构造可配置词法分析器

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/
2007-11-8

一、问题概述

随着计算机语言的结构越来越复杂，为了开发优秀的编译器，人们已经渐渐感到将词法分析独立出来做研究的重要性。不过词法分析器的作用却不限于此。回想一下我们的老师刚刚开始向我们讲述程序设计的时候，总是会出一道题目：给出一个填入了四则运算式子的字符串，写程序计算该式子的结果。除此之外，我们有时候建立了比较复杂的配置文件，譬如 XML 的时候，分析器首先也要对该文件进行词法分析，把整个字符串断成了一个一个比较短小的记号(指的是具有某种属性的字符串)，之后才进行结构上的分析。再者，在实现某种控制台应用程序的时候，程序需要分析用户打进屏幕的命令。如果该命令足够复杂的话，我们也首先要对这个命令进行词法分析，之后得到的结果会大大方便进行接下去的工作。

当然，这些问题大部分已经得到了解决，而且历史上也有人做出了各种各样专门的或者通用的工具(Lex、正则表达式引擎等)来解决这一类问题。我们在使用这种工具的时候，为了更加高效地书写配置，或者我们在某种特殊情况下需要自己制作类似的工具，就需要了解词法分析背后的原理。本文将给出一个构造通用词法分析工具所需要的原理。由于实现的代码过长，本文将不附带实现。

究竟什么是“把一个字符串断成一些记号”呢？我们先从四则运算式子入手。一个四则运算式子是一个字符数列，可是我们关心的对象实际上是操作符、括号和数字。于是此法分析的作用就是把一个字符串断开成我们关心的带有属性的记号。举个例子：(11+22)*(33+44)是一个合法的四则运算式子，如果输入是(左括号,”(“) (数字,”11”) (一级操作符,”+”) (数字,”22”) (右括号,”)”) (二级操作符,”*”) (左括号,”(“) (数字,”33”) (一级操作符,”+”) (数字,”44”) (右括号,”)”)的话，我们在检查结构的时候只需要关心这个记号的属性(也就是左括号、右括号、数字、操作符等)就行了，具体计算的时候才需要关心这个记号实际上的内容。如果式子里边有空格的话，我们也仅仅需要把空格当成是一种记号类型，在词法分析得出结果之后，将具有空格属性的记号丢弃掉就可以了，接下去的步骤不需变化。

但需要注意的是，词法分析得到的结果是没有层次结构的，所有的记号都是等价的对象。我们在计算表达式的时候把+和*看成了不同层次的操作符，类似的结构是具有嵌套的层次的。词法分析不能得出嵌套层次结构的信息，最多只能得到关于重复结构的信息。

二、正则表达式

我们现在需要寻找一种可以描述记号类型的工具，在此之前我们首先研究一下常见的记号的结构。为了表示出具有某种共性的字符串的集合，我们需要书写出一些能代表字符串集合的规则。这个集合中的所有成员都将被认为是一种特定类型的记号。

首先，规则可以把一个特定的字符或者是空字符串认为是一种类型的记号的全部。上文所说到的四则运算式子的例子，“左括号”这种类型的记号就仅仅对应着字符”(“，其他的字符或者字符串都不可能是“左括号”这个类型的记号。

其次，规则可以进行串联。串联的意思是这样的，我们可以让一个字符串的前缀符合某一个指定的规则，剩下的部分的前缀符合第二个规则，剩下的部分的前缀符合第三个规则等等，一直到最后一个部分的全部要符合最后一个规则。如果我们把”function”这个字符串作为一个记号类型来处理的话，我们可以把”function”这个字符串替换成 8 个串联的规则：”f”,”u”,”n”,”c”,”t”,”i”,”o”,”n”。首先，字符串”function”的前缀”f”符合规则”f”，剩下的部分”unction”的前缀”u”符合规则”u”，等等，一直到最后一个部分”n”的全部符合规则”n”。

第三，规则可以进行并联。并联的意思就是，如果一个字符串符合一系列规则中的其中一个的话，我们就说这个字符串符合这一些规则的并联。于是这些规则的并联就构成了一个新的规则。一个典型的例子就是判断一个字符串是否关键字。关键字可以是”if”，可以是”else”，可以是”while”等等。当然，一个关键字是不可能同时符合这些规则的，不过只要一个字符串符合这些规则的其中一个的话，我们就说这个字符串是关键字。于是，关键字这个规则就是”if”、”else”、”while”等规则的并联。

第四，一个规则可以是可选的。可选的规则实际上是属于并联的一种特殊形式。加入我们需要规则”abc”和”abcde”并联，我们会发现这两个规则有着相同的前缀”abc”，而且这个前缀恰好就是其中的一个规则。于是我们可以把规则改写成”abc”与””和”de”的并联的串联。但是规则””指定的规则是空串，因此这个规则与”de”的并联就可以看成是一个可选的规则”de”。

第五，规则可以被重复。有限次的重复可以使用串联表示，但是如果我们不想限制重复的次数的话，串联就没法表示这个规则了，于是我们引入了“重复”。一个典型的例子就是程序设计语言的标识符。标识符可以是一个变量的名字或者是其他东西。一门语言通常没有规定变量名的最大长度。因此为了表示这个规则，就需要将 52 个字母进行并联，然后对这个规则进行重复。

上述的 5 种构造规则的方法中，后面的 4 个方法被用于把规则组合成为更大的规则。为了给出这种规则的形式化表示，我们引入了一种范式。这种范式有以下语法：

1：字符用双引号包围起来，空串使用 ε 代替。
2：两个规则头尾连接代表规则的串联。
3：两个规则使用 | 隔开代表规则的并联。
4：规则使用[]包围代表该规则是可选的，规则使用{}包围代表该规则是重复的。
5：规则使用()包围代表该规则是一个整体，通常用于改变操作符 | 的优先级。

举个例子，一个实数的规则书写如下：
{“0”|”1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9”}”.”[{“0”|”1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9”}]。

但是，我们如何表示“不是数字的其他字符呢”？字符的数量是有限的，因此我们可以使用规则的并联来表示。但是所有的字符实在是太多（ASCII 字符集有 127 个字符，UTF-16

字符集有 65535 个字符），因此后来人们想出了各种各样的简化规则书写的办法。比较著名的有 BNF 范式。BNF 范式经常被用于理论研究，但是更加实用的是正则表达式。

正则表达式的字符不需要用双引号括起来，但是如果需要表示一些被定义了的字符(如“|”)的话，就使用转义字符的方法表示(如 “\|”)。其次，X?代表[X]，X+代表{X}，X*代表[{X}]。字符集合可以用区间来表示，[0-9]可以表示“0”|”1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9”，[^0-9]则表示“除了数字以外的其他字符”。正则表达式还有各种各样的其他规则来简化我们的书写，不过由于本文并不是“精通正则表达式”，因此我们只保留若干比较本质的操作来进行词法分析原理的描述。

正则表达式的表达能力极强，小数的规则可以使用[0-9]+.[0-9]来表示，C 语言的注释可以表示为/*([^*]|*+[^*/])**+/来表示。

三、有穷状态自动机

人阅读正则表达式会比较简单，但是机器阅读正则表达式就是一件非常困难的事情了。而且，直接使用正则表达式进行匹配配的话，不仅工作量大，而且速度缓慢。因此我们还需要另外一种专门为机器设计的表达方式。本文在以后的章节中会给出一种算法把正则表达式转换为机器可以阅读的形式，就是这一章节所描述的有穷状态自动机。

有穷状态自动机这个名字听起来比较可怕，不过实际上这种自动机并没有想象中的那么复杂。状态机的这种概念被广泛的应用在各种各样的领域中。软件工程的统一建模语言(UML)有状态图，数字逻辑中也有状态转移图。不过这些各种各样的图在本质上都跟状态机没有什么区别。我将会通过一个例子来讲述状态的实际意义。

假设我们现在需要检查一个字符串中 a 的数量和 b 的数量是否都是偶数。当然我们可以用一个正则表达式来描述它。不过对于这个问题来说，用正则表达式来描述远远不如构造状态机方便。我们可以设计出一个状态的集合，然后指定集合中的某一个元素为“起始状态”。其实状态就是在工作还没开始的时候，分析器所处的状态。分析器在每一次进行一项新的工作的时候，都要把状态重置为起始状态。分析器每读入一个字符就修改一次状态，修改的方法我们也可以指定。分析器在读完所有的字符以后，必然停留在一个确定的状态中。如果这个状态跟我们所期望的状态一致的话，我们就说这个分析器接受了这个字符串，否则我们就说这个分析器拒绝了这个字符串。

