Java核心API (正则表达式)

正则表达式（regular expressions）是一种描述字符串集的方法，它是以字符串集中各字符串的共有特征为依据的。正则表达式可以用于搜索、编辑或者是操作文本和数据。它超出了 Java 程序设计语言的标准语法，因此有必要去学习特定的语法来构建正则表达式。正则表达式的变化是复杂的，一旦你理解了它们是如何被构造的话，你就能解析或者构建任意的正则表达式了

java.util.regex 包主要由三个类所组成：Pattern、Matcher 和 PatternSyntaxException。

* Pattern 对象表示一个已编译的正则表达式。Pattern 类没有提供公共的构造方法。要构建一个模式，首先必须调用公共的静态 compile 方法，它将返回一个 Pattern 对象。这个方法接受正则表达式作为第一个参数。
* Matcher 是一个靠着输入的字符串来解析这个模式和完成匹配操作的对象。与 Pattern 相似，Matcher 也没有定义公共的构造方法，需要通过调用 Pattern 对象的 matcher 方法来获得一个 Matcher 对象。
* PatternSyntaxException 对象是一个未检查异常，指示了正则表达式中的一个语法错误。

# 测试用具

|  |
| --- |
| **public** **class** RegexTestHarnessV5 {  **public** **static** **void** main(String[] args) {  Scanner scanner = **new** Scanner(System.***in***);  **while** (**true**) {  System.***out***.printf("%nEnter your regex: ");  Pattern pattern = Pattern.*compile*(scanner.nextLine());  System.***out***.printf("Enter input string to search: ");  Matcher matcher = pattern.matcher(scanner.nextLine());  **boolean** found = **false**;  **while** (matcher.find()) {  System.***out***.printf("I found the text \"%s\" starting at index %d and ending at index %d.%n",  matcher.group(), matcher.start(), matcher.end());  found = **true**;  }  **if** (!found) {  System.***out***.printf("No match found.%n");  }  }  }  } |

# 正则表达式

## 元字符

API 所支持的元字符有：([{\^-$|}])?\*+.

有两种方法可以强制将元字符处理成为普通字符：

* . 在元字符前加上反斜线（\）；
* 把它放在\Q（引用开始）和\E（引用结束）之间。在使用这种技术时，\Q和\E能被放于表达式中的任何位置（假设先出现\Q[6]）

## 字符类（[]）

字符类是放在方括号里的字符集，指定了一些字符中的一个能被给定的字符串所匹配。

|  |  |  |
| --- | --- | --- |
| Pattern | 说明 | Pattern解析 |
|  | a, b 或 c（简单类） | [...]表示取...中的一个 |
| [^abc] | 除 a, b 或 c 之外的任意字符 | ^：表示取反 |
| [a-zA-Z] | a 到 z，或 A 到 Z | -：表示范围选取 |
| [a-d[m-p]] | a 到 d，或 m 到 p：[a-dm-p] | 并集 |
| [a-z&&[def]] | d，e 或 f | &&：交集 |
| [a-z&&[^bc]] | 除 b 和 c 之外的 a 到 z 字符：[ad-z] | 差集 |
| [a-z&&[^m-p]] | a 到 z，并且不包括 m 到 p：[a-lq-z] | 差集 |

## 预定义字符类

|  |
| --- |
| . 任何字符（匹配或者不匹配行结束符）  \d 数字字符：[0-9]  \D 非数字字符：[^0-9]  \s 空白字符：[\t\n\x0B\f\r]  \S 非空白字符：[^\s]  \w 单词字符：[a-zA-Z\_0-9]  \W 非单词字符：[^\w] |

