

- [2. 软件设计基本原则\(135\)](#)
- [3. HttpClient, 使用 C#操作 Web\(39\)](#)
- [4. 用正则表达式找出不包含连续字符串 abc 的单词\(31\)](#)
- [5. 使用 LumiSoft.Net.POP3.Client 接收邮件\(30\)](#)

正则表达式 30 分钟入门教程

来园子之前写的一篇[正则表达式教程](#)，部分翻译自 codeproject 的 [The 30 Minute Regex Tutorial](#)。

由于评论里有过长的 URL,所以本页排版比较混乱,推荐你到原处查看,看完了如果有问题,再到这里来提出。

一些要说的话：

- 如果你没有正则表达式的基础，请跟着教程“一步步来”。请不要大概地扫两眼就说看不懂——以这种态度我写成什么样你也看不懂。当我告诉你这是“30 分钟入门教程”时，请不要试图在 30 秒内入门。

事实是，我身边有个才接触电脑，对操作都不是很熟练的人通过自己学习这篇教程，最后都能在文章采集系统中使用正则表达式完成任务。而且，他写的表达式中，还使用了“零宽断言”等“高级”技术。

所以，如果你能具体地说明你的问题，我很愿意帮助你。但是如果你概括地说看不懂，那不是我的问题。

- 欢迎转载，但请声明作者以及来源。

正则表达式 30 分钟入门教程

版本：v2.31 (2009-4-11) 作者：[deerchao](#) 转载请注明[来源](#)

目录

[跳过目录](#)

- [本文目标](#)
- [如何使用本教程](#)
- [正则表达式到底是什么东西？](#)
- [入门](#)

- [测试正则表达式](#)
- [元字符](#)
- [字符转义](#)
- [重复](#)
- [字符类](#)
- [分枝条件](#)
- [反义](#)
- [分组](#)
- [后向引用](#)
- [零宽断言](#)
- [负向零宽断言](#)
- [注释](#)
- [贪婪与懒惰](#)
- [处理选项](#)
- [平衡组/递归匹配](#)
- [还有什么东西没提到](#)
- [联系作者](#)
- [网上的资源及本文参考文献](#)
- [更新纪录](#)

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——除非你是超人 :)

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？

清除格式 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)** *对其进行匹配的源字符串* 对正则表达式或其中一部分的说明

隐藏边注 本文右边有一些注释，主要是用来提供一些相关信息，或者给没有程序员背景的读者解释一些基本概念，通常可以忽略。

正则表达式到底是什么东西？

字符是计算机软件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**是 0 个或多个字符的序列。**文本**也就是文字，字符串。说某个字符串**匹配**某个正则表达式，通常是指这个字符串里有一部分（或几部分分别）能满足表达式给出的条件。

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)。

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**。

这几乎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，标点符号或者换行来分隔的，但是 `\b` 并不匹配这些单词分隔字符中的任何一个，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使用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。

换行符就是 `'\n'`,ASCII 编码为 10(十六进制 0x0A)的字符。

如果同时使用其它元字符，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` 是个新的元字符，匹配一位数字(0，或 1，或 2，或.....)。`-`不是元字符，只匹配它本身——连字符(或者减号，或者中横线，或者随你怎么称呼它)。

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

其它可用的测试工具：

- [RegexBuddy](#)
- [Javascript 正则表达式在线测试工具](#)

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，所以找一种工具对正则表达式进行测试是很有必要的。

不同的环境下正则表达式的一些细节是不相同的，本教程介绍的是微软 .Net Framework 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`。正则表达式里还有更多的元字符，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等。

对中文/汉字的特殊处理是由 .Net 提供的正则表达式引擎支持的，其它环境下的具体情况请查看相关文档。

下面来看看更多的例子：

`\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：就是不少于一个的连续的`\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 个字符的单词。

表 1.常用的元字符	
代码	说明
<code>.</code>	匹配除换行符以外的任意字符
<code>\w</code>	匹配字母或数字或下划线或汉字
<code>\s</code>	匹配任意的空白符
<code>\d</code>	匹配数字
<code>\b</code>	匹配单词的开始或结束
<code>^</code>	匹配字符串的开始
<code>\$</code>	匹配字符串的结束

