Java正则表达式

1.正则表达式概念

可粗略估计一下，Linu x用户基本都会遇到正则表达式。正则表达式是个极端强大的工具，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，比如最著名的Perl。

1.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。

正则表达式实例: 一个字符串其实就是一个简单的正则表达式，例如 **Hello World** 正则表达式匹配 "Hello World" 字符串。**.**（点号）也是一个正则表达式，它匹配任何一个字符如："a" 或 "1"。

下表列出了一些正则表达式的实例及描述：

正则表达式	描述
this is text	匹配字符串 "this is text"
this\s+is\s+text	注意字符串中的 **\s+**。匹配单词 "this" 后面的 **\s+** 可以匹配多个空格，之后匹配 is 字符串，再之后 **\s+** 匹配多个空格然后再跟上 text 字符串。可以匹配这个实例：this is text
^\d+(.\d+)?	^ 定义了以什么开始\d+ 匹配一个或多个数字? 设置括号内的选项是可选的. 匹配 "."可以匹配的实例："5", "1.5" 和 "2.21"。

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。

2.正则表达式语法

在其他语言中，**** 表示：**我想要在正则表达式中插入一个普通的（字面上的）反斜杠，请不要给它任何特殊的意义。**

在 Java 中，**** 表示：**我要插入一个正则表达式的反斜线，所以其后的字符具有特殊的意义。**

所以，在其他的语言中（如Perl），一个反斜杠 **** 就足以具有转义的作用，而在 Java 正则表达式中则需要有两个反斜杠才能被解析为其他语言中的转义作用。也可以简单的理解在 Java 的正则表达式中，两个 **** 代表其他语言中的一个 ****，这也就是为什么**表示一位数字的正则表达式是 \\d**，而表示一个普通的反斜杠是 ****。

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")"。
	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

(?:*pattern*)	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 (©)。

3.常见正则表达式

[abc] a、b 或 c（简单类）

[^abc] 任何字符，除了 a、b 或 c（否定）

[a-zA-Z] a到 z 或 A 到 Z，两头的字母包括在内（范围）

[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]] a 到 z，除了 b 和 c：[ad-z]（减去）

[a-z&&[^m-p]] a 到 z，而非 m 到 p：[a-lq-z]（减去）

. 任何字符（与行结束符可能匹配也可能不匹配）

\d 数字：[0-9]

\D 非数字： [^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]

\p{Lower} 小写字母字符：[a-z]

\p{Upper} 大写字母字符：[A-Z]

X? X，一次或一次也没有

X* X，零次或多次

X+ X，一次或多次

X{n} X，恰好 n 次

X{n,} X，至少 n 次

X{n,m} X，至少 n 次，但是不超过 m 次

(X) X，作为捕获组

非捕获

(?=X) X，通过零宽度的正lookahead

(?!X) X，通过零宽度的负lookahead

(?<=X) X，通过零宽度的正lookbehind

(?<!X) X，通过零宽度的负lookbehind

(?>X) X，作为独立的非捕获组

2.Java正则表达式

1.Java正则式工具包

Java作为一种开发语言，有许多值得推荐的地方，但是一直以来没有自带对正则表达式的支持。直到JDK1.4，Java双脚跳进了正则表达式的世界。java.util.regex包在支持正则表达式上有它的过人之处。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**新的异常类，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陌生，你也会发现，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。

java正则表达式通过**java.util.regex包**下的**Pattern类与Matcher类**实现(建议在阅读本文时,打开java API文档,当介绍到哪个方法时,查看java API中的方法说明,效果会更佳)。**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是一个正则表达式**经编译后的表现模式。 **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。

Pattern与Matcher一起合作。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**。

2.Pattern类

Pattern类用于**创建一个正则表达式**,也可以说创建一个匹配模式,它的**构造方法是私有的**,不可以直接创建,但可以通过**Pattern.complie(String regex)**简单工厂方法创建一个正则表达式, Java代码示例:

```
Pattern p=Pattern.compile("\\w+");
p.pattern();//返回 \w+
```

pattern() 返回正则表达式的字符串形式,其实就是返回**Pattern.complile(String regex)**的regex参数

1.Pattern.split(CharSequence input)

Pattern有一个split(CharSequence input)方法,**用于分隔字符串**,并返回一个String[],我猜String.split(String regex)就是通过Pattern.split(CharSequence input)来实现的. Java代码示例:

```
Pattern p=Pattern.compile("\\d+");
String[] str=p.split("我的QQ是:456456我的电话是:0532214我的邮箱是:aaa@aaa.com");
```

结果:str[0]="我的QQ是:" str[1]="我的电话是:" str[2]="我的邮箱是:aaa@aaa.com"

2.Pattern.matcher(String regex,CharSequence input)

