**1 概述**贪婪与非贪婪模式影响的是被量词修饰的子表达式的匹配行为，贪婪模式在整个表达式匹配成功的前提下，尽可能多的匹配，而非贪婪模式在整个表达式匹配成功的前提下，尽可能少的匹配。非贪婪模式只被部分NFA引擎所支持。  
  
属于贪婪模式的量词，也叫做匹配优先量词，包括：  
  
“{m,n}”、“{m,}”、“?”、“\*”和“+”。  
  
在一些使用NFA引擎的语言中，在匹配优先量词后加上“?”，即变成属于非贪婪模式的量词，也叫做忽略优先量词，包括：  
  
“{m,n}?”、“{m,}?”、“??”、“\*?”和“+?”。  
  
从正则语法的角度来讲，被匹配优先量词修饰的子表达式使用的就是贪婪模式，如“(Expression)+”；被忽略优先量词修饰的子表达式使用的就是非贪婪模式，如“(Expression)+?”。  
  
对于贪婪模式，各种文档的叫法基本一致，但是对于非贪婪模式，有的叫懒惰模式或惰性模式，有的叫勉强模式，其实叫什么无所谓，只要掌握原理和用法，能够运用自如也就是了。个人习惯使用贪婪与非贪婪的叫法，所以文中都会使用这种叫法进行介绍。  
  
**2 贪婪与非贪婪模式匹配原理**  
对于贪婪与非贪婪模式，可以从应用和原理两个角度进行理解，但如果想真正掌握，还是要从匹配原理来理解的。  
  
先从应用的角度，回答一下“什么是贪婪与非贪婪模式？”  
  
2.1 从应用角度分析贪婪与非贪婪模式  
2.1.1 什么是贪婪与非贪婪模式  
先看一个例子  
  
举例：  
  
源字符串：aa<div>test1</div>bb<div>test2</div>cc  
  
正则表达式一：<div>.\*</div>  
  
匹配结果一：<div>test1</div>bb<div>test2</div>  
  
正则表达式二：<div>.\*?</div>  
  
匹配结果二：<div>test1</div>（这里指的是一次匹配结果，所以没包括<div>test2</div>）  
  
根据上面的例子，从匹配行为上分析一下，什是贪婪与非贪婪模式。  
  
正则表达式一采用的是贪婪模式，在匹配到第一个“</div>”时已经可以使整个表达式匹配成功，但是由于采用的是贪婪模式，所以仍然要向右尝试匹配，查看是否还有更长的可以成功匹配的子串，匹配到第二个“</div>”后，向右再没有可以成功匹配的子串，匹配结束，匹配结果为“<div>test1</div>bb<div>test2</div>”。当然，实际的匹配过程并不是这样的，后面的匹配原理会详细介绍。  
  
仅从应用角度分析，可以这样认为，贪婪模式，就是在整个表达式匹配成功的前提下，尽可能多的匹配，也就是所谓的“贪婪”，通俗点讲，就是看到想要的，有多少就捡多少，除非再也没有想要的了。  
  
正则表达式二采用的是非贪婪模式，在匹配到第一个“</div>”时使整个表达式匹配成功，由于采用的是非贪婪模式，所以结束匹配，不再向右尝试，匹配结果为“<div>test1</div>”。  
  
仅从应用角度分析，可以这样认为，非贪婪模式，就是在整个表达式匹配成功的前提下，尽可能少的匹配，也就是所谓的“非贪婪”，通俗点讲，就是找到一个想要的捡起来就行了，至于还有没有没捡的就不管了。  
  
2.1.2 关于前提条件的说明  
在上面从应用角度分析贪婪与非贪婪模式时，一直提到的一个前提条件就是“整个表达式匹配成功”，为什么要强调这个前提，我们看下下面的例子。  
  
正则表达式三：<div>.\*</div>bb  
  
匹配结果三：<div>test1</div>bb  
  
修饰“.”的仍然是匹配优先量词“\*”，所以这里还是贪婪模式，前面的“<div>.\*</div>”仍然可以匹配到“<div>test1</div>bb<div>test2</div>”，但是由于后面的“bb”无法匹配成功，这时“<div>.\*</div>”必须让出已匹配的“bb<div>test2</div>”，以使整个表达式匹配成功。这时整个表达式匹配的结果为“<div>test1</div>bb”，“<div>.\*</div>”匹配的内容为“<div>test1</div>”。可以看到，在“整个表达式匹配成功”的前提下，贪婪模式才真正的影响着子表达式的匹配行为，如果整个表达式匹配失败，贪婪模式只会影响匹配过程，对匹配结果的影响无从谈起。  
  
**非贪婪模式也存在同样的问题，来看下面的例子。**  
正则表达式四：<div>.\*?</div>cc  
  
匹配结果四：<div>test1</div>bb<div>test2</div>cc  
  
这里采用的是非贪婪模式，前面的“<div>.\*?</div>”仍然是匹配到“<div>test1</div>”为止，此时后面的“cc”无法匹配成功，要求“<div>.\*?</div>”必须继续向右尝试匹配，直到匹配内容为“<div>test1</div>bb<div>test2</div>”时，后面的“cc”才能匹配成功，整个表达式匹配成功，匹配的内容为“<div>test1</div>bb<div>test2</div>cc”，其中“<div>.\*?</div>”匹配的内容为“<div>test1</div>bb<div>test2</div>”。可以看到，在“整个表达式匹配成功”的前提下，非贪婪模式才真正的影响着子表达式的匹配行为，如果整个表达式匹配失败，非贪婪模式无法影响子表达式的匹配行为。  
  