正则表达式引擎通常会提供一个“测试指定的字符串是否匹配一个正则表达式”的方法，如 JavaScript 里的 `RegExp.test()` 方法或 .NET 里的 `Regex.IsMatch()` 方法。这里的匹配是指是字符串里有没有符合表达式规则的部分。如果不使用`^`和`$`的话，对于`\d{5,12}`而言，使用这样的方法就只能保证字符串里包含 5 到 12 连续位数字，而不是整个字符串就是 5 到 12 位数字。

元字符`^`（和数字 6 在同一个键位上的符号）和`$`都匹配一个位置，这和`\b`有点类似。`^`匹配你要用来查找的字符串的开头，`$`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：你没办法指定它们，因为它们会被解释成别的意思。这时你就得使用`\`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：`unibetter\.com` 匹配 `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
<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

`^\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，你只需要在方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!)。

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-zA-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：`\(?:\d{2}[-]?\d{8}`。

`"`(`"`和`"`)也是元字符，后面的分组节里会提到，所以在这里需要使用转义。

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})。

分枝条件

不幸的是，刚才那个表达式也能匹配010)12345678或(022-87654321这样的“不正确”的格式。要解决这个问题，我们需要用到**分枝条件**。正则表达式里的**分枝条件**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。你可以试试用分枝条件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：**使用分枝条件时，要注意各个条件的顺序**。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)。原因是匹配分枝条件时，将会从左到右地测试每个条件，如果满足了某个分枝的话，就不会去再管其它的条件了。

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}`)。

IP 地址中每个数字都不能大于 255，大家千万不要被《24》第三季的编剧给忽悠了.....

不幸的是，它也将匹配 `256.300.888.999` 这种不可能存在的 IP 地址。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。

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

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。

呃.....其实,组号分配还不像我刚说得那么简单:

- 分组 **0** 对应整个正则表达式
- 实际上组号分配过程是要从左向右扫描两遍的：第一遍只给未命名组分配，第二遍只给命名组分配——因此所有命名组的组号都大于未命名的组号
- 你可以使用 **(?:exp)** 这样的语法来剥夺一个分组对组号分配的参与权。

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*，或者 *kitty kitty*。这个表达式首先是一个单词，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**)，这个单词会被捕获到编号为 **1** 的分组中，然后是 **1** 个或几个空白符(**\s+**)，最后是**分组 1** 中捕获的内容（也就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.常用分组语法		
分类	代码/语法	说明
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)	这种类型的分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，也不会拥有组号。“我为什么会想要这样做？”——好问题，你觉得为什么呢？

零宽断言

地球人，是不是觉得这些术语名称太复杂，太难记了？我也有同感。知道有这么一种东西就行了，它叫什么，随它去吧！人若无名，便可专心练剑；物若无名，便可随意取舍.....

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(即断言)，因此它们也被称为**零宽断言**。最好还是拿例子来说明吧：

断言用来声明一个应该为真的事实。正则表达式中只有当断言为真时才会继续进行匹配。

(?=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)`，零宽度负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。

请详细分析表达式 `(?<=<(\w+)>).*?(?=<\1>)`，这个表达式最能表现零宽断言的真正用途。

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 `\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 `` 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通过语法 `(?#comment)` 来包含注释。例如：

