[1 正则表达式基础知识 2](#_Toc265283882)

[1.1 句点符号 . 2](#_Toc265283883)

[1.2 方括号符号 [ ] 2](#_Toc265283884)

[1.3 “或”符号 | 2](#_Toc265283885)

[1.4 表示匹配次数的符号 \* + ? {n} {n, m} 3](#_Toc265283886)

[1.5 “否”符号 ^ 4](#_Toc265283887)

[1.6 圆括号和空白符号 () 4](#_Toc265283888)

[1.7 其它符号 \d \D \w \W \s \S 4](#_Toc265283889)

[2 Jakarta-ORO库 (第三方库) 5](#_Toc265283890)

[▲ PatternCompiler对象 5](#_Toc265283891)

[▲ Pattern对象 6](#_Toc265283892)

[▲ PatternMatcher对象 6](#_Toc265283893)

[3 应用实例 7](#_Toc265283894)

[3.1 日志文件处理 7](#_Toc265283895)

[3.2 HTML处理实例一 9](#_Toc265283896)

[3.3 HTML处理实例二 11](#_Toc265283897)

[元字符? 与 () 13](#_Toc265283898)

[\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 13](#_Toc265283899)

[RegExp对象 13](#_Toc265283900)

[RegExp对象的属性 14](#_Toc265283901)

[静态属性 14](#_Toc265283902)

[实例属性 15](#_Toc265283903)

[RegExp对象的方法 15](#_Toc265283904)

[String对象中的相关方法 16](#_Toc265283905)

[限定符 与 选择匹配符 18](#_Toc265283906)

[用限定符指定连续出现的次数 19](#_Toc265283907)

[贪婪匹配 与 非贪婪匹配 19](#_Toc265283908)

[用选择匹配符进行匹配选择 19](#_Toc265283909)

[分组组合 与 反向引用符 20](#_Toc265283910)

[分组组合 20](#_Toc265283911)

[反向引用 20](#_Toc265283912)

[非捕获匹配 20](#_Toc265283913)

[正向"预测先行"匹配 20](#_Toc265283914)

[反向"预测先行"匹配 20](#_Toc265283915)

[特殊字符 与 字符匹配符 20](#_Toc265283916)

[定位符 与 原义字符 20](#_Toc265283917)

[实例举例 20](#_Toc265283918)

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。

# 1 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：



## 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：



## 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：



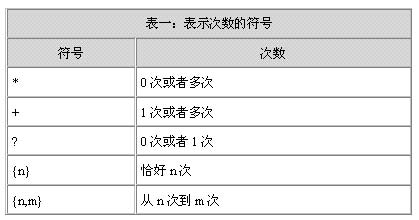
## 1.3 “或”符号 |

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。

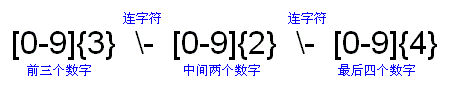


## 1.4 表示匹配次数的符号 \* + ? {n} {n, m}

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：

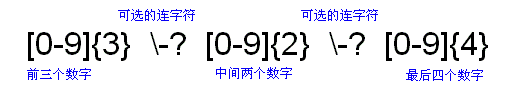


假设我们要在文本文件中搜索美国的社会安全号码。这个号码的格式是999-99-9999。用来匹配它的正则表达式如图一所示。在正则表达式中，连字符（“-”）有着特殊的意义，它表示一个范围，比如从0到9。因此，匹配社会安全号码中的连字符号时，它的前面要加上一个转义字符“\”。



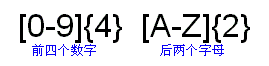
图一：匹配所有123-12-1234形式的社会安全号码

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：



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。



图三：匹配典型的美国汽车牌照号码，如8836KV

## 1.5 “否”符号 ^

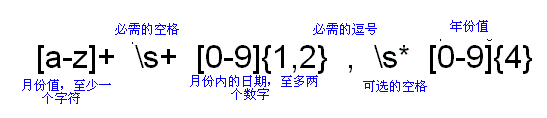
“^”符号称为“否”符号。如果用在方括号内，“^”表示不想要匹配的字符。例如，图四的正则表达式匹配所有单词，但以“X”字母开头的单词除外。



