　在Sun的**Java**JDK 1.40版本中，**Java**自带了支持正则表达式的包，本文就抛砖引玉地介绍了如何使用java.util.regex包。

　　可粗略估计一下，除了偶尔用**Linux**的外，其他Linu x用户都会遇到正则表达式。正则表达式是个极端强大工具，而且在字符串模式-匹配和字符串模式-替换方面富有弹性。在Unix世界里，正则表达式几乎没有什么限制，可肯定的是，它应用非常之广泛。

　　正则表达式的引擎已被许多普通的Unix工具所实现，包括grep，awk，vi和Emacs等。此外，许多使用比较广泛的脚本语言也支持正则表达式，比如**Python**，Tcl，**JavaScript**，以及最著名的Perl。

　　我很早以前就是个Perl方面的黑客，如果你和我一样话，你也会非常依赖你手边的这些强大的text-munging工具。近几年来，像其他程序开发者一样，我也越来越关注Java的开发。

　　Java作为一种开发语言，有许多值得推荐的地方，但是它一直以来没有自带对正则表达式的支持。直到最近，借助于第三方的类库，Java开始支持正则表达式，但这些第三方的类库都不一致、兼容性差，而且维护代码起来很糟糕。这个缺点，对我选择Java作为首要的开发工具来说，一直是个巨大的顾虑之处。

　　你可以想象，当我知道Sun的**java**JDK 1.40版本包含了java.util.regex(一个完全开放、自带的正则表达式包)时，是多么的高兴!很搞笑的说，我花好些时间去挖掘这个被隐藏起来的宝石。我非常惊奇的是，Java这样的一个很大改进(自带了java.util.regex包)为什么不多公开一点呢?!

　　最近，Java双脚都跳进了正则表达式的世界。java.util.regex包在支持正则表达也有它的过人之处，另外Java也提供详细的相关说明文档。使得朦朦胧胧的regex神秘景象也慢慢被拨开。有一些正则表达式的构成(可能最显著的是，在于糅合了字符类库)在Perl都找不到。

　　在regex包中，包括了两个类，Pattern(模式类)和Matcher(匹配器类)。Pattern类是用来表达和陈述所要搜索模式的对象，Matcher类是真正影响搜索的对象。另加一个新的例外类，PatternSyntaxException，当遇到不合法的搜索模式时，会抛出例外。

　　即使对正则表达式很熟悉，你会发现，通过java使用正则表达式也相当简单。要说明的一点是，对那些被Perl的单行匹配所宠坏的Perl狂热爱好者来说，在使用java的regex包进行替换操作时，会比他们所以前常用的方法费事些。

　　本文的局限之处，它不是一篇正则表达式用法的完全教程。如果读者要对正则表达进一步了解的话，推荐阅读Jeffrey Frieldl的Mastering Regular Expressions，该书由O’Reilly出版社出版。我下面就举一些例子来教读者如何使用正则表达式，以及如何更简单地去使用它。

　　设计一个简单的表达式来匹配任何电话号码数字可能是比较复杂的事情，原因在于电话号码格式有很多种情况。所有必须选择一个比较有效的模式。比如:(212) 555-1212， 212-555-1212和212 555 1212，某些人会认为它们都是等价的。

　　首先让我们构成一个正则表达式。为简单起见，先构成一个正则表达式来识别下面格式的电话号码数字:(nnn)nnn-nnnn。

　　第一步，创建一个pattern对象来匹配上面的子字符串。一旦程序运行后，如果需要的话，可以让这个对象一般化。匹配上面格式的正则表达可以这样构成:(/d{3})/s/d{3}-/d{4}，其中/d单字符类型用来匹配从0到9的任何数字，另外{3}重复符号，是个简便的记号，用来表示有3个连续的数字位，也等效于(/d/d/d)。/s也另外一个比较有用的单字符类型，用来匹配空格，比如Space键，tab键和换行符。

　　是不是很简单?但是，如果把这个正则表达式的模式用在java程序中，还要做两件事。对java的解释器来说，在反斜线字符(/)前的字符有特殊的含义。在java中，与regex有关的包，并不都能理解和识别反斜线字符(/)，尽管可以试试看。但为避免这一点，即为了让反斜线字符(/)在模式对象中被完全地传递，应该用双反斜线字符(/)。此外圆括号在正则表达中两层含义，如果想让它解释为字面上意思(即圆括号)，也需要在它前面用双反斜线字符(/)。也就是像下面的一样:

　　//(//d{3}//)//s//d{3}-//d{4}

　　现在介绍怎样在java代码中实现刚才所讲的正则表达式。要记住的事，在用正则表达式的包时，在你所定义的类前需要包含该包，也就是这样的一行:

　　import java.util.regex.\*;

　　下面的一段代码实现的功能是，从一个文本文件逐行读入，并逐行搜索电话号码数字，一旦找到所匹配的，然后输出在控制台。

　　BufferedReader in;

　　Pattern pattern = Pattern.compile("//(//d{3}//)//s//d{3}-//d{4}");

　　in = new BufferedReader(new FileReader("phone"));

　　String s;

　　while ((s = in.readLine()) != null)

　　{

　　Matcher matcher = pattern.matcher(s);

　　if (matcher.find())

　　{

　　System.out.println(matcher.group());

　　}

　　}

　　in.close();

　　对那些熟悉用**python**或**javascript**来实现正则表达式的人来说，这段代码很平常。在Python和Javascript这些语言中，或者其他的语言，这些正则表达式一旦明确地编译过后，你想用到哪里都可以。与Perl的单步匹配相比，看起来多多做了些工作，但这并不很费事。

　　find()方法，就像你所想象的，用来搜索与正则表达式相匹配的任何目标字符串，group()方法，用来返回包含了所匹配文本的字符串。应注意的是，上面的代码，仅用在每行只能含有一个匹配的电话号码数字字符串时。可以肯定的说，java的正则表达式包能用在一行含有多个匹配目标时的搜索。本文的原意在于举一些简单的例子来激起读者进一步去学习java自带的正则表达式包，所以对此就没有进行深入的探讨。

　　这相当漂亮吧! 但是很遗憾的是，这仅是个电话号码匹配器。很明显，还有两点可以改进。如果在电话号码的开头，即区位号和本地号码之间可能会有空格。我们也可匹配这些情况，则通过在正则表达式中加入/s?来实现，其中?元字符表示在模式可能有0或1个空格符。

　　第二点是，在本地号码位的前三位和后四位数字间有可能是空格符，而不是连字号，更有胜者，或根本就没有分隔符，就是7位数字连在一起。对这几种情况，我们可以用(-|)?来解决。这个结构的正则表达式就是转换器，它能匹配上面所说的几种情况。在()能含有管道符|时，它能匹配是否含有空格符或连字符，而尾部的?元字符表示是否根本没有分隔符的情况。

　　最后，区位号也可能没有包含在圆括号内，对此可以简单地在圆括号后附上?元字符，但这不是一个很好的解决方法。因为它也包含了不配对的圆括号，比如"(555" 或 "555)"。相反，我们可以通过另一种转换器来强迫让电话号码是否带有有圆括号:(/(/d{3}/)|/d{3})。如果我们把上面代码中的正则表达式用这些改进后的来替换的话，上面的代码就成了一个非常有用的电话号码数字匹配器:

　　Pattern pattern =

　　Pattern.compile("(//(//d{3}//)|//d{3})//s?//d{3}(-|)?//d{4}");

　　可以确定的是，你可以自己试着进一步改进上面的代码。

　　现在看看第二个例子，它是从Friedl的中改编过来的。其功能是用来检查文本文件中是否有重复的单词，这在印刷排版中会经常遇到，同样也是个语法检查器的问题。

　　匹配单词，像其他的一样，也可以通过好几种的正则表达式来完成。可能最直接的是/b/w+/b，其优点在于只需用少量的regex元字符。其中/w元字符用来匹配从字母a到u的任何字符。+元字符表示匹配匹配一次或多次字符，/b元字符是用来说明匹配单词的边界，它可以是空格或任何一种不同的标点符号(包括逗号，句号等)。

　　现在，我们怎样来检查一个给定的单词是否被重复了三次?为完成这个任务，需充分利用正则表达式中的所熟知的向后扫描。如前面提到的，圆括号在正则表达式中有几种不同的用法，一个就是能提供组合类型，组合类型用来保存所匹配的结果或部分匹配的结果(以便后面能用到)，即使遇到有相同的模式。在同样的正则表达中，可能(也通常期望)不止有一个组合类型。在第n个组合类型中匹配结果可以通过向后扫描来获取到。向后扫描使得搜索重复的单词非常简单:/b(/w+)/s+/1/b。

　　圆括号形成了一个组合类型，在这个正则表示中它是第一组合类型(也是仅有的一个)。向后扫描/1，指的是任何被/w+所匹配的单词。我们的正则表达式因此能匹配这样的单词，它有一个或多个空格符，后面还跟有一个与此相同的单词。注意的是，尾部的定位类型(/b)必不可少，它可以防止发生错误。如果我们想匹配"Paris in the the**spring**"，而不是匹配"Java's regex package is the theme of this article"。根据java现在的格式，则上面的正则表达式就是:Pattern pattern =Pattern.compile("//b(//w+)//s+//1//b");

　　最后进一步的修改是让我们的匹配器对大小写敏感。比如，下面的情况:"The the theme of this article is the Java's regex package."，这一点在regex中能非常简单地实现，即通过使用在Pattern类中预定义的静态标志CASE\_INSENSITIVE :

　　Pattern pattern =Pattern.compile("//b(//w+)//s+//1//b",

　　Pattern.CASE\_INSENSITIVE);

