yellowgreen", arrowhead="none"];

36:

37: }

38:
```

状态图

有限自动机示意图



上图是一个简易有限自动机，接受a及a结尾的任意长度的串。其脚本定义如下：

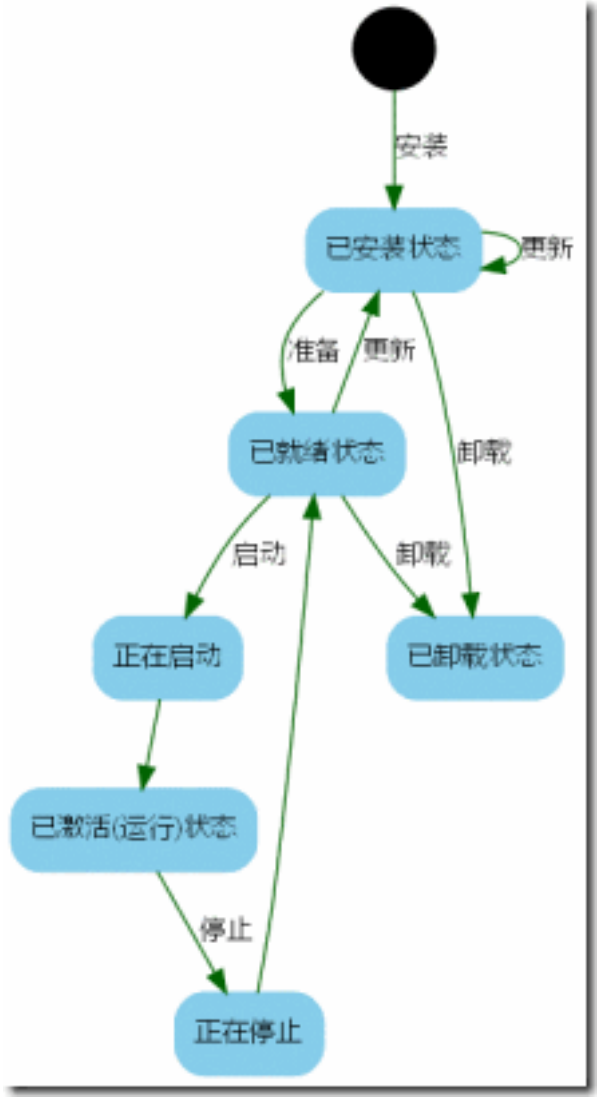
```
digraph automata_0 {

}

size = "8.5, 11";

fontname = "Microsoft YaHei";

fontsize = 10;

node [shape = circle, fontname = "Microsoft YaHei", fontsize = 10];

edge [fontname = "Microsoft YaHei", fontsize = 10];

0 [ style = filled, color=lightgrey ];

2 [ shape = doublecircle ];
```

```
12:

13: 0 -> 2 [ label = "a " ];

14: 0 -> 1 [ label = "other " ];

15: 1 -> 2 [ label = "a " ];

16: 1 -> 1 [ label = "other " ];

17: 2 -> 2 [ label = "a " ];

18: 2 -> 1 [ label = "other " ];

19:

20: "Machine: a" [ shape = plaintext ];

21: }
```

形状值为plaintext的表示不用绘制边框，仅展示纯文本内容，这个在绘图中，绘制指示性的文本时很有用，如上图中的”Machine: a”。

OSGi中模块的生命周期图

OSGi中，模块具有生命周期，从安装到卸载，可能的状态具有已安装，已就绪，正在启动，已启动，正在停止，已卸载等。如下图所示：



对应的脚本如下：

```
1: digraph module_lc{

2:

3: rankdir=TB;

4: fontname = "Microsoft YaHei";

5: fontsize = 12;

6:

7: node [fontname = "Microsoft YaHei", fontsize = 12, shape = "Mrecord", color="skybl

8: edge [fontname = "Microsoft YaHei", fontsize = 12, color="darkgreen" ];

9:
```

```
10: installed [label="已安装状态"];

11: resolved [label="已就绪状态"];

12: uninstalled [label="已卸载状态"];

13: starting [label="正在启动"];

14: active [label="已激活(运行)状态"];