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 ()

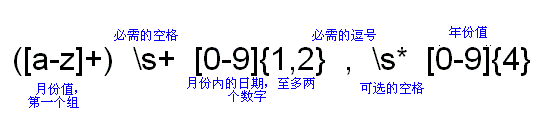
假设要从格式为“June 26, 1951”的生日日期中提取出月份部分，用来匹配该日期的正则表达式可以如图五所示：



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：



图六：匹配所有Month DD,YYYY格式的日期，定义月份值为第一个组

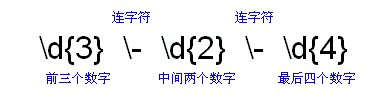
## 1.7 其它符号 \d \D \w \W \s \S

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



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：



图七：匹配所有123-12-1234格式的社会安全号码

# 2 Jakarta-ORO库 (第三方库)

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”：



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：



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)

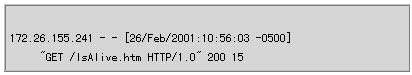
· boolean contains(PatternMatcherInput input, Pattern pattern)

# 3 应用实例

下面我们来看看Jakarta-ORO库的一些应用实例。

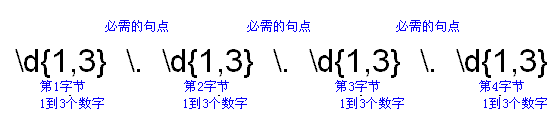
## 3.1 日志文件处理

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：



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。

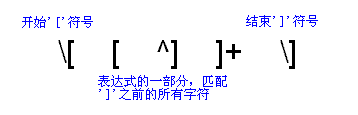
首先我们来看看IP地址。IP地址有4个字节构成，每一个字节的值在0到255之间，各个字节通过一个句点分隔。因此，IP地址中的每一个字节有至少一个、最多三个数字。图八显示了为IP地址编写的正则表达式：



图八：匹配IP地址

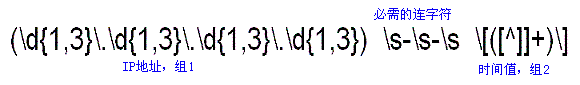
IP地址中的句点字符必须进行转义处理（前面加上“\”），因为IP地址中的句点具有它本来的含义，而不是采用正则表达式语法中的特殊含义。句点在正则表达式中的特殊含义本文前面已经介绍。

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。



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。



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：

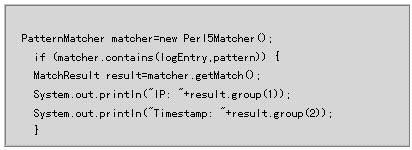


这里使用的正则表达式与图十的正则表达式差不多完全相同，但有一点例外：在Java中，你必须对每一个向前的斜杠（“\”）进行转义处理。图十不是Java的表示形式，所以我们要在每个“\”前面加上一个“\”以免出现编译错误。遗憾的是，转义处理过程很容易出现错误，所以应该小心谨慎。你可以首先输入未经转义处理的正则表达式，然后从左到右依次把每一个“\”替换成“\\”。如果要复检，你可以试着把它输出到屏幕上。

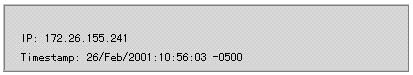
初始化字符串之后，实例化PatternCompiler对象，用PatternCompiler编译正则表达式创建一个Pattern对象：



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



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：



## 3.2 HTML处理实例一

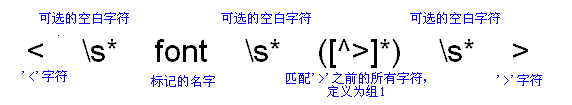
下面一个任务是分析HTML页面内FONT标记的所有属性。HTML页面内典型的FONT标记如下所示：

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：

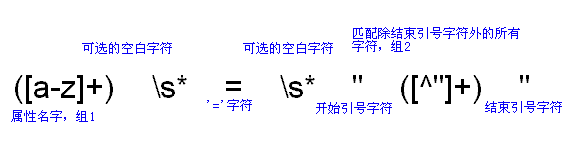


在这种情况下，我建议你使用两个正则表达式。第一个如图十一所示，它从字体标记提取出“"face="Arial, Serif" size="+2" color="red"”。



