构造正则表达式引擎

陈梓瀚
华南理工大学计算机软件学院软件工程 05 级本科
vczh@163.com
http://www.cppblog.com/vczh/
2008-5-22

注意：阅读本篇前需要阅读《**构造可配置词法分析器**》，并且最好具有扎实的数据结构基础。

一、问题概述

《**构造可配置词法分析器**》中说明了一个可配置的词法分析器所需要的理论知识，不过仅仅有这些知识还是不足以构造一个正则表达式引擎的。因为正则表达式引擎中有一些功能并不是附加了属性 DFA 所能够表达的。本篇文章主要解决三个问题：

- 如何分析正则表达式
- 如何构造一个基于 DFA 的纯匹配引擎
- 如何扩展 NFA 以便表达正则表达式的高级功能（预查、捕获等）

关于状态机的知识，本篇文章不再重复，请自行阅读《**构造可配置词法分析器**》。

二、正则表达式

现在的绝大多数正则表达式引擎是通过字符串来表达的。之前曾经看见一些在 C++中使用操作符重载以便达到在代码中直接书写正则表达式的正则表达式引擎。实际上这种方法不仅可以在编译的时候就能检查正则表达式是否正确，而且还能省掉处理正则表达式这一步骤。不过硬编码还是有硬编码的局限性的，因此在这里稍微介绍一下如何把一个字符串表达的正则表达式处理成我们所需要的结构。

在《**构造可配置词法分析器**》曾经提到正则表达式是由字符集合、串联、并联、可选以及重复组成的。但是为了使用的方便，流行的正则表达式都扩展出来非常多的语法。不过由于各家的正则表达式引擎都不尽相同，**所以本章并不能作为标准正则表达式教程**。但是本章仍然会提出一种实用的正则表达式语法，以便更容易地学习如何处理正则表达式。

正则表达式通常具有如下语法：

字符集合。字符集合有很多种。举个例子，A 就是一个字符集合，所描述的范围是只有一个字符“A”的集合。[a-z]也是一个字符集合，所描述的范围是所有小写字母。为什么这样写呢？因为小写字母的 ASCII 码（或者 Utf16 编码）是相连的，这样计算机看到的结果跟人看到的结果就一致了。[a-zA-Z0-9_]也是一个字符集合，描述了 C++所有能用来表示“名称”的字符。[^a-zA-Z]也是一个字符集合，描述了除了字母以外的所有字符。除此之外，还有很多转义集合（例如\d 或\x0041 等），在此不赘述。

并联串联。有了字符集合之后，我们就可以表达各种各样的词汇了。譬如我们使用[0-9][0-9][0-9]来表示三个数字构成的字符串，使用[0-9][0-9] | [a-z][a-z][a-z]来表达两个数字或

者三个小写字母构成的字符串，还有更复杂的譬如(a | b)(c | d)来表达集合{ac,ad,bc,bd}等。

重复。仅仅由并联串联是不够的，我们还可以将一个正则表达式进行“重复”以便表达更加复杂的集合。通常我们使用?来表达 0-1 次重复，*表达 0 次或以上重复，+表达 1 次或以上重复，{m}表达重复 m 次，{m,n}表达 m 次到 n 次重复，{m,}表达 m 次或以上重复。实际上?就是{0,1}，+就是{1,}，而*就是{0,}。每一个重复仅作用于在该重复左边最短的合法的正则表达式。譬如 ab+表示{ab,abb,abbb,…}，(ab)+则表达式{ab,abab,ababab,….}。

表达式引用。表达式引用可以让我们方便地表达出现很多次的复杂结构（并不是所有正则表达式引擎都支持，虽然实现起来并无难度）。举个例子，我们想使用正则表达式表达一个 IPv4 下的 IP 地址。我们知道 IP 地址的书写方法是 XXX.XXX.XXX.XXX，但是每一个 XXX 的范围只能是 0 到 255，因此我们可以把一个 XXX 分解成如下模式

- 25[0-5]
- 2[0-4][0-9]
- 1[0-9][0-9]
- [1-9][0-9]
- [0-9]

将它们并联起来我们得到：25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9]。显然，如果我们的正则表达式中需要书写 4 个这样长长的模式将会带来很多困扰。于是，这个时候表达式引用就大显身手了。我们为这个小表达式建立一个名字叫 sec，然后就可以将正则表达式写成：(?:<sec>25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9][0-9]|[0-9])(.(?:<sec>)){3}了。为表达式命名的语法是(?:<名字>表达式)，而使用命名的表达式的语法是(?:<名字>)，它们的区别在于有没有在括号内书写表达式。当然，重复定义是不对的，引用无定义的表达式也是不对的，而且在一个命名表达式内部使用自己也是不对的（这是上下文无关文法的能力范围）。

在本文中，仅仅由以上语法构造的正则表达式称为**纯正则表达式**。因为这种表达式还没超出**编译原理**中定义的 Type-3 文法的表达能力。下面还要介绍一些非常实用但是超出了 Type-3 的语法，在本文中称为**扩展正则表达式**。

正向预查。正向预查的语法是(?=纯正则表达式)。代表这个地方一定要匹配所包含的表达式，但是在匹配结束之后指针并没有继续往后走。譬如我们需要在文章中搜索所有在 98 和 2000 前面的“Windows”，我们使用正则表达式 Windows(?=98|2000)来完成这个任务。

反向预查。与正向预查相反，反向预查搜索所有不匹配的东西。譬如我们需要在文章中搜索所有**不在** 98 和 2000 前面的“Windows”，我们使用 Windows(?!98|2000)来完成这个任务。这个很容易记住，因为 C++中代表“否”用的刚好就是感叹号。嘿嘿。

匿名捕获。匿名捕获的语法是(?正则表达式)，如果一个正则表达式的匹配路径走过这条路的话，那么括号内所匹配的字符串将会被装进一个列表里面。举个例子，我们需要把字符串 123,456,78,90 中的所有数字都拿出来，我们使用(?[0-9]+)(.(?[0-9]+))*来完成这个任务。如果不想捕获的话，无论使用(表达式)或者是(?:表达式)都是可以的。不过(?表达式)还有其他作用。

命名捕获。在(?表达式)或(?:表达式)内我们都可以使用表达式引用、命名捕获和命名检

查三种功能。其中命名捕获跟匿名捕获类似，它将匹配到的字符串装进一个命了名的列表里面。不过现在流行的正则表达式引擎的做法是命名捕获之能保存最后一个捕获。命名捕获的语法是<#名字>表达式。譬如我们想获得 email 地址的“@”两边的内容并装进两个不同的列表里面，我们使用(?:<#user>[a-zA-Z0-9_-.]+)@(?:<#host>[a-zA-Z0-9_-.]+)来完成这个任务。当这个正则表达式匹配了一个 email 地址之后，会分别把用户名和服务器装进 user 和 host 两个列表里面。

命名检查。命名检查在于检查接下来的字符串的前缀或全部是否跟捕获的字符串一致。命名检查的语法是<$名字>。举个例子，我们想检查一个字符串的“.”两边是否一致，譬如 ab.ab、123.123、+-*/.+-*/等，我们可以使用(?:<#part>[^.]+).(?:<$part>)来完成这个任务。当字符串顺利到达“.”的时候，前边的部分就被放进了 part 表中。接下来他会检查接下来字符串是否跟表 part 中所保留的捕获一致。如果是 ab.cd 的话，part={ab}，因此匹配 cd 的时候就会失败。如果是 cab.abc 的话就会成功，而且能匹配到 ab.ab。