2.1.3 贪婪还是非贪婪——应用的抉择  
通过应用角度的分析，已基本了解了贪婪与非贪婪模式的特性，那么在实际应用中，究竟是选择贪婪模式，还是非贪婪模式呢，这要根据需求来确定。  
  
对于一些简单的需求，比如源字符为“aa<div>test1</div>bb”，那么取得div标签，使用贪婪与非贪婪模式都可以取得想要的结果，使用哪一种或许关系不大。  
  
但是就2.1.1中的例子来说，实际应用中，一般一次只需要取得一个配对出现的div标签，也就是非贪婪模式匹配到的内容，贪婪模式所匹配到的内容通常并不是我们所需要的。  
  
那为什么还要有贪婪模式的存在呢，从应用角度很难给出满意的解答了，这就需要从匹配原理的角度去分析贪婪与非贪婪模式。  
  
2.2 从匹配原理角度分析贪婪与非贪婪模式  
如果想真正了解什么是贪婪模式，什么是非贪婪模式，分别在什么情况下使用，各自的效率如何，那就不能仅仅从应用角度分析，而要充分了解贪婪与非贪婪模式的匹配原理。  
  
2.2.1 从基本匹配原理谈起  
NFA引擎基本匹配原理参考：正则基础之——NFA引擎匹配原理。  
  
这里主要针对贪婪与非贪婪模式涉及到的匹配原理进行介绍。先看一下贪婪模式简单的匹配过程。  
  
源字符串："Regex"  
  
正则表达式：".\*"

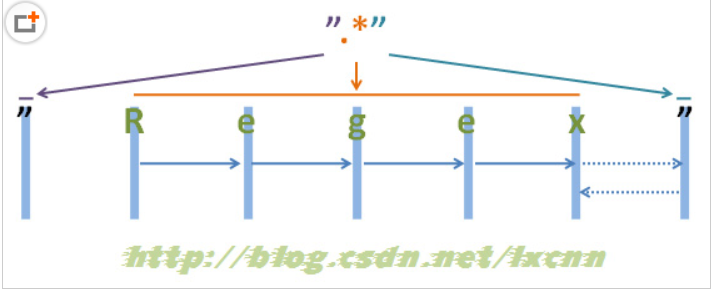


图2-1  
  
注：为了能够看清晰匹配过程，上面的空隙留得较大，实际源字符串为“”Regex””，下同。  
  
来看一下匹配过程。首先由第一个“"”取得控制权，匹配位置0位的“"”，匹配成功，控制权交给“.\*”。  
  
“.\*”取得控制权后，由于“\*”是匹配优先量词，在可匹配可不匹配的情况下，优先尝试匹配。从位置1处的“R”开始尝试匹配，匹配成功，继续向右匹配，匹配位置2处的“e”，匹配成功，继续向右匹配，直到匹配到结尾的“””，匹配成功，由于此时已匹配到字符串的结尾，所以“.\*”结束匹配，将控制权交给正则表达式最后的“"”。  
  
“"”取得控制权后，由于已经在字符串结束位置，匹配失败，向前查找可供回溯的状态，控制权交给“.\*”，由“.\*”让出一个字符，也就是字符串结尾处的“””，再把控制权交给正则表达式最后的“"”，由“"”匹配字符串结尾处的“"”，匹配成功。  
  
此时整个正则表达式匹配成功，其中“.\*”匹配的内容为“Regex”，匹配过程中进行了一次回溯。  
  
接下来看一下非贪婪模式简单的匹配过程。  
  
源字符串："Regex"  
  
正则表达式：".\*?"

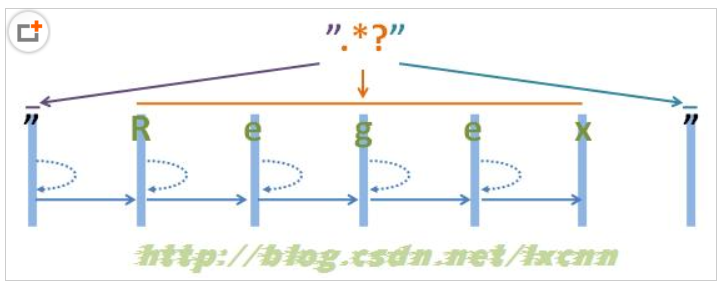


图2-2  
  
看一下非贪婪模式的匹配过程。首先由第一个“"”取得控制权，匹配位置0位的“"”，匹配成功，控制权交给“.\*?”。  
  
“.\*?”取得控制权后，由于“\*?”是忽略优先量词，在可匹配可不匹配的情况下，优先尝试不匹配，由于“\*”等价于“{0,}”，所以在忽略优先的情况下，可以不匹配任何内容。从位置1处尝试忽略匹配，也就是不匹配任何内容，将控制权交给正则表达式最后的“””。  
  