　　有关正则表达式的话题是非常丰富，而且复杂的，用Java来实现也非常广泛，则需要对regex包进行的彻底研究，我们在这里所讲的只是冰山一角。即使你对正则表达式比较陌生，使用regex包后会很快发现它强大功能和可伸缩性。如果你是个来自Perl或其他语言王国的老练的正则表达式的黑客，使用过regex包后，你将会安心地投入到java的世界，而放弃其他的工具，并把java的regex包看成是手边必备的利器。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CharSequence**  JDK 1.4定义了一个新的接口，叫CharSequence。它提供了String和StringBuffer这两个类的字符序列的抽象：  interface CharSequence {  charAt(int i);  length();  subSequence(int start, int end);  toString(); }  为了实现这个新的CharSequence接口，String，StringBuffer以及CharBuffer都作了修改。很多正则表达式的操作都要拿CharSequence作参数。  **Pattern和Matcher**  先给一个例子。下面这段程序可以**测试**正则表达式是否匹配字符串。第一个参数是要匹配的字符串，后面是正则表达式。正则表达式可以有多个。在Unix/**linux**环境下，命令行下的正则表达式还必须用引号。   |  | | --- | | //: c12:TestRegularExpression.java // Allows you to easly try out regular expressions. // {Args: abcabcabcdefabc "abc+" "(abc)+" "(abc){2,}" } import java.util.regex.\*; publicclass TestRegularExpression { publicstaticvoid main(String[] args) { if(args.length < 2) {  System.out.println("Usage:/n" + "java TestRegularExpression " + "characterSequence regularExpression+");  System.exit(0);  }  System.out.println("Input: /"" + args[0] + "/""); for(int i = 1; i < args.length; i++) {  System.out.println( "Regular expression: /"" + args[i] + "/"");  Pattern p = Pattern.compile(args[i]);  Matcher m = p.matcher(args[0]); while(m.find()) {  System.out.println("Match /"" + m.group() + "/" at positions " +  m.start() + "-" + (m.end() - 1));  }  }  } } ///:~ |   **Java的正则表达式是由java.util.regex的Pattern和Matcher类实现的**。Pattern对象表示经编译的正则表达式。静态的compile( )方法负责将表示正则表达式的字符串编译成Pattern对象。正如上述例程所示的，只要给Pattern的matcher( )方法送一个字符串就能获取一个Matcher对象。此外，Pattern还有一个能快速判断能否在input里面找到regex的  staticboolean matches(?regex, ?input)  以及能返回String数组的split( )方法，它能用regex把字符串分割开来。  只要给Pattern.matcher( )方法传一个字符串就能获得Matcher对象了。接下来就能用Matcher的方法来查询匹配的结果了。  boolean matches() boolean lookingAt() boolean find() boolean find(int start)  matches( )的前提是Pattern匹配整个字符串，而lookingAt( )的意思是Pattern匹配字符串的开头。  **find( )**  Matcher.find( )的功能是发现CharSequence里的，与pattern相匹配的多个字符序列。例如：   |  | | --- | | //: c12:FindDemo.java import java.util.regex.\*; import com.bruceeckel.simpletest.\*; import java.util.\*; publicclass FindDemo { privatestatic Test monitor = new Test(); publicstaticvoid main(String[] args) {  Matcher m = Pattern.compile("//w+")  .matcher("Evening is full of the linnet's wings"); while(m.find())  System.out.println(m.group()); int i = 0; while(m.find(i)) {  System.out.print(m.group() + " ");  i++;  }  monitor.expect(new String[] { "Evening", "is", "full", "of", "the", "linnet", "s", "wings", "Evening vening ening ning ing ng g is is s full " + "full ull ll l of of f the the he e linnet linnet " + "innet nnet net et t s s wings wings ings ngs gs s "  });  } } ///:~ |   "//w+"的意思是"一个或多个单词字符"，因此它会将字符串直接分解成单词。find( )像一个迭代器，从头到尾扫描一遍字符串。第二个find( )是带int参数的，正如你所看到的，它会告诉方法从哪里开始找——即从参数位置开始查找。  **Groups**  Group是指里用括号括起来的，能被后面的表达式调用的正则表达式。Group 0 表示整个表达式，group 1表示第一个被括起来的group，以此类推。所以；  A(B(C))D  里面有三个group：group 0是ABCD， group 1是BC，group 2是C。  你可以用下述Matcher方法来使用group：  public int groupCount( )返回matcher对象中的group的数目。不包括group0。  public String group( ) 返回上次匹配操作(比方说find( ))的group 0(整个匹配)  public String group(int i)返回上次匹配操作的某个group。如果匹配成功，但是没能找到group，则返回null。  public int start(int group)返回上次匹配所找到的，group的开始位置。  public int end(int group)返回上次匹配所找到的，group的结束位置，最后一个字符的下标加一。   |  | | --- | | //: c12:Groups.java import java.util.regex.\*; import com.bruceeckel.simpletest.\*; publicclass Groups { privatestatic Test monitor = new Test(); staticpublicfinal String poem = "Twas brillig, and the slithy toves/n" + "Did gyre and gimble in the wabe./n" + "All mimsy were the borogoves,/n" + "And the mome raths outgrabe./n/n" + "Beware the Jabberwock, my son,/n" + "The jaws that bite, the claws that catch./n" + "Beware the Jubjub bird, and shun/n" + "The frumious Bandersnatch."; publicstaticvoid main(String[] args) {  Matcher m =  Pattern.compile("(?m)(//S+)//s+((//S+)//s+(//S+))\_\_\_FCKpd\_\_\_6quot;)  .matcher(poem); while(m.find()) { for(int j = 0; j <= m.groupCount(); j++)  System.out.print("[" + m.group(j) + "]");  System.out.println();  }  monitor.expect(new String[]{ "[the slithy toves]" + "[the][slithy toves][slithy][toves]", "[in the wabe.][in][the wabe.][the][wabe.]", "[were the borogoves,]" + "[were][the borogoves,][the][borogoves,]", "[mome raths outgrabe.]" + "[mome][raths outgrabe.][raths][outgrabe.]", "[Jabberwock, my son,]" + "[Jabberwock,][my son,][my][son,]", "[claws that catch.]" + "[claws][that catch.][that][catch.]", "[bird, and shun][bird,][and shun][and][shun]", "[The frumious Bandersnatch.][The]" + "[frumious Bandersnatch.][frumious][Bandersnatch.]"  });  } } ///:~ |   这首诗是***Through the Looking Glass***的，Lewis Carroll的"Jabberwocky"的第一部分。可以看到这个正则表达式里有很多用括号括起来的group，它是由任意多个连续的非空字符('/S+')和任意多个连续的空格字符('/s+')所组成的，其最终目的是要捕获每行的最后三个单词；'$'表示一行的结尾。但是'$'通常表示整个字符串的结尾，所以这里要明确地告诉正则表达式注意换行符。这一点是由'(?m)'标志完成的(模式标志会过一会讲解)。  **start( )和end( )**  如果匹配成功，start( )会返回此次匹配的开始位置，end( )会返回此次匹配的结束位置，即最后一个字符的下标加一。如果之前的匹配不成功(或者没匹配)，那么无论是调用start( )还是end( )，都会引发一个IllegalStateException。下面这段程序还演示了matches( )和lookingAt( )：   |  | | --- | | //: c12:StartEnd.java import java.util.regex.\*; import com.bruceeckel.simpletest.\*; publicclass StartEnd { privatestatic Test monitor = new Test(); publicstaticvoid main(String[] args) {  String[] input = new String[] { "Java has regular expressions in 1.4", "regular expressions now expressing in Java", "Java represses oracular expressions"  };  Pattern  p1 = Pattern.compile("re//w\*"),  p2 = Pattern.compile("Java.\*"); for(int i = 0; i < input.length; i++) {  System.out.println("input " + i + ": " + input[i]);  Matcher  m1 = p1.matcher(input[i]),  m2 = p2.matcher(input[i]); while(m1.find())  System.out.println("m1.find() '" + m1.group() + "' start = "+ m1.start() + " end = " + m1.end()); while(m2.find())  System.out.println("m2.find() '" + m2.group() + "' start = "+ m2.start() + " end = " + m2.end()); if(m1.lookingAt()) // No reset() necessary  System.out.println("m1.lookingAt() start = "  + m1.start() + " end = " + m1.end()); if(m2.lookingAt())  System.out.println("m2.lookingAt() start = "  + m2.start() + " end = " + m2.end()); if(m1.matches()) // No reset() necessary  System.out.println("m1.matches() start = "  + m1.start() + " end = " + m1.end()); if(m2.matches())  System.out.println("m2.matches() start = "  + m2.start() + " end = " + m2.end());  }  monitor.expect(new String[] { "input 0: Java has regular expressions in 1.4", "m1.find() 'regular' start = 9 end = 16", "m1.find() 'ressions' start = 20 end = 28", "m2.find() 'Java has regular expressions in 1.4'" + " start = 0 end = 35", "m2.lookingAt() start = 0 end = 35", "m2.matches() start = 0 end = 35", "input 1: regular expressions now " + "expressing in Java", "m1.find() 'regular' start = 0 end = 7", "m1.find() 'ressions' start = 11 end = 19", "m1.find() 'ressing' start = 27 end = 34", "m2.find() 'Java' start = 38 end = 42", "m1.lookingAt() start = 0 end = 7", "input 2: Java represses oracular expressions", "m1.find() 'represses' start = 5 end = 14", "m1.find() 'ressions' start = 27 end = 35", "m2.find() 'Java represses oracular expressions' " + "start = 0 end = 35", "m2.lookingAt() start = 0 end = 35", "m2.matches() start = 0 end = 35"  });  } } ///:~ |   注意，只要字符串里有这个模式，find( )就能把它给找出来，但是lookingAt( )和matches( )，只有在字符串与正则表达式一开始就相匹配的情况下才能返回true。matches( )成功的前提是正则表达式与字符串完全匹配，而lookingAt( )成功的前提是，字符串的开始部分与正则表达式相匹配。  **匹配的模式(Pattern flags)**  compile( )方法还有一个版本，它需要一个控制正则表达式的匹配行为的参数：  Pattern Pattern.compile(String regex, int flag)  flag的取值范围如下：   |  |  | | --- | --- | | **编译标志** | **效果** | | Pattern.CANON\_EQ | 当且仅当两个字符的"正规分解(canonical decomposition)"都完全相同的情况下，才认定匹配。比如用了这个标志之后，表达式"a/u030A"会匹配"?"。默认情况下，不考虑"规范相等性(canonical equivalence)"。 | | Pattern.CASE\_INSENSITIVE (?i) | 默认情况下，大小写不明感的匹配只适用于US-ASCII字符集。这个标志能让表达式忽略大小写进行匹配。要想对Unicode字符进行大小不明感的匹配，只要将UNICODE\_CASE与这个标志合起来就行了。 | | Pattern.COMMENTS (?x) | 在这种模式下，匹配时会忽略(正则表达式里的)空格字符(注：不是指表达式里的"//s"，而是指表达式里的空格，tab，回车之类)。注释从#开始，一直到这行结束。可以通过嵌入式的标志来启用Unix行模式。 | | Pattern.DOTALL (?s) | 在这种模式下，表达式'.'可以匹配任意字符，包括表示一行的结束符。默认情况下，表达式'.'不匹配行的结束符。 | | Pattern.MULTILINE (?m) | 在这种模式下，'^'和'$'分别匹配一行的开始和结束。此外，'^'仍然匹配字符串的开始，'$'也匹配字符串的结束。默认情况下，这两个表达式仅仅匹配字符串的开始和结束。 | | Pattern.UNICODE\_CASE (?u) | 在这个模式下，如果你还启用了CASE\_INSENSITIVE标志，那么它会对Unicode字符进行大小写不明感的匹配。默认情况下，大小写不明感的匹配只适用于US-ASCII字符集。 | | Pattern.UNIX\_LINES (?d) | 在这个模式下，只有'/n'才被认作一行的中止，并且与'.'，'^'，以及'$'进行匹配。 |   在这些标志里面，Pattern.CASE\_INSENSITIVE，Pattern.MULTILINE，以及Pattern.COMMENTS是最有用的(其中Pattern.COMMENTS还能帮我们把思路理清楚，并且/或者做文档)。注意，你可以用在表达式里插记号的方式来启用绝大多数的模式。这些记号就在上面那张表的各个标志的下面。你希望模式从哪里开始启动，就在哪里插记号。  可以用"OR" ('|')运算符把这些标志合使用：   |  | | --- | | //: c12:ReFlags.java import java.util.regex.\*; import com.bruceeckel.simpletest.\*; publicclass ReFlags { privatestatic Test monitor = new Test(); publicstaticvoid main(String[] args) {  Pattern p = Pattern.compile("^java",  Pattern.CASE\_INSENSITIVE | Pattern.MULTILINE);  Matcher m = p.matcher( "java has regex/nJava has regex/n" + "JAVA has pretty good regular expressions/n" + "Regular expressions are in Java"); while(m.find())  System.out.println(m.group());  monitor.expect(new String[] { "java", "Java", "JAVA"  });  } } ///:~ |   这样创建出来的正则表达式就能匹配以"java"，"Java"，"JAVA"...开头的字符串了。此外，如果字符串分好几行，那它还会对每一行做匹配(匹配始于字符序列的开始，终于字符序列当中的行结束符)。注意，group( )方法仅返回匹配的部分。  **split( )**  所谓分割是指将以正则表达式为界，将字符串分割成String数组。  String[] split(CharSequence charseq) String[] split(CharSequence charseq, int limit)  这是一种既快又方便地将文本根据一些常见的边界标志分割开来的方法。   |  | | --- | | //: c12:SplitDemo.java import java.util.regex.\*; import com.bruceeckel.simpletest.\*; import java.util.\*; publicclass SplitDemo { privatestatic Test monitor = new Test(); publicstaticvoid main(String[] args) {  String input = "This!!unusual use!!of exclamation!!points";  System.out.println(Arrays.asList(  Pattern.compile("!!").split(input))); // Only do the first three:  System.out.println(Arrays.asList(  Pattern.compile("!!").split(input, 3)));  System.out.println(Arrays.asList( "Aha! String has a split() built in!".split(" ")));  monitor.expect(new String[] { "[This, unusual use, of exclamation, points]", "[This, unusual use, of exclamation!!points]", "[Aha!, String, has, a, split(), built, in!]"  });  } } ///:~ |   第二个split( )会限定分割的次数。  正则表达式是如此重要，以至于有些功能被加进了String类，其中包括split( )(已经看到了)，matches( )，replaceFirst( )以及replaceAll( )。这些方法的功能同Pattern和Matcher的相同。  **替换操作**  正则表达式在替换文本方面特别在行。下面就是一些方法：  replaceFirst(String replacement)将字符串里，第一个与模式相匹配的子串替换成replacement。  replaceAll(String replacement)，将输入字符串里所有与模式相匹配的子串全部替换成replacement。  appendReplacement(StringBuffer sbuf, String replacement)对sbuf进行逐次替换，而不是像replaceFirst( )或replaceAll( )那样，只替换第一个或全部子串。这是个非常重要的方法，因为它可以调用方法来生成replacement(replaceFirst( )和replaceAll( )只允许用固定的字符串来充当replacement)。有了这个方法，你就可以编程区分group，从而实现更强大的替换功能。  调用完appendReplacement( )之后，为了把剩余的字符串拷贝回去，必须调用appendTail(StringBuffer sbuf, String replacement)。  下面我们来演示一下怎样使用这些替换方法。说明一下，这段程序所处理的字符串是它自己开头部分的注释，是用正则表达式提取出来并加以处理之后再传给替换方法的。   |  | | --- | | //: c12:TheReplacements.java import java.util.regex.\*; import java.io.\*; import com.bruceeckel.util.\*; import com.bruceeckel.simpletest.\*; /\*! Here's a block of text to use as input to  the regular expression matcher. Note that we'll  first extract the block of text by looking for  the special delimiters, then process the  extracted block. !\*/ publicclass TheReplacements { privatestatic Test monitor = new Test(); publicstaticvoid main(String[] args) throws Exception {  String s = TextFile.read("TheReplacements.java"); // Match the specially-commented block of text above:  Matcher mInput =  Pattern.compile("///\*!(.\*)!//\*/", Pattern.DOTALL)  .matcher(s); if(mInput.find())  s = mInput.group(1); // Captured by parentheses // Replace two or more spaces with a single space:  s = s.replaceAll(" {2,}", " "); // Replace one or more spaces at the beginning of each // line with no spaces. Must enable MULTILINE mode:  s = s.replaceAll("(?m)^ +", "");  System.out.println(s);  s = s.replaceFirst("[aeiou]", "(VOWEL1)");  StringBuffer sbuf = new StringBuffer();  Pattern p = Pattern.compile("[aeiou]");  Matcher m = p.matcher(s); // Process the find information as you // perform the replacements: while(m.find())  m.appendReplacement(sbuf, m.group().toUpperCase()); // Put in the remainder of the text:  m.appendTail(sbuf);  System.out.println(sbuf);  monitor.expect(new String[]{ "Here's a block of text to use as input to", "the regular expression matcher. Note that we'll", "first extract the block of text by looking for", "the special delimiters, then process the", "extracted block. ", "H(VOWEL1)rE's A blOck Of tExt tO UsE As InpUt tO", "thE rEgUlAr ExprEssIOn mAtchEr. NOtE thAt wE'll", "fIrst ExtrAct thE blOck Of tExt by lOOkIng fOr", "thE spEcIAl dElImItErs, thEn prOcEss thE", "ExtrActEd blOck. "  });  } } ///:~ |   用TextFile.read( )方法来打开和读取文件。mInput的功能是匹配'/\*!' 和 '!\*/' 之间的文本(注意一下分组用的括号)。接下来，我们将所有两个以上的连续空格全都替换成一个，并且将各行开头的空格全都去掉(为了让这个正则表达式能对所有的行，而不仅仅是第一行起作用，必须启用多行模式)。这两个操作都用了String的replaceAll( )(这里用它更方便)。注意，由于每个替换只做一次，因此除了预编译Pattern之外，程序没有额外的开销。  replaceFirst( )只替换第一个子串。此外，replaceFirst( )和replaceAll( )只能用常量(literal)来替换，所以如果每次替换的时候还要进行一些操作的话，它们是无能为力的。碰到这种情况，得用appendReplacement( )，它能在进行替换的时候想写多少代码就写多少。在上面那段程序里，创建sbuf的过程就是选group做处理，也就是用正则表达式把元音字母找出来，然后换成大写的过程。通常你得在完成全部的替换之后才调用appendTail( )，但是如果要模仿replaceFirst( )(或"replace n")的效果，你也可以只替换一次就调用appendTail( )。它会把剩下的东西全都放进sbuf。  你还可以在appendReplacement( )的replacement参数里用"$g"引用已捕获的group，其中'g' 表示group的号码。不过这是为一些比较简单的操作准备的，因而其效果无法与上述程序相比。  **reset( )**  此外，还可以用reset( )方法给现有的Matcher对象配上个新的CharSequence。   |  | | --- | | //: c12:Resetting.java import java.util.regex.\*; import java.io.\*; import com.bruceeckel.simpletest.\*; publicclass Resetting { privatestatic Test monitor = new Test(); publicstaticvoid main(String[] args) throws Exception {  Matcher m = Pattern.compile("[frb][aiu][gx]")  .matcher("fix the rug with bags"); while(m.find())  System.out.println(m.group());  m.reset("fix the rig with rags"); while(m.find())  System.out.println(m.group());  monitor.expect(new String[]{ "fix", "rug", "bag", "fix", "rig", "rag"  });  } } ///:~ |   如果不给参数，reset( )会把Matcher设到当前字符串的开始处。 |