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。



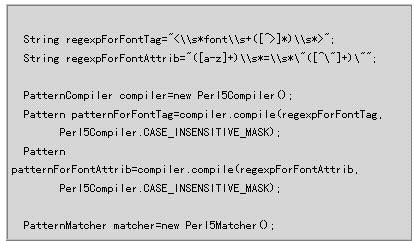
图十二：匹配单个属性，并把它分割成名字-值对

分割结果为：

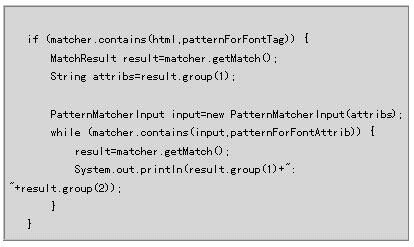


现在我们来看看完成这个任务的Java代码。首先创建两个正则表达式字符串，用Perl5Compiler把它们编译成Pattern对象。编译正则表达式的时候，指定Perl5Compiler.CASE\_INSENSITIVE\_MASK选项，使得匹配操作不区分大小写。

接下来，创建一个执行匹配操作的Perl5Matcher对象。



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：



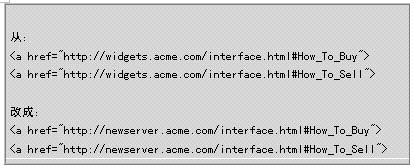
接下来创建一个PatternMatcherInput对象。这个对象允许你从最后一次匹配的位置开始继续进行匹配操作，因此，它很适合于提取FONT标记内属性的名字-值对。创建PatternMatcherInput对象，以参数形式传入待匹配的字符串。然后，用匹配器实例提取出每一个FONT的属性。这通过指定PatternMatcherInput对象（而不是字符串对象）为参数，反复地调用PatternMatcher对象的contains()方法完成。PatternMatcherInput对象之中的每一次迭代将把它内部的指针向前移动，下一次检测将从前一次匹配位置的后面开始。

本例的输出结果如下：



## 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：



执行这个搜索的正则表达式如图十三所示：



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：

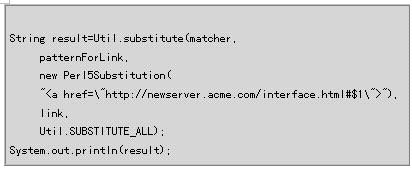
http://edu.yesky.com/imagelist/05/10/s82f65t25m75.jpg

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。

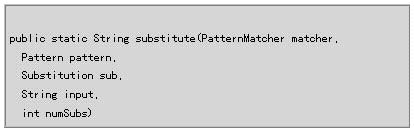
现在，返回Java。就象前面我们所做的那样，你必须创建测试字符串，创建把正则表达式编译到Pattern对象所必需的对象，以及创建一个PatternMatcher对象：



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：



Util.substitute()方法的语法如下：



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。

# 元字符? 与 ()

|  |  |
| --- | --- |
| ? | 表示仅靠其前的元素可选 |
| () | 将多个元素组合成一个组合项, 括起来的表达式部分称为一个子匹配(子表达式)  子表达式匹配的内容按照从左到右的顺序存储在缓冲区, 该过程称为捕获.  字匹配捕获的内容可在表达式中位于该括号之后的地方被反向引用, 第一个圆括号捕获的内容用\1表示, 第二个用\2表示, 以此类推. |