如何通过设计状态及其转移方法来实现一个分析器呢？当然，如果一个字符串仅仅包含 a 或者 b 的话，那么分析器的状态只有四种：“奇数 a 奇数 b”、“奇数 a 偶数 b”、“偶数 a 奇数 b”、“偶数 a 偶数 b”。我们把这些状态依次命名为 aa、aB、Ab、AB。大写代表偶数，小写代表奇数。当工作还没开始的时候，分析器已经读入的字符串是空串，那么理所当然的起始状态应当是 AB。当分析器读完所有字符的时候，我们期望读入的字符串的 a 和 b 的数量都是偶数，那么结束的状态也应该是 AB。于是我们给出这样的一个状态图：

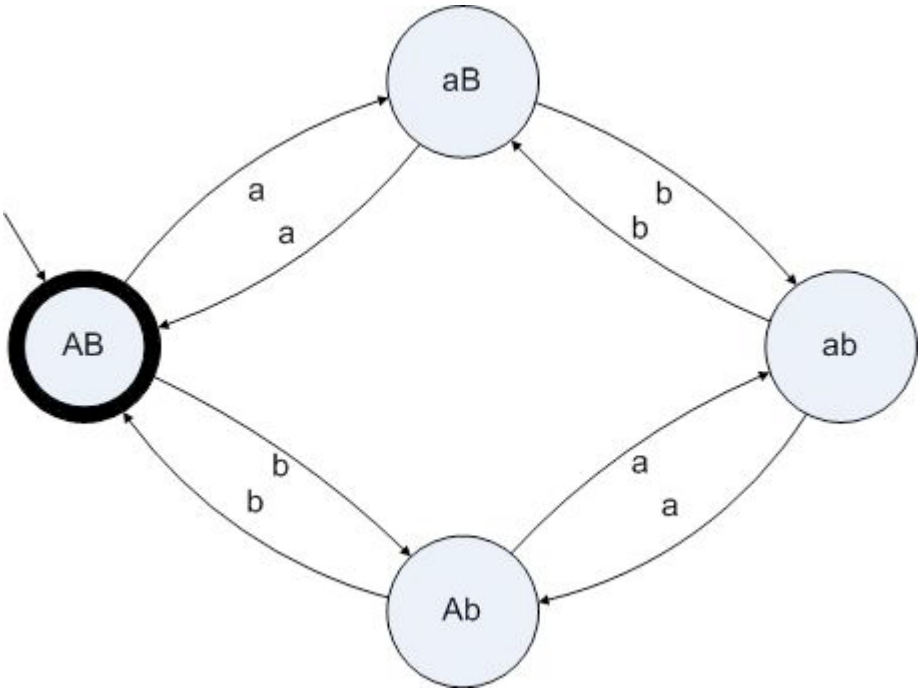

图 3.1
检查一个字符串是否由偶数个 a 和偶数个 b 组成的状态图

在这个状态图里，有一个短小的箭头指向了 AB，代表 AB 这个状态是初始状态。AB 状态有粗的边缘，代表 AB 这个状态是结束的可接受状态。一个状态图的结束状态可以是一个或者多个。在这个例子里，起始状态和结束状态刚好是同一个状态。标有字符”a”的箭头从 AB 指向 aB，代表如果分析器处于状态 AB 并且读入的字符是 a 的话，就转移到状态 aB 上。

我们把这个状态图应用在两个字符串上，分别是”abaabbba”和”aababbaba”。其中，第一个字符串是可以接受的，第二个字符串是不可接受的(因为有 5 个 a 和 4 个 b)。

分析第一个字符串的时候，状态机所经过的状态是：
AB[a]aB[b]ab[a]Ab[a]ab[b]aB[b]ab[b]aB[a]**AB**
分析第二个字符串的时候，状态机所经过的状态是：
AB[a]aB[a]**AB**[b]Ab[a]ab[b]aB[b]ab[a]Ab[b]**AB**[a]aB

第一个字符串”abaabbba”让状态机在状态 AB 上停了下来，于是这个字符串是可以接受的。第二个字符串”aababbaba”让状态机在状态 aB 上停了下来，于是这个字符串是不可以接受的。

在机器内部表示这个状态图的话，我们可以使用一种比较简单的方法。这种方法仅仅把状态与状态之间的箭头、起始状态和结束状态集合记录下来。对应于这个状态图的话，我们就可以把这个状态图表示成以下形式：

起始状态：AB
结束状态集合：AB

(AB,a,aB)
(AB,b,Ab)
(aB,a,AB)
(aB,b,ab)
(Ab,a,ab)
(Ab,b,AB)
(ab,a,Ab)
(ab,b,aB)

用一个状态图来表示状态机的时候有时候会遇到确定性与非确定性的问题。所谓的确定性就是指对于任何一个状态，输入一个字符都可以跳转到另一个确定的状态中去。确定性和非确定性的区别有一个直观的描述：状态图的任何一个状态都可以有不定数量的边指向另一个状态，如果在这些边里面，存在两条边，它们所承载的字符如果相同，那么这个状态输入这个就字符可以跳转到另外两个状态中去，于是该状态机就是不确定的。如图所示：

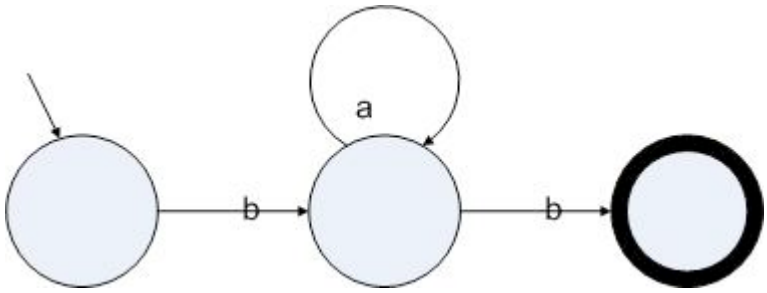

图 3.2

正则表达式 ba*b 的一个确定的状态机表示

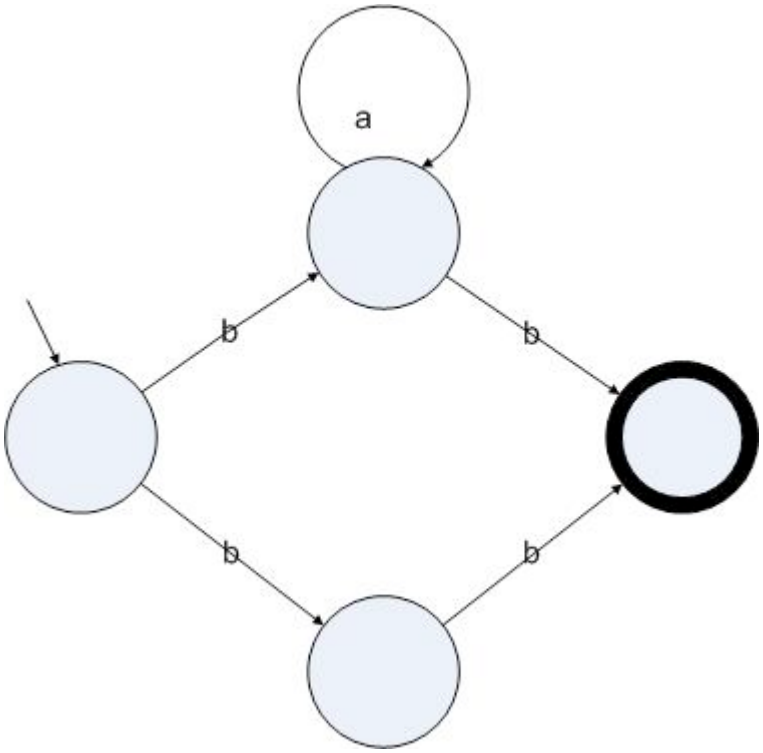

图 3.3

正则表达式 ba*b 的一个非确定的状态机表示

图 3.3 中的状态机的起始状态读入字符 b 后可以跳转到中间的两个状态里，因此这个状态机是非确定的。相反，图 3.2 中的状态机，虽然功能跟图 3.3 的状态机一致，但却是确定的。我们还可以使用一种特殊的边来进行状态的转换。我们用 ε 边来表示一个状态可以**不读入字符就跳转到另一个状态**上。下图给出了一个跟图 3.3 功能一致的包含 ε 边的非确定的状态机：