`2[0-4]\d(?#200-249)|25[0-5](?#250-255)|[01]?[0-9]\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：

```
(?<=      # 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。以这个表达式为例：**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**a.*?**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**（第一到第三个字符）和 **ab**（第四到第五个字符）。

为什么第一个匹配是 **aab**（第一到第三个字符）而不是 **ab**（第二到第三个字符）？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	
代码 / 语法	说明
*?	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

在 C# 中，你可以使用 `Regex(String, RegexOptions)` 构造函数来设置正则表达式的处理选项。如：`Regex regex = new Regex(@"\ba\w{6}\b", RegexOptions.IgnoreCase);`

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 <code>^</code> 和 <code>\$</code> 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， <code>\$</code> 的精确含意是：匹配 <code>\n</code> 之前的位置以及字符串结束前的位置。)
Singleline(单行模式)	更改 <code>.</code> 的含义，使它与每一个字符匹配（包括换行符 <code>\n</code> ）。
IgnorePatternWhitespace(忽略空白)	忽略表达式中的非转义空白并启用由 <code>#</code> 标记的注释。
ExplicitCapture(显式捕获)	仅捕获已被显式命名的组。

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

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(Stack)**
- `(?'-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（或者你自称程序员但是不知道堆栈是什么东西），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，就在压入一个"`Open`"，每碰到一个右括号，就弹出一个，到了最后就看看堆栈是否为空——如果不为空那就证明左括号比右括号多，那匹配就应该失败。正则表达式引擎会进行回溯(放弃最前面或最后面的一些字符)，尽量使整个表达式得到匹配。

```
<                                #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

<div>标签:

```
<div[ ^> ]* > [ ^<> ]* ( ( ( ? 'Open' <div[ ^> ]* > ) [ ^<> ]* ) + ( ( ? '-'Open' </div> v> ) [ ^<> ]* ) + ) * ( ? (Open) (?!)) </div> .
```

还有什么东西没提到

上边已经描述了构造正则表达式的大量元素, 但是还有很多没有提到的东西。下面是一些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 .net 下正则表达式详细的文档。

这里的介绍很简略, 如果你需要更详细的信息, 而又没有在电脑上安装 MSDN Library, 可以查看[关于正则表达式语言元素的 MSDN 在线文档](#)。

表 7. 尚未详细讨论的语法

代码/语法	说明
\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

表 7.尚未详细讨论的语法	
代码/语法	说明
(?<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

联系作者

好吧,我承认,我骗了你,读到这里你肯定花了不少 30 分钟.相信我,这是我的错,而不是因为你太笨.我之所以说"30 分钟",是为了让你有信心,有耐心继续下去.既然你看到了这里,那证明我的阴谋成功了.被忽悠的感觉很爽吧?

要投诉我,或者觉得我其实可以忽悠得更高明,或者有任何其它问题,欢迎来我的[博客](#)让我知道.

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 \(Second Edition\)](#)

更新纪录

- 2006-3-27 第一版
- 2006-10-12 第二版
 - 修正了几个细节上的错误和不准确的地方
 - 增加了对处理中文时的一些说明
 - 更改了几个术语的翻译（采用了 MSDN 的翻译方式）
 - 增加了平衡组的介绍
 - 放弃了对 The Regulator 的介绍，改用 Regex Tester
- 2007-3-12 V2.1

- 修正了几个小的错误
- 增加了对处理选项(RegexOptions)的介绍
- 2007-5-28 V2.2
 - 重新组织了对零宽断言的介绍
 - 删除了几个不太合适的示例，添加了几个实用的示例
 - 其它一些微小的更改
- 2007-8-3 V2.21
 - 修改了几处文字错误
 - 修改/添加了对\$, \b 的精确说明
 - 承认了作者是个骗子
 - 给 RegexTester 添加了 Singleline 选项的相关功能
- 2008-4-13 v2.3
 - 调整了部分章节的次序
 - 修改了页面布局，删除了专门的参考节
 - 针对读者的反馈，调整了部分内容
- 2009-4-11 v2.31
 - 修改了几处文字错误
 - 添加了一些注释说明
 - 调整了一些措词

标签: [正则表达式](#), [Regex](#), [教程](#)

绿色通道: [好文要顶](#) [关注我](#) [收藏该文](#) [与我联系](#)

[deerchao](#)

关注 - 5

粉丝 - 64

[+加关注](#)

24

0

(请您对文章做出评价)

» 博主后一篇: [正在改写 C#版的 ICTCLAS](#)

posted on 2006-08-24 22:04 [deerchao](#) 阅读(79741) 评论(790) 编辑 收藏

评论

评论共 8 页: [上一页](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#)

#684 楼 2009-12-22 23:54 呀 你真能忽悠[未注册用户]

