

# 揭开正则表达式的神秘面纱

[出处: <http://www.regexlab.com/zh/regref.htm>]

## 引言

正则表达式 (regular expression) 描述了一种字符串匹配的模式, 可以用来: (1) 检查一个串中是否含有符合某个规则的字串, 并且可以得到这个子串; (2) 根据匹配规则对字符串进行灵活的替换操作。

正则表达式学习起来其实是很简单的, 不多的几个较为抽象的概念也很容易理解。之所以很多人感觉正则表达式比较复杂, 一方面是因为大多数的文档没有做到由浅入深地讲解, 概念上没有注意先后顺序, 给读者的理解带来困难; 另一方面, 各种引擎自带的文档一般都要介绍它特有的功能, 然而这部分特有的功能并不是我们首先要理解的。

文章中的每一个举例, 都可以点击进入测试页面进行测试。闲话少说, 开始。

## 1. 正则表达式规则

### 1.1 普通字符

字母、数字、汉字、下划线、以及后边章节中没有特殊定义的标点符号, 都是"普通字符"。表达式中的普通字符, 在匹配一个字符串的时候, 匹配与之相同的一个字符。

**举例 1:** 表达式 "c", 在匹配字符串 "abcde" 时, 匹配结果是: 成功; 匹配到的内容是: "c"; 匹配到的位置是: 开始于 2, 结束于 3。(注: 下标从 0 开始还是从 1 开始, 因当前编程语言的不同而可能不同)

**举例 2:** 表达式 "bcd", 在匹配字符串 "abcde" 时, 匹配结果是: 成功; 匹配到的内容是: "bcd"; 匹配到的位置是: 开始于 1, 结束于 4。

### 1.2 简单的转义字符

一些不便书写的字符, 采用在前面加 "\" 的方法。这些字符其实我们都已经熟知了。

表达式	可匹配
\r, \n	代表回车和换行符
\t	制表符
\\	代表 "\" 本身

还有其他一些在后边章节中有特殊用处的标点符号，在前面加 `"\"` 后，就代表该符号本身。比如：`^`，`$` 都有特殊意义，如果要想匹配字符串中 `"^"` 和 `"$"` 字符，则表达式就需要写成 `"\"` 和 `"\"`。

表达式	可匹配
<code>\^</code>	匹配 <code>^</code> 符号本身
<code>\\$</code>	匹配 <code>\$</code> 符号本身
<code>\.</code>	匹配小数点 (.) 本身

这些转义字符的匹配方法与 "普通字符" 是类似的。也是匹配与之相同的一个字符。

举例 1：表达式 `"\$d"`，在匹配字符串 `"abc$de"` 时，匹配结果是：成功；匹配到的内容是：`"$d"`；匹配到的位置是：开始于 3，结束于 5。

### 1.3 能够与 '多种字符' 匹配的表达式

正则表达式中的一些表示方法，可以匹配 '多种字符' 其中的任意一个字符。比如，表达式 `"\d"` 可以匹配任意一个数字。虽然可以匹配其中任意字符，但是只能是一个，不是多个。这就好比玩扑克牌时候，大小王可以代替任意一张牌，但是只能代替一张牌。

表达式	可匹配
<code>\d</code>	任意一个数字，0~9 中的任意一个
<code>\w</code>	任意一个字母或数字或下划线，也就是 <code>A~Z,a~z,0~9,_</code> 中任意一个
<code>\s</code>	包括空格、制表符、换页符等空白字符的其中任意一个
<code>.</code>	小数点可以匹配除了换行符 ( <code>\n</code> ) 以外的任意一个字符

举例 1：表达式 `"\d\d"`，在匹配 `"abc123"` 时，匹配的结果是：成功；匹配到的内容是：`"12"`；匹配到的位置是：开始于 3，结束于 5。

举例 2：表达式 `"a.\d"`，在匹配 `"aaa100"` 时，匹配的结果是：成功；匹配到的内容是：`"aa1"`；匹配到的位置是：开始于 1，结束于 4。

### 1.4 自定义能够匹配 '多种字符' 的表达式

使用方括号 `[ ]` 包含一系列字符，能够匹配其中任意一个字符。用 `[^ ]` 包含一系列字符，则能够匹配其中字符之外的任意一个字符。同样的道理，虽然可以匹配其中任意一个，但是只能是一个，不是多个。

表达式	可匹配
[ab5@]	匹配 "a" 或 "b" 或 "5" 或 "@"
[^abc]	匹配 "a","b","c" 之外的任意一个字符
[f-k]	匹配 "f"~"k" 之间的任意一个字母
[^A-F0-3]	匹配 "A"~"F","0"~"3" 之外的任意一个字符

举例 1: 表达式 "[bcd][bcd]" 匹配 "abc123" 时, 匹配的结果是: 成功; 匹配到的内容是: "bc"; 匹配到的位置是: 开始于 1, 结束于 3。

举例 2: 表达式 "[^abc]" 匹配 "abc123" 时, 匹配的结果是: 成功; 匹配到的内容是: "1"; 匹配到的位置是: 开始于 3, 结束于 4。

## 1.5 修饰匹配次数的特殊符号

前面章节中讲到的表达式, 无论是只能匹配一种字符的表达式, 还是可以匹配多种字符其中任意一个的表达式, 都只能匹配一次。如果使用表达式再加上修饰匹配次数的特殊符号, 那么不用重复书写表达式就可以重复匹配。

使用方法是: "次数修饰"放在"被修饰的表达式"后边。比如: "[bcd][bcd]" 可以写成 "[bcd]{2}"。

表达式	作用
{n}	表达式重复 n 次, 比如: "\w{2}" 相当于 "\w\w"; "a{5}" 相当于 "aaaaa"
{m,n}	表达式至少重复 m 次, 最多重复 n 次, 比如: "ba{1,3}"可以匹配 "ba"或"baa"或"baaa"
{m,}	表达式至少重复 m 次, 比如: "\w\d{2,}"可以匹配 "a12","_456","M12344"...
?	匹配表达式 0 次或者 1 次, 相当于 {0,1}, 比如: "a[cd]?"可以匹配 "a","ac","ad"
+	表达式至少出现 1 次, 相当于 {1,}, 比如: "a+b"可以匹配 "ab","aab","aaab"...
*	表达式不出现或出现任意次, 相当于 {0,}, 比如: "\^*b"可以匹配 "b","^^^b"...

