**Java 正则表达式**

正则表达式定义了字符串的模式。

正则表达式可以用来搜索、编辑或处理文本。

正则表达式并不仅限于某一种语言，但是在每种语言中有细微的差别。

Java正则表达式和Perl的是最为相似的。

java.util.regex包主要包括以下三个类：

* **Pattern类：**

pattern对象是一个正则表达式的编译表示。Pattern类没有公共构造方法。要创建一个Pattern对象，你必须首先调用其公共静态编译方法，它返回一个Pattern对象。该方法接受一个正则表达式作为它的第一个参数。

* **Matcher类：**

Matcher对象是对输入字符串进行解释和匹配操作的引擎。与Pattern类一样，Matcher也没有公共构造方法。你需要调用Pattern对象的matcher方法来获得一个Matcher对象。

* **PatternSyntaxException：**

PatternSyntaxException是一个非强制异常类，它表示一个正则表达式模式中的语法错误。

**捕获组**

捕获组是把多个字符当一个单独单元进行处理的方法，它通过对括号内的字符分组来创建。

例如，正则表达式(dog) 创建了单一分组，组里包含"d"，"o"，和"g"。

捕获组是通过从左至右计算其开括号来编号。例如，在表达式（（A）（B（C））），有四个这样的组：

* ((A)(B(C)))
* (A)
* (B(C))
* (C)

可以通过调用matcher对象的groupCount方法来查看表达式有多少个分组。groupCount方法返回一个int值，表示matcher对象当前有多个捕获组。

还有一个特殊的组（组0），它总是代表整个表达式。该组不包括在groupCount的返回值中。

**实例**

下面的例子说明如何从一个给定的字符串中找到数字串：

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

public static void main( String args[] ){

// 按指定模式在字符串查找

String line = "This order was placed for QT3000! OK?";

String pattern = "(.\*)(\\d+)(.\*)";

// 创建 Pattern 对象

Pattern r = Pattern.compile(pattern);

// 现在创建 matcher 对象

Matcher m = r.matcher(line);

if (m.find( )) {

System.out.println("Found value: " + m.group(0) );

System.out.println("Found value: " + m.group(1) );

System.out.println("Found value: " + m.group(2) );

} else {

System.out.println("NO MATCH");

}

}

}

以上实例编译运行结果如下：

Found value: This order was placed for QT3000! OK?