呵呵 本教程很不错 赞一个 [回复](#) [引用](#)

#685 楼 2009-12-22 23:55 呀 你真能忽悠[未注册用户]

我把你的这些转载到我的主页，不介意吧，便于日后的学习，呵呵 [回复](#)
[引用](#)

#686 楼 2009-12-24 16:56 baiyuxiong[未注册用户]

相当不错。

谢谢。

顺便帮你点了几个广告。 [回复](#) [引用](#)

#687 楼 2009-12-30 16:51 厨子

很早就看过了,今天才看到这个链接.来此仅表达一下对作者的感激之情.占了个位置.呵呵~~ [回复](#) [引用](#) [查看](#)

#688 楼 2010-01-19 13:30 airy

2010 第一个来的 [回复](#) [引用](#) [查看](#)

#689 楼 2010-01-19 13:30 airy

楼主这篇，成了我的手册。 [回复](#) [引用](#) [查看](#)

#690 楼 2010-02-23 15:51 next~[未注册用户]

thx~ [回复](#) [引用](#)

#691 楼 2010-02-23 17:43 happyhappy[未注册用户]

我想问下,我有这样一些字符串:

Q0032|550.2.530|902.01.01.02.01.01.02,902.01.01.04.01,902.01.0
9.02|11805837

我想去掉第 2 个|后面的所有字符串 该怎么写啊?

谢谢 [回复](#) [引用](#)

#692 楼 2010-02-25 09:26 小超。

哈哈，写的真好，佩服中.... [回复](#) [引用](#) [查看](#)

#693 楼 2010-03-16 01:08 Ryusuke

谢谢分享!!! [回复](#) [引用](#) [查看](#)

#694 楼 2010-04-10 12:10 132811

有一点疑问:

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

其中的“.”只是匹配一个字符吗？还是表示其它后匹配的所有字符都不能有回车？ [回复](#) [引用](#) [查看](#)

#695 楼 2010-04-12 01:20 **amwukddx**

零宽度正回顾后发断言在 JavaScript 中表示，貌似有点问题：
在 js 里用(?<![":])\b[a-zA-Z]{1,2}\d+\b(?![":])来匹配 A1:B2 时，浏览器提示"正则表达式语法错误",当我去掉左尖括号'<'时，就没有问题，当然这种校验已经没有意义了，忘楼能帮忙看看 [回复](#) [引用](#) [查看](#)

#696 楼 2010-04-12 01:24 **amwukddx**

@132811

要连着看 .*意为着所有不带回车的字符（串）（没有字符也可以） [回复](#) [引用](#) [查看](#)

#697 楼 2010-04-12 01:27 **amwukddx**

@happyhappy

比较笨的办法：[^\|]*\[^\|]*\| [回复](#) [引用](#) [查看](#)

#698 楼 2010-04-12 01:39 **amwukddx**

@guozhangliang

(?:\<img.*>)\w* [回复](#) [引用](#) [查看](#)

#699 楼 2010-04-14 17:46 **jianshaohui**

楼主，我非常感谢你。。我看书 之后 再看你这个，理解得更透彻 [回复](#)
[引用](#) [查看](#)

#700 楼 2010-05-19 17:35 **chunchill**

我转载一下，写的太好了，做个笔记：） [回复](#) [引用](#) [查看](#)

#701 楼 2010-06-10 16:34 **慕唯**

@smoketest

貌似

(\b(2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?) 也可以。 [回复](#) [引用](#) [查看](#)

#702 楼 2010-06-10 16:35 **慕唯**

[引用](#) happyhappy:

我想问下,我有这样一些字符串:

Q0032|550.2.530|902.01.01.02.01.01.02,902.01.01.04.01,902.01.09.02|
11805837

我想去掉第 2 个|后面的所有字符串 该怎么写啊?