## 量词

贪婪（greedy）、勉强（reluctant）和侵占（possessive）量词，来匹配指定表达式X的次数。量词（quantifiers）允许指定匹配出现的次数，方便起见，当前 Pattern API 规范下，描述了贪婪、勉强和侵占三种量词。首先粗略地看一下，量词X?、X??和X?+都允许匹配 X 零次或一次，精确地做同样的事情，但它们之间有着细微的不同之处。

|  |  |  |  |
| --- | --- | --- | --- |
| 量词种类 | | | 意义 |
| 贪婪 | 勉强 | 侵占 |
| X? | X?? | X?+ | 匹配 X 零次或一次 |
| X\* | X\*? | X\*+ | 匹配 X 零次或多次 |
| X+ | X+? | X++ | 匹配 X 一次或多次 |
| X{n} | X{n}? | X{n}+ | 匹配 X n 次 |
| X{n,} | X{n,}? | X{n,}+ | 匹配 X 至少 n 次 |
| X{n,m} | X{n,m}? | X{n,m}+ | 匹配 X 至少 n 次，但不多于 m 次 |

注意：量词可以在字符类和捕获组()后面。0a1a2a3a4b5匹配的位置从在字符之间，包含所有字符

#贪婪、勉强和侵占量词间的不同？

贪婪量词之所以称之为"贪婪的"，这是由于它们强迫匹配器读入（或者称之为吃掉）整个输入的字符串，来优先尝试第一次匹配，如果第一次尝试匹配（对于整个输入的字符串）失败，匹配器会通过回退整个字符串的一个字符再一次进行尝试，不断地进行处理直到找到一个匹配，或者左边没有更多的字符来用于回退了。赖于在表达式中使用的量词，最终它将尝试地靠着 1 或 0 个字符的匹配。

勉强量词采用相反的途径：从输入字符串的开始处开始，因此每次勉强地吞噬一个字符来寻找匹配，最终它们会尝试整个输入的字符串。

侵占量词始终是吞掉整个输入的字符串，尝试着一次（仅有一次）匹配。不像贪婪量词那样，侵占量词绝不会回退，即使这样做是允许全部的匹配成功

|  |
| --- |
| 第一个例子使用贪婪量词.\*，寻找紧跟着字母“f”“o”“o”的“任何东西”零次或者多次。由于量词是贪婪的，表达式的.\*部分第一次“吃掉”整个输入的字符串。在这一点，全部表达式不能成功地进行匹配，这是由于最后三个字母（“f”“o”“o”）已经被消耗掉了。那么匹配器会慢慢地每次回退一个字母，直到返还的“foo”在最右边出现，这时匹配成功并且搜索终止。  Enter your regex: .\*foo  Enter input string to search: xfooxxxxxxfoo  I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.  第二个例子采用勉强量词，因此通过首次消耗“什么也没有”作为开始。由于“foo”并没有出现在字符串的开始，它被强迫吞掉第一个字母（“x”），在 0 和 4 处触发了第一个匹配。测试用具会继续处理，直到输入的字符串耗尽为止。在 4 和 13 找到了另外一个匹配。  Enter your regex: .\*?foo  Enter input string to search: xfooxxxxxxfoo  I found the text " xfoo" starting at index 0 and ending at index 5.  I found the text "xxxxxxfoo" starting at index 5 and ending at index 14.  第三个例子的量词是侵占，所以在寻找匹配时失败了。在这种情况下，整个输入的字符串被.\*+消耗了，什么都没有剩下来满足表达式末尾的“foo”。  Enter your regex: .\*+foo  Enter input string to search: xfooxxxxxxfoo  No match found. |

## 捕获组

捕获组（capturing group）是将多个字符作为单独的单元来对待的一种方式。构建它们可以通过把字符放在一对圆括号中而成为一组。例如，正则表达式(dog)建了单个的组，包括字符“d”“o”和“g”。匹配捕获组输入的字符串部分将会存放于内存中，稍后通过反向引用再次调用。

### 编号方式

在 Pattern 的 API 描述中，捕获组通过从左至右计算开始的圆括号进行编号。例如，在表达式((A)(B(C)))中，有下面的四组：

|  |
| --- |
| ((A)(B(C)))  (A)  (B(C))  (C) |

　　要找出当前的表达式中有多少组，通过调用 Matcher 对象的 groupCount 方法。groupCount 方法返回 int 类型值，表示当前 Matcher 模式中捕获组的数量。例如，groupCount 返回 4 时，表示模式中包含有 4 个捕获组。