举例 1: 表达式 "\d+\.\?d\*" 在匹配 "It costs \$12.5" 时, 匹配的结果是: 成功; 匹配到的内容是: "12.5"; 匹配到的位置是: 开始于 10, 结束于 14。

举例 2: 表达式 "go{2,8}gle" 在匹配 "Ads by gooooooogle" 时, 匹配的结果是: 成功; 匹配到的内容是: "gooooooogle"; 匹配到的位置是: 开始于 7, 结束于 17。

## 1.6 其他一些代表抽象意义的特殊符号

一些符号在表达式中代表抽象的特殊意义：

表达式	作用
<code>^</code>	与字符串开始的地方匹配，不匹配任何字符
<code>\$</code>	与字符串结束的地方匹配，不匹配任何字符
<code>\b</code>	匹配一个单词边界，也就是单词和空格之间的位置，不匹配任何字符

进一步的文字说明仍然比较抽象，因此，举例帮助大家理解。

举例 1：表达式 `"^aaa"` 在匹配 `"xxx aaa xxx"` 时，匹配结果是：失败。因为 `"^"` 要求与字符串开始的地方匹配，因此，只有当 `"aaa"` 位于字符串的开头的时候，`"^aaa"` 才能匹配，比如：`"aaa xxx xxx"`。

举例 2：表达式 `"aaa$"` 在匹配 `"xxx aaa xxx"` 时，匹配结果是：失败。因为 `"$"` 要求与字符串结束的地方匹配，因此，只有当 `"aaa"` 位于字符串的结尾的时候，`"aaa$"` 才能匹配，比如：`"xxx xxx aaa"`。

举例 3：表达式 `".\b."` 在匹配 `"@@@abc"` 时，匹配结果是：成功；匹配到的内容是：`"@a"`；匹配到的位置是：开始于 2，结束于 4。

进一步说明：`"\b"` 与 `"^"` 和 `"$"` 类似，本身不匹配任何字符，但是它要求它在匹配结果中所处位置的左右两边，其中一边是 `"\w"` 范围，另一边是非 `"\w"` 的范围。

举例 4：表达式 `"\bend\b"` 在匹配 `"weekend,endfor,end"` 时，匹配结果是：成功；匹配到的内容是：`"end"`；匹配到的位置是：开始于 15，结束于 18。

一些符号可以影响表达式内部的子表达式之间的关系：

表达式	作用
<code> </code>	左右两边表达式之间 "或" 关系，匹配左边或者右边
<code>( )</code>	(1). 在被修饰匹配次数的时候，括号中的表达式可以作为整体被修饰 (2). 取匹配结果的时候，括号中的表达式匹配到的内容可以被单独得到

举例 5：表达式 `"Tom|Jack"` 在匹配字符串 `"I'm Tom, he is Jack"` 时，匹配结果是：成功；匹配到的内容是：`"Tom"`；匹配到的位置是：开始于 4，结束于 7。匹配下一个时，匹配结果是：成功；匹配到的内容是：`"Jack"`；匹配到的位置是：开始于 15，结束于 19。

举例 6：表达式 `"(go\s*)+"` 在匹配 `"Let's go go go!"` 时，匹配结果是：成功；匹配到内容是：`"go go go"`；匹配到的位置是：开始于 6，结束于 14。

举例 7：表达式 `"¥(\d+\.\d*)"` 在匹配 `"$ 10.9,¥20.5"` 时，匹配的结果是：成功；匹配到的内容是：`"¥20.5"`；匹配到的位置是：开始于 6，结束于 10。单独获取括号范围匹配到的内容是：`"20.5"`。

## 2. 正则表达式中的一些高级规则

### 2.1 匹配次数中的贪婪与非贪婪

在使用修饰匹配次数的特殊符号时，有几种表示方法可以使同一个表达式能够匹配不同的次数，比如：`"{m,n}"`，`"{m,}"`，`"?"`，`"*"`，`"+"`，具体匹配的次数随被匹配的字符串而定。这种重复匹配不定次数的表达式在匹配过程中，总是尽可能多的匹配。比如，针对文本 `"dxxxdxxxd"`，举例如下：

表达式	匹配结果
<code>(d)(w+)</code>	<code>"\w+"</code> 将匹配第一个 <code>"d"</code> 之后的所有字符 <code>"xxxdxxxd"</code>
<code>(d)(w+)(d)</code>	<code>"\w+"</code> 将匹配第一个 <code>"d"</code> 和最后一个 <code>"d"</code> 之间的所有字符 <code>"xxxdxxx"</code> 。虽然 <code>"\w+"</code> 也能够匹配上最后一个 <code>"d"</code> ，但是为了使整个表达式匹配成功， <code>"\w+"</code> 可以 "让出" 它本来能够匹配的最后一个 <code>"d"</code>

由此可见，`"\w+"` 在匹配的时候，总是尽可能多的匹配符合它规则的字符。虽然第二个举例中，它没有匹配最后一个 `"d"`，但那也是为了让整个表达式能够匹配成功。同理，带 `"*"` 和 `"{m,n}"` 的表达式都是尽可能地多匹配，带 `"?"` 的表达式在可匹配可不匹配的时候，也是尽可能的 "要匹配"。这种匹配原则就叫作 "贪婪" 模式。

非贪婪模式：

在修饰匹配次数的特殊符号后再加上一个 `"?"` 号，则可以使匹配次数不定的表达式尽可能少的匹配，使可匹配可不匹配的表达式，尽可能的 "不匹配"。这种匹配原则叫作 "非贪婪" 模式，也叫作 "勉强" 模式。如果少匹配就会导致整个表达式匹配失败的时候，与贪婪模式类似，非贪婪模式会最小限度的再匹配一些，以使整个表达式匹配成功。举例如下，针对文本 `"dxxxdxxxd"` 举例：

表达式	匹配结果
<code>(d)(w+?)</code>	<code>"\w+?"</code> 将尽可能少的匹配第一个 <code>"d"</code> 之后的字符，结果是： <code>"\w+?"</code> 只匹配了一个 <code>"x"</code>
<code>(d)(w+?)(d)</code>	为了让整个表达式匹配成功， <code>"\w+?"</code> 不得不匹配 <code>"xxx"</code> 才可以让后边的 <code>"d"</code> 匹配，从而使整个表达式匹配成功。因此，结果是： <code>"\w+?"</code> 匹配 <code>"xxx"</code>

更多的情况，举例如下：

举例 1：表达式 `"<td>(.*?)</td>"` 与字符串 `"<td><p>aa</p></td><td><p>bb</p></td>"` 匹配时，匹配的结果是：成功；匹配到的内容是 `"<td><p>aa</p></td> <td><p>bb</p></td>"` 整个字符串，表达式中的 `"</td>"` 将与字符串中最后一个 `"</td>"` 匹配。

举例 2: 相比之下, 表达式 `"<td>(.*?)</td>"` 匹配举例 1 中同样的字符串时, 将只得到 `"<td><p>aa</p></td>"`, 再次匹配下一个时, 可以得到第二个 `"<td><p>bb</p></td>"`。