谢谢

用 substring 啦。 [回复](#) [引用](#) [查看](#)

#703 楼 2010-06-17 16:41 我哈哈

万分感谢 被你忽悠的感觉老好了 呵呵 [回复](#) [引用](#) [查看](#)

#704 楼 2010-06-21 16:41 kisstefani

写的不错 我入门了。 [回复](#) [引用](#) [查看](#)

#705 楼 2010-06-21 17:01 kisstefani

引用慕唯:

引用 happyhappy:

我想问下,我有这样一些字符串:

Q0032|550.2.530|902.01.01.02.01.01.02,902.01.01.04.01,902.01.09.02|
11805837

我想去掉第 2 个|后面的所有字符串 该怎么写啊?

谢谢

用 substring 啦。

(?<![^\|])[\^|]*\|[\^|]* 求高手。 [回复](#) [引用](#) [查看](#)

#706 楼 2010-06-23 11:03 小布丁~

太佩服了~~好崇拜哦~~ [回复](#) [引用](#) [查看](#)

#707 楼 2010-06-26 04:04 gaoyongbo026

看了两个小时 [回复](#) [引用](#) [查看](#)

#708 楼 2010-07-01 09:04 jamesNiu

请问楼主,如果匹配某个 img 标签 不包含某个连续字符串

, 我需要确保里面不含 www.使用

^<img((?!www)[^>])*>\$无法成功,请楼主指教 [回复](#) [引用](#) [查看](#)

#709 楼 2010-07-07 21:12 gbwin

老师,这样的正则该怎么处理呢?

得出不在<...>中的 A,即得出第 1 个和最后 2 个 A

A

中华人民共和国

<

A

.....

A

A

'A

>

中华人民共和国

A、A [回复](#) [引用](#) [查看](#)

#710 楼 2010-08-09 14:19 guobosheng

非常感谢博主的文章！我发现这里说的正则表达式不是全部可以在 notepad++ 中使用，这是因为 notepad++ 不支持它吗？还是另的原因？ [回](#)
[复](#) [引用](#) [查看](#)

#711 楼 2010-08-10 15:16 MichaelWu

总算看到最后了，谢谢 lz [回复](#) [引用](#) [查看](#)

#712 楼 2010-09-02 17:21 meetyouever

LZ 你个狗日的，把老子骗来了，显然 30 分钟，我只能看个大概！！

不过还是谢谢你的好文章 [回复](#) [引用](#) [查看](#)

#713 楼 2010-09-15 14:00 answers

好文章！特申请号上来顶。。 [回复](#) [引用](#) [查看](#)

#714 楼 2010-09-21 15:34 早念

虽然很多东西没看懂，但是还是学到了不少东西！谢谢嘎！！ [回复](#) [引](#)
[用](#) [查看](#)

#715 楼 2010-10-19 15:17 一周寂寞七天

灰常不错....还没看完... [回复](#) [引用](#) [查看](#)

#716 楼 2010-10-21 17:26 一周寂寞七天

今天看完了...过一忽悠一圈..... [回复](#) [引用](#) [查看](#)

#717 楼 2010-10-21 20:05 Gnoweb

@amwukddx

JavaScript 暂时不支持 (?<=、(?<! 零宽度正负回顾后发断言 [回复](#) [引用](#)
[查看](#)

#718 楼 2010-10-28 16:27 西湖龙井

看了好几天才看完 还有很多不懂 真的被骗了 [回复](#) [引用](#) [查看](#)

#719 楼 2010-10-29 19:01 hulu

"零宽断言"部分:

原文:

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。

似乎应该是这样:

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 1,234,567,890 进行查找时结果是 234 567 890。

[回复](#) [引用](#) [查看](#)

#720 楼 2010-10-29 22:33 hulu

@流浪者 2

当然有问题 又没有限制 ip 地址的前后是数字还是字母还是空格 所以仍然会从非法的 ip 中匹配出合法的 ip 来 [回复](#) [引用](#) [查看](#)

#721 楼 2010-10-29 22:37 hulu

引用 hi: (?=exp) 匹配 exp 前面的位置
(?!exp) 匹配 exp 后面的位置
貌似这个反了

没有反 第一个是 exp 为后缀 第二个是 exp 为前缀 [回复](#) [引用](#) [查看](#)

#722 楼 2010-10-30 00:15 病学者

deerchao 您好, 你的教程让我能学会了点简单的“正则表达式”, 但是我遇到了个比较难的问题。如下:

<a

href="javascript:page.common.openPlayer('mjwadycbogr5uecbogw')"

我想获取: mjwadycbogr5uecbog

不知能实现吗?

