正则表达式 30 分钟入门教程

目录

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

最重要的是——请给我 **30 分钟**，如果你没有使用正则表达式的经验，请不要试图在 30 **秒**内入门。当然，如果你是超人，那自然得另当别论。

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没有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——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的原理，以后你还需要多练习，多使用，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——你看，我自己也没能把所有的东西记下来，不是吗？

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 清除格式

正则表达式到底是什么？

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，用来查找所有以 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(一个是,一个不是或不存在)\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*前边的内容可以连续重复出现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。

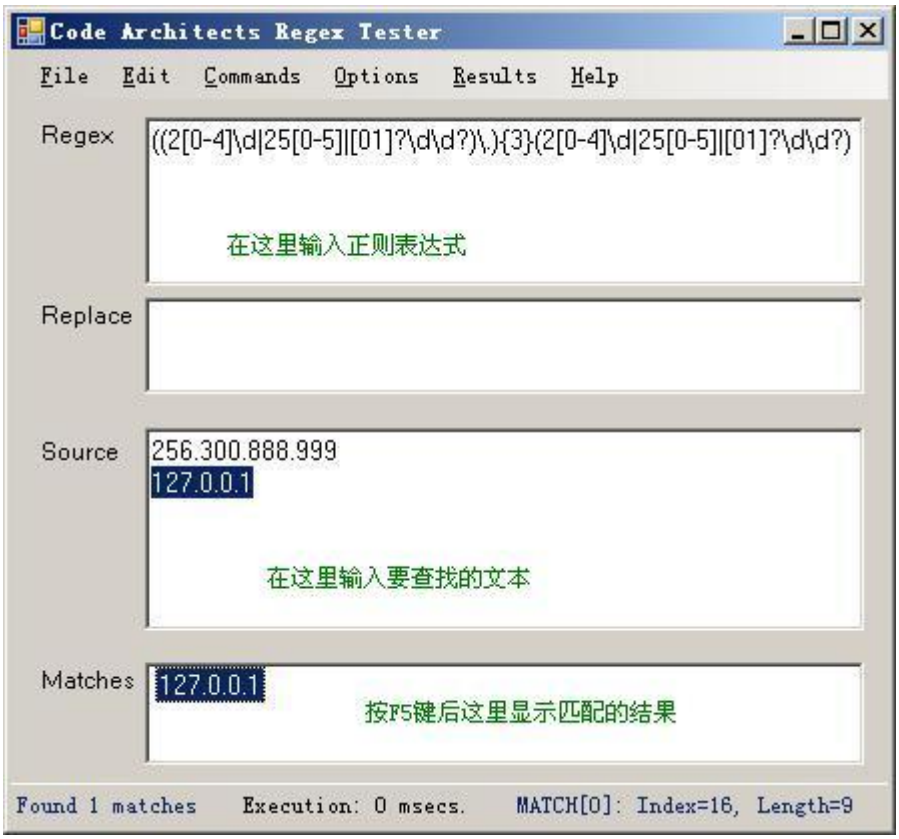
为了避免那么多烦人的重复，我们也可以这样写这个表达式：0\d{2}-\d{8}。 这里\d 后面的{2}({8})的意思是前面\d 必须连续重复匹配 2 次(8 次)。

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。这是个绿色软件，下载完后打开压缩包,直接运行 RegexTester.exe 就可以了。

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可用，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等。

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，现在我们说说正则表达式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
.	匹配除换行符以外的任意字符
\w	匹配字母或数字或下划线或汉字
\s	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^（和数字 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}则是重复的次数不能少于 5 次，不能多于 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。

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的限定符(指定数量的代码，例如*,{5,12}等)：

表 2.常用的限定符	
代码/语法	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复 n 次
{n,}	重复 n 次或更多次
{n,m}	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 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合的元字符，但是如果你想匹配没有预定义元字符的字符集合(比如元音字母 a,e,i,o,u),应该怎么办？

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

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

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）；但如果想要重复多个字符又该怎么办？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。题外话，好像反恐 24 小时第三季的编剧不知道这一点，汗...)。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：((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[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来它的意义。