是一个**静态方法**,用于**快速匹配字符串**,该方法适合用于**只匹配一次,且匹配全部字符串**。Java代码示例:

```
Pattern.matches("\\d+","2223");//返回true
Pattern.matches("\\d+","2223aa");//返回false,需要匹配到所有字符串才能返回true,这里aa不能匹配到
Pattern.matches("\\d+","22bb23");//返回false,需要匹配到所有字符串才能返回true,这里bb不能匹配到
```

3.Pattern.matcher(CharSequence input)

说了这么多,终于轮到**Matcher类**登场了, **Pattern.matcher(CharSequence input)返回一个Matcher对象**. Matcher类的构造方法也是私有的,**不能随意创建**,只能**通过Pattern.matcher(CharSequence input)方法得到该类的实例**. Pattern类只能做一些简单的匹配操作,要想得到**更强更便捷的正则匹配操作**,那就需要将Pattern与Matcher一起合作。Java代码示例:

```
Pattern p=Pattern.compile("\\d+");
Matcher m=p.matcher("22bb23");
m.pattern();//返回p 也就是返回该Matcher对象是由哪个Pattern对象的创建的
```

4.Pattern类方法集锦

1、Pattern complie(String regex)：编译正则表达式，并创建Pattern类。

由于Pattern的构造函数是私有的，不可以直接创建，所以**通过静态的简单工厂方法compile(String regex)方法来创建**，将给定的正则表达式编译并赋予给Pattern类。

2、String pattern()：返回正则表达式的字符串形式。

其实就是**返回Pattern.complile(Stringregex)的regex参数**。示例如下：

```
String regex ="\\?|\\*";
Pattern pattern= Pattern.compile(regex);
StringpatternStr = pattern.pattern();//返回\?\*
```

3、Pattern compile(String regex, int flags)。

方法功能和compile(Stringregex)相同，不过增加了flag参数，flag参数用来**控制正则表达式的匹配行为**，可取值范围如下：

- Pattern.CANON_EQ：启用**规范等价**。当且仅当两个字符的“正规分解(canonicaldecomposition)”都完全相同的情况下，才认定匹配。默认情况下，不考虑“规范相等性(canonical equivalence)”。
- Pattern.CASE_INSENSITIVE：启用**不区分大小写的匹配**。默认情况下，大小写不敏感的匹配只适用于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字符进行大小不敏感的匹配，只要将UNICODE_CASE与这个标志合起来就行了。
- Pattern.COMMENTS：模式中**允许空白和注释**。在这种模式下，匹配时会忽略(正则表达式里的)空格字符(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：启用dotall模式。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.LITERAL：启用模式的**字面值解析**。
- Pattern.MULTILINE：启用**多行模式**。在这种模式下，**‘^’和‘$’也匹配字符串的结束**。默认情况下，这两个表达式仅仅匹配字符串的开始和结束。
- Pattern.UNICODE_CASE：启用Unicode**感知的大小写折叠**。在这个模式下，如果你还启用了CASE_INSENSITIVE标志，那么它会对Unicode字符进行大小写不敏感的匹配。默认情况下，大小写不敏感的匹配只适用于US-ASCII字符集。
- Pattern.UNIX_LINES： 启用Unix行模式。在这个模式下，**只有‘\n’才被认作一行的中止**，并且与‘.’、‘^’、以及‘$’进行匹配。

4、int flags()：返回当前Pattern的匹配flag参数。

5、String[] split(CharSequence input)。

Pattern有一个split(CharSequence input)方法，用于**分隔字符串**，并**返回一个String[]**。此外String[] split(CharSequence input, int limit)功能和String[] split(CharSequence input)相同，增加**参数limit目的在于要指定分割的段数**。

6、static boolean matches(String regex, CharSequenceinput)。

是一个静态方法，用于**快速匹配字符串**，该方法适合用于只匹配一次，且匹配全部字符串。方法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式开展匹配**，该方法只进行一次匹配工作，并**不需要生成一个Matcher实例**。

7、**Matcher matcher(CharSequence input)。**

Pattern.matcher(CharSequenceinput)**返回一个Matcher对象**。Matcher类的构造方法也是私有的，不能随意创建，只能通过Pattern.matcher(CharSequence input)方法得到该类的实例。Pattern类只能做一些简单的匹配操作，要想得到更强更便捷的正则匹配操作，那就需要将Pattern与Matcher一起合作。Matcher类提供了对正则表达式的分组支持，以及对正则表达式的多次匹配支持。

5.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。

3.Matcher类

Matcher类提供了**对正则表达式的分组支持,**以及对正则表达式的**多次匹配支持**。

1.Matcher.matches()/ Matcher.lookingAt()/ Matcher.find()