Found value: This order was placed for QT300

Found value: 0

**正则表达式语法**

|  |  |
| --- | --- |
| **字符** | **说明** |
| \ | 将下一字符标记为特殊字符、文本、反向引用或八进制转义符。例如，"n"匹配字符"n"。"\n"匹配换行符。序列"\\"匹配"\"，"\("匹配"("。 |
| ^ | 匹配输入字符串开始的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，^ 还会与"\n"或"\r"之后的位置匹配。 |
| $ | 匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，$ 还会与"\n"或"\r"之前的位置匹配。 |
| \* | 零次或多次匹配前面的字符或子表达式。例如，zo\* 匹配"z"和"zoo"。\* 等效于 {0,}。 |
| + | 一次或多次匹配前面的字符或子表达式。例如，"zo+"与"zo"和"zoo"匹配，但与"z"不匹配。+ 等效于 {1,}。 |
| ? | 零次或一次匹配前面的字符或子表达式。例如，"do(es)?"匹配"do"或"does"中的"do"。? 等效于 {0,1}。 |
| {*n*} | *n*是非负整数。正好匹配 *n* 次。例如，"o{2}"与"Bob"中的"o"不匹配，但与"food"中的两个"o"匹配。 |
| {*n*,} | *n*是非负整数。至少匹配 *n*次。例如，"o{2,}"不匹配"Bob"中的"o"，而匹配"foooood"中的所有 o。"o{1,}"等效于"o+"。"o{0,}"等效于"o\*"。 |
| {*n*,*m*} | *M* 和 *n* 是非负整数，其中 *n* <= *m*。匹配至少 *n* 次，至多 *m* 次。例如，"o{1,3}"匹配"fooooood"中的头三个 o。'o{0,1}' 等效于 'o?'。注意：您不能将空格插入逗号和数字之间。 |
| ? | 当此字符紧随任何其他限定符（\*、+、?、{*n*}、{*n*,}、{*n*,*m*}）之后时，匹配模式是"非贪心的"。"非贪心的"模式匹配搜索到的、尽可能短的字符串，而默认的"贪心的"模式匹配搜索到的、尽可能长的字符串。例如，在字符串"oooo"中，"o+?"只匹配单个"o"，而"o+"匹配所有"o"。 |
| . | 匹配除"\r\n"之外的任何单个字符。若要匹配包括"\r\n"在内的任意字符，请使用诸如"[\s\S]"之类的模式。 |
| (*pattern*) | 匹配 *pattern* 并捕获该匹配的子表达式。可以使用 **$0…$9** 属性从结果"匹配"集合中检索捕获的匹配。若要匹配括号字符 ( )，请使用"\("或者"\)"。 |
| (?:*pattern*) | 匹配 *pattern* 但不捕获该匹配的子表达式，即它是一个非捕获匹配，不存储供以后使用的匹配。这对于用"or"字符 (|) 组合模式部件的情况很有用。例如，'industr(?:y|ies) 是比 'industry|industries' 更经济的表达式。 |
| (?=*pattern*) | 执行正向预测先行搜索的子表达式，该表达式匹配处于匹配 *pattern* 的字符串的起始点的字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?=95|98|NT|2000)' 匹配"Windows 2000"中的"Windows"，但不匹配"Windows 3.1"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。 |
| (?!*pattern*) | 执行反向预测先行搜索的子表达式，该表达式匹配不处于匹配 *pattern* 的字符串的起始点的搜索字符串。它是一个非捕获匹配，即不能捕获供以后使用的匹配。例如，'Windows (?!95|98|NT|2000)' 匹配"Windows 3.1"中的 "Windows"，但不匹配"Windows 2000"中的"Windows"。预测先行不占用字符，即发生匹配后，下一匹配的搜索紧随上一匹配之后，而不是在组成预测先行的字符后。 |
| *x*|*y* | 匹配 *x* 或 *y*。例如，'z|food' 匹配"z"或"food"。'(z|f)ood' 匹配"zood"或"food"。 |
| [*xyz*] | 字符集。匹配包含的任一字符。例如，"[abc]"匹配"plain"中的"a"。 |
| [^*xyz*] | 反向字符集。匹配未包含的任何字符。例如，"[^abc]"匹配"plain"中"p"，"l"，"i"，"n"。 |
| [*a-z*] | 字符范围。匹配指定范围内的任何字符。例如，"[a-z]"匹配"a"到"z"范围内的任何小写字母。 |
| [^*a-z*] | 反向范围字符。匹配不在指定的范围内的任何字符。例如，"[^a-z]"匹配任何不在"a"到"z"范围内的任何字符。 |
| \b | 匹配一个字边界，即字与空格间的位置。例如，"er\b"匹配"never"中的"er"，但不匹配"verb"中的"er"。 |
| \B | 非字边界匹配。"er\B"匹配"verb"中的"er"，但不匹配"never"中的"er"。 |
| \c*x* | 匹配 *x* 指示的控制字符。例如，\cM 匹配 Control-M 或回车符。*x* 的值必须在 A-Z 或 a-z 之间。如果不是这样，则假定 c 就是"c"字符本身。 |
| \d | 数字字符匹配。等效于 [0-9]。 |
| \D | 非数字字符匹配。等效于 [^0-9]。 |
| \f | 换页符匹配。等效于 \x0c 和 \cL。 |
| \n | 换行符匹配。等效于 \x0a 和 \cJ。 |
| \r | 匹配一个回车符。等效于 \x0d 和 \cM。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等。与 [ \f\n\r\t\v] 等效。 |
| \S | 匹配任何非空白字符。与 [^ \f\n\r\t\v] 等效。 |
| \t | 制表符匹配。与 \x09 和 \cI 等效。 |
| \v | 垂直制表符匹配。与 \x0b 和 \cK 等效。 |
| \w | 匹配任何字类字符，包括下划线。与"[A-Za-z0-9\_]"等效。 |
| \W | 与任何非单词字符匹配。与"[^A-Za-z0-9\_]"等效。 |
| \x*n* | 匹配 *n*，此处的 *n* 是一个十六进制转义码。十六进制转义码必须正好是两位数长。例如，"\x41"匹配"A"。"\x041"与"\x04"&"1"等效。允许在正则表达式中使用 ASCII 代码。 |
| \*num* | 匹配 *num*，此处的 *num* 是一个正整数。到捕获匹配的反向引用。例如，"(.)\1"匹配两个连续的相同字符。 |
| \*n* | 标识一个八进制转义码或反向引用。如果 \*n* 前面至少有 *n* 个捕获子表达式，那么 *n* 是反向引用。否则，如果 *n* 是八进制数 (0-7)，那么 *n* 是八进制转义码。 |
| \*nm* | 标识一个八进制转义码或反向引用。如果 \*nm* 前面至少有 *nm* 个捕获子表达式，那么 *nm* 是反向引用。如果 \*nm* 前面至少有 *n* 个捕获，则 *n* 是反向引用，后面跟有字符 *m*。如果两种前面的情况都不存在，则 \*nm* 匹配八进制值 *nm*，其中 *n*和 *m* 是八进制数字 (0-7)。 |
| \nml | 当 *n* 是八进制数 (0-3)，*m* 和 *l* 是八进制数 (0-7) 时，匹配八进制转义码 *nml*。 |
| \u*n* | 匹配 *n*，其中 *n* 是以四位十六进制数表示的 Unicode 字符。例如，\u00A9 匹配版权符号 (©)。 |

**Matcher类的方法**

**索引方法**

索引方法提供了有用的索引值，精确表明输入字符串中在哪能找到匹配：

|  |  |
| --- | --- |
| **序号** | **方法及说明** |
| 1 | **public int start()** 返回以前匹配的初始索引。 |
| 2 | **public int start(int group)**  返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引 |
| 3 | **public int end()** 返回最后匹配字符之后的偏移量。 |
| 4 | **public int end(int group)** 返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量。 |