“"”取得控制权后，从位置1处尝试匹配，由“"”匹配位置1处的“R”，匹配失败，向前查找可供回溯的状态，控制权交给“.\*?”，由“.\*?”吃进一个字符，匹配位置1处的“R”，再把控制权交给正则表达式最后的“"”。  
  
“"”取得控制权后，从位置2处尝试匹配，由“"”匹配位置1处的“e”，匹配失败，向前查找可供回溯的状态，重复以上过程，直到由“.\*?”匹配到“x”为止，再把控制权交给正则表达式最后的“"”。  
  
“"”取得控制权后，从位置6处尝试匹配，由“"”匹配字符串最后的“"”，匹配成功。  
  
此时整个正则表达式匹配成功，其中“.\*?”匹配的内容为“Regex”，匹配过程中进行了五次回溯。  
  
2.2.2 贪婪还是非贪婪——匹配效率的抉择  
通过匹配原理的分析，可以看到，在匹配成功的情况下，贪婪模式进行了更少的回溯，而回溯的过程，需要进行控制权的交接，让出已匹配内容或匹配未匹配内容，并重新尝试匹配，在很大程度上降低匹配效率，所以贪婪模式与非贪婪模式相比，存在匹配效率上的优势。  
  
但2.2.1中的例子，仅仅是一个简单的应用，读者看到这里时，是否会存在这样的疑问，贪婪模式就一定比非贪婪模式匹配效率高吗？答案是否定的。  
  
举例：  
  
需求：取得两个“"”中的子串，其中不能再包含“"”。  
  
正则表达式一：".\*"  
  
正则表达式二：".\*?"  
  
情况一：当贪婪模式匹配到更多不需要的内容时，可能存在比非贪婪模式更多的回溯。比如源字符串为“The word "Regex" means regular expression.”。  
  
情况二：贪婪模式无法满足需求。比如源字符串为“The phrase "regular expression" is called "Regex" for short.”。  
  
对于情况一，正则表达式一采用的贪婪模式，“.\*”会一直匹配到字符串结束位置，控制权交给最后的“””，匹配不成功后，再进行回溯，由于多匹配的内容“means regular expression.”远远超过需匹配内容本身，所以采用正则表达式一时，匹配效率会比使用正则表达式二的非贪婪模式低。  
  
对于情况二，正则表达式一匹配到的是“"regular expression" is called "Regex"”，连需求都不满足，自然也谈不上什么匹配效率的高低了。  
  
以上两种情况是普遍存在的，那么是不是为了满足需求，又兼顾效率，就只能使用非贪婪模式了呢？当然不是，根据实际情况，变更匹配优先量词修饰的子表达式，不但可以满足需求，还可以提高匹配效率。  
  
源字符串："Regex"  
  
给出正则表达式三："[^"]\*"  
  
看一下正则表达式三的匹配过程。

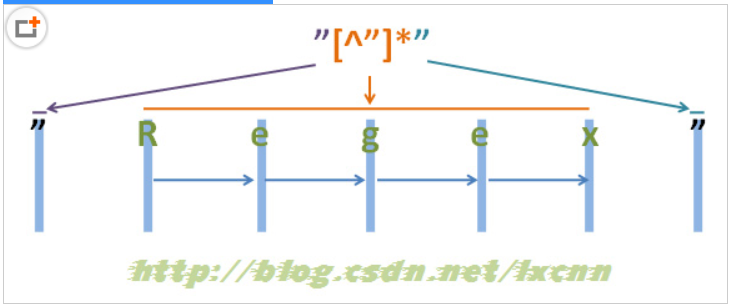


图2-3  
  
首先由第一个“"”取得控制权，匹配位置0位的“"”，匹配成功，控制权交给“[^"]\*”。  
  
“[^"]\*”取得控制权后，由于“\*”是匹配优先量词，在可匹配可不匹配的情况下，优先尝试匹配。从位置1处的“R”开始尝试匹配，匹配成功，继续向右匹配，匹配位置2处的“e”，匹配成功，继续向右匹配，直到匹配到“x”，匹配成功，再匹配结尾的“””时，匹配失败，将控制权交给正则表达式最后的“"”。  
  
“””取得控制权后，匹配字符串结尾处的“””，匹配成功。  
  
此时整个正则表达式匹配成功，其中“[^"]\*”匹配的内容为“Regex”，匹配过程中没有进行回溯。  
  
将量词修饰的子表达式由范围较大的“.”，换成了排除型字符组“[^"]”，使用的仍是贪婪模式，很完美的解决了需求和效率问题。当然，由于这一匹配过程没有进行回溯，所以也不需要记录回溯状态，这样就可以使用固化分组，对正则做进一步的优化。  
  
给出正则表达式四："(?>[^"]\*)"  
  
固化分组并不是所有语言都支持的，如.NET支持，而Java就不支持，但是在Java中却可以使用更简单的占有优先量词来代替："[^"]\*+"。  
  
**3 贪婪还是非贪婪模式——再谈匹配效率**一般来说，贪婪与非贪婪模式，如果量词修饰的子表达式相同，比如“.\*”和“.\*?”，它们的应用场景通常是不同的，所以效率上一般不具有可比性。  
  
而对于改变量词修饰的子表达式，以满足需求时，比如把“.\*”改为“[^"]\*”，由于修饰的子表达式已不同，也不具有直接的可对比性。但是在相同的子表达式，又都可以满足需求的情况下，比如“[^"]\*”和“[^"]\*?”，贪婪模式的匹配效率通常要高些。  
  