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**(也就是此分组捕获的内容)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**组名**。要指定一个子表达式的组名，请使用这样的语法：(?<Word>\w+)(或者把尖括号换成'也行：(?'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	
(exp)	匹配 exp,并捕获文本到自动命名的组里
(?<name>exp)	匹配 exp,并捕获文本到名称为 name 的组里，也可以写成 (?'name'exp)
(?:exp)	匹配 exp,不捕获匹配的文本，也不给此分组分配组号
零宽断言	
(?=exp)	匹配 exp 前面的位置
(?<=exp)	匹配 exp 后面的位置
(?!exp)	匹配后面跟的不是 exp 的位置
(?<!exp)	匹配前面不是 exp 的位置
注释	
(?#comment)	这种类型的组不对正则表达式的处理产生任何影响，用于提供注释让人阅读

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。

零宽断言

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，也就是说它们像\b,^,$那样用于指定一个位置，这个位置应该满足一定的条件(断言)，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**零宽度正预测先行断言**，它断言自身出现的位置的后面能匹配表达式 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)，如查找 *I'm singing while you're dancing.*时，它会匹配 sing 和 danc。

(?<=exp)也叫**零宽度正回顾后发断言**，它断言自身出现的位置的前面能匹配表达式 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，用它对 *1234567890* 进行查找时结果是 234567890。

下面这个例子同时使用了这两种断言：(?<=\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)。

负向零宽断言

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)。但是如果我们只是想要**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]总要匹配一个字符，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*。**负向零宽断言**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。

零宽度负预测先行断言(?!exp)，断言此位置的后面不能匹配表达式 exp。例如：\d{3}(?!\d)匹配三位数字，而且这三位数字的后面不能是数字；\b((?!abc)\w)+\b 匹配不包含连续字符串 abc 的单词。

同理，我们可以用(?<!exp),**零宽度正回顾后发断言**来断言此位置的前面不能匹配表达式 exp：(?<![a-z])\d{7}匹配前面不是小写字母的七位数字。

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)，然后是.*(任意的字符串),最后是一个**后缀**(?=<\/\1>)。注意后缀里的\/，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的话，后缀就是了。整个表达式匹配的是和之间的内容(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被忽略。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：

```
(?<=     # 断言要匹配的文本的前缀
<(\w+)>  # 查找尖括号括起来的字母或数字(即 HTML/XML 标签)
)        # 前缀结束
.*       # 匹配任意文本
(?=      # 断言要匹配的文本的后缀
<\/\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，只要在它后面加上一个问号?。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，因为正则表达式有另一条规则，比懒惰 / 贪婪规则的优先级更高：最先开始的匹配拥有最高的优先权——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

处理选项

上面介绍了几个选项如忽略大小写，处理多行等，这些选项能用来改变处理正则表达式的方式。下面是.Net 中常用的正则表达式选项：

表 6.常用的处理选项	
名称	**说明**
IgnoreCase(忽略大小写)	匹配时不区分大小写。
Multiline(多行模式)	更改^和$的含义，使它们分别在任意一行的行首和行尾匹配，而不仅仅在整个字符串的开头和结尾匹配。(在此模式下,$的精确含意是:匹配\n 之前的位置以及字符串结束前的位置.)
Singleline(单行模式)	更改.的含义，使它与每一个字符匹配（包括换行符\n）。
IgnorePatternWhitespace(忽略空白)	忽略表达式中的非转义空白并启用由#标记的注释。
RightToLeft(从右向左查找)	匹配从右向左而不是从左向右进行。
ExplicitCapture(显式捕获)	仅捕获已被显式命名的组。
ECMAScript(JavaScript 兼容模式)	使表达式的行为与它在 JavaScript 里的行为一致。

一个经常被问到的问题是：是不是只能同时使用多行模式和单行模式中的一种？答案是：不是。这两个选项之间没有任何关系，除了它们的名字比较相似（以至于让人感到疑惑）以外。

平衡组/递归匹配

注意：这里介绍的平衡组语法是由.Net Framework 支持的；其它语言/库不一定支持这种功能，或者支持此功能但需要使用不同的语法。

有时我们需要匹配像(100 * (50 + 15))这样的可嵌套的层次性结构，这时简单地使用\(.+\)则只会匹配到最左边的左括号和最右边的右括号之间的内容(这里我们讨论的是贪婪模式，懒惰模式也有下面的问题)。假如原来的字符串里的左括号和右括号出现的次数不相等，比如*(5 / (3 + 2)))*，那我们的匹配结果里两者的个数也不会相等。有没有办法在这样的字符串里匹配到最长的，配对的括号之间的内容呢？

为了避免(和\(把你的大脑彻底搞糊涂，我们还是用尖括号代替圆括号吧。现在我们的问题变成了如何把 *xx <aa <bbb> <bbb> aa> yy* 这样的字符串里，最长的配对的尖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"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。