15: stopping [label="正在停止"];

16: start [label="", shape="circle", width=0.5, fixedsize=true, style="filled", color=

17:

18: start -> installed [label="安装"];

19: installed -> uninstalled [label="卸载"];

20: installed -> resolved [label="准备"];

21: installed -> installed [label="更新"];

22: resolved -> installed [label="更新"];

23: resolved -> uninstalled [label="卸载"];

24: resolved -> starting [label="启动"];

25: starting -> active [label=""];

26: active -> stopping [label="停止"];
```

```
27: stopping -> resolved [label=""];
```

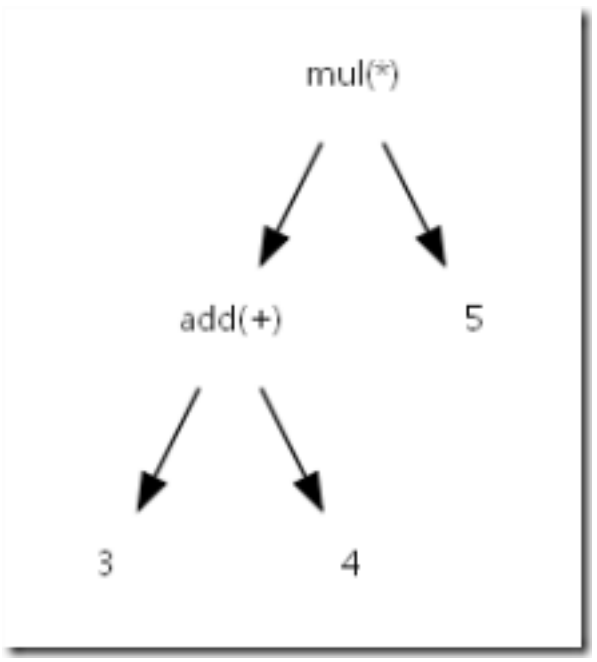
```
28:
```

```
29: }
```

其他实例

一棵简单的抽象语法树(AST)

表达式 (3+4)*5 在编译时期，会形成一棵语法树，一边在计算时，先计算3+4的值，最后与5相乘。



对应的脚本如下：

```
1: digraph ast{
```

```
2: fontname = "Microsoft YaHei";
```

```
3: fontsize = 10;
```

```
4:
```

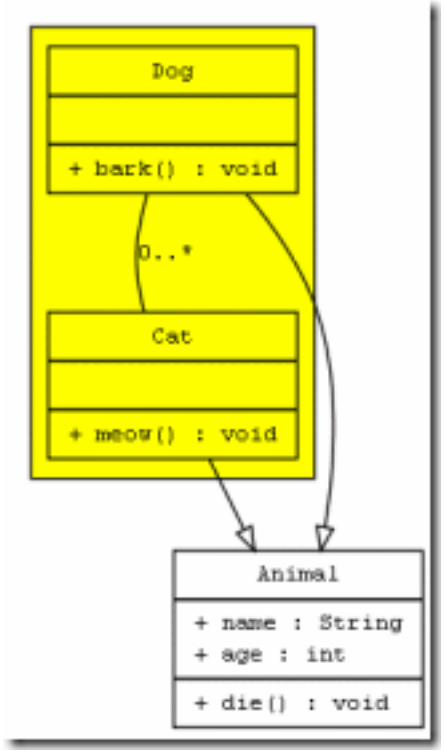
```
5: node [shape = circle, fontname = "Microsoft YaHei", fontsize = 10];
```

```
6: edge [fontname = "Microsoft YaHei", fontsize = 10];
```

```
 7: node [shape="plaintext"];

 8:

 9: mul [label="mul(*)"];

10: add [label="add(+)"];

11:

12: add -> 3

13: add -> 4;

14: mul -> add;

15: mul -> 5;