边界。我们使用^代表字符串开头，使用$代表字符串结尾。因此在上面的例子中，如果我们想匹配整个字符串而不是一部分的话，我们使用^(?:<#part>[^.]+).(?:<$part>)$来完成这个任务。这样的话 ab.ab 就会成功匹配，而 cab.abc 就不会成功匹配了（而不是找到子串 ab.ab）。如果我们想表达”^”这个字符怎么办呢？答案是使用转义\^。正则表达式的转义字符很多，不过这个跟实现是没什么关系的，到时候特殊处理一下便是。

非贪婪重复。在一个重复后加“?”代表让前面的重复次数尽可能少。

我们需要在内存中表达一个正则表达式。通过上面的分析我们可以看出，正则表达式的结构是一棵树。我们如何表达树呢？XML？万能 struct？使用了多态的 class 族？这些都不是问题。反正只要是树就好了。不过个人观点是，使用了多态的 class 族写起来会比较舒服。这里可以参考**设计模式**中的 **Visitor 模式**。

我们讲上面所讲的所有语法进行整理，我们可以得到以下几种元素以及塔门各自所需要的数据：

- 字符集合 ：一个范围的列表
- 串联 ：子树列表
- 并联 ：子树列表
- 重复 ：最小次数，最大次数，是否无限
- 左边界
- 右边界
- 功能 ：子树
  功能（正常、匿名捕获、正向预查、反向预查）
  描述（表达式命名、表达式引用、命名捕获、命名检查）
  名字

为了更加直观的描述上面的抽象的作用，我们讲一个复杂的正则表达式变成跟上面的描述一致的树。假设我们需要分析一个 HTML 的标记。为了简化起见，我们让 HTML 的属性值必须是双引号的字符串，而且内部不能有引号。这样的话我们可以把正则表达式写成：

\(?:<#markup>\w+)(\s+(?:<#attribute>\w+)=(?:<#value>”[^”]+”))?)*\)

这里\w 代表[a-zA-Z0-9_]，\s 代表空格。\(和\)代表括号。这个表达式由如下部分组成

- \(
- (?:<#markup>\w+)
- (\s+(?:<#attribute>\w+)=(?:<#value>”[^”]+”))*
- \)

我们拿着条表达式去分析<img src=”c:\X.jpg” alt=”unavailable picture”>的时候，我们捕获到的数据就是

markup={img}

attribute={src , alt}

value={“c:\X.jpg” , “unavailable picture”}

在这里需要再次提醒的就是，不是所有的正则表达式都支持命名捕获，并且可以捕获一个列表。因此这个表达式在很多流行的正则表达式引擎里是没用的。

那么，我们可以根据上面的描述，使用下面这棵树来表示这个长长的正则表达式：

如何把一个表达正则表达式的字符串转换成上面那棵树呢？这对于某些人来说可能很简单，对于另外一些人来说可能很复杂。不过不用怕，回想一下我们小时候总是会写的那种分析带括号和单目操作符的四则运算式子的程序就知道，其实两种分析实际上是差不多的。用一个操作符堆栈和一个操作数堆栈就可以分析一个四则运算式子。现在我们把四则运算式子变成正则表达式，操作符稍微改一下，操作数换成上面那棵树的节点。就可以用同样的方法解决这个问题了。

不过这个问题还有一种通用的解决办法，可以参考**编译原理**中**上下文无关文法分析**的两个知识点：

- 消除左递归
- 递归下降分析法

这两种方法结合起来非常适合**手写**分析器。关于语法分析的问题我可能会在下一篇文章中详细描述。

三、匹配纯正则表达式

在讲述如何匹配纯正则表达式之前，我们首先需要了解一下状态机是如何表示的。我们知道一个状态机是由边和状态构成的。边所需要的数据有三种，第一种是匹配的内容，第二种是执行边所需要的附加动作（在纯正则表达式中不需要动作），第三种就是边的起始和终结两个状态。状态所需要的数据有两种，第一种是状态的类型（是否终结状态），第二种是从这个状态出去和进到这个状态中的两个边的列表。

我们可以做出如下定义：

```
struct Status
{
	vector<Edge*> InEdges;
	vector<Edge*> OutEdges;
	bool FinalStatus;
};

struct Edge
{
	XXX MatchContent;
	vector<YYY> Actions;
	Status* Start;
	Status* End;
};

vector<Status*> AllStatuses;
vector<Edge*> AllEdges;
```

AllStatuses 和 AllEdges 在我自己的实现里面换成了两个对象池。独立出两个列表的好处

是便于内存的管理。至于 Edge 中的 XXX 和 YYY 两个类型就根据实际需要定义了。不过一般来说 XXX 描述的信息包含字符范围、左边界、右边界和命名检查的列表名字等。而且在实际的应用中，知道一个 DFA 状态是由哪几个 NFA 状态组成有时候也是很有用的（譬如在做词法分析器的时候）。这样的话就需要变通一下了。因此在实际的开发中，我们很有必要使用 boost 里的图或者自己开发一个状态机的模板库来实现这些数据结构。因为各种状态机的结构都是一样的，但是每一种状态机里面的数据都会多多少少有些区别。

我们有了正则表达式和状态机在内存中的表示方法之后，根据《**构造可配置词法分析器**》我们可以得到一个表达这个**纯正则表达式**的 DFA 了。扩展正则表达式的匹配方法在下一章中描述。

我们现在需要考虑的一个问题就是，我们使用的字符串包含的字符是很多的。使用现在流行的 wchar_t 来保存的话，那么一共有 65535 个字符。而且 c#和 java 等语言字符串的内部表示也是 Utf16 的。这样就带来一个问题，生成的 DFA 可能会很大。那么我们如何更改《**构造可配置词法分析器**》的 DFA 生成方法来构造更小的更快的 DFA 呢？

我们考虑一下平时书写的正则表达式，我们会发现在大部分情况下只要是数字，不管是 0 还是 9，其意义都是一样的。这样我们可以得到一个启发：使用字符范围而不是字符来描述 DFA 的转换。

考虑一下如下正则表达式：value\s+\w+|list(\s+\w+)*。我们在将这个正则表达式转换成树之后，找到所有**字符集合**节点，可以得到这个正则表达式的字符集合一共有 v、a、l、u、e、i、s、t、\s 和\w。这 10 个集合我们命名为 Ai(i=1..10)。然后我们要构造集合 Bi 使得 a 和 b 满足以下条件：

对于所有的 Ai 和 Bj，Ai 与 Bj 的交集要么是 Bj 要么是空集。

每一个 Bj 都必须是连续的。也就是说对于每一个 Bj 的最大值和最小值 min 和 max，只要一个字符 c 满足 min<=c<=max，那么 c 属于 Bj。

根据这些条件，我们可以得到很多种 Bj 的解。于是我们取一个 Bj 的数量最少的解，可以得到 Bj 为：

a、[b-d]、e、[f-h]、i、[j-k]、l、[m-r]、s、t、u、v、[w-z]、[A-Z]、[0-9]、_和\s

于是我们对这些集合进行编号（注意：\s 在很多种实现里面都是不连续的若干字符，为了方便才放在一起的）。可以得到如下表格：

	\s	_	0-9	A-Z	a	b-d	e	f-h	i	j-k	l	m-r	s	t	u	v	w-z
-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