```
<                       #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
```

平衡组的一个最常见的应用就是匹配 HTML,下面这个例子可以匹配嵌套的 标签：<div[^>]*>[^<>]*(((?'Open'<div[^>]*>)[^<>]*)+((?'-Open')[^<>]*)+)*(?(Open)(?!)) .

还有些什么东西没提到

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 7.尚未详细讨论的语法	
\a	报警字符(打印它的效果是电脑嘀一声)
\b	通常是单词分界位置，但如果在字符类里使用代表退格
\t	制表符，Tab
\r	回车
\v	竖向制表符
\f	换页符
\n	换行符
\e	Escape
\0nn	ASCII 代码中八进制代码为 nn 的字符
\xnn	ASCII 代码中十六进制代码为 nn 的字符
\unnnn	Unicode 代码中十六进制代码为 nnnn 的字符
\cN	ASCII 控制字符。比如\cC 代表 Ctrl+C
\A	字符串开头(类似^，但不受处理多行选项的影响)
\Z	字符串结尾或行尾(不受处理多行选项的影响)
\z	字符串结尾(类似$，但不受处理多行选项的影响)
\G	当前搜索的开头

\p{name}	Unicode 中命名为 name 的字符类，例如\p{IsGreek}
(?>exp)	贪婪子表达式
(?<x>-<y>exp)	平衡组
(?im-nsx:exp)	在子表达式 exp 中改变处理选项
(?im-nsx)	为表达式后面的部分改变处理选项
(?(exp)yes|no)	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
(?(exp)yes)	同上，只是使用空表达式作为 no
(?(name)yes|no)	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
(?(name)yes)	同上，只是使用空表达式作为 no

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 个或更多个字符的序列。

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。

断言
声明一个应该为真的事实。只有当断言为真时才会对正则表达式继续进行匹配。

网上的资源及本文参考文献

- 微软的正则表达式教程
- System.Text.RegularExpressions.Regex 类(MSDN)
- 专业的正则表达式教学网站(英文)
- 关于.Net 下的平衡组的详细讨论（英文）
- Mastering Regular Expressions (Second Edition)

正则表达式语法

正则表达式是一种文本模式，包括普通字符（例如，a 到 z 之间的字母）和特殊字符（称为"元字符"）。模式描述在搜索文本时要匹配的一个或多个字符串。

下面是正则表达式的一些示例：

表达式	匹配
/^\s*$/	匹配空行。
/\d{2}-\d{5}/	验证由两位数字、一个连字符再加 5 位数字组成的 ID 号。
/<\s*(\S+)(\s[^>]*)?>[\s\S]*<\s*\/\1\s*>/	匹配 HTML 标记。

下表包含了元字符的完整列表以及它们在正则表达式上下文中的行为：

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*。至少匹配 *n* 次，至多匹配 *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“\n”之外的任何单个字符。若要匹配包括“\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“或”字符 (|) 组合模式部件的情况很有用。例如，与“industry|industries”相比，“industr(?:y| ies)”是一个更加经济的表达式。
(?=*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)”与“Windows 2000”中的“Windows”匹配，但不与“Windows 3.1”中的“Windows”匹配。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)”与“Windows 3.1”中的“Windows”匹配，但

	不与“Windows 2000”中的“Windows”匹配。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*	与 *x* 或 *y* 匹配。例如，“z| food”与“z”或“food”匹配。“(z| f)ood”与“zood”或“food”匹配。
[*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”中的“p”。
[*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*	匹配由 *x* 指示的控制字符。例如，\cM 匹配一个 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。等价于 [^ \f\n\r\t\v]。
\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	任何非字字符匹配。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，那么 *n* 是反向引用，后面跟 *m*。如果前面的条件均不存在，那么当 *n* 和 *m* 是八进制数 (0-7) 时，*nm* 匹配八进制转义码 *nm*。
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 (©)。

优先级顺序

正则表达式从左到右进行计算，并遵循优先级顺序，这与算术表达式非常类似。

下表从最高到最低说明了各种正则表达式运算符的优先级顺序：

运算符	说明
\	转义符

(), (?:), (?=), []	括号和中括号
*, +, ?, {n}, {n,}, {n,m}	限定符
^, $, *anymetacharacter*, *anycharacter*	定位点和序列
|	替换

字符的优先级比替换运算符高，替换运算符允许“m|food”与“m”或“food”匹配。若要匹配“mood”或“food”，请使用括号创建子表达式，从而产生“(m|f)ood”。

特殊字符

许多元字符要求在试图匹配它们时特别对待。若要匹配这些特殊字符，必须首先使字符“转义”，即，将反斜杠字符 (\) 放在它们前面。下表列出了特殊字符以及它们的含义：

特殊字符	注释
$	匹配输入字符串结尾的位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，那么 $ 还匹配 \n 或 \r 前面的位置。若要匹配 $ 字符本身，请使用 \$。
( )	标记子表达式的开始和结束。可以捕获子表达式以供以后使用。若要匹配这两个字符，请使用 \(和 \)。
*	零次或多次匹配前面的字符或子表达式。若要匹配 * 字符，请使用 *。
+	一次或多次匹配前面的字符或子表达式。若要匹配 + 字符，请使用 \+。
.	匹配除换行符 \n 之外的任何单个字符。若要匹配 .，请使用 \。
[ ]	标记中括号表达式的开始。若要匹配这些字符，请使用 \[和 \]。
?	零次或一次匹配前面的字符或子表达式，或指示“非贪心”限定符。若要匹配 ? 字符，请使用 \?。
\	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，字符 n 匹配字符 n。\n 匹配换行符。序列 \\ 匹配 \，序

	列 \(匹配 (。
/	表示文本正则表达式的开始或结束。若要匹配 / 字符，请使用 \/。
^	匹配输入字符串开始处的位置，但在中括号表达式中使用的情况除外，在那种情况下它对字符集求反。若要匹配 ^ 字符本身，请使用 \^。
{ }	标记限定符表达式的开始。若要匹配这些字符，请使用 \{ 和 \}。
|	指出在两个项之间进行选择。若要匹配 | ，请使用 |

不可打印字符

非打印字符也可以是正则表达式的组成部分。下表列出了表示非打印字符的转义序列：

字符	含义
\c*x*	匹配由 *x* 指示的控制字符。例如，\cM 匹配一个 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。
\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。等价于 [^ \f\n\r\t\v]。
\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。

字符匹配

句点 (.) 匹配字符串中的各种打印或非打印字符，只有一个字符例外。这个例外就是换行符 (\n)。下面的正则表达式匹配 aac、abc、acc、adc 等等，以及 a1c、a2c、a-c 和 a#c：