---

## 2.2 反向引用 \1, \2...

表达式在匹配时, 表达式引擎会将小括号 `"( )"` 包含的表达式所匹配到的字符串记录下来。在获取匹配结果的时候, 小括号包含的表达式所匹配到的字符串可以单独获取。这一点, 在前面的举例中, 已经多次展示了。在实际应用场合中, 当用某种边界来查找, 而所要获取的内容又不包含边界时, 必须使用小括号来指定所要的范围。比如前面的 `"<td>(.*?)</td>"`。

其实, "小括号包含的表达式所匹配到的字符串" 不仅是在匹配结束后才可以使用, 在匹配过程中也可以使用。表达式后边的部分, 可以引用前面 "括号内的子匹配已经匹配到的字符串"。引用方法是 `"\"` 加上一个数字。`"\1"` 引用第 1 对括号内匹配到的字符串, `"\2"` 引用第 2 对括号内匹配到的字符串.....以此类推, 如果一对括号内包含另一对括号, 则外层的括号先排序号。换句话说, 哪一对的左括号 `"("` 在前, 那这一对就先排序号。

举例如下:

举例 1: 表达式 `"('|')(.*?)(\1)"` 在匹配 `" 'Hello', 'World' "` 时, 匹配结果是: 成功; 匹配到的内容是: `" 'Hello' "`。再次匹配下一个时, 可以匹配到 `" 'World' "`。

举例 2: 表达式 `"(\w)\1{4,}"` 在匹配 `"aa bbbb abcdefg ccccc 111121111 999999999"` 时, 匹配结果是: 成功; 匹配到的内容是 `"cccc"`。再次匹配下一个时, 将得到 `999999999`。这个表达式要求 `"\w"` 范围的字符至少重复 5 次, 注意与 `"\w{5,}"` 之间的区别。

举例 3: 表达式 `"<(\w+)\s*(\w+(\('='|").*?\4)?\s*)*>.*?</\1>"` 在匹配 `"<td id='td1' style='bgcolor:white'></td>"` 时, 匹配结果是成功。如果 `"<td>"` 与 `"</td>"` 不配对, 则会匹配失败; 如果改成其他配对, 也可以匹配成功。

---

## 2.3 预搜索, 不匹配; 反向预搜索, 不匹配

前面的章节中, 我讲到了几个代表抽象意义的特殊符号: `"^"`, `"$"`, `"\b"`。它们都有一个共同点, 那就是: 它们本身不匹配任何字符, 只是对 "字符串的两头" 或者 "字符之间的缝隙" 附加了一个条件。理解到这个概念以后, 本节将继续介绍另外一种对 "两头" 或者 "缝隙" 附加条件的, 更加灵活表示方法。

正向预搜索: `"(?=xxxxx)"`, `"(?!xxxxx)"`

格式: `"(?=xxxxx)"`, 在被匹配的字符串中, 它对所处的 "缝隙" 或者 "两头" 附加的条件

是：所在缝隙的右侧，必须能够匹配上 `xxxxx` 这部分的表达式。因为它只是在此作为这个缝隙上附加的条件，所以它并不影响后边的表达式去真正匹配这个缝隙之后的字符。这就类似 `"\b"`，本身不匹配任何字符。`"\b"` 只是将所在缝隙之前、之后的字符取来进行了一下判断，不会影响后边的表达式来真正的匹配。

举例 1：表达式 `"Windows (?=NT|XP)"` 在匹配 `"Windows 98, Windows NT, Windows 2000"` 时，将只匹配 `"Windows NT"` 中的 `"Windows"`，其他的 `"Windows"` 字样则不被匹配。

举例 2：表达式 `"(\w)((?=\1\1\1)(\1))+"` 在匹配字符串 `"aaa fffff 999999999"` 时，将可以匹配 6 个 `"f"` 的前 4 个，可以匹配 9 个 `"9"` 的前 7 个。这个表达式可以读解成：重复 4 次以上的字母数字，则匹配其剩下最后 2 位之前的部分。当然，这个表达式可以不这样写，在此的目的是作为演示之用。

格式：`"(?!xxxxx)"`，所在缝隙的右侧，必须不能匹配 `xxxxx` 这部分表达式。

举例 3：表达式 `"((?!\bstop\b).)+"` 在匹配 `"fdjka ljfdl stop fjdsld fdj"` 时，将从头一直匹配到 `"stop"` 之前的位置，如果字符串中没有 `"stop"`，则匹配整个字符串。

举例 4：表达式 `"do(?:\w)"` 在匹配字符串 `"done, do, dog"` 时，只能匹配 `"do"`。在本条举例中，`"do"` 后边使用 `"(?:\w)"` 和使用 `"\b"` 效果是一样的。

反向预搜索：`"(?<=xxxxx)"`，`"(?<!\xxxxx)"`

这两种格式的概念和正向预搜索是类似的，反向预搜索要求的条件是：所在缝隙的“左侧”，两种格式分别要求必须能够匹配和必须不能够匹配指定表达式，而不是去判断右侧。与“正向预搜索”一样的是：它们都是对所在缝隙的一种附加条件，本身都不匹配任何字符。

举例 5：表达式 `"(?<=\d{4})\d+(?=\d{4})"` 在匹配 `"1234567890123456"` 时，将匹配除了前 4 个数字和后 4 个数字之外的中间 8 个数字。由于 `JScript.RegExp` 不支持反向预搜索，因此，本条举例不能够进行演示。很多其他的引擎可以支持反向预搜索，比如：`Java 1.4` 以上的 `java.util.regex` 包，`.NET` 中 `System.Text.RegularExpressions` 命名空间，`boost::regex` 以及 `GRETA` 正则表达式库等。

### 3. 其他通用规则

还有一些在各个正则表达式引擎之间比较通用的规则，在前面的讲解过程中没有提到。

3.1 表达式中，可以使用 `"\xXX"` 和 `"\uXXXX"` 表示一个字符（`"X"` 表示一个十六进制数）

形式	字符范围
<code>\xXX</code>	编号在 0 ~ 255 范围的字符，比如：空格可以使用 <code>"\x20"</code> 表示



\uXXXX	任何字符可以使用 "\u" 再加上其编号的 4 位十六进制数表示，比如："\u4E2D"
--------	--

3.2 在表达式 "\s", "\d", "\w", "\b" 表示特殊意义的同时，对应的大写字母表示相反的意义

表达式	可匹配
\S	匹配所有非空白字符（"\s" 可匹配各个空白字符）
\D	匹配所有的非数字字符
\W	匹配所有的字母、数字、下划线以外的字符
\B	匹配非单词边界，即左右两边都是 "\w" 范围或者左右两边都不是 "\w" 范围时的字符缝隙