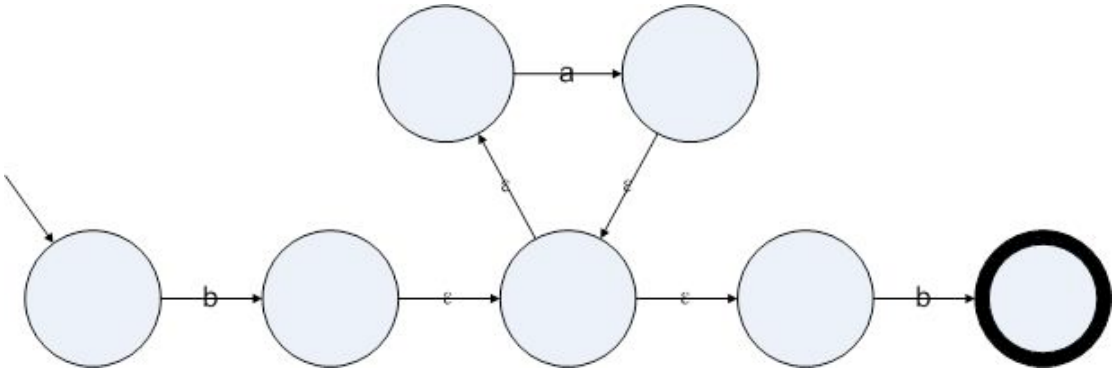

图 3.4

正则表达式 ba*b 的一个带有 ε 边的非确定的状态机

在教科书中，通常把确定的有穷状态自动机(有穷状态自动机也就是本文讨论的这种状态机)称为 DFA，把非确定的有穷状态自动机称为 NFA，把带有 ε 边的非确定的状态机称为 ε -NFA。下文中也将采用这几个术语来指示各种类型的有穷状态自动机。

在刚刚接触到 ε 边的时候，一个通常的疑问就是这种边存在的理由。事实上如果是人直接画状态机的话，有时候也可以直接画出一个确定的状态机，复杂一点的话也可以画出一个非确定的状态机，在有些极端的情况下我们需要使用 ε 边来更加简洁的表示我们的意图。不过 ε 边存在的最大的理由就是：我们可以通过使用 ε 边来给出一个简洁的算法把一个正则表达式转换成 ε -NFA。

四、从正则表达式到 ε -NFA

通过第二节所描述的内容，我们知道一个正则表达式的基本元素就是字符集。通过对规则的串联、并联、重复、可选等操作，我们可以构造除更复杂的正则表达式。如果从正则表达式构造状态机的时候也可以用这几种操作对状态图进行组合的话，那么方法将会变得很简单。接下来我们将一一对这 5 个构造正则表达式的方法进行讨论。使用下文描述的算法构造出来的所有 ε -NFA 都**有且只有一个结束状态**。

1：字符集

字符集是正则表达式最基本的元素，因此反映到状态图上，字符集也会是构成状态图的基本元素。对于字符集 C，如果有一个规则只接受 C 的话，这个规则对应的状态图将会被构造成以下形式：

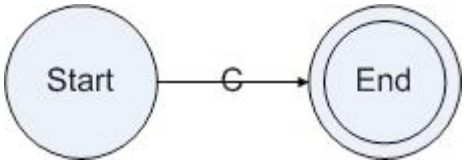

图 4.1

这个状态图的初始状态是 Start，结束状态是 End。Start 状态读入字符集 C 跳转到 End 状态，不接受其他字符集。

2：串联

如果我们使用 A⊙B 表示规则 A 和规则 B 的串联，我们可以很容易的知道串联这个操作具有结合性，也就是说(A⊙B)⊙C=A⊙(B⊙C)。因此对于 n 个规则的串联，我们只需要先将前 n-1 个规则进行串连，然后把得到的规则看成一个整体，跟最后一个规则进行串联，那么就得到了所有规则的串联。如果我们知道如何将两个规则串联起来的话，也就等于知道了如何把 n 个规则进行串联。

为了将两个串联的规则转换成一个状态图，我们只需要先将这两个规则转换成状态图，然后让第一个状态的结束状态跳转到第二个状态图的起始状态。这种跳转必须是不读入字符的跳转，也就是令这两个状态等价。因此，第一个状态图跳转到了结束状态的时候，就可以当成第二个状态图的起始状态，继续第二个规则的检查。因此我们使用了 ε 边连接两个状态图：

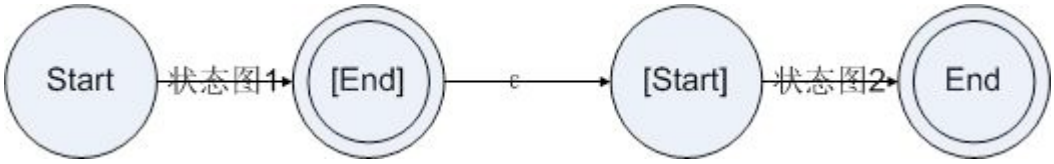

图 4.2

3：并联

并联的方法跟串联类似。为了可以在起始状态读入一个字符的时候就知道这个字符可能走的是并联的哪一些分支并进行跳转，我们需要先把所有分支的状态图构造出来，然后把起始状态连接到所有分支的起始状态上。而且，在某个分支成功接受了一段字符串之后，为了让那个状态图的结束状态反映在整个状态图的结束状态上，我们也把所有分支的结束状态都连接到大规则的结束状态上。如下所示：

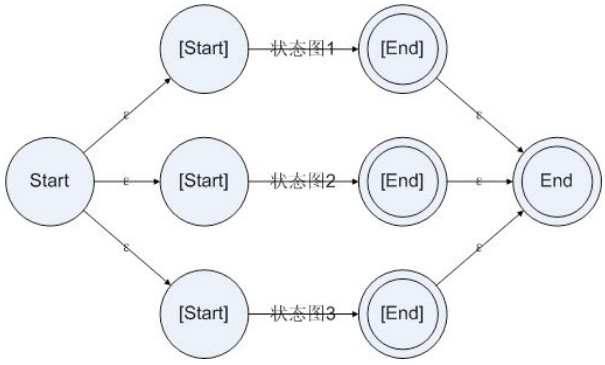

图 4.3

4：重复

对于一个重复，我们可以设立两个状态。第一个状态是起始状态，第二个状态是结束状态。当状态走到结束状态的时候，如果遇到一个可以让规则接受的字符串，则再次回到结束状态。这样的话就可以用一个状态图来表示重复了。于是对于重复，我们可以构造状态图如下所示：

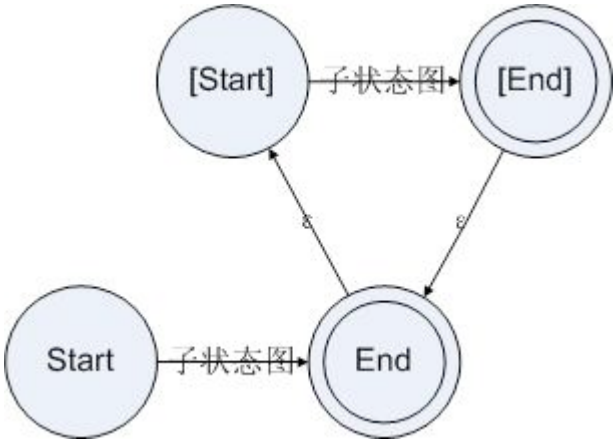

图 4.4

5：可选

为可选操作建立状态图比较简单。为了完成可选操作，我们需要在接受一个字符的时候，如果字符串的前缀被当前规则接受则走当前规则的状态图，如果可选规则的后续规则接受了字符串则走后续规则的状态图，如果都接受的话就两个图都要走。为了达到这个目的，我们把规则的状态图的起始状态和结束状态连接起来，得到了如下状态图：

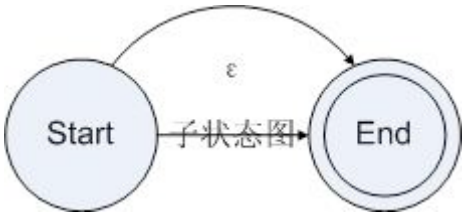

图 4.5

如果重复使用的是 0 次以上重复，也就是原来的重复加上可选的结果，那么可以简单地把图 4.4 的 Start 状态去掉，让 End 状态同时拥有起始状态和结束状态两个角色，[Start]和[End]则保持原状。