```
/a.c/
```

若要匹配包含文件名的字符串，而句点 (.) 是输入字符串的组成部分，请在正则表达式中的句点前面加反斜扛 (\) 字符。举例来说明，下面的正则表达式匹配 filename.ext：

```
/filename\.ext/
```

这些表达式只让您匹配"任何"单个字符。可能需要匹配列表中的特定字符组。例如，可能需要查找用数字表示的章节标题（Chapter 1、Chapter 2 等等）。

中括号表达式

若要创建匹配字符组的一个列表，请在方括号（[和]）内放置一个或更多单个字符。当字符括在中括号内时，该列表称为"中括号表达式"。与在任何别的位置一样，普通字符在中括号内表示其本身，即，它在输入文本中匹配一次其本身。大多数特殊字符在中括号表达式内出现时失去它们的意义。不过也有一些例外，如：

- 如果] 字符不是第一项，它结束一个列表。若要匹配列表中的] 字符，请将它放在第一位，紧跟在开始 [后面。
- \ 字符继续作为转义符。若要匹配 \ 字符，请使用 \\。

括在中括号表达式中的字符只匹配处于正则表达式中该位置的单个字符。以下正则表达式匹配 Chapter 1、Chapter 2、Chapter 3、Chapter 4 和 Chapter 5：

```
/Chapter [12345]/
```

请注意，单词 *Chapter* 和后面的空格的位置相对于中括号内的字符是固定的。中括号表达式指定的只是匹配紧跟在单词 *Chapter* 和空格后面的单个字符位置的字符集。这是第九个字符位置。

若要使用范围代替字符本身来表示匹配字符组，请使用连字符 (-) 将范围中的开始字符和结束字符分开。单个字符的字符值确定范围内的相对顺序。下面的正则表达式包含范围表达式，该范围表达式等效于上面显示的中括号中的列表。