# \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# RegExp对象

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 有两种方法创建对象  第一种: 使用RegExp的显示构造函数: new RegExp("pattern"[, "flags"]);  第二种: 使用RegExp的隐式构造函数, 采用纯文本格式: /pattern[flags];  pattern部分为要使用的正则表达式模式文本,  在第一种方式中, pattern部分以JavaScript字符串形式存在, 需要使用双引号或单引号括起来;  在第二种方式中, pattern部分嵌套在两个"/"之间, 不能使用引号.  flags部分设置正则表达式的标志信息, 是可选项. 如果设置flags部分,  在第一种方式中, 以字符串的形式存在;  在第二种方式中, 以文本的形式紧接在最后一个"/"字符之后.  flag可以是以下标志字符的组合:   |  |  | | --- | --- | | g | g是全局标志, 如果设置了这个标志, 对某个文本执行搜索和替换操作时, 将对文本中所有匹配的部分起作用. 如果不设置这个标志, 则仅搜索和替换最先匹配的内容. | | i | i是忽略大小写标志, 如果不设置这个标志, 进行匹配比较时, 将忽略大小写. | | m | m是多行标志,  如果不设置这个标志, 那么元字符"^"只与整个被搜索字符串的开始位置相匹配, 而元字符"$"只与被搜索字符串的结束位置相匹配.  如果设置了这个标志, "^"还可以与被搜索字符串中的"\n"或"\r"之后的位置(即下一行的行首)相匹配, 而"$"还可以与被搜索字符串中的"\n"或"\r"之后的位置(即下一行的行尾)相匹配. |   例子:  var re1 = new RegExp("jin");  var re2 = /\d{5}/;  var re3 = /y/;  注意: 正则表达式模式文本中的转义字符也是"\", 如果正则表达式中要匹配原义字符"\", 在正则表达式模式文本中要以"\\"来表示, 当使用显示构造函数方式创建RegExp实例对象的时候, 就需要使用"\\\\"来表示原义字符"\"  var re3 = new RegExp("\\\\"); 等同于 var re4 = /\\/ |

# RegExp对象的属性

## 静态属性

|  |  |
| --- | --- |
| index | 是当前表达式模式首次匹配内容的开始位置, 从0开始计数. 其初始值为-1, 每次成功匹配时, index属性都会随之改变. |
| input | 返回当前所作用的字符串, 可以简写为$\_, 初始值为空字符串"". |
| lastIndex | 是当前表达式模式首次匹配内容中最后一个字符的下一个位置. 从0开始计数, 常被作为继续搜索时的起始位置, 初始值为-1, 表示从起始位置开始搜索.每次成功匹配, lastIndex值都会改变. |
| lastParen | 如果表达式模式中有括起来的字匹配, 是当前表达式模式中最后的字匹配所匹配到的字符串, 可以简写为$+, 其初始值为空字符串"", 每次成功匹配时, lastParen值都会改变. |
| leftContext | 是当前表达式模式最后一个匹配字符串左边的所有内容, 可以简写为$` (其中`为"Esc"下边的反单引号). 初始值为空字符串"". 每次成功匹配leftContext值都会改变. |
| rightContext | 是当前表达式模式最后一个匹配字符串右边的所有内容, 可以简写为$', 初始值为空字符串, 每次匹配成功, 其属性值都会改变. |
| $1...$9 | 这些属性值是只读的, 如果表达式模式中有括起来的字匹配, $1...$9属性值分别是第1个到第9个字匹配所捕获到的内容, 如果有超过9个以上的字匹配, $1...$9属性分别代表对应的最后9个字匹配. 在一个表达式模式中, 可以指定人一多个带括号的字匹配, 但RegExp对象只能存储最后的9个字匹配的结果. 在RegExp实例对象的一些方法所返回的结果数组中, 可以捕获得所有圆括号内的字匹配结果. |

## 实例属性

|  |  |
| --- | --- |
| global | 返回创建RegExp对象实例时指定的global标志(g)的状态, 如果创建RegExp对象实例时设置了g标志, 该属性返回true, 否则返回false, 默认返回false. |
| ignoreCase | 返回创建RegExp对象实例时指定的ignoreCase标志(i)的状态, 如果创建RegExp对象实例时设置了i标志, 该属性返回true, 否则返回false, 默认返回false. |
| multiLine | 返回创建RegExp对象实例时指定的multiLine标志(m)的状态, 如果创建RegExp对象实例时设置了m标志, 该属性返回true, 否则返回false. 默认返回false. |
| source | 返回创建RegExp对象实例时指定的表达式文本字符串. |