3.3 在表达式中有特殊意义，需要添加 "\" 才能匹配该字符本身的字符汇总

字符	说明
^	匹配输入字符串的开始位置。要匹配 "^" 字符本身，请使用 "\\^"
\$	匹配输入字符串的结尾位置。要匹配 "\$" 字符本身，请使用 "\\\$"
( )	标记一个子表达式的开始和结束位置。要匹配小括号，请使用 "\\(" 和 "\\)"
[ ]	用来自定义能够匹配 '多种字符' 的表达式。要匹配中括号，请使用 "\\[" 和 "\\]"
{ }	修饰匹配次数的符号。要匹配大括号，请使用 "\\{" 和 "\\}"
.	匹配除了换行符（\n）以外的任意一个字符。要匹配小数点本身，请使用 "\\."
?	修饰匹配次数为 0 次或 1 次。要匹配 "?" 字符本身，请使用 "\\?"
+	修饰匹配次数为至少 1 次。要匹配 "+" 字符本身，请使用 "\\+"
*	修饰匹配次数为 0 次或任意次。要匹配 "*" 字符本身，请使用 "\\*"
	左右两边表达式之间 "或" 关系。匹配 " " 本身，请使用 "\\ "

3.4 括号 "( )" 内的子表达式，如果希望匹配结果不进行记录供以后使用，可以使用 "(?:xxxxx)" 格式

举例 1: 表达式 "(?:\\w\\1)+" 匹配 "a bbccdd efg" 时，结果是 "bbccdd"。括号 "(?:)" 范围的匹配结果不进行记录，因此 "(\\w)" 使用 "\\1" 来引用。

3.5 常用的表达式属性设置简介: Ignorecase, Singleline, Multiline, Global

表达式属性	说明
Ignorecase	默认情况下，表达式中的字母是要区分大小写的。配置为 Ignorecase 可使匹配时不区分大小写。有的表达式引擎，把 "大小写" 概念延伸至 UNICODE 范围的大小写。



Singleline	默认情况下，小数点 "." 匹配除了换行符（\n）以外的字符。配置为 <b>Singleline</b> 可使小数点可匹配包括换行符在内的所有字符。
Multiline	<p>默认情况下，表达式 "^" 和 "\$" 只匹配字符串的开始 ① 和结尾 ④ 位置。如：</p> <pre>①xxxxxxxx②\n ③xxxxxxxx④</pre> <p>配置为 <b>Multiline</b> 可以使 "^" 还可以匹配换行符之后，下一行开始前 ③ 的位置，使 "\$" 还可以匹配换行符之前，一行结束 ② 的位置。</p>
Global	主要在将表达式用来替换时起作用，配置为 <b>Global</b> 表示替换所有的匹配。

## 4. 综合提示

**4.1** 如果要要求表达式所匹配的内容是整个字符串，而不是从字符串中找一部分，那么可以在表达式的首尾使用 "^" 和 "\$"，比如："**^\d+\$**" 要求整个字符串只有数字。

**4.2** 如果要求匹配的内容是一个完整的单词，而不会是单词的一部分，那么在表达式首尾使用 "\b"，比如：使用 "**\b(if|while|else|void|int.....)\b**" 来匹配程序中的关键字。

**4.3** 表达式不要匹配空字符串。否则会一直得到匹配成功，而结果什么都没有匹配到。比如：准备写一个匹配 "123"、"123."、"123.5"、".5" 这几种形式的表达式时，整数、小数点、小数数字都可以省略，但是不要将表达式写成："**\d\*\.?\d\***"，因为如果什么都没有，这个表达式也可以匹配成功。更好的写法是："**\d+\.?\d\*|\.\d+**"。

**4.4** 能匹配空字符串的子匹配不要循环无限次。如果括号内的子表达式中的每一部分都可以匹配 0 次，而这个括号整体又可以匹配无限次，那么情况可能比上一条所说的更严重，匹配过程中可能死循环。虽然现在有些正则表达式引擎已经通过办法避免了这种情况出现死循环了，比如 .NET 的正则表达式，但是我们仍然应该尽量避免出现这种情况。如果我们在写表达式时遇到了死循环，也可以从这一点入手，查找一下是否是本条所说的原因。

**4.5** 合理选择贪婪模式与非贪婪模式。

**4.6** 或 "|" 的左右两边，对某个字符最好只有一边可以匹配，这样，不会因为 "|" 两边的表达式因为交换位置而有所不同。

# 正则表达式 **30** 分钟入门教程(第二版)

作者: [deerchao](#) 来源:[unibetter](#) 大学生社区 转载请注明来源 本文第一版

## 目录

1. 本文目标
2. 如何使用本教程
3. 什么是正则表达式?
4. 入门
5. 测试正则表达式
6. 元字符
7. 字符转义
8. 重复
9. 字符类
10. 反义
11. 替换
12. 分组
13. 后向引用
14. 位置指定
15. 负向位置指定
16. 注释
17. 贪婪与懒惰
18. 平衡组
19. 还有什么东西没提到
20. 一些我认为你可能已经知道的术语的参考
21. 网上的资源及本文参考文献
22. 第二版更新说明

## 本文目标

**30** 分钟内让你明白正则表达式是什么, 并对它有一些基本的了解, 让你可以在自己的程序或网页里使用它。

## 如何使用本教程

别被下面那些复杂的表达式吓倒, 只要跟着我一步一步来, 你会发现正则表达式其实并不像你想像中的那么困难。当然, 如果你看完了这篇教程之后发现自己明白了 很多, 却又几乎什么都记不得, 那也是很正常的——其实我认为没接触过正则表达式的人在看完这篇教程后能把提到过的语法记住 **80%** 以上的可能性为零。这里只 是让你明白基本道理, 以后你还需要多练习, 多查资料, 才能熟练掌握正则表达式。

除了作为入门教程之外，本文还试图成为可以在日常工作中使用的正则表达式语法参考手册（就作者本人的经历来说，这个目标还是完成得不错的）。

文本格式约定：专业术语 元字符/语法格式 正则表达式 正则表达式中的一部分 (用于分析) 用于在其中搜索的字符串 对正则表达式或其中一部分的说明

## 什么是正则表达式？

在编写处理字符串的程序或网页时，经常会有查找符合某些复杂规则的字符串的需要。正则表达式就是用于描述这些规则的工具。换句话说，正则表达式就是记录文本规则的代码。

很可能你使用过 Windows/Dos 下用于文件查找的通配符(wildcard)，也就是\*和?。如果你想查找某个目录下的所有的 Word 文档的话，你会搜索\*.doc。在这里，\*会被解释成任意的字符串。和通配符类似，正则表达式也是用来进行文本匹配的工具，只不过比起通配符它能更精确地描述你的需求——当然，代价就是更复杂。比如你可以编写一个正则表达式来查找所有以0开头，后面跟着2-3个数字，然后是一个连字号“-”，最后是7或8位数字的字符串(像010-12345678或0376-7654321)。