```
/Chapter [1-5]/
```

当以这种方式指定范围时，开始值和结束值两者都包括在范围内。注意，还有一点很重要，按 Unicode 排序顺序，开始值必须在结束值的前面。

若要在中括号表达式中包括连字符，请采用下列方法之一：

- 用反斜扛将它转义：

```
[\-]
```

- 将连字符放在中括号列表的开始或结尾。下面的表达式匹配所有小写字母和连字符：

- ```
  [-a-z]
  ```

```
[a-z-]
```

- 创建一个范围，在该范围中，开始字符值小于连字符，而结束字符值等于或大于连字符。下面的两个正则表达式都满足这一要求：

- ```
  [!--]
  ```

```
[!~~]
```

若要查找不在列表或范围内的所有字符，请将插入符号 (^) 放在列表的开头。如果插入字符出现在列表中的其他任何位置，则它匹配其本身。下面的正则表达式匹配编号大于 5 的章节标题：

```
/Chapter [^12345]/
```

在上面的示例中，表达式在第九个位置匹配 1、2、3、4 或 5 之外的任何数字字符。这样，例如，Chapter 7 就是一个匹配项，Chapter 9 也是一个匹配项。

上面的表达式可以使用连字符 (-) 来表示：

```
/Chapter [^1-5]/
```

中括号表达式的典型用途是指定任何大写或小写字母或任何数字的匹配。下面的表达式指定这样的匹配：

```
/[A-Za-z0-9]/
```

限定符

如果您不能指定构成匹配的字符的数量，那么正则表达式支持限定符的概念。这些限定符使您能够指定，为使匹配为真，正则表达式的某个给定组件必须出现多少次。

下表说明各种限定符以及它们的含义：

字符	说明
*	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+ 匹配 zo 和 zoo，但不匹配 z。+ 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} 不匹配 Bob 中的 o，但匹配 food 中的两个 o。
{*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*。至少匹配 *n* 次，至多匹配 *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。

由于章节编号在大的输入文档中会很可能超过九，所以您需要一种方式来处理两位或三位章节编号。限定符给您这种能力。下面的正则表达式匹配编号为任何位数的章节标题：

```
/Chapter [1-9][0-9]*/
```

请注意，限定符出现在范围表达式之后。因此，它应用于整个范围表达式，在本例中，只指定从 0 到 9 的数字（包括 0 和 9）。

这里不使用 + 限定符，因为在第二个位置或后面的位置不一定需要有一个数字。也不使用？字符，因为它将章节编号限制到只有两位数。您需要至少匹配 Chapter 和空格字符后面的一个数字。

如果您知道章节编号被限制为只有 99 章，可以使用下面的表达式来至少指定一位但至多两位数字。

```
/Chapter [0-9]{1,2}/
```

上面的表达式的缺点是，大于 99 的章节编号仍只匹配开头两位数字。另一个缺点是 Chapter 0 也将匹配。只匹配两位数字的更好的表达式如下：

```
/Chapter [1-9][0-9]?/
```

或

```
/Chapter [1-9][0-9]{0,1}/
```

*、+ 和 ? 限定符都被称为"贪心的"，因为它们匹配尽可能多的文本。但是，有时您只需要最小的匹配。

例如，您可能搜索 HTML 文档，以查找括在 H1 标记内的章节标题。该文本在您的文档中如下：

```
<H1>Chapter 1 - Introduction to Regular Expressions</H1>
```

下面的表达式匹配从开始小于符号 (<) 到关闭 H1 标记的大于符号 (>) 之间的所有内容。