# RegExp对象的方法

|  |  |
| --- | --- |
| test(str) | 检查一个字符串中是否存在创建RegExp对象实例时所指定的表达式模式, 如果存在就返回true, 否则返回false.  如果找到匹配项, 则会更新RegExp对象中的有关静态属性, 以反映匹配情况. |
| exec(str) | 该方法使用创建RegExp对象实例时所指定的表达式模式对一个字符串进行搜索, 并返回一个包含搜索结果的数组.  如果正则表达式设置了全局标志(g), 可以通过多次调用exec和test方法在字符串中进行连续搜索, 每次都从RegExp对象的lastIndex属性值指定的位置开始搜索字符串.  如果没有设置全局标志(g), 则exec和test方法忽略RegExp对象的lastIndex属性值. 从字符串的起始位置开始搜索.  如果exec方法没有找到匹配, 返回null; 如果找到匹配, 则返回一个数值, 并更新RegExp对象中的有关静态属性以反映匹配情况. 返回数组中的元素0包含了完整的匹配结果.  而元素1~n依次是表达式模式中定义的各个字匹配的结果.  exec方法返回的数组有3个属性, 分别是input, index, lastIndex.  input属性: 整个被搜索的字符串.  index属性: 是指匹配在整个被搜索字符串中的位置.  lastIndex属性: 是指匹配的字符串中的最后一个字符的下一个字符的位置.  例子   |  | | --- | | var myString = "aaa 111 bbb 222 ccc 111 222ddd";  var regex = /111/;  var array = regex.exec(myString);  if(array){  var str = "找到匹配字符串!"  +"\n返回数组的值为: " + array  +"\n数组元素个数为: " + array.length  +"\n被搜索的字符串为: " + array.input  +"\n匹配字串的开始位置为: " + array.index  +"\n匹配字串后面第一个字符的位置为: " + regex.lastIndex;  alert(str);  }else{  alert("未找到匹配的字符串!");  } | |
| compile("pattern"[, ""flags"]) | 该方法可以更换RegExp对象实例所使用的表达式模式, 并将新的表达式模式编译为内部格式, 从而使以后的匹配过程执行更快.  如果要在循环中重复使用某个表达式, 对其进行编译将使执行加速. 但是如果在程序中使用了任何其他表达式模式后, 再使用原来编译过的表达式模式, 则这种编译毫无益处. |

# String对象中的相关方法

|  |  |  |
| --- | --- | --- |
| match(rgExp) | 这个方法的作用于RegExp对象的exec方法类似, 使用正则表达式模式对字符串进行搜索, 并返回一个包含搜索结果的数组.  如果没有为正则表达式设置全局标志(g), match方法产生的结果与没有设置全局标志(g)的exec方法结果完全相同;  如果设置了全局标志(g), match方法返回的数组中包含所有完整的匹配结果, 元素0~n依次是每个完整的匹配结果.  传递给match方法的参数是一个RegExp类型的对象实例, 即用表达式作为match方法的参数去搜索字符串,  而传递给exec方法的参数是一个String类型的对象实例, 即用表达式对象去搜索作为exec方法参数的字符串.   |  | | --- | | var strSrc="xxalb01c001yya2b02c002zzasdfa2b02c002asdfasda2b02c002";  var re=/a(\d)b(\d{2})c(\d{3})/gi;  var arr;  arr = strSrc.match(re);  for(var i=0; i<arr.length; i++){  document.write("[\""+arr[i]+"\"],");  } | |
| search(rgExp) | 该方法返回使用表达式搜索时, 第一个匹配的字符串在整个被搜索字符串中的位置.   |  | | --- | | var strSrc="xxalb01c001yya2b02c002zzasdfa2b02c002asdfasda2b02c002";  var re=/a(\d)b(\d{2})c(\d{3})/gi;  var index = strSrc.search(re);  alert("在字符串的第"+index+"个字符位置找到匹配模式: " + re); | |
| replace(rgExp, replaceText) | 该方法使用表达式模式对字符串执行搜索, 并对搜索到的内容用指定的字符串替换, 返回一个字符串对象, 包含了替换后的内容.  replace方法执行后, 将更新RegExp对象中的有关静态属性以反映匹配情况, 该方法需要两个参数. 如下:   |  |  | | --- | --- | | rgExp | 搜索时要使用的表达式对象, 如果是字符串, 不按正则表达式的方式进行模糊搜索, 而进行精确搜索. | | replaceText | 用于替换搜索到的内容的字符串, 其中可以使用一些特殊的字符组合来表示匹配变量, 其中,  $&是整个表达式模式在被搜索字符串中所匹配的字符串,  $是表达式模式在被搜索字符串中所匹配的字符串左边的所有内容,  $'是表达式模式在被搜索字符串中所匹配的字符串右边的所有内容,  $01...$99分别是第一个到第99个字匹配所捕获到的内容, 当$n和$nn(n为整数)在表达式中没有对应的字匹配时, 将被解释成普通字符. |   例子   |  | | --- | | var strSrc="a12b34c56"; //替换前的字符串  var re=/(\d)(\d)/gi;  var strDest = strSrc.replace(re, "$2$1"); // strDest是替换后的字符串  alert(strSrc + "被转换为: " + strDest); | |
| split([separator[, limit]]) | 该方法返回按照某种分隔标志符将一个字符串拆分成若干个子字符串时所产生的子字符串数组.  参数separator是分隔标识符参数, 可以是多个字符或一个正则表达式, 并不作为返回到数组元素的一部分.  参数limit限制返回元素的个数.  例子:   |  | | --- | | var splitArray = new Array();  var string = "太平洋, 大西洋, 印度洋, 北冰洋";  var regex=/, /;  splitArray = string.split(regex);  for(i=0; i<splitArray.length; i++){  document.write(splitArray[i] + "<br />");  } | |