同时还有一个事实就是，非贪婪模式可以实现的，通过优化量词修饰的子表达式的贪婪模式都可以实现，而贪婪模式可以实现的一些优化效果，却未必是非贪婪模式可以实现的。  
  
贪婪模式还有一点优势，就是在匹配失败时，贪婪模式可以更快速的报告失败，从而提升匹配效率。下面将全面考察贪婪与非贪婪模式的匹配效率。  
  
3.1 效率提升——演进过程  
在了解了贪婪与非贪婪模式的匹配基本原理之后，我们再来重新看一下正则效率提升的演进过程。  
  
需求：取得两个“"”中的子串，其中不能再包含“"”。  
  
源字符串：The phrase "regular expression" is called "Regex" for short.  
  
正则表达式一：".\*"  
  
正则表达式一匹配的内容为“"regular expression" is called "Regex"”，不符合要求。  
  
提出正则表达式二：".\*?"  
  
首先“"”取得控制权，由位置0位开始尝试匹配，直到位置11处匹配成功，控制权交给“.\*?”，匹配过程同2.2.1中非贪婪模式的匹配过程。“.\*?”匹配的内容为“Regex”，匹配过程中进行了四次回溯。  
  
如何消除回溯带来的匹配效率的损失，就是使用更小范围的子表达式，采用贪婪模式，提出正则表达式三："[^"]\*"  
  
首先“"”取得控制权，由位置0位开始尝试匹配，直到位置11处匹配成功，控制权交给“[^"]\*”，匹配过程同2.2.2节中非贪婪模式的匹配过程。“[^"]\*”匹配的内容为“Regex”，匹配过程中没有进行回溯。  
  
3.2 效率提升——更快的报告失败  
以上讨论的是匹配成功的演进过程，而对于一个正则表达式，在匹配失败的情况下，如果能够以最快的速度报告匹配失败，也会提升匹配效率，这或许是我们设计正则过程中最容易忽略的。而在源字符串数据量非常大，或正则表达式比较复杂的情况下，是否能够快速报告匹配失败，将对匹配效率产生直接的影响。  
  
下面将构建匹配失败的正则表达式，对匹配过程进行分析。  
  
以下匹配过程分析中，源字符串统一为：The phrase "regular expression" is called "Regex" for short.  
  
3.2.1 非贪婪模式匹配失败过程分析

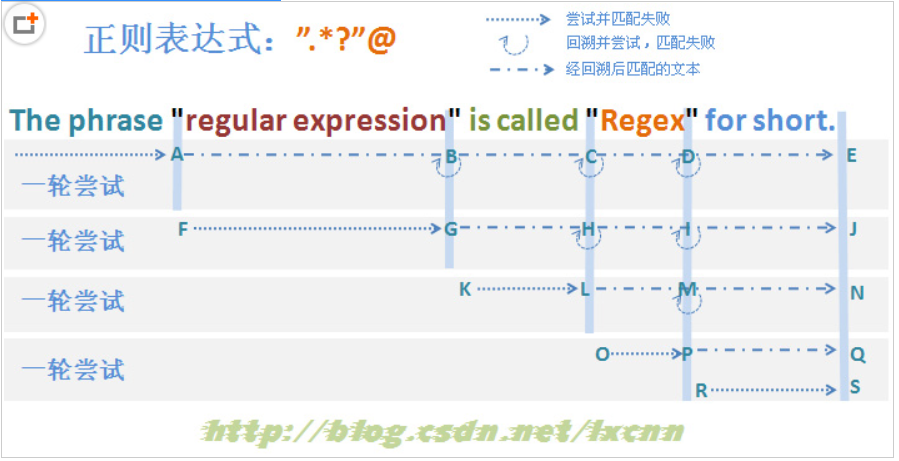


图3-1  
  
构建匹配失败的非贪婪模式的正则表达式：".\*?"@  
  
由于最后的“@”的存在，这个正则表达式最后一定是匹配失败的，那么看一下匹配过程。  
  
首先由“"”取得控制权，由位置0处开始尝试匹配，匹配失败，直到图中标示的A处匹配成功，控制权交给“.\*?”。  
  
“.\*?”取得控制权后，由A后面的位置开始尝试匹配，由于是非贪婪模式，首先忽略匹配，将控制权交给“"”，同时记录一下回溯状态。“"”取得控制权后，由A后面的位置开始尝试匹配，匹配字符“r”失败，查找可供回溯的状态，将控制权交给“.\*?”，由“.\*?”匹配字符“r”。重复以上过程，直到“.\*?”匹配了B处前面的字符“n”，“"”匹配了B处的字符“””，将控制权交给“@”。由“@”匹配接下来的空格“ ”，匹配失败，查找可供回溯的状态，控制权交给“.\*?”，由“.\*?”匹配空格。继续重复以上匹配过程，直到由“.\*?”匹配到字符串结束位置，将控制权交给“"”。由于已经是字符串结束位置，匹配失败，报告整个表达式在位置11处匹配失败，一轮匹配尝试结束。  
  