16: }

17:
```

简单的UML类图

下面是一简单的UML类图，Dog和Cat都是Animal的子类，Dog和Cat同属一个包，且有可能有联系(0..n)。

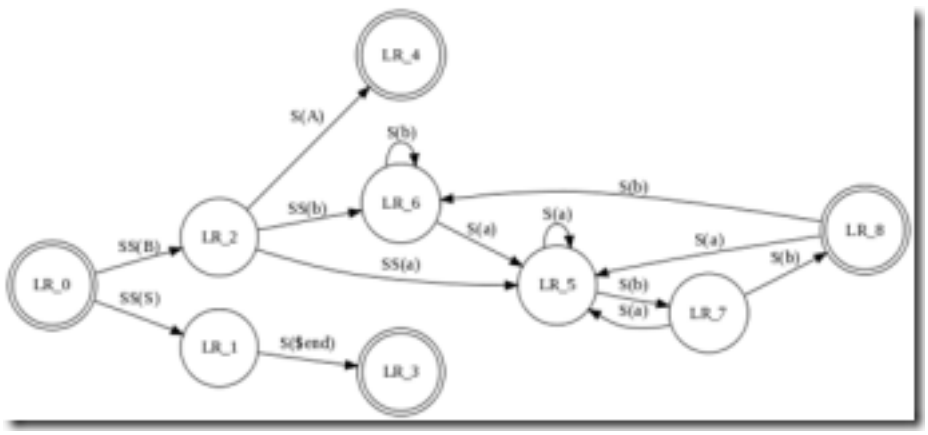


脚本：

```
1: digraph G{
2:
3: fontname = "Courier New"
4: fontsize = 10
5:
6: node [ fontname = "Courier New", fontsize = 10, shape = "record" ];
7: edge [ fontname = "Courier New", fontsize = 10 ];
8:
9: Animal [ label = "{Animal |+ name : String\l+ age : int\l|+ die() : void\l}" ];
10:
11:     subgraph clusterAnimalImpl{
```

```
12:         bgcolor="yellow"

13:         Dog [ label = "{Dog||+ bark() : voidl}" ];

14:         Cat [ label = "{Cat||+ meow() : voidl}" ];

15:     };

16:

17: edge [ arrowhead = "empty" ];

18:

19: Dog->Animal;

20: Cat->Animal;

21: Dog->Cat [arrowhead="none", label="0..*"];

22: }
```

状态图



脚本：

```
1: digraph finite_state_machine {

2:
```

```
 3: rankdir = LR;

 4: size = "8,5"

 5:

 6: node [shape = doublecircle];

 7:

 8: LR_0 LR_3 LR_4 LR_8;

 9:

10: node [shape = circle];

11:

12: LR_0 -> LR_2 [ label = "SS(B)" ];

13: LR_0 -> LR_1 [ label = "SS(S)" ];

14: LR_1 -> LR_3 [ label = "S($end)" ];

15: LR_2 -> LR_6 [ label = "SS(b)" ];

16: LR_2 -> LR_5 [ label = "SS(a)" ];

17: LR_2 -> LR_4 [ label = "S(A)" ];

18: LR_5 -> LR_7 [ label = "S(b)" ];