正则表达式是用于进行文本匹配的工具，所以本文里多次提到了在字符串里搜索/查找，这种说法的意思是在给定的字符串中，寻找与给定的正则表达式相匹配的部分。有可能字符串里有不止一个部分满足给定的正则表达式，这时每一个这样的部分被称为一个匹配。匹配在本文里可能会有三种意思：一种是形容词性的，比如说一个字符串匹配一个表达式；一种是动词性的，比如说在字符串里匹配正则表达式；还有一种是名词性的，就是刚刚说到的“字符串中满足给定的正则表达式的一部分”。

## 入门

学习正则表达式的最好方法是从例子开始，理解例子之后再自己对例子进行修改，实验。下面给出了不少简单的例子，并对它们作了详细的说明。

假设你在一篇英文小说里查找hi，你可以使用正则表达式hi。

这是最简单的正则表达式了，它可以精确匹配这样的字符串：由两个字符组成，前一个字符是h,后一个是i。通常，处理正则表达式的工具会提供一个忽略大小写的选项，如果选中了这个选项，它可以匹配hi,HI,Hi,hI这四种情况中的任意一种。

不幸的是，很多单词里包含hi这两个连续的字符，比如him,history,high等等。用hi来查找的话，这里边的hi也会被找出来。如果要精确地查找hi这个单词的话，我们应该使用\bhi\b。

`\b` 是正则表达式规定的一个特殊代码（好吧，某些人叫它**元字符**，**metacharacter**），代表着单词的开头或结尾，也就是单词的分界处。虽然通常英文的单词是由空格或标点符号或换行来分隔的，但是`\b`并不匹配这些单词分隔符中的任何一个，它**只匹配一个位置**。（如果需要更精确的说法，`\b`匹配这样的位置：它的前一个字符和后一个字符不全是`\w`）

假如你要找的是 hi 后面不远处跟着一个 Lucy，你应该用`\bhi\b.*\bLucy\b`。

这里，`.`是另一个元字符，匹配除了换行符以外的任意字符。`*`同样是元字符，不过它代表的不是字符，也不是位置，而是数量--它指定`*`前边的内容可以重复任意次以使整个表达式得到匹配。因此，`.*`连在一起就意味着任意数量的不包含换行的字符。现在`\bhi\b.*\bLucy\b`的意思就很明显了：先是一个单词 hi, 然后是任意个任意字符(但不能是换行)，最后是 Lucy 这个单词。

如果同时使用其它的一些元字符，我们就能构造出功能更强大的正则表达式。比如下面这个例子：

`0\d\d-\d\d\d\d\d\d\d\d` 匹配这样的字符串：以 0 开头，然后是两个数字，然后是一个连字号“-”，最后是 8 个数字(也就是中国的电话号码。当然，这个例子只能匹配区号为 3 位的情形)。

这里的`\d`是一个新的元字符，匹配任意的数字(0，或 1，或 2，或.....)。-不是元字符，只匹配它本身——连字号。

为了避免那么多烦人的重复，我们也可以这样写这个表达式：`0\d{2}-\d{8}`

这里`\d`后面的`{2}`(`{8}`)的意思是前面`\d` 必须连续重复匹配 2 次(8 次)。

## 测试正则表达式

如果你不觉得正则表达式很难读写的话，要么你是一个天才，要么，你不是地球人。正则表达式的语法很令人头疼，即使对经常使用它的人来说也是如此。由于难于读写，容易出错，所以很有必要创建一种工具来测试正则表达式。

由于在不同的环境下正则表达式的一些细节是不相同的，本教程介绍的是 Microsoft .Net 2.0 下正则表达式的行为，所以，我向你介绍一个 .Net 下的工具 [Regex Tester](#)。首先你确保已经安装了 [.Net Framework 2.0](#)，然后下载 [Regex Tester](#)，下载完后打开压缩包,直接运行 `RegexTester.exe`。

下面是 `Regex Tester` 运行时的截图：



## 元字符

现在你已经知道几个很有用的元字符了，如**`\b`**、**`.`**、**`*`**，还有**`\d`**。当然还有更多的元字符，比如**`\s`**匹配任意的空白符，包括空格，制表符(Tab)，换行符，中文全角空格等。**`\w`**匹配字母或数字或下划线或汉字。

下面来试试更多的例子：

**`\ba\w*\b`** 匹配以字母 **a** 开头的单词——先是某个单词开始处(**`\b`**)，然后是字母 **a**，然后是任意数量的字母或数字(**`\w*`**)，最后是单词结束处(**`\b`**)（好吧，现在我们说说这里的单词是什么意思吧：就是几个连续的**`\w`**。不错，这与学习英文时要背的成千上万个同名的东西的确关系不大）。

**`\d+`** 匹配 **1** 个或更多连续的数字。这里的**`+`**是和**`*`**类似的元字符，不同的是**`*`**匹配重复任意次(可能是 **0** 次)，而**`+`**则匹配重复 **1** 次或更多次。

**`\b\w{6}\b`** 匹配刚好 **6** 个字母/数字的单词。

表 1.常用的元字符

代码	说明
<b><code>.</code></b>	匹配除换行符以外的任意字符
<b><code>\w</code></b>	匹配字母或数字或下划线或汉字
<b><code>\s</code></b>	匹配任意的空白符

`\d` 匹配数字  
`\b` 匹配单词的开始或结束  
`^` 匹配字符串的开始  
`$` 匹配字符串的结束

元字符`^`（和`6`在同一个键位上的符号）以及`$`和`\b`有点类似，都匹配一个位置。`^`匹配你要用来查找的字符串的开头，`$`匹配结尾。这两个代码在验证输入的内容时非常有用，比如一个网站如果要求你填写的QQ号必须为5位到12位数字时，可以使用：`^\d{5,12}$`。

这里的`{5,12}`和前面介绍过的`{2}`是类似的，只不过`{2}`匹配只能不多不少重复2次，`{5,12}`则是必须重复最少5次，最多12次，否则都不匹配。

因为使用了`^`和`$`，所以输入的整个字符串都要用来和`\d{5,12}`来匹配，也就是说整个输入必须是5到12个数字，因此如果输入的QQ号能匹配这个正则表达式的话，那就符合要求了。

和忽略大小写的选项类似，有些正则表达式处理工具还有一个处理多行的选项。如果选中了这个选项，`^`和`$`的意义就变成了匹配行的开始处和结束处。

## 字符转义

如果你想查找元字符本身的话，比如你查找`.`或者`*`，就出现了问题：你没法指定它们，因为它们会被解释成其它的意思。这时你就必须使用`\`来取消这些字符的特殊意义。因此，你应该使用`\.`和`\*`。当然，要查找`\`本身，你也得用`\\`。