到了这里，DFA 的边就可以用这些数字来表示输入的字符范围了。我们在分析的时候就可以构造一张转换表，通常是一个长度 65536（sizeof(wchar_t)）的数组 CharClass，其中 CharClass[0]=-2，其他字符保存上面那张表的结果。这样，如果我们所查到的一个字符的 CharClass 是-2 或者-1 的话，那么这个输入就无效了，因为不存在任何状态接受这些字符。利用 CharClass 我们可以用 O(1)的时间复杂度来得到一个字符所对应的种类编号。譬如说 CharClass[L'a']的结果是 4。不过在这里要注意的就是，不是所有的编译器都会把 wchar_t 当成 unsigned short 的。

将《**构造可配置词法分析器**》所描述的 DFA 生成算法和上面的 CharClass 两种方法结合

起来，我们可以得到正则表达式 value\s+\w+|list(\s+\w+)*的 DFA：

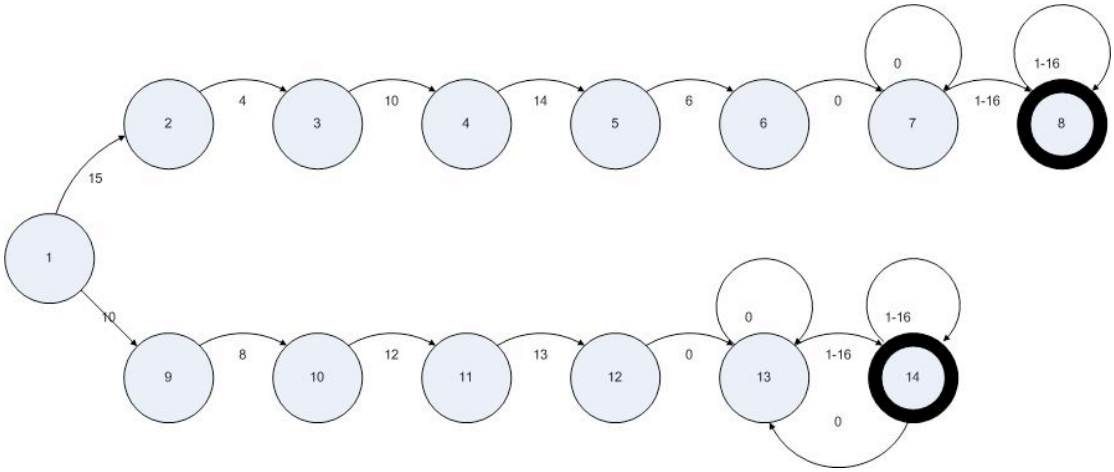

图 3-1

其中状态 1 是起始状态，状态 8 和 14 是终结状态，边上的数字是 CharClass 的结果。其中本来是\w 的边的数字是 1-16，代表从 1 到 16 都可以。生成了这个 DFA 之后，我们可以将这个 DFA 转换成 2 维数组，纵坐标是状态，横坐标是输入的字符的 CharClass：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1											9					2	
2					3												
3											4						
4															5		
5							6										
6	7																
7	7	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
[8]		8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
9									10								
10													11				
11														12			
12	13																
13	13	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
[14]	13	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14

表中[状态]是终结状态。

这样的话，我们从状态 1 开始。每一次输入一个字符都先查看 CharClass，然后在二维数组里面根据**状态**和 **CharClass** 查找转换后的新状态。这个时候可能有几种结果

- **CharClass 的结果是-1 或者-2**：检查是不是曾经到达过终结状态。是的话则用那个时候的字符指针作为匹配结果，否则匹配失败。
- **表中查找不到转换后的状态**：检查是不是曾经到达过终结状态。是的话则用那个时候的字符指针作为匹配结果，否则匹配失败。
- **成功转换后到达终结状态**：字符指针往后移 1 位，记录字符指针和状态。
- **成功转换后到达一般状态**：字符指针往后移 1 位。

现在我们有了 CharClass，有了 DFA 表格，有了匹配算法，我们可以大概写出一个在字符串中寻找第一个匹配的代码。在这里我们让初始状态为 0，并且表格中查找不到的状态都设置为-1：

参数：

CharClass ：CharClass 数组
DFA ：DFA 二维数组
Final ：长度为状态数量的数组，记录某个状态是否终结状态
Input ：输入的字符串
Begin ：成功匹配的起始位置，0 为失败
Length ：成功匹配的长度

```
int RunDFA(int* CharClass , int** DFA , bool* Final , wchar_t* Input)
{
    int LastFinalLength=-1;
    int Status=0;
    wchar_t* Init=Input;
    while(Status!=-1)
    {
        If(Final[Status])
        {
            LastFinalLength=Input-Init;
        }
        int cc=CharClass[*Input++];
        if(cc<0)break;
        Status=DFA[Status][cc];
    }
    return LastFinalLength;
}

void Match(
    int* CharClass , int** DFA , bool* Final , wchar_t* Input ,
    wchar_t** Begin ,int* Length)
{
    *Begin=0;
    while(*Input)
    {
        Int Len=RunDFA(CharClass,DFA,Final,Input);
        If(Len!=-1)
        {
            *Begin=Input;
            *Length=Len;
            return;
        }
        Input++;
    }
}
```

四、为扩展正则表达式构造 NFA

我在《**构造可配置词法分析器**》中曾经描述了一种构造 ε -NFA 的方法，可是这个方法并不支持扩展的正则表达式语法。所以在这里扩展一下构造 ε -NFA 的方法。所有的新边都不是 ε 边，在消除 ε 边算法那里需要全部保留。

现在需要添加若干种类型的边，在这里称为**功能边**。功能边用于在 NFA 中加入关于循环、捕获、预查和检查等的信息，然后还会提出一种 **NFA 压缩算法**整理这些边，以便让贪婪模型的匹配更加高速。在这里提一下，**在进行正则表达式匹配的时候，匹配的过程是很快的，匹配的过程产生的数据结构才要占用大量的时间。**所以建议不要用 STL 来处理这些极端的问题，最好为正则表达式开发专用的数据结构。

重复

根据上文，我们可以知道重复分贪婪和不贪婪两种。用一种直观的方法来表达，我们使用两个正则表达式来匹配一个字符串 12345

- (?[0-9]+) (?[0-9]+?)
- (?[0-9]+?) (?[0-9]+)

第一个正则表达式的捕获是{1234 , 5}，而第二个正则表达式的捕获是{1 , 2345}。这个区别是重复后面的“?”指定的。于是我们用来构造“重复”的 ε -NFA 的时候，就要注意**边在状态里面存放的顺序**了。这跟贪婪模型的匹配算法有关系。在这里不妨留意个悬念，读者看完之后可以回过头来想想这个方法之中蕴含的道理。

重复有两种，一种是有限的，一种是无限的。我们分别使用[0-9]{2,4}和[0-9]{2,}作为例子讲解如何生成新的 NFA。循环的头部和尾部需要功能边 loop/end 来标记。

[0-9]{2,4}:

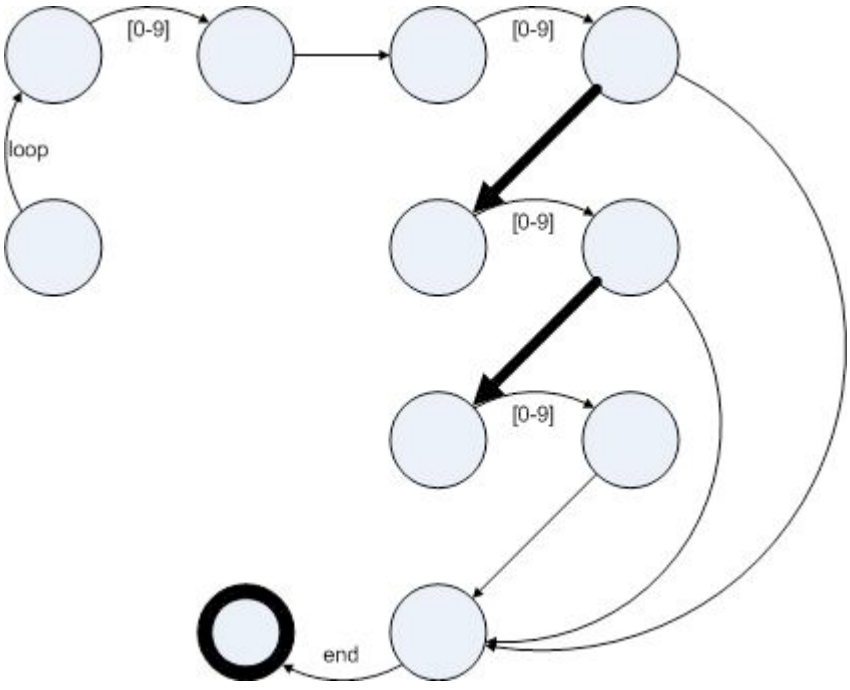

图 4-1，粗线为状态的输出边中位置靠前的边

[0-9]{2,4}?：

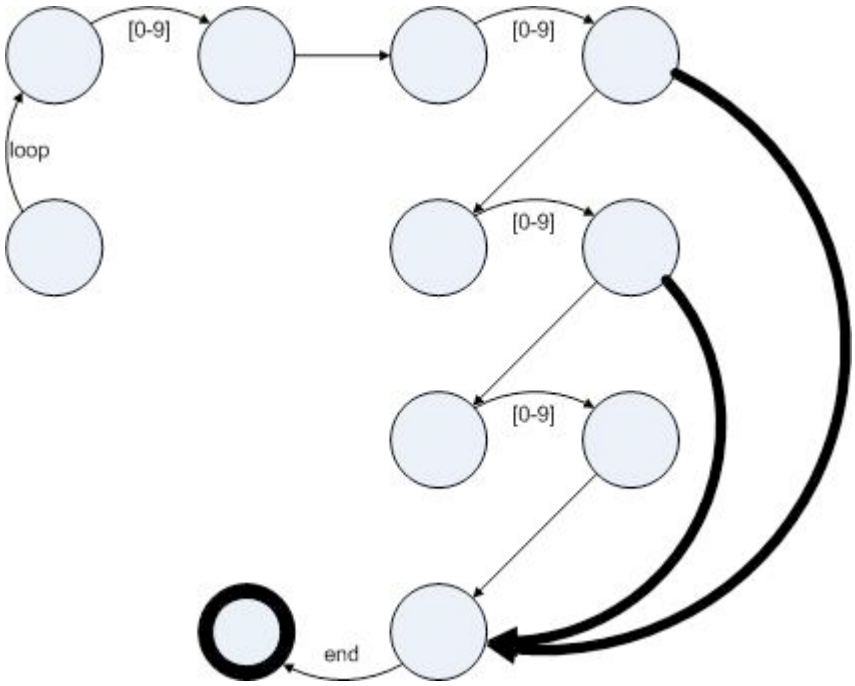

图 4-2，粗线为状态的输出边中位置靠前的边

[0-9]{2,}：

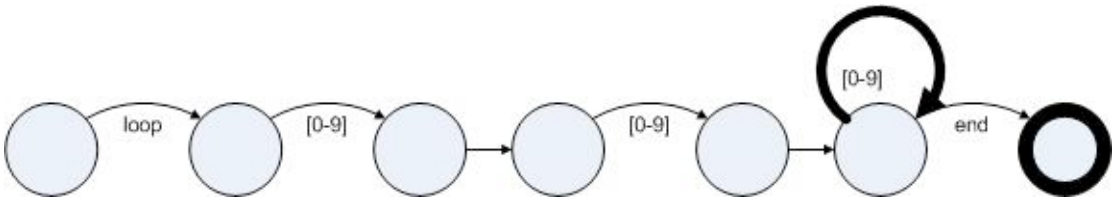

图 4-3，粗线为状态的输出边中位置靠前的边

[0-9]{2,}?：

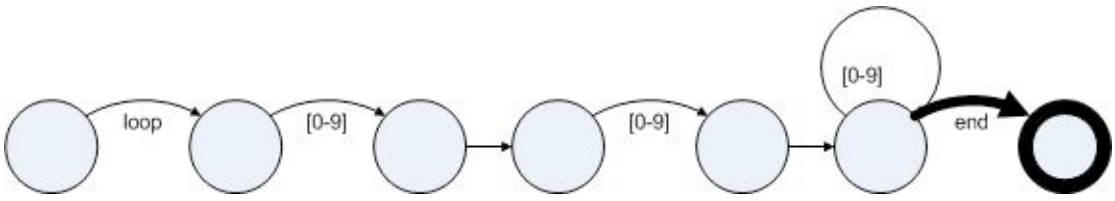

图 4-2，粗线为状态的输出边中位置靠前的边

正向预查/反向预查

正向预查和反向预查的 ε -NFA 引入了两种功能边，分别叫 Positive 和 Negative。这组功能边都有一个属性，指向预查用的纯正则表达式的 DFA。关于 DFA 的匹配请看第三章。

因为 Positive 和 Negative 边在输入的时候只有通过和不通过两种状态，不会影响其他状态，所以做了特殊处理。分析正则表达式的时候把所有的预查表达式都分离出来，做成一张

DFA 表。然后每一条 Positive 或 Negative 边都记录着一个指向表中 DFA 的索引。在匹配的时候就把当前的指针交给相应的 DFA 判断。因此 Positive 和 Negative 的 ε -NFA 构造起来非常简单。这里使用 a(?=b)|b(?!a)作为例子，这条正则表达式寻找的是 b 前面的 a 或者不在 a 前面的 b。在这里注意以下，使用贪婪模型的 ε -NFA，分支的顺序跟表达式的位置是一一对应的。