# 限定符 与 选择匹配符

|  |
| --- |
| 正则表达式的语法主要就是: 多各个元字符功能的描述  元字符从功能上划分为: 限定符, 选择匹配符, 分组组合, 反向引用符, 特殊字符, 字符匹配符, 定位符 |

## 用限定符指定连续出现的次数

|  |  |
| --- | --- |
| {n} | 规定前面的元素或组合项连续出现n次. 例如: "o{2}"不能与"Bob"匹配, 但可以与"food"匹配, 也可以与"fooood"中的任意两个连续的o匹配 |
| {n,} | 规定前面的元素或组合项最少连续出现n次. 例如"o{2,}"不能与"Bob"匹配, 但可以与"fooood"中的所有o匹配 |
| {n,m} | 规定前面的元素或组合项最少连续出现n次, 最多连续出现m次, 其中n<=m, 逗号和数字之间不能有空格. 例如:"o{1,3}"即可匹配"food"中的两个o, 又可匹配"fooood"中的三个连续的o. |
| + | 规定前面的元素或组合项必须出现一次或连续多次, 例如, "zo+"与"zo" 和"zoo"等在字母z后面连续出现一个或多个字母o的字符串匹配, 但与一个单独的z不匹配. +等效于{1,}. |
| \* | 规定前面的元素或组合项可以出现多次或零次, 例如"zo\*", 与"z" 和 "zoo"都能匹配, \*等效于{0,}. |
| ? | 规定前面的元素或组合项零次或一次, ?等效于{0,1}. |
|  |  |

## 贪婪匹配 与 非贪婪匹配

|  |
| --- |
| 正则表达式默认使用最长(贪婪)匹配原则  当字符"?"紧跟任何其他限定符(\*, +, ?, {n,}, {n,m})之后时, 匹配模式变成使用最短(非贪婪)匹配原则. |

## 用选择匹配符进行匹配选择

|  |
| --- |
| 选择匹配符只有一个"|"  其两个选项是"|", 字符两边尽可能最大的表达式.  "chapter|session 1"匹配的是"chapter"或"session 1"  如果要匹配"chapter 1"或"session 1", 应该使用括号创建子表达式(chapter|session) 1 |

# 分组组合 与 反向引用符

## 分组组合

## 反向引用

## 非捕获匹配

## 正向"预测先行"匹配

## 反向"预测先行"匹配

# 特殊字符 与 字符匹配符

特殊字符

字符匹配符

# 定位符 与 原义字符

用于文本验证的定位符

对特殊字符进行转义

# 实例举例

|  |  |
| --- | --- |
| 数字 | ^[0-9]\*$ |
|  |  |
|  |  |
|  |  |