```
/<.*>/
```

如果您只需要匹配开始 H1 标记，下面的"非贪心"表达式只匹配 <H1>。

```
/<.*?>/
```

通过在 *、+ 或 ? 限定符之后放置 ?，该表达式从“贪心”表达式转换为“非贪心”表达式或者最小匹配。

定位点

本节前面的主题中的示例只涉及章节标题查找。字符串 Chapter 后面跟空格和数字的任何匹配项可能是实际章节标题，或者也可能是指向另一章的交叉引用。由于真正的章节标题总是出现在行的开始，所以设计一种方法只查找标题而不查找交叉引用可能很有用。

定位点提供该能力。定位点使您能够将正则表达式固定到行首或行尾。它们还使您能够创建这样的正则表达式，这些正则表达式出现在一个单词内、在一个单词的开头或者一个单词的结尾。下表包含正则表达式定位点以及它们的含义的列表：

字符	说明
^	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。
\b	匹配一个字边界，即字与空格间的位置。
\B	非字边界匹配。

不能将限定符与定位点一起使用。由于在紧靠换行或者字边界的前面或后面不能有一个以上位置，因此不允许诸如 ^* 之类的表达式。

若要匹配一行文本开始处的文本，请在正则表达式的开始使用 ^ 字符。不要将 ^ 的这种用法与中括号表达式内的用法混淆。

若要匹配一行文本的结束处的文本，请在正则表达式的结束处使用 $ 字符。

若要在搜索章节标题时使用定位点，下面的正则表达式匹配一个章节标题，该标题只包含两个尾随数字，并且出现在行首：

```
/^Chapter [1-9][0-9]{0,1}/
```

真正的章节标题不仅出现行的开始处，而且它还是该行中仅有的文本。它即出现在行首又出现在同一行的结尾。下面的表达式能确保指定的匹配只匹配章节而不匹配交叉引用。通过创建只匹配一行文本的开始和结尾的正则表达式，就可做到这一点。

```
/^Chapter [1-9][0-9]{0,1}$/
```

匹配字边界稍有不同，但向正则表达式添加了很重要的能力。字边界是单词和空格之间的位置。非字边界是任何其他位置。下面的表达式匹配单词 *Chapter* 的开头三个字符，因为这三个字符出现字边界后面：

```
/\bCha/
```

\b 字符的位置是非常重要的。如果它位于要匹配的字符串的开始，它在单词的开始处查找匹配项。如果它位于字符串的结尾，它在单词的结尾处查找匹配项。例如，下面的表达式匹配单词 Chapter 中的字符串 *ter*，因为它出现在字边界的前面：

```
/ter\b/
```

下面的表达式匹配 Chapter 中的字符串 *apt*，但不匹配 aptitude 中的字符串 *apt*：

```
/\Bapt/
```

字符串 *apt* 出现在单词 *Chapter* 中的非字边界处，但出现在单词 *aptitude* 中的字边界处。对于 \B 非字边界运算符，位置并不重要，因为匹配不关心究竟是单词的开头还是结尾。

替换和分组

替换使用 | 字符来允许在两个或多个替换选项之间进行选择。例如，可以扩展章节标题正则表达式，以返回比章标题范围更广的匹配项。但是，这并不象您可能认为的那样简单。替换匹配 | 字符两边的尽可能最大的表达式。您可能认为，下面的表达式匹配出现在行首和行尾、后面跟一个或两个数字的 Chapter 或 Section：

```
/^Chapter|Section [1-9][0-9]{0,1}$/
```

很遗憾，上面的正则表达式要么匹配行首的单词 *Chapter*，要么匹配行尾的单词 *Section* 及跟在其后的任何数字。如果输入字符串是 Chapter 22，那么上面的表达式只匹配单词 *Chapter*。如果输入字符串是 Section 22，那么该表达式匹配 Section 22。

若要使正则表达式更易于控制，可以使用括号来限制替换的范围，即，确保它只应用于两个单词 *Chapter* 和 *Section*。但是，括号也用于创建子表达式，并可能捕获它们以供以后使用，这一点在有关反向引用的那一节讲述。通过在上面的正则表达式的适当位置添加括号，就可以使该正则表达式匹配 Chapter 1 或 Section 3。

下面的正则表达式使用括号来组合 Chapter 和 Section，以便表达式正确地起作用：

```
/^(Chapter|Section) [1-9][0-9]{0,1}$/
```

虽然这些表达式正确发挥作用，但 Chapter| Section 两边的括号还会使得两个匹配单词中的任何一个被捕获以供将来使用。由于在上面的表达式中只有一组括号，因此，只有一个被捕获的“子匹配项”。可以通过使用 **RegExp** 对象的 **$1-$9** 属性来引用此子匹配项。

在上面的示例中，您只需要使用括号来组合单词 *Chapter* 和 *Section* 之间的选择。若要防止匹配被保存以备将来使用，请在括号内正则表达式模式之前放置 ?:。下面的修改提供相同的能力而不保存子匹配项：