至此，我们已经将 5 种构造状态图的办法都对应到了 5 种构造规则的办法上了。对于任意的一个正则表达式，我们仅需要把这个表达式还原成那 5 种构造的嵌套，然后把每一步构造都对应到一个状态图的构造上，就可以将一个正则表达式转换成一个 ε -NFA 了。举个例子，我们使用正则表达式来表达“一个字符串仅包含偶数个 a 和偶数个 b”，然后把它转换成 ε -NFA。

我们先对这个问题进行分析。如果一个字符串仅包含偶数个 a 和偶数个 b 的话，那么这个字符串一定是偶数长度的。于是我们可以把这个字符串分割成两个两个的字符段。而且这些字符段只有四种：aa、bb、ab 和 ba。对于 aa 和 bb 来说，无论出现多少次都不会影响字符串中 a 和 b 的数量的奇偶性(理由：在一个模 2 加法系统里，0 是不变项，也就是说对于任何属于模 2 加法的数 X 有 X+0 = 0+X = X)。对于 ab 和 ba 的话，如果一个字符串的开始和结束是 ab 或者 ba，中间的部分是 aa 或者 bb 的任意组合，这个字符串也是具有偶数个 a 和偶数个 b 的。我们现在得到了两种构造偶数个 a 和偶数个 b 的字符串的方法。把串联、并联、可选、重复等操作应用在这些字符串上，仍然会得到具有偶数个 a 和偶数个 b 的字符串。于是我们可以把正则表达式书写成以下形式：

((aa|bb)|((ab|ba)(aa|bb)*(ab|ba)))*

根据上文提到的方法，我们可以把这个正则表达式转换成以下状态机：

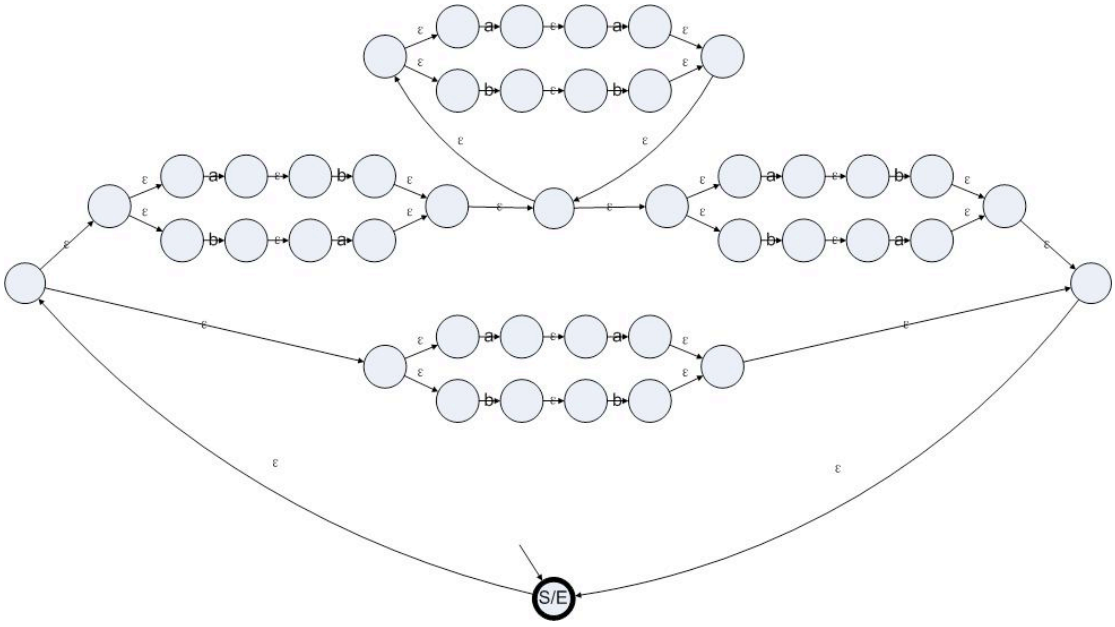

图 4.6

至此，我们已经得到了把一个正则表达式转换为 ε -NFA 的方法了。但是只得到 ε -NFA 还是不行的，因为 ε -NFA 的不确定性太大了，直接根据 ε -NFA 跑的话，每一次都会得到大量的临时状态集合，会极大地降低效率。因此，我们还需要一个办法消除一个状态机的非确定性。

五、消除非确定性

消除 ε 边算法

我们见到的有穷状态自动机一共有三种：ε -NFA、NFA 和 DFA。现在我们需要将 ε -NFA 转换为 DFA。一个 DFA 中不可能出现 ε 边，所以我们首先要消除 ε 边。消除 ε 边算法基于一个很简单的想法：如果状态 A 通过 ε 边到达状态 B 的话，那么状态 A 无需读入字符就可以直达状态 B。如果状态 B 需要读入字符 x 才可以到达状态 C 的话，那么状态 A 读入 x 也可以到达状态 C。因为从 A 到 C 的路径是 A B C，其中 A 到 B 不需要读入字符。

于是我们会得到一个很自然的想法：消除从状态 A 出发的 ε 边，只需要寻找所有从 A 开始仅通过 ε 边就可以到达的状态，并把从这些状态触出发的非 ε 边复制到 A 上即可。剩下的工作就是删除所有的 ε 边和一些因为消除 ε 边而变得不可到达的状态。为了更加形象地描述消除 ε 边算法，我们从正则表达式(ab|cd)*构造一个 ε -NFA，并在此状态机上应用消除 ε 边算法。

正则表达式(ab|cd)*的状态图如下所示：

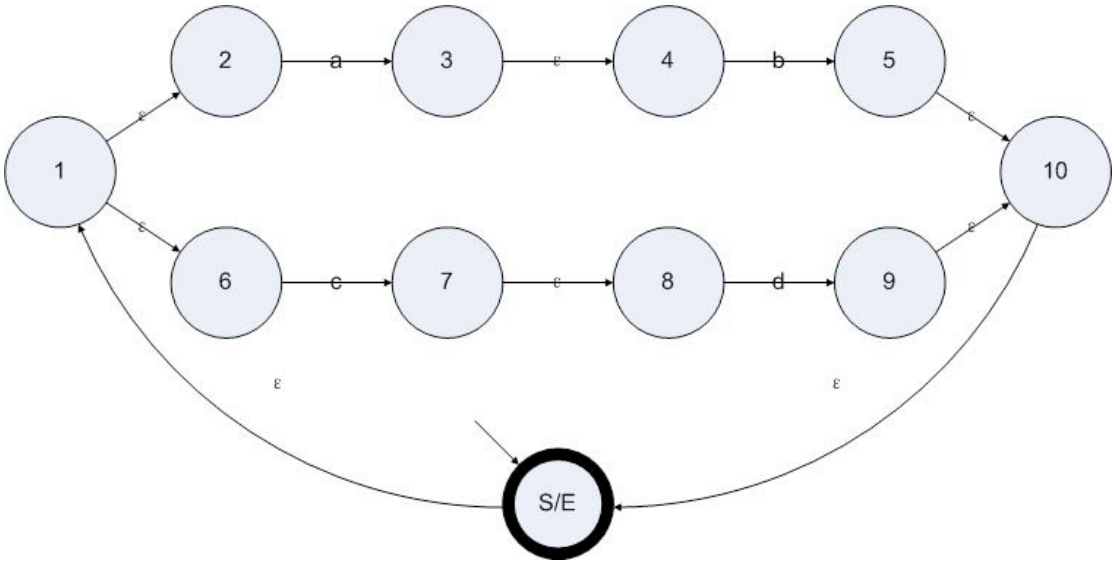

图 5.1

1：找到所有有效状态。

有效状态就是在完成了消除 ε 边算法之后仍然存在的状态。我们可以在开始整个算法之前就预先计算出所有有效状态。有效状态的特点是存在非 ε 边的输入。同时，起始状态也是一个有效状态。结束状态不一定是有效状态，但是如果存在一个有效状态可以仅通过 ε 边到达结束状态的话，那么这个状态应该被标记为结束状态。因此对一个 ε -NFA 应用消除 ε 边算法产生的 NFA 可能出现多个结束状态。不过起始状态仍然只有一个。