正则引擎传动装置使正则向前传动，进入下一轮尝试。后续匹配过程与第一轮尝试匹配过程基本类似，可以参考图3-1。  
  
从匹配过程中可以看到，非贪婪模式的匹配失败过程，几乎每一步都伴随着回溯过程，对匹配效率的影响是很大的。  
  
3.2.2 贪婪模式匹配失败过程分析——大范围子表达式

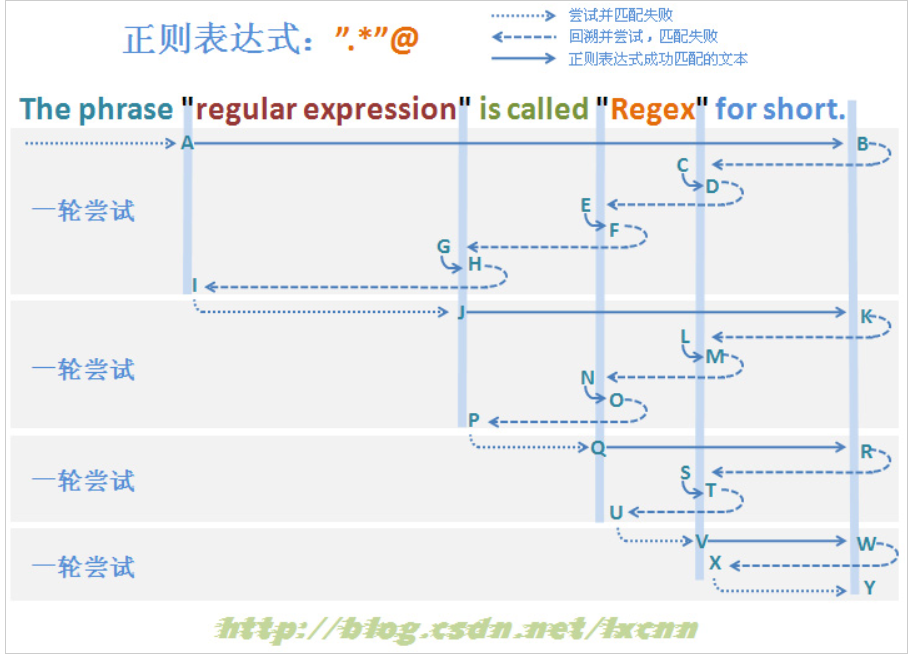


图3-2  
  
PS：以上分析过程图示参考了《精通正则表达式》一书相关章节图示。  
  
构建匹配失败的贪婪模式的正则表达式：".\*"@  
  
其中量词修饰的子表达式为匹配范围较大的“.”，由于最后的“@”的存在，这个正则表达式最后也是一定匹配失败的，看一下匹配过程。  
  
首先由“"”取得控制权，由位置0处开始尝试匹配，匹配失败，直到图中标示的A处匹配成功，控制权交给“.\*”。  
  
“.\*”取得控制权后，由A后面的位置开始尝试匹配，由于是贪婪模式，优化尝试匹配，一直匹配到字符串的结束位置，将控制权交给“"”。“"”取得控制权后，由于已经是字符串的结束位置，匹配失败，查找可供回溯的状态，将控制权交给“.\*”，由“.\*”让出已匹配字符“.”。重复以上过程，直到后面“"”匹配了C处后面的字符“””，将控制权交给“@”。由“@”匹配接下来D处的空格“ ”，匹配失败，查找可供回溯的状态，控制权交给“.\*”，由“.\*”让出已匹配文本。继续重复以上匹配过程，直到由“.\*”让出所有已匹配的文本到I处，将控制权交给“"”。“"”匹配失败，由于已经没有可供回溯的状态，报告整个表达式在位置11处匹配失败，一轮匹配尝试结束。  
  
正则引擎传动装置使正则向前传动，进入下一轮尝试。后续匹配过程与第一轮尝试匹配过程基本类似，可以参考图3-2。  
  
从匹配过程中可以看到，大范围子表达式贪婪模式的匹配失败过程，从总体上看，与非贪婪模式没有什么区别，最终进行的回溯次数与非贪婪模式基本一致，对匹配效率的影响仍然很大。  
  