　　有一个特别的组——组 0，它表示整个表达式。这个组不包括在 groupCount 的报告范围内。以(?开始的组是纯粹的非捕获组（non-capturing group），它不捕获文本，也不作为组总数而计数。

Matcher 中的一些方法，可以指定 int 类型的特定组号作为参数，因此理解组是如何编号的是尤为重要的。

public int start(int group)：返回之前的匹配操作期间，给定组所捕获的子序列的初始索引。

public int end(int group)：返回之前的匹配操作期间，给定组所捕获子序列的最后字符索引加 1。

public String group (int group)：返回之前的匹配操作期间，通过给定组而捕获的输入子序列。

### 反向引用

匹配输入字符串的捕获组部分会存放在内存中，通过反向引用（backreferences）稍后再调用。在正则表达式中，反向引用使用反斜线（\）后跟一个表示需要再调用组号的数字来表示。例如，表达式(\d\d)定义了匹配一行中的两个数字的捕获组，通过反向引用\1，表达式稍候会被再次调用。

　　匹配两个数字，且后面跟着两个完全相同的数字时，就可以使用(\d\d)\1作为正则表达式：

|  |
| --- |
| Enter your regex: (\d\d)\1  Enter input string to search: 1111  I found the text "1111" starting at index 0 and ending at index 4. |

## 边界匹配

界匹配器（boundary matchers）的信息，可以使模式匹配更为精确。比如说你对某个特定的单词感兴趣，并且它只出现在行首或者是行尾时。又或者你想知道匹配发生在单词边界（word boundary），或者是上一个匹配的尾部。

|  |
| --- |
| ^ 行首  $ 行尾  \b 单词边界  \B 非单词边界  \A 输入的开头  \G 上一个匹配的结尾  \Z 输入的结尾，仅用于最后的结束符（如果有的话）  \z 输入的结尾 |

# Pattern 类的方法

## 使用标志构建模式

Pattern 类定义了备用的 compile 方法，用于接受影响模式匹配方式的标志集。标志参数是一个位掩码，可以是下面公共静态字段中的任意一个：

Pattern.CANON\_EQ：启用规范等价。在指定此标志后，当且仅当在其完整的规范分解匹配时，两个字符被视为匹配。例如，表达式a\u030A[8]在指定此标志后，将匹配字符串“\u00E5”（即字符 å）。默认情况下，匹配不会采用规范等价。指定此标志可能会对性能会有一定的影响。

Pattern.CASE\_INSENSITIVE:启用不区分大小写匹配。默认情况下，仅匹配 US-ASCII 字符集中的字符。Unicode 感知（Unicode-aware）的不区分大小写匹配，可以通过指定 UNICODE\_CASE 标志连同此标志来启用。不区分大小写匹配也能通过内嵌标志表达式(?i)来启用。指定此标志可能会对性能会有一定的影响。

Pattern.COMMENTS：模式中允许存在空白和注释。在这种模式下，空白和以#开始的直到行尾的内嵌注释会被忽略。注释模式也能通过内嵌标志表达式(?x)来启用。

Pattern.DOTALL:启用 dotall 模式。在 dotall 模式下，表达式.匹配包括行结束符在内的任意字符。默认情况下，表达式不会匹配行结束符。dotall 模式也通过内嵌标志表达式(?x)来启用。［s 是“单行（single-line）”模式的助记符，与 Perl 中的相同。］

Pattern.LITERAL：启用模式的字面分析。指定该标志后，指定模式的输入字符串作为字面上的字符序列来对待。输入序列中的元字符和转义字符不具有特殊的意义了。CASE\_INSENSITIVE 和 UNICODE\_CASE 与此标志一起使用时，会对匹配产生一定的影响。其他的标志就变得多余了。启用字面分析没有内嵌标志表达式。

Pattern.MULTILINE：启用多行（multiline）模式。在多行模式下，表达式^和$分别匹配输入序列行结束符前面和行结束符的前面。默认情况下，表达式仅匹配整个输入序列的开始和结尾。多行模式也能通过内嵌标志表达式(?m)来启用。