谢谢! [回复](#) [引用](#) [查看](#)

#723 楼 2010-11-01 16:22 dawnzhu

专门注册了一个号来告诉你,我为了表达对你提供的信息的感谢,我点你的广告两次. [回复](#) [引用](#) [查看](#)

#724 楼 2010-11-02 09:43 **马老虎**

点击广告!! [回复](#) [引用](#) [查看](#)

#725 楼 2010-11-03 04:46 **mybluedog24**

我也是专门注册个号来感谢!!~

我是在其他网站看到的~看到最后直接点击进来感谢!!~~~

[回复](#) [引用](#) [查看](#)

#726 楼 2010-11-19 12:54 **June Zhang**

呵呵呵 刚被楼主的 30 分钟“忽悠了”,我以前不怎么了解正则表达式,现在看了 了解了一点 [回复](#) [引用](#) [查看](#)

#727 楼 2010-11-22 14:17 **nancy71**

hi 楼主,你写的真是不错,请问我可以转载这篇入门教程吗?谢谢! [回](#)
[复](#) [引用](#) [查看](#)

#728 楼[楼主] 2010-11-22 14:20 **deerchao**

[引用](#) nancy71: hi 楼主,你写的真是不错,请问我可以转载这篇入门教程吗?谢谢!
在保留原出处链接的情况下,可以. [回复](#) [引用](#) [查看](#)

#729 楼 2010-11-30 11:52 **hl3292**

太好了 [回复](#) [引用](#) [查看](#)

#730 楼 2010-12-02 16:00 **帽帽-阿杰**

730 楼的谢谢楼主 [回复](#) [引用](#) [查看](#)

#731 楼 2010-12-05 17:06 **chasefornone**

博主太威武了,获益匪浅啊,继续来骗我吧... [回复](#) [引用](#) [查看](#)

#732 楼 2010-12-07 09:51 **Stephen.Huang**

楼主, \w 表示的是[a-zA-Z_]吧,我试过汉字不行。是不是不同的 flavor
不一样? [回复](#) [引用](#) [查看](#)

#733 楼[楼主] 2010-12-07 17:50 **deerchao**

@Stephen.Huang

不同的正则表达式引擎的实现不一样.

.Net BCL 里的是支持的. [回复](#) [引用](#) [查看](#)

#734 楼 2010-12-08 08:57 **Stephen.Huang**

@deerchao

试了一下。果然不同的引擎不一样。。。 [回复](#) [引用](#) [查看](#)

#735 楼 2010-12-08 11:49 ddj2688

请问各位大虾:在"中华人民共和国.....和人民..."这个文本中要查找到"人民共和"与"和人民"这样的文字,正则表达怎么写? [回复](#) [引用](#) [查看](#)

#736 楼[楼主] 2010-12-08 11:53 deerchao

@ddj2688

人民共和|和人民 [回复](#) [引用](#) [查看](#)

#737 楼 2010-12-08 12:15 ddj2688

@deerchao

非常感谢!有没有更简单的? [回复](#) [引用](#) [查看](#)

#738 楼 2010-12-08 14:45 ddj2688

我没说清楚,应该是文本中要查找到"人...和"与"和...人"这样的文字,正则表达怎么写? [回复](#) [引用](#) [查看](#)

#739 楼 2010-12-11 09:43 ernesto

对博主大感谢啊!以后看到正则表达式不用头疼了。

另外发现文中介绍分组时举的匹配 IP 地址的数字部分的例子,可以也匹配 001、01 这样的数字。有的情况下需要排除这样的匹配,我的可以排除这样的数字的版本是(`[1-9]?\\d|1\\d\\d|2[0-4]\\d|25[0-5]`)。如果是从一个包含其它内容的字符串里匹配 IP 地址,需要用到零宽断言,
(`?<![\\d.]`)(`([1-9]?\\d|1\\d\\d|2[0-4]\\d|25[0-5])[.]`){3}(`[1-9]?\\d|1\\d\\d|2[0-4]\\d|25[0-5])(?![\\d.]`)。