|  |
| --- |
| 如果你曾经用过Perl或任何其他内建正则表达式支持的语言，你一定知道用正则表达式处理文本和匹配模式是多么简单。如果你不熟悉这个术语，那么“正则表达式”（Regular Expression）就是一个字符构成的串，它定义了一个用来搜索匹配字符串的模式。 |

|  |
| --- |
| 许多语言，包括Perl、PHP、Python、JavaScript和JScript，都支持用正则表达式处理文本，一些文本编辑器用正则表达式实现高级“搜索-替换”功能。那么Java又怎样呢？本文写作时，一个包含了用正则表达式进行文本处理的Java规范需求（Specification Request）已经得到认可，你可以期待在JDK的下一版本中看到它。 |

|  |
| --- |
| 然而，如果现在就需要使用正则表达式，又该怎么办呢？你可以从Apache.org下载源代码开放的Jakarta-ORO库。本文接下来的内容先简要地介绍正则表达式的入门知识，然后以Jakarta-ORO API为例介绍如何使用正则表达式。 |

|  |
| --- |
| **一、正则表达式基础知识** |

|  |
| --- |
| 我们先从简单的开始。假设你要搜索一个包含字符“cat”的字符串，搜索用的正则表达式就是“cat”。如果搜索对大小写不敏感，单词“catalog”、“Catherine”、“sophisticated”都可以匹配。也就是说： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/j43nu4ndo6z8.jpg |

|  |
| --- |
| **1.1 句点符号** |

|  |
| --- |
| 假设你在玩英文拼字游戏，想要找出三个字母的单词，而且这些单词必须以“t”字母开头，以“n”字母结束。另外，假设有一本英文字典，你可以用正则表达式搜索它的全部内容。要构造出这个正则表达式，你可以使用一个通配符——句点符号“.”。这样，完整的表达式就是“t.n”，它匹配“tan”、“ten”、“tin”和“ton”，还匹配“t#n”、“tpn”甚至“t n”，还有其他许多无意义的组合。这是因为句点符号匹配所有字符，包括空格、Tab字符甚至换行符： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/9i9f280hrzkz.jpg |

|  |
| --- |
| **1.2 方括号符号** |

|  |
| --- |
| 为了解决句点符号匹配范围过于广泛这一问题，你可以在方括号（“[]”）里面指定看来有意义的字符。此时，只有方括号里面指定的字符才参与匹配。也就是说，正则表达式“t[aeio]n”只匹配“tan”、“Ten”、“tin”和“ton”。但“Toon”不匹配，因为在方括号之内你只能匹配单个字符： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/t1e74w8itp99.jpg |

|  |
| --- |
| **1.3 “或”符号** |

|  |
| --- |
| 如果除了上面匹配的所有单词之外，你还想要匹配“toon”，那么，你可以使用“|”操作符。“|”操作符的基本意义就是“或”运算。要匹配“toon”，使用“t(a|e|i|o|oo)n”正则表达式。这里不能使用方扩号，因为方括号只允许匹配单个字符；这里必须使用圆括号“()”。圆括号还可以用来分组，具体请参见后面介绍。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/i1fus8ynpz96.jpg |