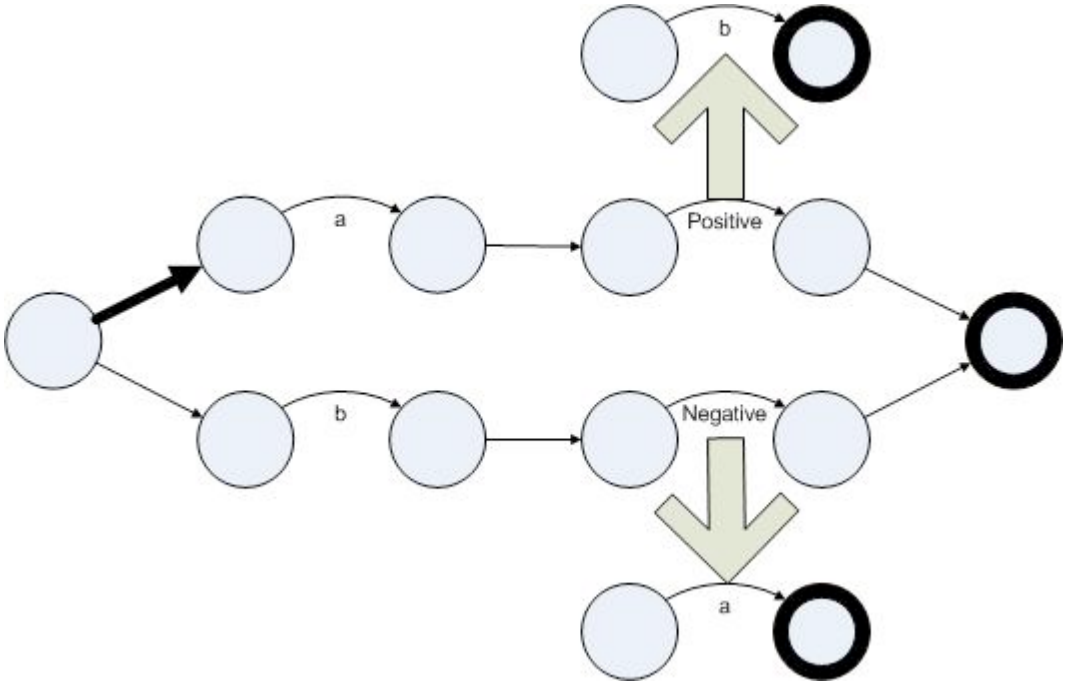

图 4-5，粗线为状态的输出边中位置靠前的边

匿名捕获/命名捕获

匿名捕获和命名捕获需要引入三种功能边，分别是 Capture、Storage 和 End。其中 Storage 记录了命名捕获的名字。举个例子，(?a)(?:<#b>c)的 ε -NFA 如下：

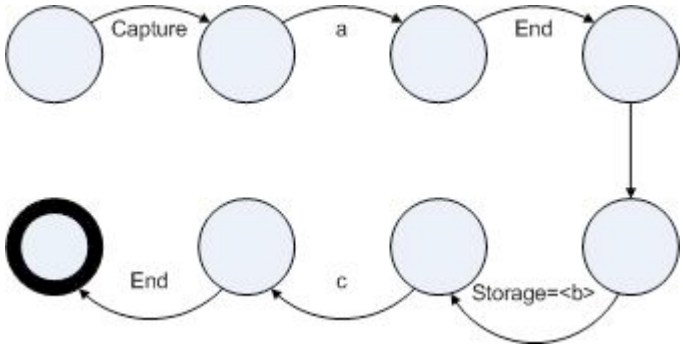

图 4-6

边界/命名检查

边界是一种输入位置的判断，需要 Head 和 Tail 两种功能边，但是其使用方法跟一般的字符集合没有区别，因此不再图示。命名检查需要一种 Check 功能边，这种边跟 Storage 一样记录了命名捕获的名字。使用方法同字符集合，不再图示。

五、NFA 压缩算法

在使用了《**构造可配置词法分析器**》中介绍的消除 ε 边算法消除了 ε 边之后，得到的 NFA 具有很多不改变输入的功能边。于是我自己想了一种办法（不知道前辈们是否也想过，没去考证），把功能边收集起来，存放在一个输入或 ε 边下面。为什么还有 ε 边呢？前面的消除 ε 边算法只是化简图，现在的 ε 边是为了匹配用的。因为类似 Check 的这种边是无法指定输入的，因为不知道那个时候究竟 Storage 了什么东西。

在介绍 NFA 压缩算法之前，首先简要地讲述一下 NFA 压缩算法的目的。NFA 压缩算法压出来的图主要是为了减少贪婪算法的回溯次数，并且在一次输入的时候可以尽可能的进行更多的运算。于是一种新的 NFA 就产生了。这种 NFA 图的边有以下属性：

- 输入（或者 ε ）
- 功能边列表
- 起始状态
- 终止状态

现在使用正则表达式[0-9]*?(?:<#sec>[0-9]+?)(?:<$sec>)+来说明 NFA 压缩算法的流程。这个正则表达式用于一串数字中的重复部分。输入 1234564567 的话 sec 将包含{456}，输入 123456789 的话匹配会失败。

使用上文描述的扩展过的 ε -NFA 生成算法，配合《**构造可配置词法分析器**》中介绍的消除 ε 边算法，我们可以得到如下 NFA 图：

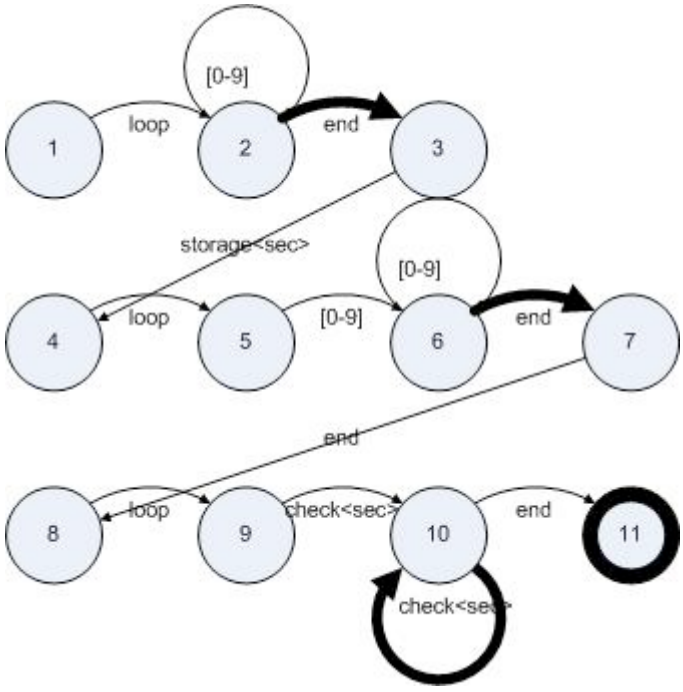

图 5-1

算法如下：

1、 标记出所有输入的边都没有消耗字符的状态，起始状态和终结状态除外。**消耗字符的边指的是字符集合以及命名检查。**

2、 遍历所有未标记的状态，寻找每一条不消耗字符的边，一直沿着边走，直到遇到消

耗字符的边为止。建立新边，标记输入为遇到的字符集合边的输入，将所有沿路遇到的功能边搜集起来放到新边的功能边列表里。**遍历一条不消耗字符的边构造新边的时候，有可能会遇到分叉，于是有可能构造多条新边。**寻找每一条消耗字符的边。将其复制到末尾变成新边。