3.2.3 贪婪模式匹配失败过程分析——改进的子表达式

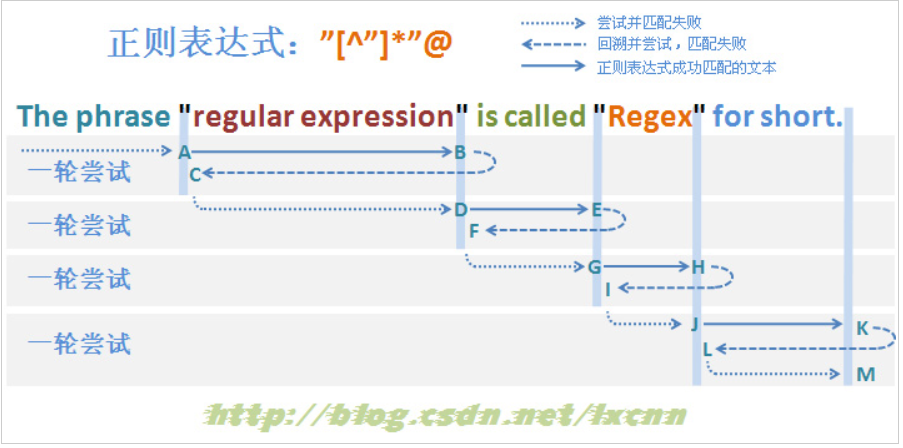


图3-3  
  
构建匹配失败的贪婪模式的正则表达式："[^"]\*"@  
  
其中量词修饰的子表达式，改为匹配范围较小的排除型字符组“[^"]”，由于最后的“@”的存在，这个正则表达式最后也是一定匹配失败的，看一下匹配过程。  
  
首先由“"”取得控制权，由位置0处开始尝试匹配，匹配失败，直到图中标示的A处匹配成功，控制权交给“[^"]\*”。  
  
“[^"]\*”取得控制权后，由A后面的位置开始尝试匹配，由于是贪婪模式，优先尝试匹配，一直匹配到B处，将控制权交给“"”。“"”匹配接下来的的字符“"”，匹配成功，将控制权交给“@”。由“@”匹配接下来的空格“ ”，匹配失败，查找可供回溯的状态，控制权交给“[^"]\*”，由“[^"]\*”让出已匹配文本。继续重复以上匹配过程，直到由“[^"]\*”让出所有已匹配的文本到C处，将控制权交给“"”。“"”匹配失败，由于已经没有可供回溯的状态，报告整个表达式在位置11处匹配失败，一轮匹配尝试结束。  
  
正则引擎传动装置使正则向前传动，进入下一轮尝试。后续匹配过程与第一轮尝试匹配过程基本类似，可以参考图3-3。  
  
从匹配过程中可以看到，使用了排除型字符组的贪婪模式的匹配失败过程，从总体上看，大量减少了每轮回溯的次数，可以有效的提升匹配效率。  
  
3.2.4 贪婪模式匹配失败过程分析——固化分组  
通过3.2.3节的分析可以知道，由于“[^"]\*”使用了排除型字符组，那么图3-3中，在A和B之间被匹配到的字符，就一定不会是字符“"”，所以B到C之间回溯过程就是多余的，也就是说在这之间的可供回溯的状态完全可以不记录。.NET中可以使用固化分组，Java中可以使用占有优先量词来实现这一效果。

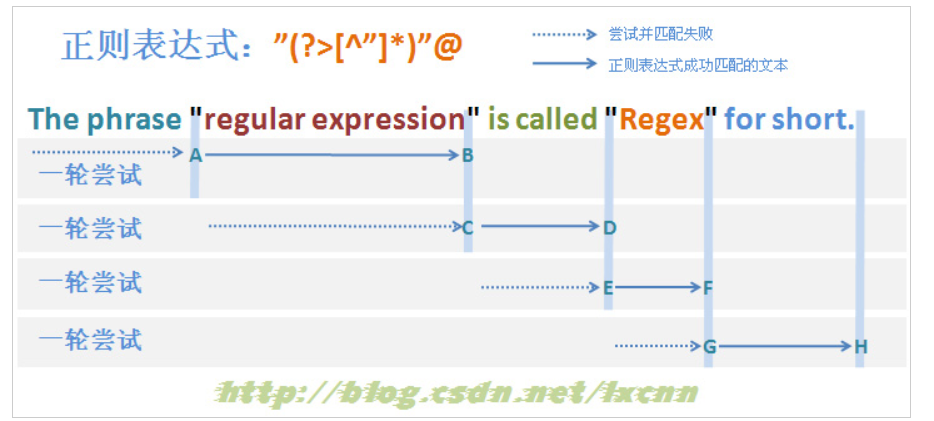


图3-4  
  
首先由“"”取得控制权，由位置0处开始尝试匹配，匹配失败，直到图中标示的A处匹配成功，控制权交给“(?>[^"]\*)”。  
  
“(?>[^"]\*)”取得控制权后，由A后面的位置开始尝试匹配，由于是贪婪模式，优先尝试匹配，一直匹配到B处，将控制权交给“"”，在这一匹配过程中，不记录任何可供回溯的状态。“"”匹配接下来的字符“””，匹配成功，将控制权交给“@”。由“@”匹配接下来的空格“ ”，匹配失败，查找可供回溯的状态，由于已经没有可供回溯的状态，报告整个表达式在位置11处匹配失败，一轮匹配尝试结束。  
  
正则引擎传动装置使正则向前传动，进入下一轮尝试。后续匹配过程与第一轮尝试匹配过程基本类似，可以参考图3-4。  
  
从匹配过程中可以看到，使用了固化分组的贪婪模式的匹配失败过程，没有涉及到回溯，可以最大限度的提升匹配效率。  
  
3.3 非贪婪模式向贪婪模式的转换  
使用匹配范围较大的子表达式时，贪婪模式与非贪婪模式匹配到的内容会有所不同，但是通过优化子表达式，非贪婪模式可以实现的匹配，贪婪模式都可以实现。  
  