|  |
| --- |
| **1.4 表示匹配次数的符号** |

|  |
| --- |
| 表一显示了表示匹配次数的符号，这些符号用来确定紧靠该符号左边的符号出现的次数： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/8k41sc2nzdy7.jpg |

|  |
| --- |
| 假设我们要在文本文件中搜索美国的社会安全号码。这个号码的格式是999-99-9999。用来匹配它的正则表达式如图一所示。在正则表达式中，连字符（“-”）有着特殊的意义，它表示一个范围，比如从0到9。因此，匹配社会安全号码中的连字符号时，它的前面要加上一个转义字符“/”。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/d0461e0196rw.gif |

|  |
| --- |
| 图一：匹配所有123-12-1234形式的社会安全号码 |

|  |
| --- |
| 假设进行搜索的时候，你希望连字符号可以出现，也可以不出现——即，999-99-9999和999999999都属于正确的格式。这时，你可以在连字符号后面加上“？”数量限定符号，如图二所示： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/4s8b1dwh8778.gif |

|  |
| --- |
| 图二：匹配所有123-12-1234和123121234形式的社会安全号码 |

|  |
| --- |
| 下面我们再来看另外一个例子。美国汽车牌照的一种格式是四个数字加上二个字母。它的正则表达式前面是数字部分“[0-9]{4}”，再加上字母部分“[A-Z]{2}”。图三显示了完整的正则表达式。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/8uvnyokp8pix.gif |

|  |
| --- |
| 图三：匹配典型的美国汽车牌照号码，如8836KV |

|  |
| --- |
| 1.5 “否”符号 |

|  |
| --- |
| “^”符号称为“否”符号。如果用在方括号内，“^”表示不想要匹配的字符。例如，图四的正则表达式匹配所有单词，但以“X”字母开头的单词除外。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/47z03c6v2q6z.gif |

|  |
| --- |
| 图四：匹配所有单词，但“X”开头的除外 |

|  |
| --- |
| 1.6 圆括号和空白符号 |

|  |
| --- |
| 假设要从格式为“June 26, 1951”的生日日期中提取出月份部分，用来匹配该日期的正则表达式可以如图五所示： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/2w4mut5a8vjn.gif |

|  |
| --- |
| 图五：匹配所有Moth DD,YYYY格式的日期 |

|  |
| --- |
| 新出现的“/s”符号是空白符号，匹配所有的空白字符，包括Tab字符。如果字符串正确匹配，接下来如何提取出月份部分呢？只需在月份周围加上一个圆括号创建一个组，然后用ORO API（本文后面详细讨论）提取出它的值。修改后的正则表达式如图六所示： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/c677z4111jo2.gif |

|  |
| --- |
| 图六：匹配所有Month DD,YYYY格式的日期，定义月份值为第一个组 |

|  |
| --- |
| **1.7 其它符号** |

|  |
| --- |
| 为简便起见，你可以使用一些为常见正则表达式创建的快捷符号。如表二所示： |

|  |
| --- |
| 表二：常用符号 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/32xa8u62viqw.jpg |

|  |
| --- |
| 例如，在前面社会安全号码的例子中，所有出现“[0-9]”的地方我们都可以使用“/d”。修改后的正则表达式如图七所示： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/4k2mcvrp04b7.gif |

|  |
| --- |
| 图七：匹配所有123-12-1234格式的社会安全号码 |

|  |
| --- |
| **二、Jakarta-ORO库** |

|  |
| --- |
| 有许多源代码开放的正则表达式库可供Java程序员使用，而且它们中的许多支持Perl 5兼容的正则表达式语法。我在这里选用的是Jakarta-ORO正则表达式库，它是最全面的正则表达式API之一，而且它与Perl 5正则表达式完全兼容。另外，它也是优化得最好的API之一。 |

|  |
| --- |
| Jakarta-ORO库以前叫做OROMatcher，Daniel Savarese大方地把它赠送给了Jakarta Project。你可以按照本文最后参考资源的说明下载它。 |

|  |
| --- |
| 我首先将简要介绍使用Jakarta-ORO库时你必须创建和访问的对象，然后介绍如何使用Jakarta-ORO API。 |

|  |
| --- |
| **▲ PatternCompiler对象** |

|  |
| --- |
| 首先，创建一个Perl5Compiler类的实例，并把它赋值给PatternCompiler接口对象。Perl5Compiler是PatternCompiler接口的一个实现，允许你把正则表达式编译成用来匹配的Pattern对象。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/x4z3923ca13a.jpg |

|  |
| --- |
| **▲ Pattern对象** |

|  |
| --- |
| 要把正则表达式编译成Pattern对象，调用compiler对象的compile()方法，并在调用参数中指定正则表达式。例如，你可以按照下面这种方式编译正则表达式“t[aeio]n”： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/2fd26rskmll5.jpg |

|  |
| --- |
| 默认情况下，编译器创建一个大小写敏感的模式（pattern）。因此，上面代码编译得到的模式只匹配“tin”、“tan”、 “ten”和“ton”，但不匹配“Tin”和“taN”。要创建一个大小写不敏感的模式，你应该在调用编译器的时候指定一个额外的参数： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/l56lk1spu66k.jpg |

|  |
| --- |
| 创建好Pattern对象之后，你就可以通过PatternMatcher类用该Pattern对象进行模式匹配。 |

|  |
| --- |
| **▲ PatternMatcher对象** |

|  |
| --- |
| PatternMatcher对象根据Pattern对象和字符串进行匹配检查。你要实例化一个Perl5Matcher类并把结果赋值给PatternMatcher接口。Perl5Matcher类是PatternMatcher接口的一个实现，它根据Perl 5正则表达式语法进行模式匹配： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/an6co93nb236.jpg |

|  |
| --- |
| 使用PatternMatcher对象，你可以用多个方法进行匹配操作，这些方法的第一个参数都是需要根据正则表达式进行匹配的字符串： |

|  |
| --- |
| · boolean matches(String input, Pattern pattern)：当输入字符串和正则表达式要精确匹配时使用。换句话说，正则表达式必须完整地描述输入字符串。 |

|  |
| --- |
| · boolean matchesPrefix(String input, Pattern pattern)：当正则表达式匹配输入字符串起始部分时使用。 |

|  |
| --- |
| · boolean contains(String input, Pattern pattern)：当正则表达式要匹配输入字符串的一部分时使用（即，它必须是一个子串）。 |

|  |
| --- |
| 另外，在上面三个方法调用中，你还可以用PatternMatcherInput对象作为参数替代String对象；这时，你可以从字符串中最后一次匹配的位置开始继续进行匹配。当字符串可能有多个子串匹配给定的正则表达式时，用PatternMatcherInput对象作为参数就很有用了。用PatternMatcherInput对象作为参数替代String时，上述三个方法的语法如下： |

|  |
| --- |
| · boolean matches(PatternMatcherInput input, Pattern pattern) |

|  |
| --- |
| · boolean matchesPrefix(PatternMatcherInput input, Pattern pattern) |

|  |
| --- |
| · boolean contains(PatternMatcherInput input, Pattern pattern) |

|  |
| --- |
| **三、应用实例** |

|  |
| --- |
| 下面我们来看看Jakarta-ORO库的一些应用实例。 |

|  |
| --- |
| **3.1 日志文件处理** |

|  |
| --- |
| 任务：分析一个Web服务器日志文件，确定每一个用户花在网站上的时间。在典型的BEA WebLogic日志文件中，日志记录的格式如下： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/242u5d50m3fr.jpg |

|  |
| --- |
| 分析这个日志记录，可以发现，要从这个日志文件提取的内容有两项：IP地址和页面访问时间。你可以用分组符号（圆括号）从日志记录提取出IP地址和时间标记。 |

|  |
| --- |
| 首先我们来看看IP地址。IP地址有4个字节构成，每一个字节的值在0到255之间，各个字节通过一个句点分隔。因此，IP地址中的每一个字节有至少一个、最多三个数字。图八显示了为IP地址编写的正则表达式： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/21g3c5o3k67u.gif |

|  |
| --- |
| 图八：匹配IP地址 |

|  |
| --- |
| IP地址中的句点字符必须进行转义处理（前面加上“/”），因为IP地址中的句点具有它本来的含义，而不是采用正则表达式语法中的特殊含义。句点在正则表达式中的特殊含义本文前面已经介绍。 |

|  |
| --- |
| 日志记录的时间部分由一对方括号包围。你可以按照如下思路提取出方括号里面的所有内容：首先搜索起始方括号字符（“[”），提取出所有不超过结束方括号字符（“]”）的内容，向前寻找直至找到结束方括号字符。图九显示了这部分的正则表达式。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/cg2u2j7n0mpg.gif |

|  |
| --- |
| 图九：匹配至少一个字符，直至找到“]” |

|  |
| --- |
| 现在，把上述两个正则表达式加上分组符号（圆括号）后合并成单个表达式，这样就可以从日志记录提取出IP地址和时间。注意，为了匹配“- -”（但不提取它），正则表达式中间加入了“/s-/s-/s”。完整的正则表达式如图十所示。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/157qqoq9ve8w.gif |

|  |
| --- |
| 图十：匹配IP地址和时间标记 |

|  |
| --- |
| 现在正则表达式已经编写完毕，接下来可以编写使用正则表达式库的Java代码了。 |

|  |
| --- |
| 为使用Jakarta-ORO库，首先创建正则表达式字符串和待分析的日志记录字符串： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/759353ra1907.jpg |

|  |
| --- |
| 这里使用的正则表达式与图十的正则表达式差不多完全相同，但有一点例外：在Java中，你必须对每一个向前的斜杠（“/”）进行转义处理。图十不是Java的表示形式，所以我们要在每个“/”前面加上一个“/”以免出现编译错误。遗憾的是，转义处理过程很容易出现错误，所以应该小心谨慎。你可以首先输入未经转义处理的正则表达式，然后从左到右依次把每一个“/”替换成“//”。如果要复检，你可以试着把它输出到屏幕上。 |

|  |
| --- |
| 初始化字符串之后，实例化PatternCompiler对象，用PatternCompiler编译正则表达式创建一个Pattern对象： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/4o3y3mdkct2x.jpg |

|  |
| --- |
| 现在，创建PatternMatcher对象，调用PatternMatcher接口的contain()方法检查匹配情况： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/75jn387mn42x.jpg |

|  |
| --- |
| 接下来，利用PatternMatcher接口返回的MatchResult对象，输出匹配的组。由于logEntry字符串包含匹配的内容，你可以看到类如下面的输出： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/vgpi8nu05ivw.jpg |