例如：`www\.unibetter\.com`匹配`www.unibetter.com`，`c:\\windows`匹配`c:\windows`，`2\^8`匹配`2^8`（通常这是2的8次方的书写方式）。

## 重复

你已经看过了前面的`*`、`+`、`{2}`、`{5,12}`这几个匹配重复的方式了。下面是正则表达式中所有指定重复的方式：

表 2. 常用的限定符

代码/语法	说明
<code>*</code>	重复零次或更多次
<code>+</code>	重复一次或更多次
<code>?</code>	重复零次或一次
<code>{n}</code>	重复 n 次
<code>{n,}</code>	重复 n 次或更多次
<code>{n,m}</code>	重复 n 到 m 次

下面是一些使用重复的例子：



**Windows\d+**匹配 Windows 后面跟 1 个或更多数字

**13\d{9}**匹配 以 13 后面跟 9 个数字(中国的手机号)

**^\w+**匹配 一行的第一个单词(或整个字符串的第一个单词，具体匹配哪个意思得看选项设置)

## 字符类

要想查找数字，字母或数字，空白是很简单的，因为已经有了对应这些字符集的元字符，但是如果你想匹配没有预定义元字符的字符集比如元音字母(a,e,i,o,u)，怎么办？

很简单，你只需要在中括号里列出它们就行了，像**[aeiou]**就匹配任何一个元音字母，**[.?!]**匹配标点符号(.或?或!)(英文语句通常只以这三个标点结束)。注意，我们不需要写成**[\.?!]**。

我们也可以轻松地指定一个字符**范围**，像**[0-9]**代表的含意与**\d**就是完全一致的：一位数字，同理**[a-z0-9A-Z\_]**也完全等同于**\w**（如果只考虑英文的话）。

下面是一个更复杂的表达式：**\(?:0\d{2}[ ]-)?\d{8}**。

这个表达式可以匹配几种格式的电话号码，像**(010)88886666**，或**022-22334455**，或**02912345678**等。我们对它进行一些分析吧：首先是一个转义字符**\**(,它能出现 0 次或 1 次(**?**),然后是一个**0**,后面跟着 2 个数字(**{2}**),然后是**)**或**-**或**空格**中的一个,它出现 1 次或不出现(**?**),最后是 8 个数字(**\d{8}**)。不幸的是，它也能匹配 **010)12345678** 或 **(022-87654321** 这样的“不正确”的格式。要解决这个问题，请在本教程的下面查找答案。

## 反义

有时需要查找不属于某个能简单定义的字符类的字符。比如想查找除了数字以外，其它任意字符都行的情况，这时需要用到**反义**：

表 3.常用的反义代码

代码/语法	说明
<b>\W</b>	<u>匹配任意不是字母，数字，下划线，汉字的字符</u>
<b>\S</b>	<u>匹配任意不是空白符的字符</u>
<b>\D</b>	<u>匹配任意非数字的字符</u>
<b>\B</b>	<u>匹配不是单词开头或结束的位置</u>
<b>[^x]</b>	<u>匹配除了 x 以外的任意字符</u>
<b>[^aeiou]</b>	<u>匹配除了 aeiou 这几个字母以外的任意字符</u>

例子：**\S+**匹配不包含空白符的字符串。



`<a[^>]+>` 匹配用尖括号括起来的以 **a** 开头的字符串。

## 替换

好了，现在终于到了解决 3 位或 4 位区号问题的时间了。正则表达式里的**替换**指的是有几种规则，如果满足其中任意一种规则都应该当成匹配，具体方法是用 `|` 把不同的规则分隔开。听不明白？没关系，看例子：

`0\d{2}-\d{8}|0\d{3}-\d{7}` 这个表达式能匹配两种以连字号分隔的电话号码：一种是三位区号，8 位本地号(如 010-12345678)，一种是 4 位区号，7 位本地号(0376-2233445)。

`\(0\d{2}\)[- ]?\d{8}|0\d{2}[- ]?\d{8}` 这个表达式匹配 3 位区号的电话号码，其中区号可以用小括号括起来，也可以不用，区号与本地号间可以用连字号或空格间隔，也可以没有间隔。你可以试试用替换 `|` 把这个表达式扩展成也支持 4 位区号的。

`\d{5}-\d{4}|\d{5}` 这个表达式用于匹配美国的邮政编码。美国邮编的规则是 5 位数字，或者用连字号间隔的 9 位数字。之所以要给出这个例子是因为它能说明一个问题：**使用替换时，顺序是很重要的**。如果你把它改成

`\d{5}|\d{5}-\d{4}` 的话，那么就只会匹配 5 位的邮编(以及 9 位邮编的前 5 位)。原因是匹配替换时，将会从左到右地测试每个分枝条件，如果满足了某个分枝的话，就不会去管其它的替换条件了。

`Windows98|Windows2000|WindosXP` 这个例子是为了告诉你替换不仅仅能用于两种规则，也能用于更多种规则。

## 分组

我们已经提到了怎么重复单个字符(直接在字符后面加上限定符就行了)；但如果想要重复一个字符串又该怎么办？你可以用小括号来指定**子表达式**(也叫做**分组**)，然后你就可以指定这个子表达式的重复次数了，你也可以对子表达式进行其它一些操作(后面会有介绍)。

`(\d{1,3}\.){3}\d{1,3}` 是一个简单的 IP 地址匹配表达式。要理解这个表达式，请按下列顺序分析它：`\d{1,3}` 匹配 1 到 3 位的数字，`(\d{1,3}\.){3}` 匹配三位数字加上一个英文句号(这个整体也就是这个**分组**)重复 3 次，最后再加上一个一到三位的数字(`\d{1,3}`)。

不幸的是，它也将匹配 256.300.888.999 这种不可能存在的 IP 地址(IP 地址中每个数字都不能大于 255)。如果能使用算术比较的话，或许能简单地解决这个问题，但是正则表达式中并不提供关于数学的任何功能，所以只能使用冗长的分组，选择，字符类来描述一个正确的 IP 地址：

`(([0-4]\d|25[0-5]|([01]?[d\d]?)\.){3}([0-4]\d|25[0-5]|([01]?[d\d]?))`。

理解这个表达式的关键是理解 `2[0-4]\d|25[0-5][01]?\d\d?`，这里我就不细说了，你自己应该能分析得出它的意义。

## 后向引用

使用小括号指定一个子表达式后，**匹配这个子表达式的文本**可以在表达式或其它程序中作进一步的处理。默认情况下，每个分组会自动拥有一个**组号**，规则是：从左向右，以分组的左括号为标志，第一个出现的分组的组号为 1，第二个为 2，以此类推。