我们可以把“存在非 ε 边的输入或者起始状态”这个判断方法应用在图 5.1 每一个状态上，计算出图 5.1 中所有的有效状态。结果如下图所示。

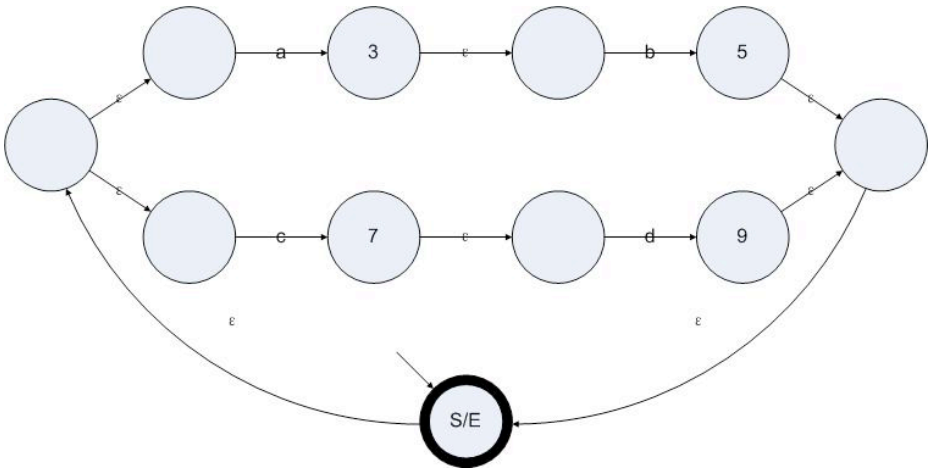

图 5.2
所有非有效状态的标签都被删除

如果一个状态同时具有 ε 边和非 ε 边输入的话，那么这个状态仍然是有效状态。因为所有的有效状态在下一步的操作中，都会得到新的输出和新的输入。

2：添加所有必要的边

接下来我们要对所有的有效状态都应用一个算法。这个算法分成两步。第一步是寻找一个状态的 ε 闭包，第二步是把这个状态的 ε 闭包看成一个整体，把所有从这个闭包中输出的边全部复制到当前状态上。从标记有效状态的结果我们得到了图 5.1 状态图的有效状态集合是{S/E 3 5 7 9}。我们依次对这些状态应用上述算法。第一步，计算 S/E 状态的 ε 闭包。**所谓一个状态的 ε 闭包就是从这个状态出发，仅通过 ε 边就可以到达的所有状态的集合。**下图中标记出了状态 S/E 的 ε 闭包：

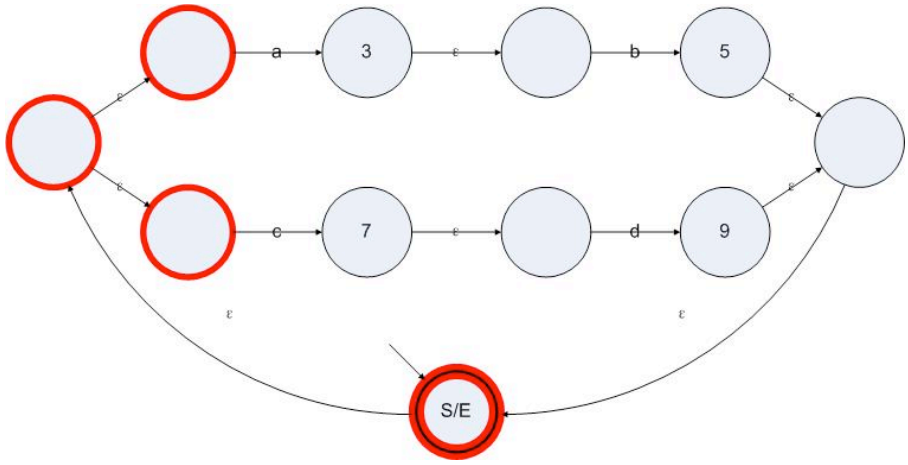

图 5.3

现在，我们把状态 S/E 从状态 S/E 的 ε 闭包中排除出去。因为从状态 A 输出的非 ε 边都属于从状态 A 的 ε 闭包中输出的非 ε 边，复制这些边是没有任何价值的。接下来就是找到从状态 S/E 的 ε 闭包中输出的非 ε 边。在图 5.3 我们可以很容易地发现，从状态 1 和状态 6(见图 5.1 的状态标签)分别输出到状态 3 和状态 7 的标记了 a 或者 b 的边，就是我们所要寻找的边。接下来我们把这些边复制到状态 S/E 上，边的目标状态仍然保持不变，可以得到下图：

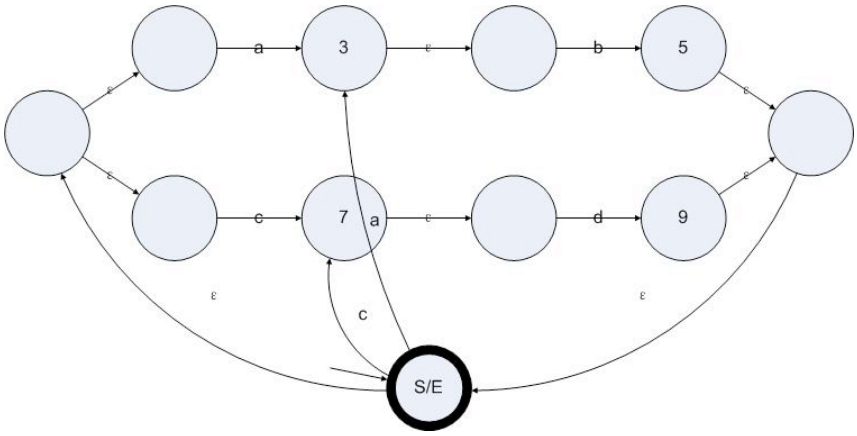

图 5.4

至此，这个算法在 S/E 上的应用就结束了，接下来我们分别对剩下的有效状态{3 5 7 9}分别应用此算法，可以得到下图：

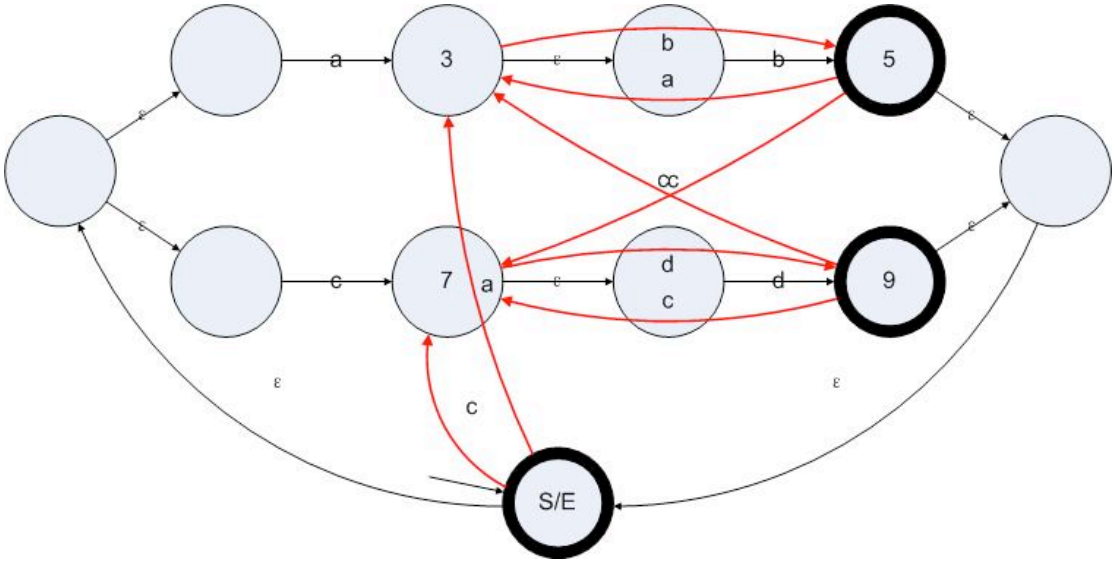

图 5.5

红色的边为新增加的边

3：删除所有 ε 边和无效状态

这一步操作是消除 ε 边算法的最后步骤。我们只需要删除所有的 ε 边和无效状态就完成了整个消除 ε 边算法。现在我们对图 5.5 的状态机应用第三步，得到如下状态图：