比如在实际应用中，匹配img标签的内容。  
  
举例：  
  
需求：取得img标签中的图片地址，src=后固定为“””  
  
源字符串：<img class="test" src="/img/logo.gif" title="测试" />  
  
正则表达式一：<img\b.\*?src="(.\*?)".\*?>  
  
匹配结果中，捕获组1的内容即为图片地址。可以看到，这个例子中使用的都是非贪婪模式，而根据上面章节的分析，后面两个非贪婪模式都可以使用排除型字符组，将非贪婪模式转换为贪婪模式。  
  
正则表达式二：<img\b.\*?src="([^"]\*)"[^>]\*>  
  
注：“src="…"”和标签结束标记符“>”之间的属性中，也可能出现字符“>”，但那是极端情况，这里不予讨论。  
  
后两处非贪婪模式，可以通过排除型字符组转换为贪婪模式，提高匹配效率，而“src=”前的非贪婪模式，由于要排除的是一个字符序列“src=”，而不是单独的某一个或几个字符，所以不能使用排除型字符组。当然也不是没有办法，可以使用顺序环视来达到这一效果。  
  
正则表达式三：<img\b(?:(?!src=).)\*src="([^"]\*)"[^>]\*>  
  
“(?!src=).”表示这样一个字符，从它开始，右侧不能是字符序列“src=”，而“(?:(?!src=).)\*”就表示符合上面规则的字符，有0个或无限多个。这样就达到排除字符序列的目的，实现的效果同排除型字符组一样，只不过排除型字符组排除的是一个或多个字符，而这种环视结构排除的是一个或多个有序的字符序列。  
  
但是以顺序环视的方式排除字符序列，由于在匹配每一个字符时，都要进行较多的判断，所以相对于非贪婪模式，是提升效率还是降低效率，要根据实际情况进行分析。对于简单的正则表达式，或是简单的源字符串，一般来说是非贪婪模式效率高些，而对于数量较大源字符串，或是复杂的正则表达式，一般来说是贪婪模式效率高些。  
  
比如上面取得img标签中的图片地址需求，基本上用正则表达二就可以了；对于复杂的应用，如平衡组中，就需要使用结合环视的贪婪模式了。  
  
以匹配嵌套div标签的平衡组为例：  
  
Regex reg = new Regex(@"(?isx) #匹配模式，忽略大小写，“.”匹配任意字符  
  
<div[^>]\*> #开始标记“<div...>”  
  
(?> #分组构造，用来限定量词“\*”修饰范围  
  
<div[^>]\*> (?<Open>) #命名捕获组，遇到开始标记，入栈，Open计数加1  
  
| #分支结构  
  
</div> (?<-Open>) #狭义平衡组，遇到结束标记，出栈，Open计数减1  
  
| #分支结构  
  
(?:(?!</?div\b).)\* #右侧不为开始或结束标记的任意字符  
  
)\* #以上子串出现0次或任意多次  
  
(?(Open)(?!)) #判断是否还有'OPEN'，有则说明不配对，什么都不匹配  
  
</div> #结束标记“</div>”  
  
");  
  
“(?:(?!</?div\b).)\*”这里使用的就是结合环视的贪婪模式，虽然每匹一个字符都要做很多判断，但这种判断是基于字符的，速度很快，而如果这里使用非贪婪模式，那么每次要做的就是分支结构“|”的判断了，而分支结构是非常影响匹配效率的，其代价远远高于对确定字符的判断。而另外一个原因，就是贪婪模式可以结合固化分组来提升效率，而对非贪婪模式使用固化分组却是没有意义的。  
  
**4 贪婪与非贪婪——最后的回顾**  
4.1 一个例子的匹配原理回顾  
再回过头来看一下2.1.1节例子中正则，前面从应用角度进行了分析，但讨论过匹配原理后会发现，匹配过程并不是那么简单的，下面从匹配原理角度分析的匹配过程。

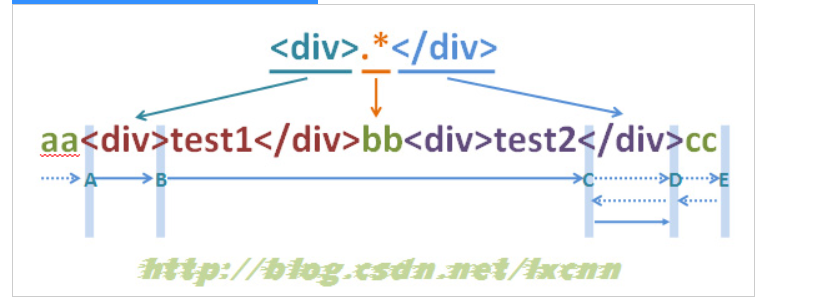


图4-1  
  
首先由“<”取得控制权，由位置0位开始尝试匹配，匹配字符“a”，匹配失败，第一轮匹配结束。第二轮匹配从位置1开始尝试匹配，同样匹配失败。第三轮从位置3开始尝试匹配，匹配字符“<”，匹配成功，控制权交给“d”。  
  