**研究方法**

研究方法用来检查输入字符串并返回一个布尔值，表示是否找到该模式：

|  |  |
| --- | --- |
| **序号** | **方法及说明** |
| 1 | **public boolean lookingAt()**  尝试将从区域开头开始的输入序列与该模式匹配。 |
| 2 | **public boolean find()** 尝试查找与该模式匹配的输入序列的下一个子序列。 |
| 3 | **public boolean find(int start）** 重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列。 |
| 4 | **public boolean matches()** 尝试将整个区域与模式匹配。 |

**替换方法**

替换方法是替换输入字符串里文本的方法：

|  |  |
| --- | --- |
| **序号** | **方法及说明** |
| 1 | **public Matcher appendReplacement(StringBuffer sb, String replacement)** 实现非终端添加和替换步骤。 |
| 2 | **public StringBuffer appendTail(StringBuffer sb)** 实现终端添加和替换步骤。 |
| 3 | **public String replaceAll(String replacement)**  替换模式与给定替换字符串相匹配的输入序列的每个子序列。 |
| 4 | **public String replaceFirst(String replacement)**  替换模式与给定替换字符串匹配的输入序列的第一个子序列。 |
| 5 | **public static String quoteReplacement(String s)** 返回指定字符串的字面替换字符串。这个方法返回一个字符串，就像传递给Matcher类的appendReplacement 方法一个字面字符串一样工作。 |

**start 和end 方法**

下面是一个对单词"cat"出现在输入字符串中出现次数进行计数的例子：

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

private static final String REGEX = "\\bcat\\b";

private static final String INPUT =

"cat cat cat cattie cat";

public static void main( String args[] ){

Pattern p = Pattern.compile(REGEX);

Matcher m = p.matcher(INPUT); // 获取 matcher 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());

}

}

}

以上实例编译运行结果如下：

Match number 1

start(): 0

end(): 3

Match number 2

start(): 4

end(): 7

Match number 3

start(): 8

end(): 11

Match number 4

start(): 19

end(): 22

可以看到这个例子是使用单词边界，以确保字母 "c" "a" "t" 并非仅是一个较长的词的子串。它也提供了一些关于输入字符串中匹配发生位置的有用信息。

Start方法返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引，end方法最后一个匹配字符的索引加1。

**matches 和lookingAt 方法**

matches 和lookingAt 方法都用来尝试匹配一个输入序列模式。它们的不同是matcher要求整个序列都匹配，而lookingAt 不要求。

这两个方法经常在输入字符串的开始使用。

我们通过下面这个例子，来解释这个功能：

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

private static final String REGEX = "foo";

private static final String INPUT = "fooooooooooooooooo";

private static Pattern pattern;

private static Matcher matcher;

public static void main( String args[] ){

pattern = Pattern.compile(REGEX);

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());

}

}

以上实例编译运行结果如下：

Current REGEX is: foo

Current INPUT is: fooooooooooooooooo

lookingAt(): true

matches(): false

**replaceFirst 和replaceAll 方法**

replaceFirst 和replaceAll 方法用来替换匹配正则表达式的文本。不同的是，replaceFirst 替换首次匹配，replaceAll 替换所有匹配。

下面的例子来解释这个功能：

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

private static String REGEX = "dog";

private static String INPUT = "The dog says meow. " +

"All dogs say meow.";

private static String REPLACE = "cat";

public static void main(String[] args) {

Pattern p = Pattern.compile(REGEX);

// get a matcher object

Matcher m = p.matcher(INPUT);

INPUT = m.replaceAll(REPLACE);

System.out.println(INPUT);

}

}

以上实例编译运行结果如下：

The cat says meow. All cats say meow.

**appendReplacement 和 appendTail 方法**

Matcher 类也提供了appendReplacement 和appendTail 方法用于文本替换：

看下面的例子来解释这个功能：

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class RegexMatches

{

private static String REGEX = "a\*b";

private static String INPUT = "aabfooaabfooabfoob";

private static String REPLACE = "-";

public static void main(String[] args) {

Pattern p = Pattern.compile(REGEX);

// 获取 matcher 对象

Matcher m = p.matcher(INPUT);

StringBuffer sb = new StringBuffer();

while(m.find()){

m.appendReplacement(sb,REPLACE);

}

m.appendTail(sb);

System.out.println(sb.toString());

}

}

以上实例编译运行结果如下：

-foo-foo-foo-

**PatternSyntaxException 类的方法**

PatternSyntaxException 是一个非强制异常类，它指示一个正则表达式模式中的语法错误。

PatternSyntaxException 类提供了下面的方法来帮助我们查看发生了什么错误。

|  |  |
| --- | --- |
| **序号** | **方法及说明** |
| 1 | **public String getDescription()** 获取错误的描述。 |
| 2 | **public int getIndex()**  获取错误的索引。 |
| 3 | **public String getPattern()** 获取错误的正则表达式模式。 |
| 4 | **public String getMessage()** 返回多行字符串，包含语法错误及其索引的描述、错误的正则表达式模式和模式中错误索引的可视化指示。 |