|  |
| --- |
| **3.2 HTML处理实例一** |

|  |
| --- |
| 下面一个任务是分析HTML页面内FONT标记的所有属性。HTML页面内典型的FONT标记如下所示： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/606h60377ssy.jpg |

|  |
| --- |
| 程序将按照如下形式，输出每一个FONT标记的属性： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/brf9m2df9ieo.jpg |

|  |
| --- |
| 在这种情况下，我建议你使用两个正则表达式。第一个如图十一所示，它从字体标记提取出“"face="Arial, Serif" size="+2" color="red"”。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/ru4fdvu24otu.gif |

|  |
| --- |
| 图十一：匹配FONT标记的所有属性 |

|  |
| --- |
| 第二个正则表达式如图十二所示，它把各个属性分割成名字-值对。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/7jng3h91d45k.gif |

|  |
| --- |
| 图十二：匹配单个属性，并把它分割成名字-值对 |

|  |
| --- |
| 分割结果为： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/7jjzyu7lfs84.jpg |

|  |
| --- |
| 现在我们来看看完成这个任务的Java代码。首先创建两个正则表达式字符串，用Perl5Compiler把它们编译成Pattern对象。编译正则表达式的时候，指定Perl5Compiler.CASE\_INSENSITIVE\_MASK选项，使得匹配操作不区分大小写。 |

|  |
| --- |
| 接下来，创建一个执行匹配操作的Perl5Matcher对象。 |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/5cf25vch72r5.jpg |

|  |
| --- |
| 假设有一个String类型的变量html，它代表了HTML文件中的一行内容。如果html字符串包含FONT标记，匹配器将返回true。此时，你可以用匹配器对象返回的MatchResult对象获得第一个组，它包含了FONT的所有属性： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/6z605d16f9gp.jpg |

|  |
| --- |
| 接下来创建一个PatternMatcherInput对象。这个对象允许你从最后一次匹配的位置开始继续进行匹配操作，因此，它很适合于提取FONT标记内属性的名字-值对。创建PatternMatcherInput对象，以参数形式传入待匹配的字符串。然后，用匹配器实例提取出每一个FONT的属性。这通过指定PatternMatcherInput对象（而不是字符串对象）为参数，反复地调用PatternMatcher对象的contains()方法完成。PatternMatcherInput对象之中的每一次迭代将把它内部的指针向前移动，下一次检测将从前一次匹配位置的后面开始。 |

|  |
| --- |
| 本例的输出结果如下： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/308i497h04uz.jpg |

|  |
| --- |
| **3.3 HTML处理实例二** |

|  |
| --- |
| 下面我们来看看另一个处理HTML的例子。这一次，我们假定Web服务器从widgets.acme.com移到了newserver.acme.com。现在你要修改一些页面中的链接： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/17341kb8z3di.jpg |

|  |
| --- |
| 执行这个搜索的正则表达式如图十三所示： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/88277rk37770.gif |

|  |
| --- |
| 图十三：匹配修改前的链接 |

|  |
| --- |
| 如果能够匹配这个正则表达式，你可以用下面的内容替换图十三的链接： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/s82f65t25m75.jpg |

|  |
| --- |
| 注意#字符的后面加上了$1。Perl正则表达式语法用$1、$2等表示已经匹配且提取出来的组。图十三的表达式把所有作为一个组匹配和提取出来的内容附加到链接的后面。 |

|  |
| --- |
| 现在，返回Java。就象前面我们所做的那样，你必须创建测试字符串，创建把正则表达式编译到Pattern对象所必需的对象，以及创建一个PatternMatcher对象：http://edu.yesky.com/imagelist/05/10/qg1rzozee4bf.jpg |

|  |
| --- |
| 接下来，用com.oroinc.text.regex包Util类的substitute()静态方法进行替换，输出结果字符串： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/o3m76k1jdp62.jpg |

|  |
| --- |
| Util.substitute()方法的语法如下： |

|  |
| --- |
| http://edu.yesky.com/imagelist/05/10/v9z05f24f3tc.jpg |

|  |
| --- |
| 这个调用的前两个参数是以前创建的PatternMatcher和Pattern对象。第三个参数是一个Substiution对象，它决定了替换操作如何进行。本例使用的是Perl5Substitution对象，它能够进行Perl5风格的替换。第四个参数是想要进行替换操作的字符串，最后一个参数允许指定是否替换模式的所有匹配子串（Util.SUBSTITUTE\_ALL），或只替换指定的次数。 |

|  |
| --- |
| **【结束语】**在这篇文章中，我为你介绍了正则表达式的强大功能。只要正确运用，正则表达式能够在字符串提取和文本修改中起到很大的作用。另外，我还介绍了如何在Java程序中通过Jakarta-ORO库利用正则表达式。至于最终采用老式的字符串处理方式（使用StringTokenizer，charAt，和substring），还是采用正则表达式，这就有待你自己决定了。 |

Jakarta-ORO篇

陈广佳 (cgjmail@163.net)  
电子信息工程系工科学士  
2001 年 12 月

由于工作的需要，本人经常要面对大量的文字电子资料的整理工作，因此曾对在JAVA中正则表达式的应用有所关注，并对其有一定的了解，希望通过本文与同行进行有关方面的心得交流。

正则表达式：  
正则表达式是一种可以用于模式匹配和替换的强有力的工具，一个正则表达式就是由普通的字符（例如字符 a 到 z）以及特殊字符（称为元字符）组成的文字模式，它描述在查找文字主体时待匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。

正则表达式在字符数据处理中起着非常重要的作用，我们可以用正则表达式完成大部分的数据分析处理工作，如:判断一个串是否是数字、是否是有效的Email地址，从海量的文字资料中提取有价值的数据等等，如果不使用正则表达式，那么实现的程序可能会很长，并且容易出错。对这点本人深有体会，面对大量工具书电子档资料的整理工作，如果不懂得应用正则表达式来处理，那么将是很痛苦的一件事情，反之则将可以轻松地完成，获得事半功倍的效果。

由于本文目的是要介绍如何在JAVA里运用正则表达式，因此对刚接触正则表达式的读者请参考有关资料，在此因篇幅有限不作介绍。

JAVA对正则表达式的支持：  
在JDK1.3或之前的JDK版本中并没有包含正则表达式库可供JAVA程序员使用，之前我们一般都在使用第三方提供的正则表达式库，这些第三方库中有源代码开放的，也有需付费购买的，而现时在JDK1.4的测试版中也已经包含有正则表达式库---java.util.regex。

故此现在我们有很多面向JAVA的正则表达式库可供选择，以下我将介绍两个较具代表性的**Jakarta-ORO**和**java.util.regex**，首先当然是本人一直在用的**Jakarta-ORO：**

Jakarta-ORO正则表达式库

1．简介：  
Jakarta-ORO是最全面以及优化得最好的正则表达式API之一，Jakarta-ORO库以前叫做OROMatcher，是由Daniel F. Savarese编写，后来他将其赠与Jakarta Project，读者可在Apache.org的网站下载该API包。

许多源代码开放的正则表达式库都是支持Perl5兼容的正则表达式语法，Jakarta-ORO正则表达式库也不例外，他与Perl 5正则表达式完全兼容。

2．对象与其方法：  
★PatternCompiler对象：  
我们在使用Jakarta-ORO API包时，最先要做的是，创建一个Perl5Compiler类的实例，并把它赋值给PatternCompiler接口对象。Perl5Compiler是PatternCompiler接口的一个实现，允许你把正则表达式编译成用来匹配的Pattern对象。

PatternCompiler compiler=new Perl5Compiler();

★Pattern对象：  
要把所对应的正则表达式编译成Pattern对象，需要调用compiler对象的compile()方法，并在调用参数中指定正则表达式。举个例子，你可以按照下面这种方式编译正则表达式"s[ahkl]y"：

|  |
| --- |
| Pattern pattern=null;  try {  pattern=compiler.compile("s[ahkl]y ");  } catch (MalformedPatternException e) {  e.printStackTrace();  } |

在默认的情况下，编译器会创建一个对大小写敏感的模式（pattern）。因此，上面代码编译得到的模式只匹配"say"、"shy"、 "sky"和"sly"，但不匹配"Say"和"skY"。要创建一个大小写不敏感的模式，你应该在调用编译器的时候指定一个额外的参数:  
pattern=compiler.compile("s[ahkl]y",Perl5Compiler.CASE\_INSENSITIVE\_MASK);

Pattern对象创建好之后，就可以通过PatternMatcher类用该Pattern对象进行模式匹配。

★PatternMatcher对象:

PatternMatcher对象依据Pattern对象和字符串展开匹配检查。你要实例化一个Perl5Matcher类并把结果赋值给PatternMatcher接口。Perl5Matcher类是PatternMatcher接口的一个实现，它根据Perl 5正则表达式语法进行模式匹配：  
PatternMatcher matcher=new Perl5Matcher();

PatternMatcher对象提供了多个方法进行匹配操作，这些方法的第一个参数都是需要根据正则表达式进行匹配的字符串：

1. boolean matches(String input, Pattern pattern)：当要求输入的字符串input和正则表达式pattern精确匹配时使用该方法。也就是说当正则表达式完整地描述输入字符串时返回真值。
2. boolean matchesPrefix(String input, Pattern pattern)：要求正则表达式匹配输入字符串起始部分时使用该方法。也就是说当输入字符串的起始部分与正则表达式匹配时返回真值。
3. boolean contains(String input, Pattern pattern)：当正则表达式要匹配输入字符串的一部分时使用该方法。当正则表达式为输入字符串的子串时返回真值。

但以上三种方法只会查找输入字符串中匹配正则表达式的第一个对象，如果当字符串可能有多个子串匹配给定的正则表达式时，那么你就可以在调用上面三个方法时用PatternMatcherInput对象作为参数替代String对象，这样就可以从字符串中最后一次匹配的位置开始继续进行匹配，这样就方便的多了。

用PatternMatcherInput对象作为参数替代String时，上述三个方法的语法如下：

1. boolean matches(PatternMatcherInput input, Pattern pattern)
2. boolean matchesPrefix(PatternMatcherInput input, Pattern pattern)
3. boolean contains(PatternMatcherInput input, Pattern pattern)

★Util.substitute()方法:  
查找后需要要进行替换，我们就要用到Util.substitute()方法，其语法如下：

|  |
| --- |
| public static String substitute(PatternMatcher matcher,  Pattern pattern,Substitution sub,String input,  int numSubs) |

前两个参数分别为PatternMatcher和Pattern对象。而第三个参数是个Substiution对象，由它来决定替换操作如何进行。第四个参数是要进行替换操作的目标字符串，最后一个参数用来指定是否替换模式的所有匹配子串（Util.SUBSTITUTE\_ALL），或只进行指定次数的替换。