Matcher类提供三个**匹配操作方法**,三个方法均**返回boolean类型**,当匹配到时返回true,没匹配到则返回false。

- matches()对**整个字符串进行匹配,**只有**整个字符串都匹配了才返回true**。Java代码示例:

```
Pattern p=Pattern.compile("\\d+");
Matcher m=p.matcher("22bb23");
m.matches();//返回false,因为bb不能被\d+匹配,导致整个字符串匹配未成功.
Matcher m2=p.matcher("2223");
m2.matches();//返回true,因为\d+匹配到了整个字符串
```

我们现在回头看一下**Pattern.matcher(String regex,CharSequence input)**,它与下面这段代码等价**Pattern.compile(regex).matcher(input).matches()** 。

- **lookingAt()对前面的字符串进行匹配**,只有匹配到的字符串**在最前面才返回true**，Java代码示例:

```
Pattern p=Pattern.compile("\\d+");
Matcher m=p.matcher("22bb23");
m.lookingAt();//返回true,因为\d+匹配到了前面的22
Matcher m2=p.matcher("aa2223");
m2.lookingAt();//返回false,因为\d+不能匹配前面的aa
```

- **find()对字符串进行匹配**,匹配到的字符串**可以在任何位置**。Java代码示例:

```
Pattern p=Pattern.compile("\\d+");
Matcher m=p.matcher("22bb23");
m.find();//返回true
Matcher m2=p.matcher("aa2223");
m2.find();//返回true
Matcher m3=p.matcher("aa2223bb");
m3.find();//返回true
Matcher m4=p.matcher("aabb");
m4.find();//返回false
```

2.Mathcer.start()/ Matcher.end()/ Matcher.group()

当使用matches(),lookingAt(),find()执行匹配操作后,就可以利用以上三个方法得到更详细的信息.

- **start()**返回匹配到的子字符串**在字符串中的索引位置**.
- **end()**返回匹配到的子字符串的**最后一个字符在字符串中的索引位置**.
- **group()**返回**匹配到的子字符串**

```
Pattern p=Pattern.compile("\\d+");
Matcher m=p.matcher("aaa2223bb");
m.find();//匹配2223
m.start();//返回3
m.end();//返回7,返回的是2223后的索引号
m.group();//返回2223

Mathcer m2=m.matcher("2223bb");
m.lookingAt();    //匹配2223
m.start();    //返回0,由于lookingAt()只能匹配前面的字符串,所以当使用lookingAt()匹配时,start()方法总是返回0
m.end();    //返回4
m.group();    //返回2223

Matcher m3=m.matcher("2223bb");
m.matches();    //匹配整个字符串
m.start();    //返回0,原因相信大家也清楚了
m.end();    //返回6,原因相信大家也清楚了,因为matches()需要匹配所有字符串
m.group();    //返回2223bb
```

下面说说正则表达式的分组在java中是怎么使用的。start(),end(),group()**均有一个重载方法**，它们是**start(int i), end(int i), group(int i)**专用于分组操作,Mathcer类还有一个**groupCount()**用于**返回有多少组**.

```
Pattern p=Pattern.compile("([a-z]+)(\\d+)");
Matcher m=p.matcher("aaa2223bb");
m.find();   //匹配aaa2223
m.groupCount(); //返回2,因为有2组
m.start(1);   //返回0 返回第一组匹配到的子字符串在字符串中的索引号
m.start(2);   //返回3
m.end(1);   //返回3 返回第一组匹配到的子字符串的最后一个字符在字符串中的索引位置.
m.end(2);   //返回7
m.group(1);   //返回aaa,返回第一组匹配到的子字符串
m.group(2);   //返回2223,返回第二组匹配到的子字符串
```

现在我们使用一下稍微高级点的正则匹配操作,例如有一段文本,里面有很多数字,而且这些数字是分开的,我们现在要将文本中所有数字都取出来,利用java的正则操作是那么的简单。Java代码示例:

```
Pattern p=Pattern.compile("\\d+");
Matcher m=p.matcher("我的QQ是:456456 我的电话是:0532214 我的邮箱
是:aaa123@aaa.com");
while(m.find()) {
     System.out.println(m.group());
}
```

输出:

```
456456
0532214
123
```

如将以上while()循环替换成

```
while(m.find()) {
     System.out.println(m.group());
     System.out.print("start:"+m.start());
     System.out.println(" end:"+m.end());
}
```

则输出:

```
456456
start:6 end:12
0532214
start:19 end:26
123
start:36 end:39
```

每次执行匹配操作后**start(),end(),group()三个方法的值都会改变**,改变成匹配到的子字符串的信息,以及它们的重载方法,也会改变成相应的信息。注意:**只有当匹配操作成功,才可以使用start(),end(),group()三个方法**,否则会抛出java.lang.IllegalStateException,也就是当matches(),lookingAt(),find()其中**任意一个方法返回true时,才可以使用**。

3.matcher类方法集锦

- **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。