**后向引用**用于重复搜索前面某个分组匹配的文本。例如，`\1` 代表分组 1 匹配的文本。难以理解？请看示例：

`\b(\w+)\b\s+\1\b` 可以用来匹配重复的单词，像 *go go, kitty kitty*。首先是一个单词，也就是单词开始处和结束处之间的多于一个的字母或数字 `(\b(\w+)\b)`，然后是 1 个或几个空白符 `(\s+)`，最后是前面匹配的那个单词 `(\1)`。

你也可以自己指定子表达式的组号或**组名**。要指定一个子表达式的组名，请使用这样的语法：`(?<Word>\w+)`，这样就把 `\w+` 的组名指定为 **Word** 了。要反向引用这个分组**捕获**的内容，你可以使用 `\k<Word>`，所以上一个例子也可以写成这样：`\b(?<Word>\w+)\b\s*\k<Word>\b`。

使用小括号的时候，还有很多特定用途的语法。下面列出了最常用的一些：

表 4. 分组语法

### 捕获

<code>(exp)</code>	<u>匹配 exp, 并捕获文本到自动命名的组里</u>
<code>(?&lt;name&gt;exp)</code>	<u>匹配 exp, 并捕获文本到名称为 name 的组里，也可以写成 (? 'name' exp)</u>
<code>(?:exp)</code>	<u>匹配 exp, 不捕获匹配的文本</u>

### 位置指定

<code>(?=exp)</code>	<u>匹配 exp 前面的位置</u>
<code>(?&lt;=exp)</code>	<u>匹配 exp 后面的位置</u>
<code>(?!exp)</code>	<u>匹配后面跟的不是 exp 的位置</u>
<code>(?&lt;!=exp)</code>	<u>匹配前面不是 exp 的位置</u>

### 注释

<code>(?#comment)</code>	<u>这种类型的组不对正则表达式的处理产生任何影响，只是为了提供让人阅读注释</u>
--------------------------	--

我们已经讨论了前两种语法。第三个 `(?:exp)` 不会改变正则表达式的处理方式，只是这样的组匹配的内容不会像前两种那样被捕获到某个组里面。

## 位置指定

接下来的四个用于查找在某些内容(但并不包括这些内容)之前或之后的东西,也就是说它们用于指定一个位置,就像**\b, ^, \$**那样,因此它们也被称为**零宽断言**。最好还是拿例子来说明吧:

**(?=exp)**也叫**零宽先行断言**,它匹配文本中的某些位置,这些位置的后面能匹配给定的后缀 **exp**。比如**\b\w+(?=ing\b)**,匹配以 **ing** 结尾的单词的前面部分(除了 **ing** 以外的部分),如果在查找 *I'm singing while you're dancing.*时,它会匹配 **sing** 和 **danc**。

**(?<=exp)**也叫**零宽后行断言**,它匹配文本中的某些位置,这些位置的前面能给定的前缀匹配 **exp**。比如**(?<=\bre)\w+\b** 会匹配以 **re** 开头的单词的后半部分(除了 **re** 以外的部分),例如在查找 *reading a book* 时,它匹配 **ading**。

假如你想要给一个很长的数字中每三位间加一个逗号(当然是从右边加起了),你可以这样查找需要在前面和里面添加逗号的部分:**((?<=\d)\d{3})\*\b**。请仔细分析这个表达式,它可能不像你第一眼看出来的那么简单。

下面这个例子同时使用了前缀和后缀:**(?<=\s)\d+(?=\s)**匹配以空白符间隔的数字(再次强调,不包括这些空白符)。

## 负向位置指定

前面我们提到过怎么查找**不是某个字符或不在某个字符类里的字符**的方法(反义)。但是如果只是想要**确保某个字符没有出现,但并不想去匹配它**时怎么办?例如,如果我们想查找这样的单词--它里面出现了字母 **q**,但是 **q** 后面跟的不是字母 **u**,我们可以尝试这样:

**\b\w\*q[^u]\w\*\b** 匹配包含后面不是字母 **u** 的字母 **q** 的单词。但是如果多做测试(或者你思维足够敏锐,直接就观察出来了),你会发现,如果 **q** 出现在单词的结尾的话,像 **Iraq, Benq**, 这个表达式就会出错。这是因为**[^u]**总是匹配一个字符,所以如果 **q** 是单词的最后一个字符的话,后面的**[^u]**将会匹配 **q** 后面的单词分隔符(可能是空格,或者是句号或其它的什么),后面的**\w+\b** 将会匹配下一个单词,于是**\b\w\*q[^u]\w\*\b** 就能匹配整个 *Iraq fighting*。负向位置指定能解决这样的问题,因为它只匹配一个位置,并不消费任何字符。现在,我们可以这样来解决这个问题: **\b\w\*q(?!u)\w\*\b**。

**零宽负向先行断言(?!exp)**,只会匹配**后缀 exp 不存在的位置**。**\d{3}(?!\d)** 匹配三位数字,而且这三位数字的后面不能是数字。

同理,我们可以用**(?<![a-z])**,零宽负向后行断言来查找前缀 **exp** 不存在的位置:**(?<![a-z])\d{7}**匹配前面不是小写字母的七位数字(实验时发现错误?注意你的“区分大小写”选项是否选中)。

一个更复杂的例子:**(?<=<(\w+)>).\*(?<=\\1>)**匹配不包含属性的简单 **HTML** 标签内里的内容。**(<?(\\w+)>)**指定了这样的前缀:被尖括号括起来的单

词(比如可能是<b>), 然后是.\* (任意的字符串), 最后是一个后缀(=?<\1>)。注意后缀里的\1, 它用到了前面提过的字符转义; \1 则是一个反向引用, 引用的正是捕获的第一组, 前面的(\w+)匹配的内容, 这样如果前缀实际上是<b>的话, 后缀就是</b>了。整个表达式匹配的是<b>和</b>之间的内容(再次提醒, 不包括前缀和后缀本身)。

## 注释

小括号的另一种用途是能过语法(?#comment)来包含注释。例如:

2[0-4]\d(?#200-249)|25[0-5](?#250-255)|[01]?d\d?(?#0-199)。

要包含注释的话, 最好是启用“忽略模式里的空白符”选项, 这样在编写表达式时能任意的添加空格, Tab, 换行, 而实际使用时这些都将忽略。启用这个选项后, 在#后面到这一行结束的所有文本都将被当成注释忽略掉。

例如, 我们可以前面的一个表达式写成这样:

(?<= # 查找前缀, 但不包含它 <(\w+)> # 查找尖括号括起来的字母或数字(标签) ) # 前缀结束 .\* # 匹配任意文本 (?= # 查找后缀, 但不包含它 <\1> # 查找尖括号括起来的内容: 前面是一个"/", 后面是先前捕获的标签 ) # 后缀结束

## 贪婪与懒惰

当正则表达式中包含能接受重复的限定符(指定数量的代码, 例如\*, {5,12}等)时, 通常的行为是(在使整个表达式能得到匹配的前提下)匹配尽可能多的字符。考虑这个表达式: a.\*b, 它将会匹配最长的以 a 开始, 以 b 结束的字符串。如果用它来搜索 aabab 的话, 它会匹配整个字符串 aabab。这被称为贪婪匹配。

有时, 我们更需要懒惰匹配, 也就是匹配尽可能少的字符。前面给出的限定符都可以被转化为懒惰匹配模式, 只要在其后面加上一个问号?。这样.\*?就意味着匹配任意数量的重复, 但是在能使整个匹配成功的前提下使用最少的重复。现在看看懒惰版的例子吧:

a.\*?b 匹配最短的, 以 a 开始, 以 b 结束的字符串。如果把它应用于 aabab 的话, 它会匹配 aab 和 ab (为什么第一个匹配是 aab 而不是 ab? 简单地说, 最先开始的匹配最有最大的优先权——The Match That Begins Earliest Wins)。

表 5. 懒惰限定符

*?	重复任意次, 但尽可能少重复
++	重复 1 次或更多次, 但尽可能少重复
??	重复 0 次或 1 次, 但尽可能少重复
{n,m}?	重复 n 到 m 次, 但尽可能少重复
{n,}??	重复 n 次以上, 但尽可能少重复

## 平衡组

如果想要匹配可嵌套的层次性结构的话，就得使用平衡组了。举个例子吧，如何把“xx <aa <bbb> <bbb> aa> yy”这样的字符串里，最长的括号内的内容捕获出来？

这里需要用到以下的语法构造：

- **(?<group>)** 把捕获的内容命名为 **group**，并压入堆栈
- **(?<-group>)** 从堆栈上弹出最后压入堆栈的名为 **group** 的捕获内容，如果堆栈本来为空，则本分组的匹配失败
- **(?(group)yes|no)** 如果堆栈上存在以名为 **group** 的捕获内容的话，继续匹配 **yes** 部分的表达式，否则继续匹配 **no** 部分
- **(?!)** 零宽负向先行断言，由于没有后缀表达式，试图匹配总是失败

如果你不是一个程序员（或者你是一个对堆栈的概念不熟的程序员），你就这样理解上面的三种语法吧：第一个就是在黑板上写一个（或再写一个）“group”，第二个就是从黑板上擦掉一个“group”，第三个就是看黑板上写的还有没有“group”，如果有就继续匹配 **yes** 部分，否则就匹配 **no** 部分。

我们需要做的是每碰到了左括号，就在黑板上写一个“group”，每碰到一个右括号，就擦掉一个，到了最后就看看黑板上还有没有——如果有那就证明左括号比右括号多，那匹配就应该失败（为了能看得更清楚一点，我用了‘(?group)’的语法）：

```
<                                #最外层的左括号    [^<>]*                #
最外层的左括号后面的不是括号的内容    (                (                (? 'Open' <)
#碰到了左括号，在黑板上写一个“Open”    [^<>>]*                #匹配左括
号后面的不是括号的内容                )+                (                (? ' -Open' >)    #
碰到了右括号，擦掉一个“Open”                [^<>]*                #匹配右括号后面
不是括号的内容                )+                )*                (? (Open) (!))                #在遇到最外层
的右括号前面，判断黑板上还有没有没擦掉的“Open”；如果还有，则匹配失败>
#最外层的右括号
```

还有什么东西没提到

我已经描述了构造正则表达式的大量元素，还有一些我没有提到的东西。下面是未提到的元素的列表，包含语法和简单的说明。你可以在网上找到更详细的参考资料 来学习它们--当你需要用到它们的时候。如果你安装了 **MSDN Library**，你也可以在里面找到关于 **.net** 下正则表达式详细的文档。

表 6.尚未详细讨论的语法

<code>\a</code>	<u>报警字符(打印它的效果是电脑嘀一声)</u>
<code>\b</code>	<u>通常是单词分界位置，但如果在字符类里使用代表退格</u>
<code>\t</code>	<u>制表符，Tab</u>
<code>\r</code>	<u>回车</u>

<code>\v</code>	<u>竖向制表符</u>
<code>\f</code>	<u>换页符</u>
<code>\n</code>	<u>换行符</u>
<code>\e</code>	<u>Escape</u>
<code>\0nn</code>	<u>ASCII 代码中八进制代码为 nn 的字符</u>
<code>\xnn</code>	<u>ASCII 代码中十六进制代码为 nn 的字符</u>
<code>\unnnn</code>	<u>Unicode 代码中十六进制代码为 nnnn 的字符</u>
<code>\cN</code>	<u>ASCII 控制字符。比如 \cC 代表 Ctrl+C</u>
<code>\A</code>	<u>字符串开头(类似^，但不受处理多行选项的影响)</u>
<code>\Z</code>	<u>字符串结尾或行尾(不受处理多行选项的影响)</u>
<code>\z</code>	<u>字符串结尾(类似\$, 但不受处理多行选项的影响)</u>
<code>\G</code>	<u>当前搜索的开头</u>
<code>\p{name}</code>	<u>Unicode 中命名为 name 的字符类，例如 \p{IsGreek}</u>
<code>(?&gt;exp)</code>	<u>贪婪子表达式</u>
<code>(?&lt;x&gt;-&lt;y&gt;exp)</code>	<u>平衡组</u>
<code>(?im-nsx:exp)</code>	<u>在子表达式 exp 中改变处理选项</u>
<code>(?im-nsx)</code>	<u>为表达式后面的部分改变处理选项</u>
<code>(?(exp)yes no)</code>	<u>把 exp 当作零宽正向先行断言，如果在这个位置能匹配，使用 yes 作为此组的表达式；否则使用 no</u>
<code>(?(exp)yes)</code>	<u>同上，只是使用空表达式作为 no</u>
<code>(?(name)yes no)</code>	<u>如果命名为 name 的组捕获到了内容，使用 yes 作为表达式；否则使用 no</u>
<code>(?(name)yes)</code>	<u>同上，只是使用空表达式作为 no</u>

#### 一些我认为你可能已经知道的术语的参考

##### 字符

程序处理文字时最基本的单位，可能是字母，数字，标点符号，空格，换行符，汉字等等。

##### 字符串

0 个或多个字符的序列。

##### 文本

文字，字符串。

##### 匹配

符合规则，检验是否符合规则，符合规则的部分。