在这里我相信有必要详细解说一下第三个参数Substiution对象，因为它将决定替换将怎样进行。

**Substiution**:  
Substiution是一个接口类，它为你提供了在使用Util.substitute()方法时控制替换方式的手段，它有两个标准的实现类：StringSubstitution与Perl5Substitution。当然，同时你也可以生成自己的实现类来定制你所需要的特殊替换动作。

**StringSubstitution：**  
StringSubstitution 实现的是简单的纯文字替换手段，它有两个构造方法：

StringSubstitution()->缺省的构造方法，初始化一个包含零长度字符串的替换对象。

StringSubstitution(java.lang.String substitution)->初始化一个给定字符串的替换对象。

**Perl5Substitution：**  
Perl5Substitution 是StringSubstitution的子类，它在实现纯文字替换手段的同时也允许进行针对MATH类里各匹配组的PERL5变量的替换，所以他的替换手段比其直接父类StringSubstitution更为多元化。

它有三个构造器：

**Perl5Substitution()**

**Perl5Substitution**(java.lang.String substitution)

**Perl5Substitution**(java.lang.String substitution, int numInterpolations)

前两种构造方法与StringSubstitution一样，而第三种构造方法下面将会介绍到。

在**Perl5Substitution**的替换字符串中可以包含用来替代在正则表达式里由小扩号围起来的匹配组的变量，这些变量是由$1, $2,$3等形式来标识。我们可以用一个例子来解释怎样使用替换变量来进行替换：

假设我们有正则表达式模式为b/d+:（也就是b[0-9]+:），而我们想把所有匹配的字符串中的"b"都改为"a",而"："则改为"-"，而其余部分则不作修改，如我们输入字符串为"EXAMPLE b123:"，经过替换后就应该变成"EXAMPLE a123-"。要做到这点，我们就首先要把不做替换的部分用分组符号小括号包起来，这样正则表达式就变为"b(/d+):"，而构造Perl5Substitution对象时其替换字符串就应该是"a$1-"，也就是构造式为Perl5Substitution（"a$1-"），表示在使用Util.substitute()方法时只要在目标字符串里找到和正则表达式" b(/d+): "相匹配的子串都用替换字符串来替换，而变量$1表示如果和正则表达式里第一个组相匹配的内容则照般原文插到$1所在的为置，如在"EXAMPLE b123："中和正则表达式相匹配的部分是"b123："，而其中和第一分组"(/d+)"相匹配的部分则是"123"，所以最后替换结果为"EXAMPLE a123-"。

有一点需要清楚的是，如果你把构造器Perl5Substitution(java.lang.String substitution,int numInterpolations)

中的numInterpolations参数设为INTERPOLATE\_ALL，那么当每次找到一个匹配字串时，替换变量（$1，$2等）所指向的内容都根据目前匹配字串来更新，但是如果numInterpolations参数设为一个正整数N时，那么在替换时就只会在前N次匹配发生时替换变量会跟随匹配对象来调整所代表的内容，但N次之后就以一致以第N次替换变量所代表内容来做为以后替换结果。

举个例子会更好理解：

假如沿用以上例子中的正则表达式模式以及替换内容来进行替换工作，设目标字符串为"Tank b123: 85 Tank b256: 32 Tank b78: 22"，并且设numInterpolations参数为INTERPOLATE\_ALL，而Util.substitute()方法中的numSub变量设为SUBSTITUTE\_ALL（请参考上文Util.substitute()方法内容），那么你获得的替换结果将会是：  
Tank a123- 85 Tank a256- 32 Tank a78- 22

但是如果你把numInterpolations设为2，并且numSubs依然设为SUBSTITUTE\_ALL，那么这时你获得的结果则会是：  
Tank a123- 85 Tank a256- 32 Tank a256- 22

你要注意到最后一个替换所用变量$1所代表的内容与第二个$1一样为"256"，而不是预期的"78"，因为在替换进行中，替换变量$1只根据匹配内容进行了两次更新，最后一次就使第二次匹配时所更新的结果，那么我们可以由此知道，如果numInterpolations设为1，那么结果将是：  
Tank a123- 85 Tank a123- 32 Tank a123- 22

3．应用示例：  
刚好前段时间公司准备出一个《伊索预言》的英语学习互动教材，其中有电子档资料的整理工作，我们就以此为例来看一下Jakarta-ORO与JDBC2.0 API结合起来对**数据库**内的资料进行简单提取与整理的实现。假设由录入部的同事送过来的存放在MS SQLSERVER 7数据库里的电子档的表结构如下（注：或许在不同的DBMS中有相应的正则表达式的应用，但这不在本文讨论范围内）：

表名：AESOP, 表中每条记录包含有三列:  
ID（int）：单词索引号  
WORD（varchar）：单词  
CONTENT(varchar)：存放单词的相关解释与例句等内容

其中CONTENT列中内容的格式如下：  
[音标] [词性] （解释）{(例句一/例句解释/例句中该词的词性: 单词在句中的意思) (例句二/例句解释/例句中该词的词性: 单词在句中的意思)}

如对应单词Kevin,CONTENT中的内容如下：  
['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词: 凯文)( Kevin is living in ZhuHai now./凯文现住在珠海/名词: 凯文)}

我们的例子主要针对CONTENT列中内容进行字符串处理。

★查找单个匹配：  
首先，让我们尝试把CONTNET列中的[音标]字段的内容列示出来，由于所有单词的记录中都有这一项并且都在字串开始位置，所以这个查找工作比较简单：

1. **确定相应的正则表达式：/[[^]]+/]**

这个是很简单的正则表达式，其意思是要求相匹配的字符串必须为以一对中括号包含的所有内容，如['kevin] 、[名词]等，但内容中不包括"]"符号，也就是要避免出现"[][]"会作为一个匹配对象的情况出现（有关正则表达式的基础知识请参照有关资料，这里不再详述）。

注意，在Java中，你必须对每一个向前的斜杠（"/"）进行转义处理。所以我们要在上面的正则表达式里每个"/"前面加上一个"/"以免出现编译错误，也就是在JAVA中初始化正则表达式的字符串的语句应该为：

String restring=" //[[^]]+//]";

并且在表达式里每个符号中间不能有空格，否则就会同样出现编译错误。

1. **实例化PatternCompiler对象，创建Pattern对象**

PatternCompiler compiler=new Perl5Compiler();

Pattern pattern=compiler.compile(restring);

1. **创建PatternMatcher对象，调用PatternMatcher接口的contain()方法检查匹配情况：**

|  |
| --- |
| PatternMatcher matcher=new Perl5Matcher();  if (matcher.contains(content,pattern)) {  //处理代码片段  } |

这里matcher.contains(content,pattern)中的参数 content是从数据库里取来的字符串变量。该方法只会查到第一个匹配的对象字符串，但是由于音标项均在CONETNET内容字符串中的起始位置，所以用这个方法就已经可以保证把每条记录里的音标项找出来了,但更为直接与合理的办法是使用boolean matchesPrefix(PatternMatcherInput input, Pattern pattern)方法，该方法验证目标字符串是否以正则表达式所匹配的字串为起始。

具体实现的完整的程序代码如下：

|  |
| --- |
| package RegularExpressions;  //import……  import org.apache.oro.text.regex.\*;  //使用Jakarta-ORO正则表达式库前需要把它加到CLASSPATH里面，如果用IDE是//JBUILDER，那么也可以在JBUILDER里直接自建新库。  public class yisuo{  public static void main(String[] args){  try{  //使用JDBC DRIVER进行DBMS连接，这里我使用的是一个第三方JDBC  //DRIVER，Microsoft本身也有一个面向SQLSERVER7/2000的免费JDBC //DRIVER，但其性能真的是奇差，不用也罢。  Class.forName("com.jnetdirect.jsql.JSQLDriver");  Connection con=DriverManager.getConnection  ("jdbc:JSQLConnect://kevin:1433","kevin chen","re");  Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_UPDATABLE);  //为使用Jakarta-ORO库而创建相应的对象  String rsstring=" //[[^]]+//]";  PatternCompiler orocom=new Perl5Compiler();  Pattern pattern=orocom.compile(rsstring);  PatternMatcher matcher=new Perl5Matcher();  ResultSet uprs = stmt.executeQuery("SELECT \* FROM aesop");  while (uprs.next()) {  Stirng word=uprs.getString("word");  Stirng content=uprs.getString("content");  if(matcher.contains(content,pattern)){  //或if(matcher.matchesPrefix(content,pattern)){  MatchResult result=matcher.getMatch();  Stirng pure=result.toString();  System.out.println(word+"的音标为："+pure);  }  }  }  catch(Exception e) {  System.out.println(e);  }  }  } |

输出结果为：kevin的音标为['kevin]

在这个处理中我是用toString()方法来取得结果，但是如果正则表达式里是用了分组符号（圆括号），那么就可以用group(int gid)的方法来取得相应各组匹配的结果，如正则表达式改为" (/[[^]]+/])"，那么就可以用以下方法来取得结果：pure=result.group(0);

用程序验证，输出结果同样为：kevin的音标为['kevin]

而如果正则表达式为（/[[^]]+/]）（/[[^]]+/]），则会查找到两个连续的方括号所包含的内容，也就找到[音标] [词性]两项，但是两项的结果分别在两个组里面，分别由下面语句获得结果：

result.group(0)->返回[音标] [词性]两项内容，也就是与整个正则表达式相匹配的结果字符串，在这里也就为['kevin] [名词]

result.group(1) ->返回[音标]项内容，结果应是['kevin]

result.group(2) ->返回[词性]项内容，结果应是[名词]

继续用程序验证，发现输出并不正确，主要是当内容有中文时就不能成功匹配，考虑到可能是Jakarta-ORO正则表达式库版本不支持中文的问题，回看一下原来我一直用的还是2.0.1的老版本，马上到Jakarta.org上下载最新的2.0.4版本装上再用程序验证，得出的结果就和预期一样正确。

★查找多个匹配：  
经过第一步的尝试使用Jakarta-ORO后，我们已经知道了如何正确使用该API包来查找目标字符串里一个匹配的子串，下面我们接着来看一看当目标字符串里包含不止一个匹配的子串时我们如何把它们一个接一个找出来进行相应的处理。

首先我们先试个简单的应用，假设我们想把CONTNET字段内容里所有用方括号包起来的字串都找出来，很清楚地，CONTNET字段的内容里面就只有两项匹配的内容：[音标]和 [词性]，刚才我们其实已经把它们分别找出来了，但是我们所用的方法是分组方法，把"[音标] [词性]"作为一整个正则表达式匹配的内容先找到，再根据分组把[音标]和 [词性]分别挑出来。但是现在我们需要做的是把[音标]和[词性]分别做为与同一个正则表达式匹配的内容，先找到一个接着再找下一个，也就是刚才我们的表达式为（/[[^]]+/]）（/[[^]]+/]）,而现在应为" /[[^]]+/] "。