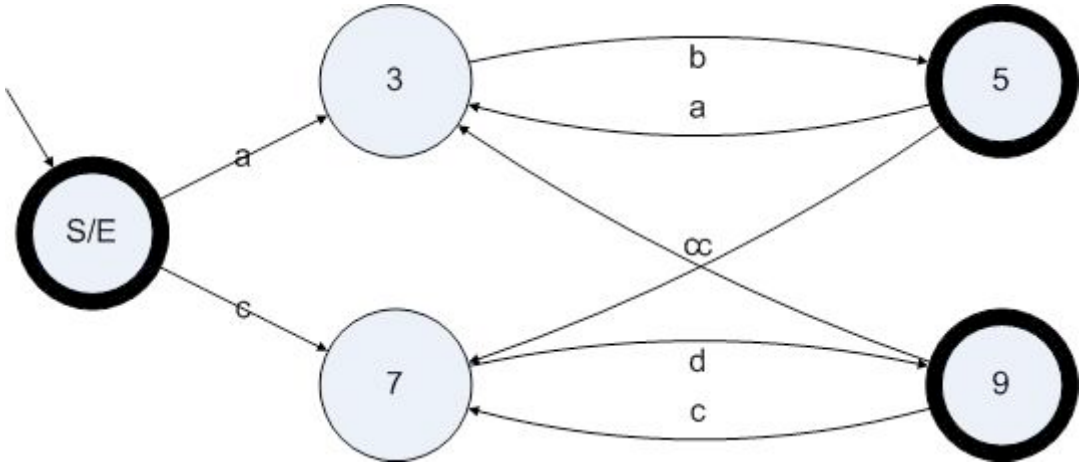

图 5.6

不过并不是只有新增的边才不被删除。根据定义，所有从有效状态出发的非 ε 边都是不能删除的边。

我们通过把消除 ε 边算法应用在图 5.1 的状态机上，得到了图 5.6 这个 DFA。但是**并不是所有的消除 ε 边算法都可以直接从 ε -NFA 直接得到 DFA**，这个其实跟正则表达式本身有

关。至于什么正则表达式可以达到这个效果这里就不深究了。但是因为有可能产生 NFA，所以我们还需要一个算法把 NFA 转换成 DFA。

从 NFA 到 DFA

NFA 是非确定性的状态机，DFA 是确定性的状态机。确定性和非确定性的最大区别就是：从一个状态读入一个字符，确定性的状态机得到一个状态，而非确定性的状态机得到一个状态的集合。如果我们把 NFA 的起始状态 S 看成一个集合{S}的话，对于一个状态集合 S'，给定一个输入，就可以用 NFA 计算出对应的状态集合 T'。因此我们在构造 DFA 的时候，只需要把起始状态对应到 S'，并且找到所有可能在 NFA 同时出现的状态集合，把这些集合都转换成 DFA 的一个状态，那么任务就完成了。因为 NFA 的状态是有限的，所以 NFA 所有状态的集合的幂集的元素个数也是有限的，因此使用这个方法构造 DFA 是完全可能的。

为了形象地表达出这个算法的过程，我们将构造一个正则表达式，然后给出该正则表达式转换成 NFA 的结果，并把构造 DFA 的算法应用在 NFA 上。假设一个字符串只有 a、b 和 c 三种字符，判断一个字符串是不是以 abc 开始并且以 cba 结束正则表达式如下：

abc(a|b|c)*cba

通过上文的算法，可以得出如下图所示的 NFA：

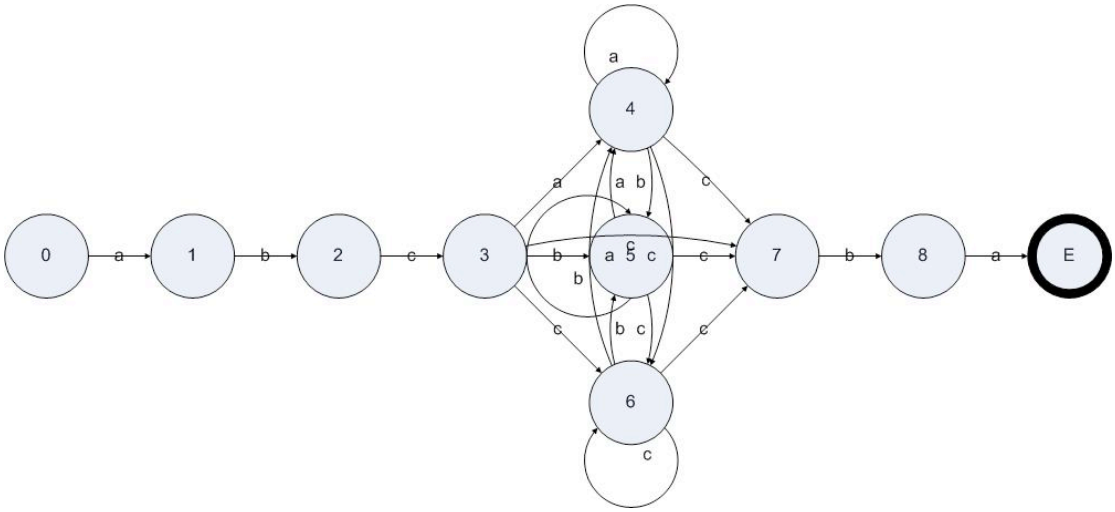

图 5.7

现在我们开始构造 DFA，具体算法如下：

1：把{S}放进队列 L 和集合 D 中。其中 S 是 NFA 的起始状态。队列 L 放置的是未被处理的已经创建了的 DFA 状态，集合 D 放置的是已经存在的 DFA 状态。根据上文的讨论，DFA 的每一个状态都对应着 NFA 的一些状态。

2：从队列 L 中取出一个状态，计算从这个状态输出的所有边所接受的字符集的并集，然后对该集合中的每一个字符寻找接受这个字符的边，把这些边的目标状态的并集 T 计算出来。如果 T∈D 则代表当前字符指向了一个已知的 DFA 状态。否则则代表当前字符指向了一个未创建的 DFA 状态，这个时候就把 T 放进 L 和 D 中。在这个步骤里有两层循环：第

一层是**遍历**所有接受的字符的并集，第二层是对每一个可以接受的字符**遍历**所有输出的边计算目标 DFA 状态所包含的 NFA 状态的集合。

3：如果 L 非空则跳到 2。

现在我们开始对图 5.7 的状态机应用 DFA 构造算法。

首先执行第一步，我们得到：

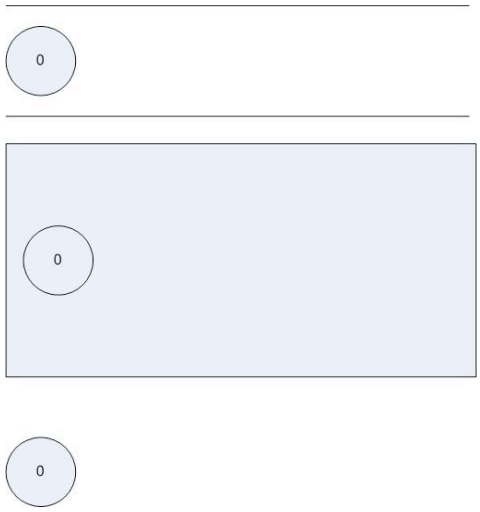

图 5.8

从上到下分别是队列 L、集合 D 和 DFA 的当前状态。就这样一直执行该算法直到状态 3 进入集合 D，我们得到：

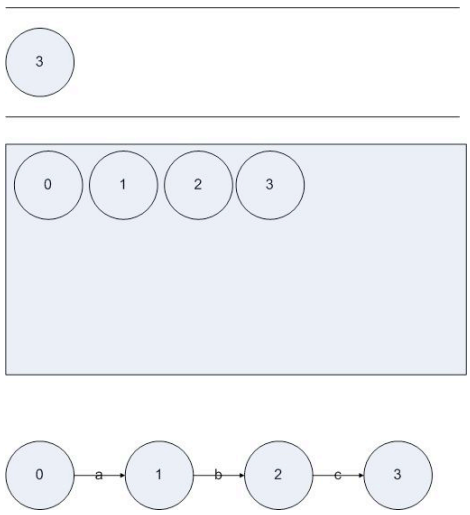

图 5.9

现在从队列 L 中取出{3}，经过分析得到状态集合{3}接受的字符集合为{a b c}。{3}读入 a 到达{4}，读入 b 到达{5}，读入 c 到达{6 7}。因为{4}、{5}和{6 7}都不属于 D，所以把它们都放入队列 L 和集合 D：