“d”尝试匹配字符“d”，匹配成功，控制权交给“i”。重复以上过程，直到由“>”匹配到字符“>”，控制权交给“.\*”。  
  
“.\*”属于贪婪模式，将从B处后的字符“t”开始，一直匹配到E处，也就是字符串结束位置，将控制权交给“<”。  
  
“<”从字符串结束位置尝试匹配，匹配失败，向前查找可供回溯的状态，把控制权交给“.\*”，由“.\*”让出一个字符“c”，把控制权再交给“<”，尝试匹配，匹配失败，向前查找可供回溯的状态。一直重复以上过程，直到“.\*”让出已匹配的字符“<”，实际上也就是让出了已匹配的子串“</div>cc”为止，“<”才匹配字符“<”成功，控制权交给“/”。  
  
接下来由“/”、“d”、“i”、“v”分别匹配对应的字符成功，此时整个正则表达式匹配完毕。  
  
**4.2 贪婪与非贪婪——量词的细节**4.2.1 区间量词的非贪婪模式  
前面提到的非贪婪模式，一直都是使用的“\*?”，而没有涉及到其它的区间量词，对于“\*?”和“+?”这样的非贪婪模式，大多数接触过正则表达式的人都可以理解，但是对于区间量词的非贪婪模式，比如“{m,n}?”，要么是没见过，要么是不理解，主要是这种应用场景非常少，所以被忽略了。  
  
首先需要明确的一点，就是量词“{m,n}”是匹配优先量词，虽然它有了上限，但是在达到上限之前，能够匹配，还是要尽可能多的匹配的。而“{m,n}?”就是对应的忽略优先量词了，在可匹配可不匹配的情况下，尽可能少的匹配。  
  
接下来举一个例子说明这种非贪婪模式的应用。  
  
举例（参考 限制字符长度与最小匹配）：  
  
需求：如何限制在长度为100的字符串中，从头匹配到最先出现的abc  
  
csdn.{1,100}abc 这样写是最大匹配(1-100个字符串中，我需要最小的)  
  
比如csdnfddabckjdsfjabc，匹配结果应为：csdnfddabc  
  
正则表达式：csdn.{1,100}?abc  
  
或许对这个例子还有人不是很理解，但是想想，其实“\*”就等价于“{0,}”，“+”就等价于“{1,}”，“\*?”也就是“{0,}?”，抽象出来也就是“{m,}?”，即上限为无穷大。如果上限为一个固定值，那就是“{m,n}?”，这样应该也就可以理解了。  
  
“{m}”没有放在匹配优先量词中，同样的，“{m}?”虽然被部分语言所支持，但是也没有放在忽略优先量词中，主要是因为这两种量词，实现的效果是一样的，只有被修饰的子表达式匹配m次才能匹配成功，且没有可供回溯的状态，所以也不存在是匹配优先还是忽略优先的问题，也就不在本文的讨论范围内。事实上即使讨论也没有意义的，只要知道它们的匹配行为也就是了。  
  
**4.2.2 忽略优先量词的匹配下限**  
对于匹配优先量词的匹配下限很好理解，“?”等价于“{0,1}”，它修饰的子表达式，最少匹配0次，最多匹配1次；“\*”等价于“{0,}”，它修饰的子表达式，最少匹配0次，最多匹配无穷多次；“+”等价于“{1,}”，它修饰的子表达式，最少匹配1次，最多匹配无穷多次。  
  
对于忽略优先量词的下限，也是容易忽略的。  
  
“??”也是忽略优先量词，被修饰的子表达式使用的也是非贪婪模式，“??”修饰的子表达式，最少匹配0次，最多匹配1次。在匹配过程中，遵循非贪婪模式匹配原则，先不匹配，即匹配0次，记录回溯状态，只有不得不匹配时，才去尝试匹配。  
  
“\*?”修饰的子表达式，最少匹配0次，最多匹配无穷多次；“+?”修饰的子表达式，最少匹配1次，最多匹配无穷多次，“+?”虽然使用的是非贪婪模式，在匹配过程中，首先要匹配一个字符，之后才是忽略匹配的，这一点也需要注意。  
  
**4.3 贪婪与非贪婪模式小结**  
Ø 从语法角度看贪婪与非贪婪  
  
被匹配优先量词修饰的子表达式，使用的是贪婪模式；被忽略优先量词修饰的子表达式，使用的是非贪婪模式。  
  
匹配优先量词包括：“{m,n}”、“{m,}”、“?”、“\*”和“+”。  
  
忽略优先量词包括：“{m,n}?”、“{m,}?”、“??”、“\*?”和“+?”。  
  
Ø 从应用角度看贪婪与非贪婪  
  
贪婪与非贪婪模式影响的是被量词修饰的子表达式的匹配行为，贪婪模式在整个表达式匹配成功的前提下，尽可能多的匹配；而非贪婪模式在整个表达式匹配成功的前提下，尽可能少的匹配。非贪婪模式只被部分NFA引擎所支持。  
  
Ø 从匹配原理角度看贪婪与非贪婪  
  
能达到同样匹配结果的贪婪与非贪婪模式，通常是贪婪模式的匹配效率较高。  
  
所有的非贪婪模式，都可以通过修改量词修饰的子表达式，转换为贪婪模式。  
  
贪婪模式可以与固化分组结合，提升匹配效率，而非贪婪模式却不可以。