我们已经知道在匹配操作的三个方法里只要用PatternMatcherInput对象作为参数替代String对象就可以从字符串中最后一次匹配的位置开始继续进行匹配，实现的程序片段如下：

|  |
| --- |
| PatternMatcherInput input=new PatternMatcherInput(content);  while (matcher.contains(input,pattern)) {  result=matcher.getMatch();  System.out.println(result.group(0))  } |

输出结果为:['kevin]  
[名词]

接着我们来做复杂一点的处理，就是我们要先把下面内容：  
['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词: 凯文)( Kevin is living in ZhuHai now. /凯文现住在珠海/名词: 凯文)}中的整个例句部分（也就是由大括号所包含的部分）找出来，再分别把例句一和例句二找出，而各例句中的各项内容（英文句、中文句、词性、解释）也要分项列出。

第一步当然是要定出相应的正则表达式，需要有两个，一是和整个例句部分（也就是由大括号包起来的部分）匹配的正则表达式："/{.+/}",

另一个则要和每个例句部分匹配（也就是小括号中的内容），：/(([^)]+/)

而且由于要把例句的各项分离出来，所以要再把里面的各部分用分组的方法匹配出来：" ([^(]+)/(.+)/(.+):([^)]+) "。

为了简便起见，我们不再和从数据库里读出，而是构造一个包含同样内容的字符串变量，程序片段如下：

|  |
| --- |
| try{  String content="['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词:凯文) (Kevin is living in ZhuHai now./凯文现住在珠海/名词: 凯文)}";  String ps1="//{.+//}";  String ps2="//([^)]+//)";  String ps3="([^(]+)/(.+)/(.+):([^)]+)";  String sentence;  PatternCompiler orocom=new Perl5Compiler();  Pattern pattern1=orocom.compile(ps1);  Pattern pattern2=orocom.compile(ps2);  Pattern pattern3=orocom.compile(ps3);  PatternMatcher matcher=new Perl5Matcher();  //先找出整个例句部分  if (matcher.contains(content,pattern1)) {  MatchResult result=matcher.getMatch();  String example=result.toString();  PatternMatcherInput input=new PatternMatcherInput(example);  //分别找出例句一和例句二  while (matcher.contains(input,pattern2)){  result=matcher.getMatch();  sentence=result.toString();  //把每个例句里的各项用分组的办法分隔出来  if (matcher.contains(sentence,pattern3)){  result=matcher.getMatch();  System.out.println("英文句: "+result.group(1));  System.out.println("句子中文翻译: "+result.group(2));  System.out.println("词性: "+result.group(3));  System.out.println("意思: "+result.group(4));  }  }  }  }  catch(Exception e) {  System.out.println(e);  } |

输出结果为：  
英文句: Kevin loves comic.  
句子中文翻译: 凯文爱漫画  
词性: 名词  
意思: 凯文  
英文句: Kevin is living in ZhuHai now.  
句子中文翻译: 凯文现住在珠海  
词性: 名词  
意思: 凯文

★查找替换：  
以上的两个应用都是单纯在查找字符串匹配方面的，我们再来看一下查找后如何对目标字符串进行替换。

例如我现在想把第二个例句进行改动，换为：Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。

也就是把  
['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词: 凯文)( Kevin is living in ZhuHai now. /凯文现住在珠海/名词: 凯文)}

改为：  
['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词: 凯文)( Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。)}

之前，我们已经了解Util.substitute()方法与Substiution接口，以及Substiution的两个实现类StringSubstitution和Perl5Substitution，我们就来看看怎么用Util.substitute()方法配合Perl5Substitution来完成我们上面提出的替换要求，确定正则表达式：

我们要先找到其中的整个例句部分，也就是由大括号包起来的字串，并且把两个例句分别分组，所以正则表达式为："/{(/([^)]+/))(/([^)]+/))/}"，如果用替换变量来代替分组，那么上面的表达式可以看为"/{$1$2/}",这样就可以更容易看出替换变量与分组间的关系。

根据上面的正则表达式Perl5Substitution类可以这样构造：  
Perl5Substitution("{$1( Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。)}")

再根据这个Perl5Substitution对象来使用Util.substitute()方法便可以完成替换了，实现的代码片段如下：

|  |
| --- |
| try{  String content="['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词: 凯文)(Kevin lives in ZhuHai now./凯文现住在珠海/名词: 凯文)}";  String ps1="//{(//([^)]+//))(//([^)]+//))//}";  String sentence;  String pure;  PatternCompiler orocom=new Perl5Compiler();  Pattern pattern1=orocom.compile(ps1);  PatternMatcher matcher=new Perl5Matcher();  String result=Util.substitute(matcher,  pattern1,new Perl5Substitution(  "{$1( Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。)}",1),  content,Util.SUBSTITUTE\_ALL);  System.out.println(result);  }  catch(Exception e) {  System.out.println(e);  } |

输出结果是正确的，为：  
['kevin] [名词]（人名凯文）{(Kevin loves comic./凯文爱漫画/名词: 凯文)( Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。)}

至于有关使用numInterpolations参数的构造器用法，读者只要根据上面的介绍自己动手试一下就会清楚了，在此就不再例述。

总结：  
本文首先介绍了Jakarta-ORO正则表达式库的对象与方法，并且接着举例让读者对实际应用有进一步的了解，虽然例子都比较简单，但希望读者们在看了该文后对Jakarta-ORO正则表达式库有一定的认知，在实际工作中有所帮助与启发。

其实在Jakarta org里除了Jakarta-ORO外还有一个百分百的纯JAVA正则表达式库，就是由Jonathan Locke赠与Jakarta ORG的Regexp，在该包里面包含了完整的文档以及一个用于调试的Applet例子，对其有兴趣的读者可以到此下载。

参考资料：

* 本文的主要参考文章，该文在介绍Jakarta-ORO的同时也为读者详尽解析了正则表达式的基本语法。
* 一个基于PERL的正则表达式详尽教程（虽然该教程是基于PERL的，但是你并不需要有PERL的经验，虽然那会有所帮助），以及一个不错的正则表达式简例教程。
* 最不可缺少的当然是Jakarta-ORO的帮助文档http://jakarta.apache.org/oro/api/

|  |
| --- |
| 关于作者  陈广佳 Kevin Chen,汕头大学电子信息工程系工科学士，台湾大新出版社珠海区开发部，现正围绕中日韩电子资料使用JAVA开发电子词典等相关项目。可通过E-mail:cgjmail@163.net于他联系。 |

java.util.regex篇

陈广佳 (cgjmail@163.net)  
电子信息工程系工科学士  
2001 年 12 月

现在JDK1.4里终于有了自己的正则表达式API包，JAVA程序员可以免去找第三方提供的正则表达式库的周折了，我们现在就马上来了解一下这个SUN提供的迟来恩物- -对我来说确实如此。

1.简介：  
java.util.regex是一个用正则表达式所订制的模式来对字符串进行匹配工作的类库包。

它包括两个类：**Pattern**和**Matcher**

|  |  |
| --- | --- |
| **Pattern** | 一个Pattern是一个正则表达式经编译后的表现模式。 |
| **Matcher** | 一个Matcher对象是一个状态机器，它依据Pattern对象做为匹配模式对字符串展开匹配检查。 |

首先一个Pattern实例订制了一个所用语法与PERL的类似的正则表达式经编译后的模式，然后一个Matcher实例在这个给定的Pattern实例的模式控制下进行字符串的匹配工作。

以下我们就分别来看看这两个类：

2.Pattern类:  
Pattern的方法如下：

|  |  |
| --- | --- |
| static Pattern | **compile**(String regex) 将给定的正则表达式编译并赋予给Pattern类 |
| static Pattern | **compile**(String regex, int flags) 同上，但增加flag参数的指定，可选的flag参数包括：CASE INSENSITIVE,MULTILINE,DOTALL,UNICODE CASE， CANON EQ |
| int | **flags**() 返回当前Pattern的匹配flag参数. |
| Matcher | **matcher**(CharSequence input) 生成一个给定命名的Matcher对象 |
| static boolean | **matches**(String regex, CharSequence input) 编译给定的正则表达式并且对输入的字串以该正则表达式为模开展匹配,该方法适合于该正则表达式只会使用一次的情况，也就是只进行一次匹配工作，因为这种情况下并不需要生成一个Matcher实例。 |
| String | **pattern**() 返回该Patter对象所编译的正则表达式。 |
| String[] | **split**(CharSequence input) 将目标字符串按照Pattern里所包含的正则表达式为模进行分割。 |
| String[] | **split**(CharSequence input, int limit) 作用同上，增加参数limit目的在于要指定分割的段数，如将limi设为2，那么目标字符串将根据正则表达式分为割为两段。 |

一个正则表达式，也就是一串有特定意义的字符，必须首先要编译成为一个Pattern类的实例，这个Pattern对象将会使用**matcher()**方法来生成一个Matcher实例，接着便可以使用该 Matcher实例以编译的正则表达式为基础对目标字符串进行匹配工作，多个Matcher是可以共用一个Pattern对象的。

现在我们先来看一个简单的例子，再通过分析它来了解怎样生成一个Pattern对象并且编译一个正则表达式，最后根据这个正则表达式将目标字符串进行分割：

|  |
| --- |
| import java.util.regex.\*;  public class Replacement{  public static void main(String[] args) throws Exception {  // 生成一个Pattern,同时编译一个正则表达式  Pattern p = Pattern.compile("[/]+");  //用Pattern的split()方法把字符串按"/"分割  String[] result = p.split(  "Kevin has seen《LEON》seveal times,because it is a good film."  +"/ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部"  +"好电影。/名词:凯文。");  for (int i=0; i<result.length; i++)  System.out.println(result[i]);  }  } |

输出结果为：

Kevin has seen《LEON》seveal times,because it is a good film.  
凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。  
名词:凯文。

很明显，该程序将字符串按"/"进行了分段，我们以下再使用 **split**(CharSequence input, int limit)方法来指定分段的段数，程序改动为：  
tring[] result = p.split("Kevin has seen《LEON》seveal times,because it is a good film./ 凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。"，2);

这里面的参数"2"表明将目标语句分为两段。

输出结果则为：

Kevin has seen《LEON》seveal times,because it is a good film.  
凯文已经看过《这个杀手不太冷》几次了，因为它是一部好电影。/名词:凯文。

由上面的例子，我们可以比较出java.util.regex包在构造Pattern对象以及编译指定的正则表达式的实现手法与我们在上一篇中所介绍的Jakarta-ORO 包在完成同样工作时的差别，Jakarta-ORO 包要先构造一个PatternCompiler类对象接着生成一个Pattern对象，再将正则表达式用该PatternCompiler类的compile()方法来将所需的正则表达式编译赋予Pattern类：

PatternCompiler orocom=new Perl5Compiler();