Pattern.UNICODE\_CASE：启用可折叠感知 Unicode（Unicode-aware case folding）大小写。在指定此标志后，需要通过 CASE\_INSENSITIVE 标志来启用，不区分大小写区配将在 Unicode 标准的意义上来完成。默认情况下，不区分大小写匹配仅匹配 US-ASCII 字符集中的字符。可折叠感知 Unicode 大小写也能通过内嵌标志表达式(?u)来启用。指定此标志可能会对性能会有一定的影响。

Pattern.UNIX\_LINES：启用 Unix 行模式。在这种模式下，.、^和$的行为仅识别“\n”的行结束符。Unix 行模式可以通过内嵌标志表达式(?d)来启用。

接下来，将修改测试用具 RegexTestHarness.java，用于构建不区分大小写匹配的模式。首先，修改代码去调用 complie 的另外一个备用的方法：

|  |
| --- |
| Pattern pattern = Pattern.compile(  console.readLine("%nEnter your regex: "),  Pttern.CASE\_INSENSITIVE  ); |

## 内嵌标志表达式

|  |
| --- |
| 常　　量 等价的内嵌标志表达式  Pattern.CANON\_EQ 没有  Pattern.CASE\_INSENSITIVE (?i)  Pattern.COMMENTS (?x)  Pattern.MULTILINE (?m)  Pattern.DOTALL (?s)  Pattern.LITERAL 没有  Pattern.UNICODE\_CASE (?u)  Pattern.UNIX\_LINES (?d) |

## 使用 matches(String, CharSequence) 方法

Pattern 类定义了一个方便的 matches 方法，用于快速地检查模式是否表示给定的输入字符串。与使用所有的公共静态方法一样，应该通过它的类名来调用 matches 方法，诸如 Pattern.matches("\\d","1");。这个例子中，方法返回 true，这是由于数字“1”匹配了正则表达式\d。

## 使用 split(String) 方法

　　split 方法是一个重要的工具，用于收集依赖于被匹配的模式任一边的文本。split 方法能从“one:two:three:four:five”字符串中解析出“one two three four five”单词：

|  |
| --- |
| @Test  public void test6(){  Pattern p = Pattern.compile(":");  String[] items = p.split("one two three four five");  for(String s : items) {  System.out.println(s);  }  } |

## 其他有用的方法

public static String quote(String s)：返回指定字符串字面模式的字符串。此方法会产生一个字符串，能被用于构建一个与字符串 s 匹配的 Pattern，好像它是一个字面上的模式。输入序列中的元字符和转义序列将没有特殊的意义了。

public String toString()：返回这个模式的字符串表现形式。这是一个编译过的模式中的正则表达式。

## 在 java.lang.String 中等价的 Pattern 方法

java.lang.String 通过模拟 java.util.regex.Pattern 行为的几个方法，也可以支持正则表达式。方便起见，下面主要摘录了出现在 API 关键的方法。

public boolean matches(String regex)：告知字符串是否匹配给定的正则表达式。调用 str.matches(regex)方法所产生的结果与作为表达式的 Pattern.matches(regex, str)的结果是完全一致。

public String[] split(String regex, int limit)：依照匹配给定的正则表达式来拆分字符串。调用 str.split(regex, n)方法所产生的结果与作为表达式的 Pattern.compile(regex).split(str, n) 的结果完全一致。

public String[] split(String regex)：依照匹配给定的正则表达式来拆分字符串。这个方法与调用两个参数的 split 方法是相同的，第一个参数使用给定的表达式，第二个参数限制为 0。在结果数组中不包括尾部的空字符串。

还有一个替换方法，把一个 CharSequence 替换成另外一个：

public String replace(CharSequence target,CharSequence replacement)：将字符串中每一个匹配替换匹配字面目标序列的子字符串，替换成指定的字面替换序列。这个替换从字符串的开始处理直至结束，例如，把字符串“aaa”中的“aa”替换成“b”，结果是“ba”，而不是“ab”