3、 终结状态的输入状态的所有新输入边都添加一条新边到达终结状态，保持各自顺序。这些边需要分别加上输入状态到终结状态的不消耗字符的功能边。

4、 删除所有旧边以及标记的状态。

现在让我们一起对上面的 NFA 应用这个算法。

- **按照步骤 1 标记边：**

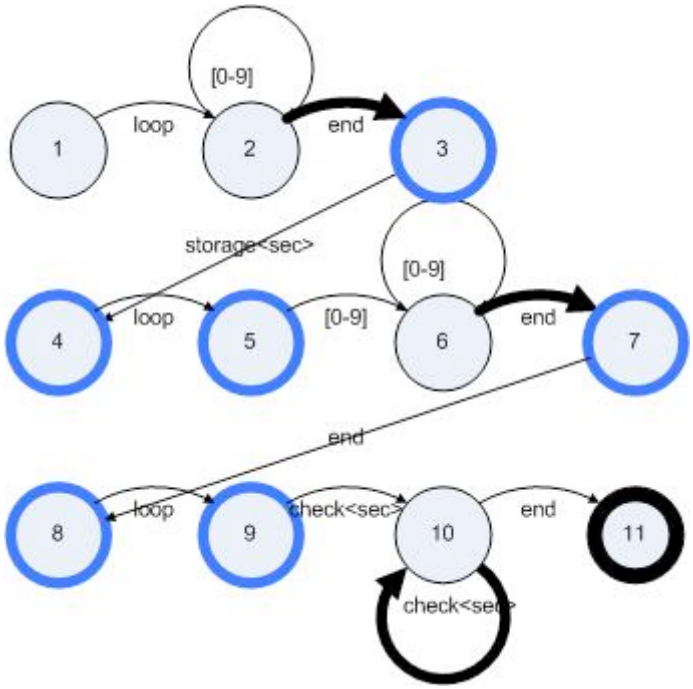

图 5-2

- **为第一个状态添加新边：**

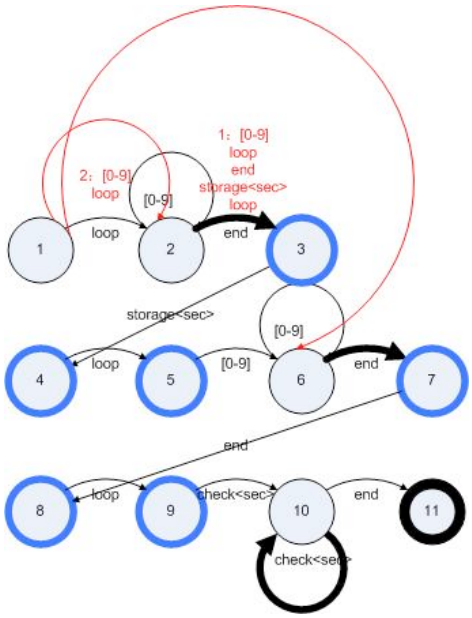

图 5-3：红色边为新边，标记的数字为边的顺序

状态已经过 loop 边到了状态 2，现在遇到分叉。分叉的第一条边是 end-3-storage-4-loop-5-[0-9]-6，因此添加一条边将 loop、end、storage 和 loop 都记载下来，输入使用最后一个路径[0-9]。同理，分叉的第二条是[0-9]=2，添加新边。