Pattern pattern=orocom.compile("REGULAR EXPRESSIONS");

PatternMatcher matcher=new Perl5Matcher();

但是在java.util.regex包里，我们仅需生成一个Pattern类，直接使用它的compile()方法就可以达到同样的效果:  
Pattern p = Pattern.compile("[/]+");

因此似乎java.util.regex的构造法比Jakarta-ORO更为简洁并容易理解。

3.Matcher类:  
Matcher方法如下：

|  |  |
| --- | --- |
| Matcher | **appendReplacement**(StringBuffer sb, String replacement) 将当前匹配子串替换为指定字符串，并且将替换后的子串以及其之前到上次匹配子串之后的字符串段添加到一个StringBuffer对象里。 |
| StringBuffer | **appendTail**(StringBuffer sb) 将最后一次匹配工作后剩余的字符串添加到一个StringBuffer对象里。 |
| int | **end**() 返回当前匹配的子串的最后一个字符在原目标字符串中的索引位置 。 |
| int | **end**(int group) 返回与匹配模式里指定的组相匹配的子串最后一个字符的位置。 |
| boolean | **find**() 尝试在目标字符串里查找下一个匹配子串。 |
| boolean | **find**(int start) 重设Matcher对象，并且尝试在目标字符串里从指定的位置开始查找下一个匹配的子串。 |
| String | **group**() 返回当前查找而获得的与组匹配的所有子串内容 |
| String | **group**(int group) 返回当前查找而获得的与指定的组匹配的子串内容 |
| int | **groupCount**() 返回当前查找所获得的匹配组的数量。 |
| boolean | **lookingAt**() 检测目标字符串是否以匹配的子串起始。 |
| boolean | **matches**() 尝试对整个目标字符展开匹配检测，也就是只有整个目标字符串完全匹配时才返回真值。 |
| Pattern | **pattern**() 返回该Matcher对象的现有匹配模式，也就是对应的Pattern 对象。 |
| String | **replaceAll**(String replacement) 将目标字符串里与既有模式相匹配的子串全部替换为指定的字符串。 |
| String | **replaceFirst**(String replacement) 将目标字符串里第一个与既有模式相匹配的子串替换为指定的字符串。 |
| Matcher | **reset**() 重设该Matcher对象。 |
| Matcher | **reset**(CharSequence input) 重设该Matcher对象并且指定一个新的目标字符串。 |
| int | **start**() 返回当前查找所获子串的开始字符在原目标字符串中的位置。 |
| int | **start**(int group) 返回当前查找所获得的和指定组匹配的子串的第一个字符在原目标字符串中的位置。 |

（光看方法的解释是不是很不好理解？不要急，待会结合例子就比较容易明白了）

一个Matcher实例是被用来对目标字符串进行基于既有模式（也就是一个给定的Pattern所编译的正则表达式）进行匹配查找的，所有往Matcher的输入都是通过CharSequence接口提供的，这样做的目的在于可以支持对从多元化的数据源所提供的数据进行匹配工作。

我们分别来看看各方法的使用：

★matches()/lookingAt ()/find()：  
一个Matcher对象是由一个Pattern对象调用其matcher()方法而生成的，一旦该Matcher对象生成,它就可以进行三种不同的匹配查找操作：

1. matches()方法尝试对整个目标字符展开匹配检测，也就是只有整个目标字符串完全匹配时才返回真值。
2. lookingAt ()方法将检测目标字符串是否以匹配的子串起始。
3. find()方法尝试在目标字符串里查找下一个匹配子串。

以上三个方法都将返回一个布尔值来表明成功与否。

★replaceAll ()/appendReplacement()/appendTail()：  
Matcher类同时提供了四个将匹配子串替换成指定字符串的方法：

1. replaceAll()
2. replaceFirst()
3. appendReplacement()
4. appendTail()

replaceAll()与replaceFirst()的用法都比较简单，请看上面方法的解释。我们主要重点了解一下appendReplacement()和appendTail()方法。

appendReplacement(StringBuffer sb, String replacement) 将当前匹配子串替换为指定字符串，并且将替换后的子串以及其之前到上次匹配子串之后的字符串段添加到一个StringBuffer对象里，而appendTail(StringBuffer sb) 方法则将最后一次匹配工作后剩余的字符串添加到一个StringBuffer对象里。

例如，有字符串fatcatfatcatfat,假设既有正则表达式模式为"cat"，第一次匹配后调用appendReplacement(sb,"dog"),那么这时StringBuffer sb的内容为fatdog，也就是fatcat中的cat被替换为dog并且与匹配子串前的内容加到sb里，而第二次匹配后调用appendReplacement(sb,"dog")，那么sb的内容就变为fatdogfatdog，如果最后再调用一次appendTail（sb）,那么sb最终的内容将是fatdogfatdogfat。

还是有点模糊？那么我们来看个简单的程序：

|  |
| --- |
| //该例将把句子里的"Kelvin"改为"Kevin"  import java.util.regex.\*;  public class MatcherTest{  public static void main(String[] args)  throws Exception {  //生成Pattern对象并且编译一个简单的正则表达式"Kelvin"  Pattern p = Pattern.compile("Kevin");  //用Pattern类的matcher()方法生成一个Matcher对象  Matcher m = p.matcher("Kelvin Li and Kelvin Chan are both working in Kelvin Chen's KelvinSoftShop company");  StringBuffer sb = new StringBuffer();  int i=0;  //使用find()方法查找第一个匹配的对象  boolean result = m.find();  //使用循环将句子里所有的kelvin找出并替换再将内容加到sb里  while(result) {  i++;  m.appendReplacement(sb, "Kevin");  System.out.println("第"+i+"次匹配后sb的内容是："+sb);  //继续查找下一个匹配对象  result = m.find();  }  //最后调用appendTail()方法将最后一次匹配后的剩余字符串加到sb里；  m.appendTail(sb);  System.out.println("调用m.appendTail(sb)后sb的最终内容是:"+ sb.toString());  }  } |

最终输出结果为：  
第1次匹配后sb的内容是：Kevin  
第2次匹配后sb的内容是：Kevin Li and Kevin  
第3次匹配后sb的内容是：Kevin Li and Kevin Chan are both working in Kevin  
第4次匹配后sb的内容是：Kevin Li and Kevin Chan are both working in Kevin Chen's Kevin  
调用m.appendTail(sb)后sb的最终内容是：Kevin Li and Kevin Chan are both working in Kevin Chen's KevinSoftShop company.

看了上面这个例程是否对appendReplacement()，appendTail()两个方法的使用更清楚呢，如果还是不太肯定最好自己动手写几行代码测试一下。

★group()/group(int group)/groupCount()：  
该系列方法与我们在上篇介绍的Jakarta-ORO中的MatchResult .group()方法类似(有关Jakarta-ORO请参考上篇的内容)，都是要返回与组匹配的子串内容，下面代码将很好解释其用法：

|  |
| --- |
| import java.util.regex.\*;  public class GroupTest{  public static void main(String[] args)  throws Exception {  Pattern p = Pattern.compile("(ca)(t)");  Matcher m = p.matcher("one cat,two cats in the yard");  StringBuffer sb = new StringBuffer();  boolean result = m.find();  System.out.println("该次查找获得匹配组的数量为："+m.groupCount());  for(int i=1;i<=m.groupCount();i++){  System.out.println("第"+i+"组的子串内容为： "+m.group(i));  }  }  } |

输出为：  
该次查找获得匹配组的数量为：2  
第1组的子串内容为：ca  
第2组的子串内容为：t

Matcher对象的其他方法因比较好理解且由于篇幅有限，请读者自己编程验证。

4．一个检验Email地址的小程序：  
最后我们来看一个检验Email地址的例程，该程序是用来检验一个输入的EMAIL地址里所包含的字符是否合法，虽然这不是一个完整的EMAIL地址检验程序，它不能检验所有可能出现的情况，但在必要时您可以在其基础上增加所需功能。

|  |
| --- |
| import java.util.regex.\*;  public class Email {  public static void main(String[] args) throws Exception {  String input = args[0];  //检测输入的EMAIL地址是否以 非法符号"."或"@"作为起始字符  Pattern p = Pattern.compile("^//.|^//@");  Matcher m = p.matcher(input);  if (m.find()){  System.err.println("EMAIL地址不能以'.'或'@'作为起始字符");  }  //检测是否以"www."为起始  p = Pattern.compile("^www//.");  m = p.matcher(input);  if (m.find()) {  System.out.println("EMAIL地址不能以'www.'起始");  }  //检测是否包含非法字符  p = Pattern.compile("[^A-Za-z0-9//.//@\_//-~#]+");  m = p.matcher(input);  StringBuffer sb = new StringBuffer();  boolean result = m.find();  boolean deletedIllegalChars = false;  while(result) {  //如果找到了非法字符那么就设下标记  deletedIllegalChars = true;  //如果里面包含非法字符如冒号双引号等，那么就把他们消去，加到SB里面  m.appendReplacement(sb, "");  result = m.find();  }  m.appendTail(sb);  input = sb.toString();  if (deletedIllegalChars) {  System.out.println("输入的EMAIL地址里包含有冒号、逗号等非法字符，请修改");  System.out.println("您现在的输入为: "+args[0]);  System.out.println("修改后合法的地址应类似: "+input);  }  }  } |

例如，我们在命令行输入：java Email www.kevin@163**.NET**

那么输出结果将会是：EMAIL地址不能以'www.'起始

如果输入的EMAIL为@kevin@163**.net**

则输出为：EMAIL地址不能以'.'或'@'作为起始字符

当输入为：cgjmail#$%@163**.Net**

那么输出就是：

输入的EMAIL地址里包含有冒号、逗号等非法字符，请修改  
您现在的输入为: cgjmail#$%@163.net  
修改后合法的地址应类似: cgjmail@163.net

5．总结：  
本文介绍了jdk1.4.0-beta3里正则表达式库--java.util.regex中的类以及其方法，如果结合与上一篇中所介绍的Jakarta-ORO API作比较，读者会更容易掌握该API的使用，当然该库的性能将在未来的日子里不断扩展，希望获得最新信息的读者最好到及时到SUN的网站去了解。

6．结束语：  
本来计划再多写一篇介绍一下需付费的正则表达式库中较具代表性的作品，但觉得既然有了免费且优秀的正则表达式库可以使用，何必还要去找需付费的呢，相信很多读者也是这么想的:，所以有兴趣了解更多其他的第三方正则表达式库的朋友可以自己到网上查找或者到我在参考资料里提供的网址去看看。

参考资料

* java.util.regex的帮助文档
* Dana Nourie 和Mike McCloskey所写的Regular Expressions and the Java" Programming Language
* 需要更多的第三方正则表达式资源以及基于它们所开发的应用程序请看http://www.meurrens.org/ip-Links/java/regex/index.html