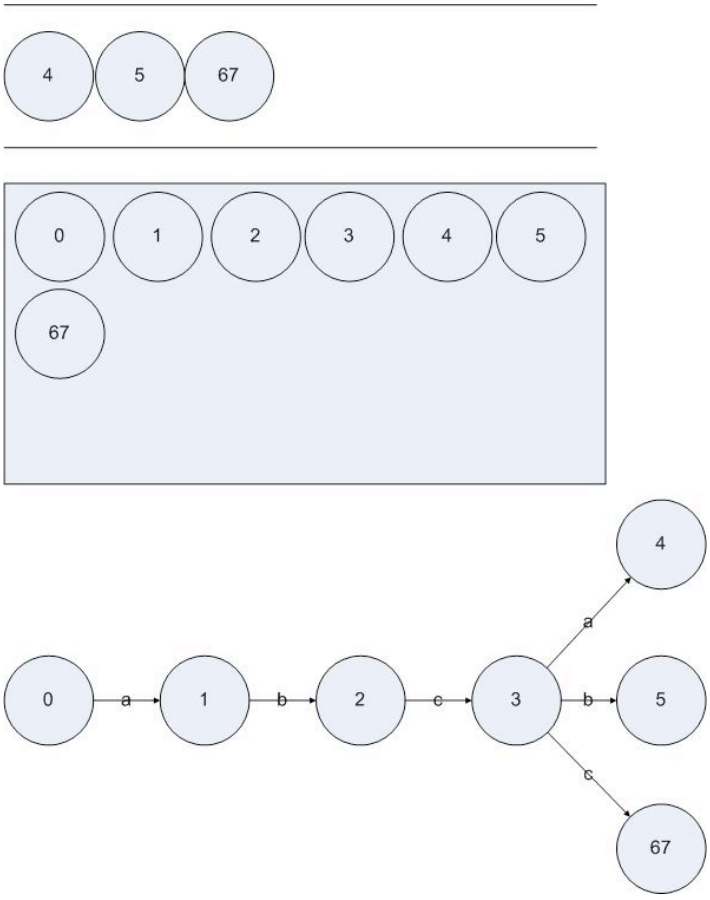

图 5.10

从队列中取出 4 进行计算，我们得到：

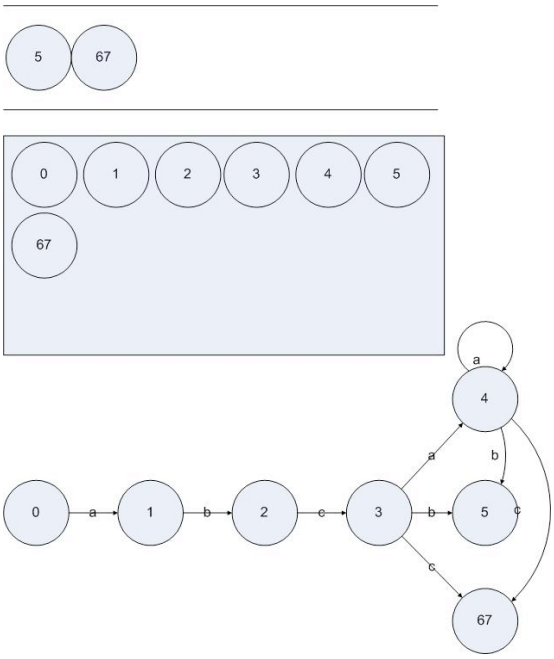

图 5.11

显然，对于状态{4}的处理并没有发现新的 DFA 状态。于是处理{5}和{6 7}，我们可以得到：

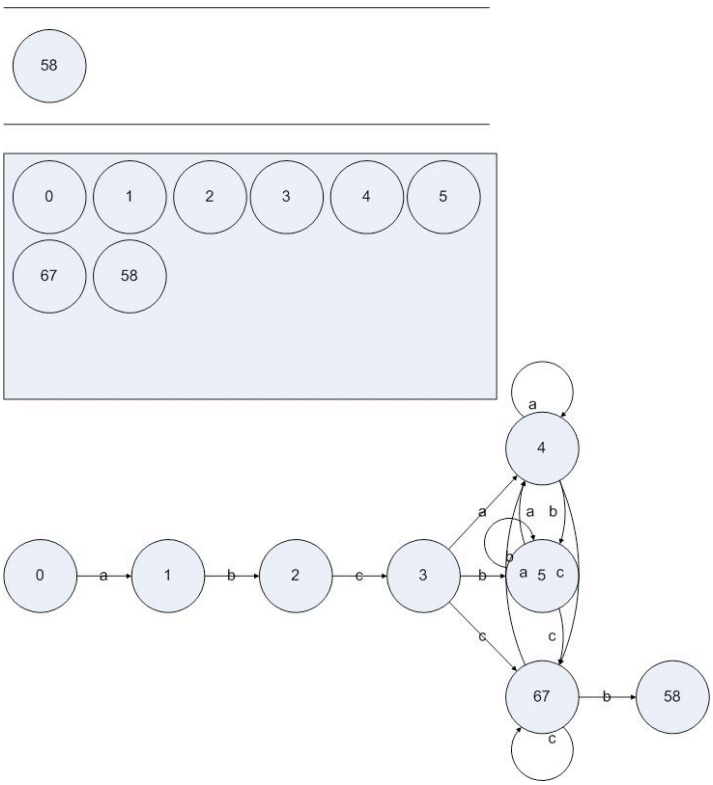

图 5.12

在处理状态{5}的时候没有发现新的 DFA 状态，处理{6 7}在输入{a c}之后的跳转也没有发现新的 DFA 状态，但是我们发现了{6 7}输入 b 却得到了新的 DFA 状态{5 8}。把算法执行完，我们得到一个 DFA：

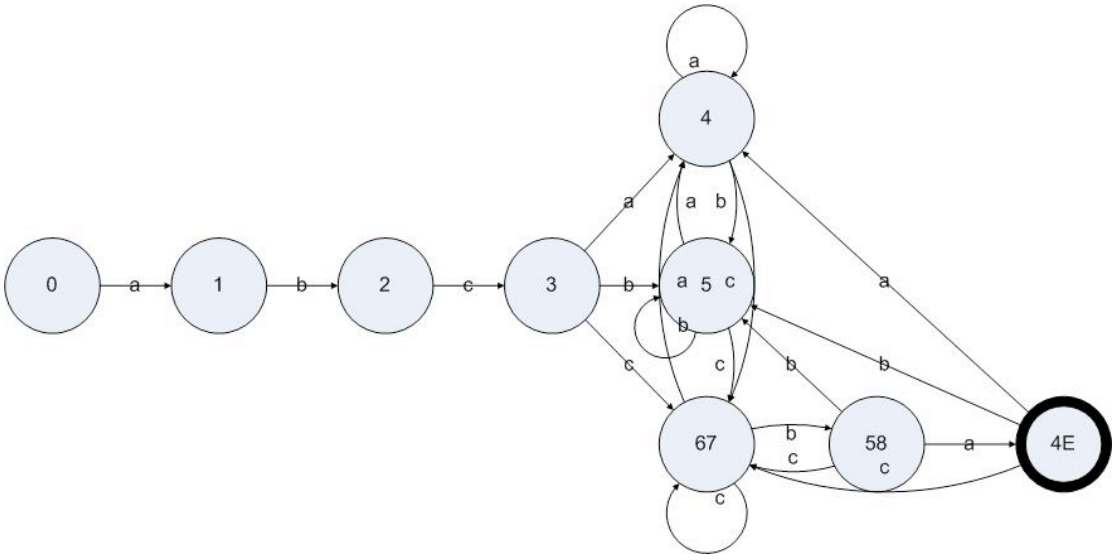

图 5.13

至此，对图 5.7 的状态机应用 DFA 构造算法的流程就结束了，我们得到了图 5.13 的 DFA，其功能与图 5.7 的 NFA 等价。在 DFA 中，起始状态是 0，结束状态是 4E。凡是包含了 NFA 的结束状态的 DFA 状态都必须是结束状态。

六、使用正则表达式构造词法分析器

判断一个字符串是否属于某规则的算法介绍到这里就结束了。回到我们一开始的问题上，我们需要使用一些规则来吧一个长的字符串断开成记号，然后计算出每一个记号对应的规则。在解决这个问题之前，我们先考察一下能够成功地被词法分析器接受的字符串应该是什么样子的。