19: LR_5 -> LR_5 [ label = "S(a)" ];
```

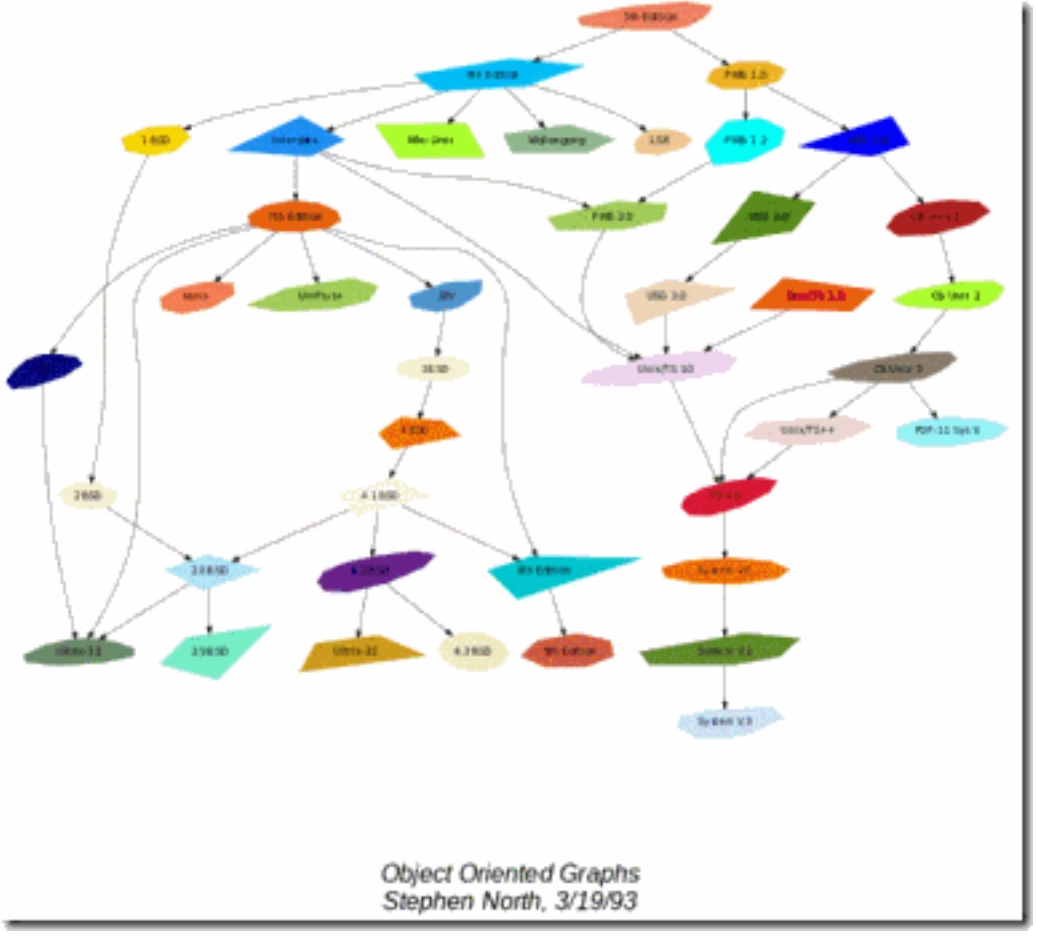
```
20: LR_6 -> LR_6 [ label = "S(b)" ];

21: LR_6 -> LR_5 [ label = "S(a)" ];

22: LR_7 -> LR_8 [ label = "S(b)" ];

23: LR_7 -> LR_5 [ label = "S(a)" ];

24: LR_8 -> LR_6 [ label = "S(b)" ];

25: LR_8 -> LR_5 [ label = "S(a)" ];

26:

27: }
```

附录

事实上，从dot的语法及上述的示例中，很容易看出，dot脚本很容易被其他语言生成。比如，使用一些简单的数据库查询就可以生成数据库中的ER图的dot脚本。

如果你追求高效的开发速度，并希望快速的将自己的想法**画**出来，那么graphviz是一个很不错的选择。

当然，graphviz也有一定的局限，比如绘制时序图(序列图)就很难实现。graphviz的节点出现在画布上的位置事实上是不确定的，依赖于所使用的布局算法，而不是在脚本中出现的位置，这可能使刚开始接触graphviz的开发人员有点不适应。graphviz的强项在于自动布局，当图中的顶点和边的数目变得很多的时候，才能很好的体会这一特性的好处：



比如上图，或者较上图更复杂的图，如果采用手工绘制显然是不可能的，只能通过graphviz提供的自动布局引擎来完成。如果仅用于展示模块间的关系，子模块与子模块间通信的方式，模块的逻辑位置等，graphviz完全可以胜任，但是如果图中对象的物理位置必须是准确的，如节点A必须位于左上角，节点B必须与A相邻等特性，使用graphviz则很难做到。毕竟，它的强项是自动布局，事实上，所有的节点对与布局引擎而言，权重在初始时都是相同的，只是在渲染之后，节点的大小，形状等特性才会影响权重。

本文只是初步介绍了graphviz的简单应用，如图的定义，顶点/边的属性定义，如果运行等，事实上还有很多的属性，如画布的大小，字体的选择，颜色列表等，大家可以通过graphviz的官网来找到更详细的资料。

Pop Jungle是我的新作，希望大家喜欢