- **为所有非终结状态添加新边：**

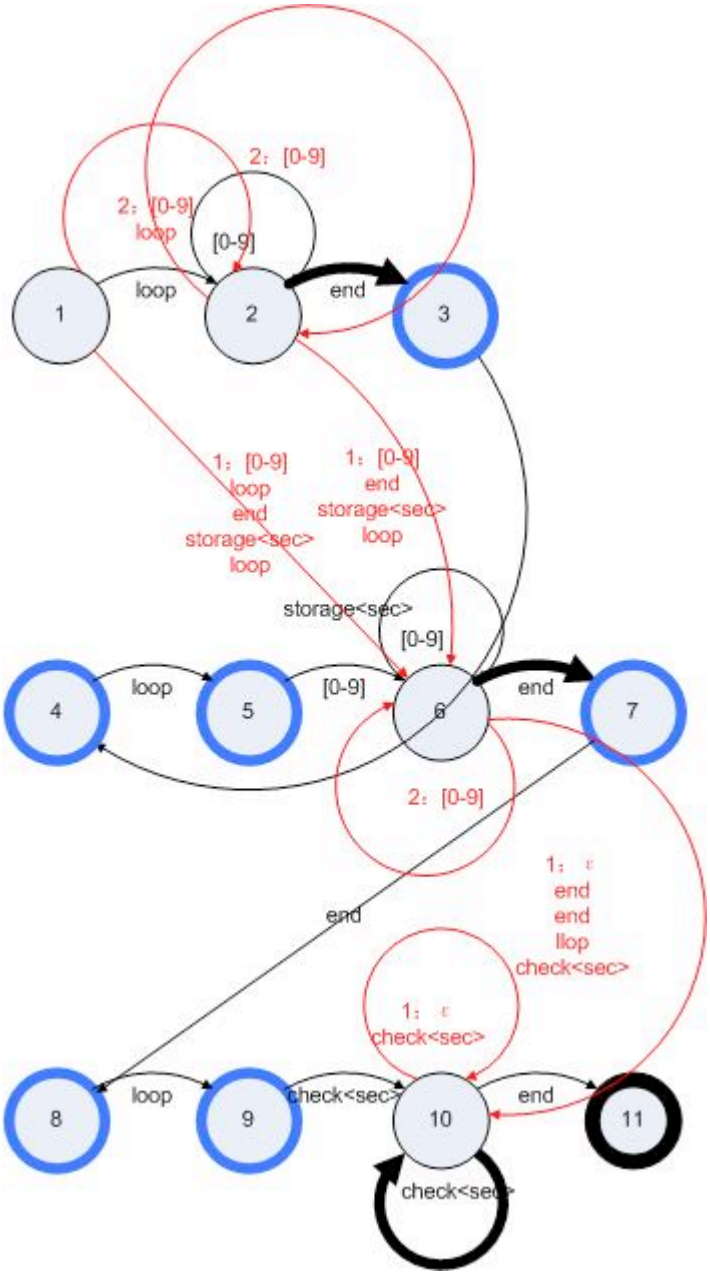

图 5-4

- **为终结状态添加新边：**

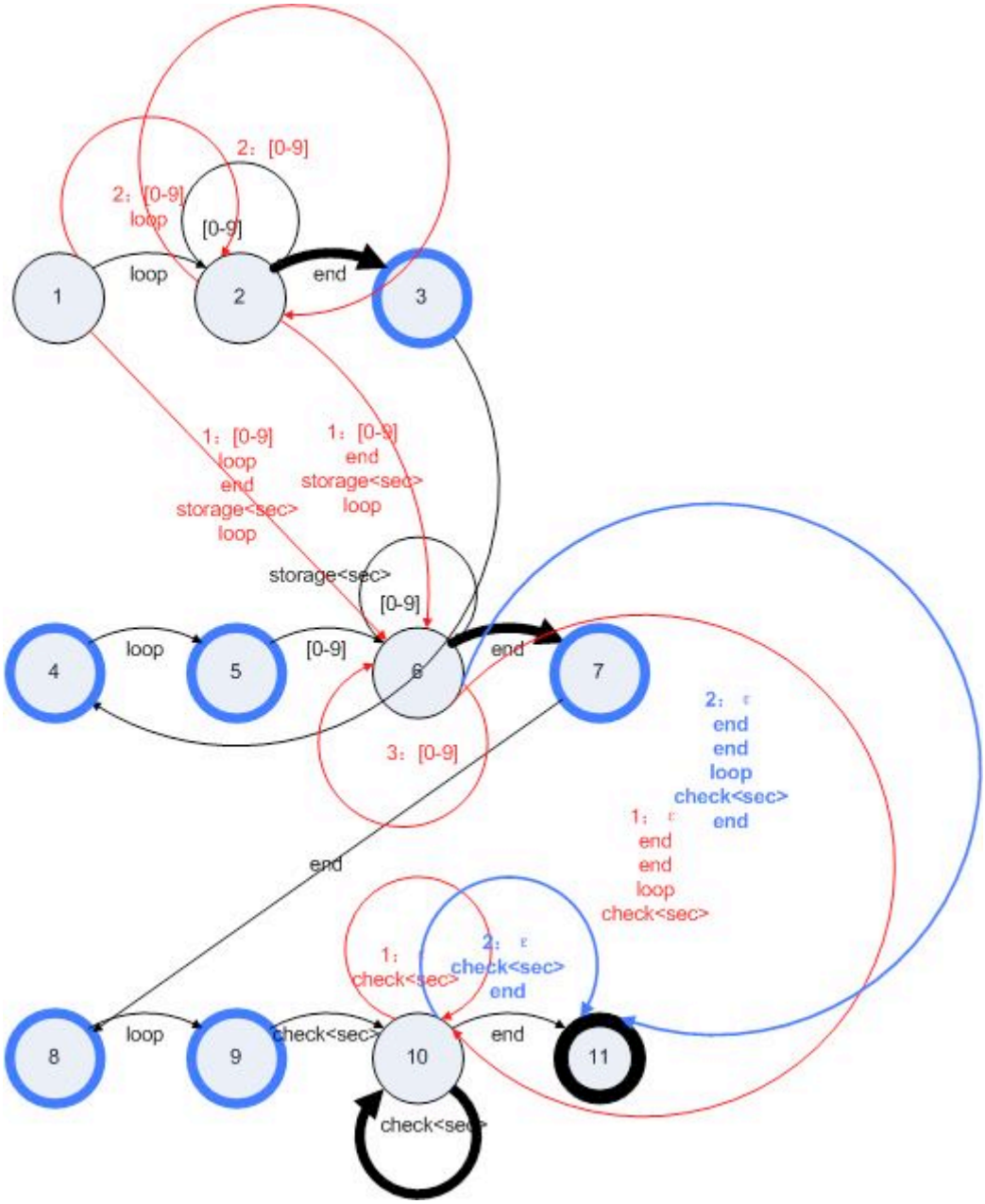

图 5-5：本步骤添加的边使用蓝色

这里需要注意的是，复制的新边在加上 10 到 11 的 end 功能边之后都放置在紧接着被复制的边的位置上。**如果 10 到 11 有不同的不消耗字符的边的话，那么就要相应的复制多几次。**

• 删除无用状态：

到了这里，所有的新边就都添加完毕了。现在删除掉旧边和被标记的状态，然后就可以得到一个被压缩过的 NFA 了。这个 NFA 将直接用来参与贪婪模型的匹配。

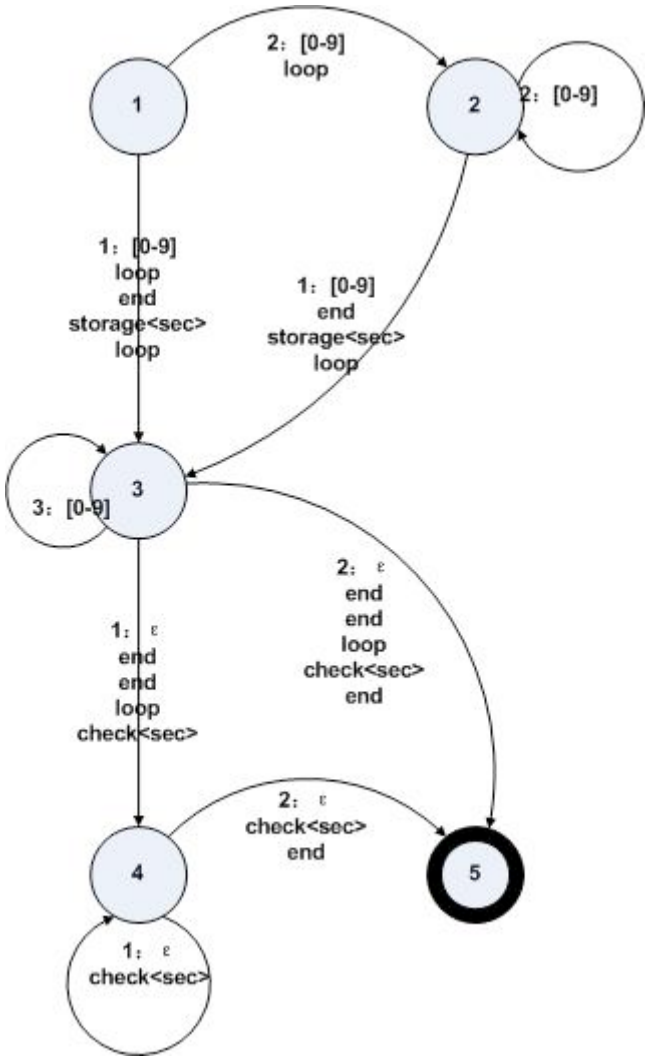

图 5-6

执行完这些步骤之后，一个带有功能边的 NFA 就压缩完毕了。那么，我们开始使用贪婪模型匹配字符串吧。

六、贪婪匹配算法

在执行贪婪算法之前，我们先回顾一下我们使用了多少功能边：

- head/tail/check
- loop/capture/storage/end
- positive/negative

这些功能边分成两种。第一种匹配输入或者输入位置，第二种修改结果所保存的信息。在匹配的时候，我们需要如下数据结构：

1、 捕获堆栈

每一个捕获堆栈的元素具有以下属性：

- 名称（包括匿名）
- 产生这个捕获元素的时候状态堆栈的深度

- 捕获的起始位置
- 捕获的长度

2、 命令堆栈

每一个命令堆栈的元素具有以下属性：

- 功能边
- 参数
- 前一个未终结指令在堆栈中的位置
- 产生这个命令的字符串起始位置

3、 状态堆栈

每一个状态堆栈的元素具有以下属性：

- NFA 状态
- 当前边序号
- 状态产生的时候命令堆栈的深度
- 字符串的起始位置

匹配的时候使用如下方法：

1、 三个堆栈清空，状态堆栈推入 NFA 的初始状态，边序号-1。

2、 在状态非空的时候

- 获得状态堆栈的栈顶，从那个状态的边序号+1 开始搜索
- 检查边的输入，如果字符集合通过或者该边为 ε 边则**尝试**执行功能边列表

  Head ：如果当前位置不在字符串的最前则失败

  Tail ：如果当前位置不再字符串的末尾则失败

  Positive ：执行 DFA，不通过则失败

  Negative ：执行 DFA，通过则失败

  Loop ：命令堆栈推入 Loop，准备好属性

  Storage ：命令堆栈推入 Storage，准备好属性

  Capture ：命令堆栈推入 Capture，准备好属性

  End ：查看前一个未终结指令，Loop 什么都不做推入 End，Storage/Capture 则往捕获堆栈中添加对象，命令堆栈推入 End。准备好属性

  Check ：搜索捕获堆栈中所有相同名字的捕获，一一检查，通过则字符串位置往后挪，全部不通过则失败。

- 如果执行列表失败则**恢复所有状态**并搜索下一条边
- 所有边都失败则将当前堆栈状态的栈顶弹出，**恢复上一个栈顶的所有状态**。
- 如果成功获取一条边，则将字符串位置挪动 1（ ε 边除外）并计算这条边的目标状态。推入状态堆栈并准备好所有属性。如果目标状态是终结状态的话则停止匹配，返回匹配结果