假设我们现在有规则 A、B、C 和 D，分别对应于四种记号类型，那么被词法分析器接受的字符串就是 A、B、C 和 D 的任意组合，也就是(A|B|C|D)*。如果规定了输入的字符串不能为空的话，那么被词法分析器接受的字符串就是(A|B|C|D)+。但是单纯地使用(A|B|C|D)+作为一个规则去应用在输入的字符串的话，我们只能够判断字符串是否是词法分析器能够接受的字符串。因此我们需要对这个方法进行修改。

首先按照上文的方法，把每一个记号类型对应的规则的正则表达式转换成 DFA，然后使用并联的方法将他们组合起来，但是并不使用“重复”。但是这次我们要做一点修改，**我们要把新的 DFA 的每一个状态对应的规则的 DFA 状态集合记住**。

这里给出一个例子，我们假设需要一个简单语言的词法分析器，规则如下：

I：[a-zA-Z_][a-zA-Z_0-9]*
N：[0-9]+
R：[0-9]+.[0-9]+
O：[=>+-*/|&]

按照规则构造出四个 DFA 并将它们组合起来：

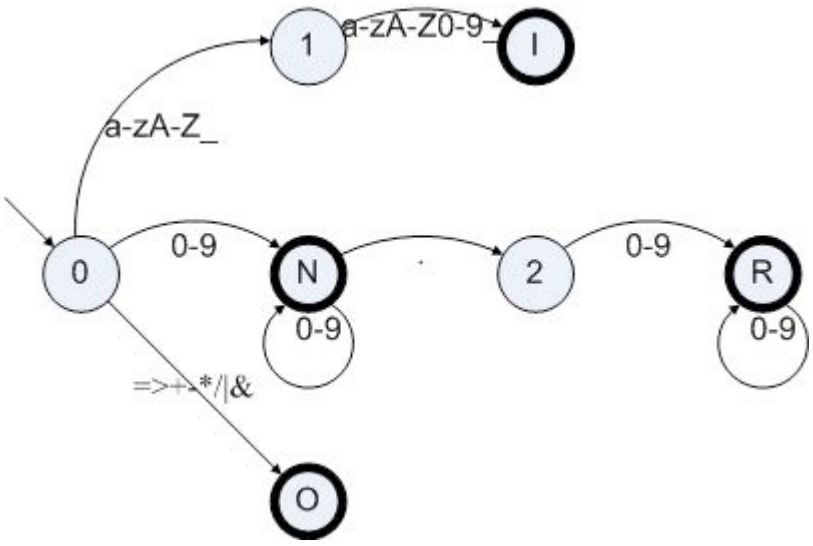

图 6.1

我们构造出了 I|N|R|O 的 DFA，并且标识出了哪些状态包含了原 DFA 的结束状态。这样做的一个好处是，当我们把一个字符串放进这样的一个 DFA 之后，我们就一直等待整个字符串被接受，或者失败。如果字符串被接受的话，我们就把当前的结束状态记下来。如果失败的话，我们就把这个状态机在分析字符串的时候经过的最后一个结束状态记下来。这个时候，结束状态所代表的原 DFA 结束状态的相应记号类型就是我们所需要的信息了。如果得不到任何结束状态的话，输入的字符串就是不背词法分析其接受的。

举个例子，使用上述状态机分析”123.ABC”。

首先从状态 0 开始，依次经过状态 N N N 2，然后宣告失败。最后一个 N(结束状态)以及当时被接受的字符串”123”被识别，结果为(N，”123”)。然后从”.ABC”开始，输入第一个记号就失败了，于是”.”被识别为不可接受字串。最后输入”ABC”，依次经过状态 0 1 I I，然后字符串结束并且被接受，于是输出(I，”ABC”)。

为什么我们在构造状态机的时候不使用“重复”呢？因为在每一个记号被识别出的时候，我们都要做一些额外的工作。如果我们使用“重复”来构造词法分析器的状态机，我们将无从知道一个记号被识别出来的确切时间。

算法到这里基本上就结束了，不过还存在一些小问题需要在细节上解决。一般来说我们给出的一些构成词法分析器的规则很少有冲突，不过偶尔会出现两个规则所代表的字符串集合存在交集的情况。有了 DFA 这个工具，我们可以很轻易地识别出规则冲突。

假如我们的词法分析器有 A 和 B 两个状态，那么我们构造词法分析器 A|B 的时候，将会得到一些包含 DFA(A)和 DFA(B)的结束状态的新状态。我们只需要检查这些状态是否具有以下特征，就可以判断 A 和 B 的关系。我们假设 DFA(A)为规则 A 的状态机，DFA(B)为规则 B 的状态机，DFA(L)为词法分析器 A|B 的状态机：

1：如果 DFA(L)存在一个或多个状态同时包含了 DFA(A)和 DFA(B)的结束状态，那么 A 和 B 所代表的字符串存在交集。

2：如果 DFA(L)不存在同时包含了 DFA(A)和 DFA(B)的结束状态的状态，那么 A 和 B 所代表的字符串不存在交集。

3：如果 DFA(L)的某些状态包含了 DFA(A)的结束状态，并且这些状态都无一例外地包含了 DFA(B)的结束状态的话，那么 A 是 B 的子集。

4：如果 DFA(L)的某些状态包含了 DFA(A)的结束状态，但是这些状态并没有无一例外地包含 DFA(B)的结束状态的话，那么 A 不是 B 的子集。

在图 6.1 的词法分析器中，我们可以很清楚地看出 I、N、R 和 O 四个规则两两之间都不存在交集。我们可以尝试构造一个冲突的规则，并看一看词法分析器的 DFA 是什么样子的：

假设词法分析器包含以下规则：

A：”if”

B：[a-z]+

对 A|B 构造 DFA，我们将会得到如下状态机：

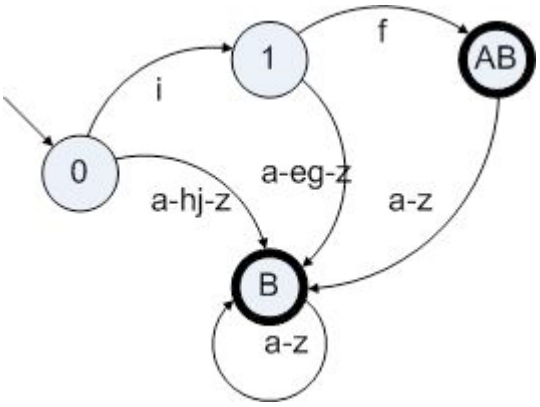

图 6.2

通过图 6.2 我们可以看出，这个状态图满足了上述的条件 3：包含了状态 A 的结束状态的状态都包含了 B 的结束状态，因此 A 是 B 的子集。显然”if”是[a-z]+的一个子集。在处理这种有冲突的规则的时候，既可以报错，也可以根据指定的优先级进行挑选。

七、尾声

使用 DFA 的方法完成的可配置词法分析器的性能是相当好的。笔者前不久曾经做过实验，首先使用本文提到的算法开发一个这样的词法分析器，然后在一份 C++代码(这份代码经过多次复制而成件，一共有 3.12M)中抽取所有数字、标识符和注释，吞吐速度高达 46 万记号/秒（笔者的台式电脑配置是奔腾 4 的超线程 2.99GHz 处理器，1G 内存），其中抽取出来的记号一共有 22 万个。在分析的过程中，只有 10%的时间花在了 DFA 上，90%的时间花在了处理结果的工作上。DFA 本身造成的消耗是很小的。不过词法分析的性能在很大程度上跟 DFA 的实现有很大关系。三个月前笔者也实现过一个同类的程序，但是吞吐速度仅有 1.1 万记号/秒。

一般来说，比较高性能的 DFA 的实现是一张二维的表。行代表字符，列代表 DFA 的状态，单元格代表该状态经输入某个字符之后进行转移的目标状态。此外还有一张表用来记录哪些状态对应哪些规则的结束状态。笔者的词法分析器是基于 UTF-16 编码的字符串，一张表一共有 65535 行显然是不现实的，因此还有另一张表把字符转换成字符类。字符类是这样定义的：假设现在已经存在了 65535 行的一张大表，如果在某个字符区间所对应的子表内，任意一列的单元格的数据都一样的话，那么这个区间内的所有字符就可以被视为是等价的，这些字符就属于同一个字符类。于是仅需要另外一张 65535 个单元的表用来把一个字符映射到字符类。这种做法可以大大的压缩 DFA 所需要的空间。在笔者的程序里，识别字符类的算法被融入了 DFA 的构造算法中

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