# Matcher 类的方法

## 索引方法

索引方法（index methods）提供一些正好在输入字符串中发现匹配的索引值：

|  |
| --- |
| public int start()：返回之前匹配的开始索引。  public int start(int group)：返回之前匹配操作中通过给定组所捕获序列的开始索引。  public int end(): 返回最后匹配字符后的偏移量。  public int end(int group): 返回之前匹配操作中通过给定组所捕获序列的最后字符之后的偏移量。 |
| @Test  public void test7() {  Pattern p = Pattern.compile("\\bdog\\b");  Matcher m = p.matcher("dog dog dog doggie dogg"); // 获得匹配器对象  int count = 0;  while (m.find()) {  count++;  System.out.println("Match number " + count);  System.out.println("start(): " + m.start());  System.out.println("end(): " + m.end());  }  } |

## 研究方法

研究方法（study methods）回顾输入的字符串，并且返回一个用于指示是否找到模式的布尔值。

|  |
| --- |
| public boolean lookingAt(): 尝试从区域开头处开始，输入序列与该模式匹配。  public boolean find(): 尝试地寻找输入序列中，匹配模式的下一个子序列。  public boolean find(int start): 重置匹配器，然后从指定的索引处开始，尝试地寻找输入序列中，匹配模式的下一个子序列。  public boolean matches(): 尝试将整个区域与模式进行匹配 |
| @Test  public void test8() {  String REGEX = "foo";  String INPUT = "fooooooooooooooooo";  // 初始化  Pattern pattern = Pattern.compile(REGEX);  Matcher matcher = pattern.matcher(INPUT);  System.out.println("Current REGEX is: " + REGEX);  System.out.println("Current INPUT is: " + INPUT);  System.out.println("lookingAt(): " + matcher.lookingAt());  System.out.println("matches(): " + matcher.matches());  }  Current REGEX is: foo  Current INPUT is: fooooooooooooooooo  lookingAt(): true  matches(): false |

## 替换方法

替换方法（replacement methods）用于在输入的字符串中替换文本有用处的方法。

|  |
| --- |
| public Matcher appendReplacement(StringBuffer sb, String replacement)：实现非结尾处的增加和替换操作。  public StringBuffer appendTail(StringBuffer sb)：实现结尾处的增加和替换操作。  public String replaceAll(String replacement)：使用给定的替换字符串来替换输入序列中匹配模式的每一个子序列。  public String replaceFirst(String replacement)：使用给定的替换字符串来替换输入序列中匹配模式的第一个子序列。  public static String quoteReplacement(String s)：返回指定字符串的字面值来替换字符串。这个方法会生成一个字符串，用作 Matcher 的 appendReplacement 方法中的字面值替换 s。所产生的字符串将与作为字面值序列的 s 中的字符序列匹配。斜线（\）和美元符号（$）将不再有特殊意义了。 |
| @Test  public void test9() {  String REGEX = "dog";  String INPUT = "The dog says meow. All dogs say meow.";  String REPLACE = "cat";  Pattern p = Pattern.compile(REGEX);  Matcher m = p.matcher(INPUT); // 获得匹配器对象  INPUT = m.replaceAll(REPLACE);  System.out.println(INPUT);  } |

## 在 java.lang.String 中等价的 Matcher 方法

|  |
| --- |
| public String replaceFirst(String regex, String replacement)：使用给定的替换字符串替换该字符串中匹配了给定正则表达式的第一个子字符串。  调用 str.replaceFirst(regex, repl)方法与使用 Pattern.compile(regex).matcher(str).replaceFirst(repl)产生的结果是完全相同的。  public String replaceAll(String regex, String replacement)：使用给定的替换字符串替换该字符串中匹配了给定正则表达式的每一个子字符串。调用 str.replaceAll(regex, repl)方法与使用 Pattern.compile(regex).matcher(str).replaceAll(repl)产生的结果是完全相同的。 |

# PatternSyntaxException 类的方法

PatternSyntaxException 是未检查异常，指示正则表达式模式中的语法错误。PatternSyntaxException 类提供了下面的一些方法，用于确定在什么地方发生了错误：

|  |
| --- |
| public String getDescription()：获得错误描述。  public int getIndex()：获得错误索引。  public String getPattern()：获得字符串形式的错误正则表达式。  public String getMessage()：获得一个多行的字符串，包括语法错误和错误的索引、错误的正则表达式模式，以及模式内可视化的索引指示。 |