3、 如果从来没有到达过终结状态则返回匹配失败。

我们以 12323 为例，使用上文压缩过的 NFA 来匹配字符串。

初始状态：（尖括号的属性值顺序与上文对堆栈元素的介绍一致）

- Storage={}

- Command={}
- Status={<1,-1,0,”12323”>}

输入|12323
- Storage={}
- Command={loop,end,storage<sec,-1,”12323”>,loop}
- Status={<1,1,0,”12323”>,<3,-1,4,”2323”>}

输入 1|2323
边 1 和边 2 在执行了 end 获得 storage 之后均通不过 check，所以走边 3
- Storage={}
- Command={loop,end,storage<sec,-1,”12323”>,loop }
- Status={<1,1,0,”12323”>,<3,3,4,”2323”>,<3,-1,4,”323”>}

输入 12|323
边 1 和边 2 在执行了 end 获得 storage 之后均通不过 check，所以走边 3
- Storage={}
- Command={loop,end,storage<sec,-1,”12323”>,loop }
- Status={<1,1,0,”12323”>,<3,3,4,”2323”>,<3,3,4,”323”>,<3,-1,4,”23”>}

输入 123|23
边 1 和边 2 在执行了 end 获得 3torage 之后均通不过 check，所以走边 3
- Storage={}
- Command={loop,end,storage<sec,-1,”12323”>,loop }
- Status={<1,1,0,”12323”>,<3,3,4,”2323”>,<3,3,4,”323”>,<3,3,4,”23”>,<3,-1,4,”3”>}

输入 1232|3
边 1 和边 2 在执行了 end 获得 storage 之后均通不过 check，所以走边 3
- Storage={}
- Command={loop,end,storage<sec,-1,”12323”>,loop }
- Status={<1,1,0,”12323”>,<3,3,4,”2323”>,<3,3,4,”323”>,<3,3,4,”23”>,<3,3,4,”3”>,<3,3,4,””>}

输入 12323|
无法继续走，因为边 1 和边 2 在执行了 end 获得 3torage 之后均通不过 check，并且边 3 需要的[0-9]已经没有输入的字符可以满足，回溯
- Storage={}
- Command={loop,end,storage<sec,-1,”12323”>,loop }
- Status={<1,1,0,”12323”>,<3,3,4,”2323”>,<3,3,4,”323”>,<3,3,4,”23”>,<3,3,4,”3”>}

原因同上，回溯
- Storage={}
- Command={loop,end,storage<sec,-1,”12323”>,loop }
- Status={<1,1,0,”12323”>,<3,3,4,”2323”>,<3,3,4,”323”>,<3,3,4,”23”>}

原因同上，回溯
- Storage={}
- Command={loop,end,storage<sec,-1,”12323”>,loop }
- Status={<1,1,0,”12323”>,<3,3,4,”2323”>,<3,3,4,”323”>}

原因同上，回溯
- Storage={}
- Command={loop,end,storage<sec,-1,”12323”>,loop }
- Status={<1,1,0,”12323”>,<3,3,4,”2323”>}

原因同上，回溯（状态栈顶的有效命令堆栈深度是 1，因此 Command 堆栈恢复状态）
- Storage={}
- Command={}
- Status={<1,1,0,”12323”>}

现在的状态堆栈的边是 1，因为被回溯，所以尝试走 2
输入|12323
- Storage={}
- Command={loop}
- Status={<1,2,0,”12323”>,<2,-1,1,”2323”>}

输入 1|2323
- Storage={}
- Command={loop,end,storage<sec,-1,”2323”>,loop }
- Status={<1,2,0,”12323”>,<2,1,1,”2323”>,<3,-1,4,”323”>}

输入 12|323
边 1 和边 2 在执行了 end 获得 storage”2”之后均通不过 check，所以走边 3
- Storage={}
- Command={loop,end,storage<sec,-1,”2323”>,loop }
- Status={<1,2,0,”12323”>,<2,1,1,”2323”>,<3,3,4,”323”>,<3,-1,4,”23”>}

输入 123|23
边 1 在执行了 end 获得 storage”23”之后，通过了 check 的检查，走边 1
- Storage={<sec,5,”2323”,2>}
- Command={loop,end,storage<sec,-1,”2323”>,loop,end,end}
- Status={<1,2,0,”12323”>,<2,1,1,”2323”>,<3,3,4,”323”>,<3,1,4,”23”>,<5,-1,6,””>}

5 是最终状态，结束匹配。
- 状态堆栈的栈顶是<5,-1,6,””>，输入是”12345”，所以匹配到的字符串是”12345”
- 捕获堆栈是<sec,5,”2323”,2>，从”2323”开始获得 2 个字符，结果是
  匿名捕获：{}
  命名捕获：{<sec,{“23”}>}

使用正则表达式[0-9]*?(?:<#sec>[0-9]+?)(?:<$sec>)+对字符串”12323”进行匹配的所有工作已经完成。现在回顾一下执行贪婪模型所需要的工作：

1、 分析正则表达式
2、 构造 ε -NFA
3、 执行消除 ε 边算法
4、 执行 NFA 压缩算法
5、 使用贪婪模型，配合压缩过的 NFA 对字符串进行匹配。

七、尾声

事实上要写一个正则表达式引擎也是一件不太容易的事情。为什么我在文章中不贴代码呢，因为代码实在是非常的长，而且我也不知道读者喜欢或者认识的语言是什么。不过在我个人看来，能够用来写正则表达式引擎的无非只有 C/C++、Delphi 和 Haskell。

根据前几天博客上读者们反应来看，大部分人还是比较信赖那些出名的正则表达式引擎的。不过就算自己不打算开发正则表达式引擎，知道正则表达式背后的理论和技术也是相当重要的。因为这是写出好的正则表达式（表达式，不是引擎）所必需知道的事实。

自从半年前《**构造可配置词法分析器**》写完之后，我收到了很多读者的来信，不过基本上都是要求阅读源代码以便“学习学习”。可能那篇文章还没把事情说透，因此我撰写了《**构造正则表达式引擎**》。本篇文章描述了一种使用二维数组处理 Utf16 输入的 DFA 的方法，描述了高级的正则表达式所需要的更加复杂的 NFA 结构，描述了如何使用扩展的 NFA 匹配字符串。不过这种扩展过的 NFA 理论上已经不属于 NFA 了，无奈找不到名字，只好这么叫着。至于如何实现 NFA 呢？这个就没有偷懒的方法了，用有向图吧。什么是有向图？请参阅《**算法导论**》。

在此我稍微说一说我认为合适的学习编程的方法：
1、 找到对自己有难度的问题
2、 在不查阅资料的情况下编写程序解决问题
3、 查阅资料并重构或重写
4、 查阅别人的代码并重构或重写
5、 根据实际要求重构或重写

这样做不仅可以练习编程，而且还能加深对问题的了解，增强学习能力，唯一的缺点是需要花大量的时间。不过我认为花时间还是值得的。话说，当时我在做正则表达式引擎和文法匹配的时候，也是重写过的。

参考文献

【1】: Alfred V.Aho　Ravi Sethi　Jeffrey D.Ullman ，Compilers Principles , Techniques , and Tools
【2】: Dick Grune　Ceriel Jacobs ， Parsing Techniques ---- A Practical Guide
【3】: Kenneth C.Louden ， Compiler Construction Principles and Practice