```
/^(?:Chapter|Section) [1-9][0-9]{0,1}$/
```

除 ?: 元字符外，两个其他非捕获元字符创建被称为“预测先行”匹配的某些内容。正向预测先行使用 ?= 指定，它匹配处于括号中匹配正则表达式模式的起始点的搜索字符串。反向预测先行使用 ?! 指定，它匹配处于与正则表达式模式不匹配的字符串的起始点的搜索字符串。

例如，假设您有一个文档，该文档包含指向 Windows 3.1、Windows 95、Windows 98 和 Windows NT 的引用。再进一步假设，您需要更新该文档，将指向 Windows 95、Windows 98 和 Windows NT 的所有引用更改为 Windows 2000。下面的正则表达式（这是一个正向预测先行的示例）匹配 Windows 95、Windows 98 和 Windows NT：

```
/Windows(?=95 |98 |NT )/
```

找到一处匹配后，紧接着就在匹配的文本（不包括预测先行中的字符）之后搜索下一处匹配。例如，如果上面的表达式匹配 Windows 98，将在 Windows 之后而不是在 98 之后继续搜索。

反向引用

正则表达式的最重要功能之一是存储匹配的模式的一部分以供以后重新使用的能力。您可能想起，若在正则表达式模式或模式的一部分两侧加上括号，就会导致表达式的一部分被存储到临时缓冲区中。可以通过使用非捕获元字符 ?:、?= 或 ?! 来重写捕获。

每个捕获的子匹配项按照它们在正则表达式模式中从左到右出现的顺序存储。缓冲区编号从 1 开始，最多可存储 99 个捕获的子表达式。可以使用 *\n* 来访问每个缓冲区，其中 *n* 是标识特定缓冲区的一位或两位十进制数字。

反向引用的最简单的、最有用的应用之一，是提供查找文本中两个相同的相邻单词的匹配项的能力。以下面的句子为例：

```
Is is the cost of of gasoline going up up?
```

上面的句子很显然有多个重复的单词。如果能设计一种方法定位该句子，而不必查找每个单词的重复出现，那该有多好。下面的正则表达式使用单个子表达式来实现这一点：

```
/\b([a-z]+) \1\b/gi
```

捕获的表达式，正如 [a-z]+ 指定的，包括一个或多个字母。正则表达式的第二部分是对以前捕获的子匹配项的引用，即，单词的第二个匹配项正好由括号表达式匹配。\1 指定第一个子匹配项。字边界元字符确保只检测整个单词。否则，诸如“is issued”或“this is”之类的词组将不能正确地被此表达式识别。

正则表达式后面的全局标记 (g) 指示，将该表达式应用到输入字符串中能够查找到的尽可能多的匹配。表达式的结尾处的不区分大小写 (i) 标记指定不区分大小写。多行标记指定换行符的两边可能出现潜在的匹配。

使用上面的正则表达式，下面的代码可以使用子匹配项信息，将文本字符串中的两个连续相同的单词的匹配项替换为同一单词的单个匹配项：

```
var ss = "Is is the cost of of gasoline going up up?.\n";
var re = /\b([a-z]+) \1\b/gim;       //Create regular expression pattern.
var rv = ss.replace(re,"$1");   //Replace two occurrences with one.
```

在 **replace** 方法内使用 **$1** 引用第一个保存的子匹配项。如果您有多个子匹配项，您将通过使用 **$2**、**$3** 等依次引用它们。

反向引用还可以将通用资源指示符 (URI) 分解为其组件。假定您想将下面的 URI 分解为协议（ftp、http 等等）、域地址和页/路径：

```
http://msdn.microsoft.com:80/scripting/default.htm
```

下面的正则表达式提供该功能：

```
/(\w+):\/\/([^/:]+)(:\d*)?([^# ]*)/
```

第一个括号子表达式捕获 Web 地址的协议部分。该子表达式匹配在冒号和两个正斜杠前面的任何单词。第二个括号子表达式捕获地址的域地址部分。该子表达式匹配不包括 / 或 : 字符的任何字符序列。第三个括号子表达式捕获端口号（如果指定了的话）。该子表达式匹配冒号后面的零个或多个数字。最后，第四个括号子表达式捕获 Web 地址指定的路径和/或页信息。该子表达式匹配 # 或空格字符之外的一个或多个字符。

将正则表达式应用到上面的 URI，各子匹配项包含下面的内容：

- **RegExp.$1** 包含"http"
- **RegExp.$2** 包含"msdn.microsoft.com"
- **RegExp.$3** 包含":80"
- **RegExp.$4** 包含"/scripting/default.htm"