[回复](#) [引用](#) [查看](#)

#740 楼 2010-12-11 13:14 wtq

不错,被忽悠得有价值 [回复](#) [引用](#) [查看](#)

#741 楼 2010-12-20 15:50 qq56973704

非常有用,谢谢了 [回复](#) [引用](#) [查看](#)

#742 楼 2010-12-23 22:09 Jack Sun

真的十分之得推荐,谢谢作者,虽然看完不能自己写复杂的表达式,但可以看的懂 呵呵~ [回复](#) [引用](#) [查看](#)

#743 楼 2010-12-31 10:32 找不到

哈哈 我花了 10 秒钟拉到最下面看到了你的忽悠 主要是么时间 有时间再来好好学习学习 谢谢您的分享 [回复](#) [引用](#) [查看](#)

#744 楼 2011-01-07 10:50 浮云的年代

想楼主伟大的姿态致敬。。。由衷的感谢你踏实的诚恳的帮助大家。 [回](#)

[复](#) [引用](#) [查看](#)

#745 楼 2011-01-07 23:06 changbalao

非常感谢楼主写的精彩内容。

不过有个问题想请教楼主。如何获取某个分组的重复的次数。

比如说我的正则表达式是 $(ab)^*$ ，语句是 ababababab，在这个语句中 ab 这个分组出现了 5 次，这个 5 次怎么获取呢。

好像在 Java 里面不支持这样的参数获取。 [回复](#) [引用](#) [查看](#)

#746 楼[楼主] 2011-01-08 08:36 deerchao

@changbalao

.Net 中是每个匹配的分组(Group)里有几个 Captures, 每个对应分组一次匹配。

Java 我不清楚, 可能类似吧。 [回复](#) [引用](#) [查看](#)

#747 楼 2011-01-09 22:14 .net 小鸟

谢谢楼主，一直想学正则，一直怕这个东西麻烦，现在看了楼主的教程，突然间许多关节自动打通，再次感谢！ [回复](#) [引用](#) [查看](#)

#748 楼 2011-01-14 10:37 求识

问个问题，正则表达式是不是根据语言的不同有很多版本？比如 C# 一个版本，JAVA 一个，数据库一个？他们之间是根本不同呢还是只是些许的细节上不一样？他们之间有什么共通的语法或者其他的吗？楼主大神的这个文章介绍的是.NET 版本的吗？谢谢各位，谢谢楼主。 [回复](#) [引用](#) [查看](#)

#749 楼 2011-01-26 14:36 飞车兔

辛苦了。偶是来看看还有什么东东能接着被忽悠。 [回复](#) [引用](#) [查看](#)

#750 楼 2011-02-22 11:29 chaochao6078

楼主看了你的博客让我受益匪浅啊

但是我按照你上面的理解

下面这句正则表达式我百思不得其解啊

$^(?!0)(?:([0-9])(?!.*?\1))+\$$

麻烦能详细解释一下吗，不甚感激！ [回复](#) [引用](#) [查看](#)

#751 楼 2011-03-02 09:45 krator

LZ，太感谢你了，分组那部分终于明白了。不过我发现一点小问题
负向零宽断言那一段

蓝色的 (`<?(\\w+)>`) 是不是搞错了，应该是 (`?<=<(\\w+)>`) 吧？ [回复](#)

[引用](#) [查看](#)

#752 楼 2011-03-04 11:55 为了谁

写得太好了，我一点不懂到现在有点懂了。 [回复](#) [引用](#) [查看](#)

#753 楼 2011-03-05 11:04 run_with_horse

好文章啊！顶一个。。。另外个人感觉有一个地方 LZ 可能不小心，打错了！
负向零宽断言

第五段 `<?(\\w+)>` 问号应该去掉的吧！ [回复](#) [引用](#) [查看](#)

#754 楼[楼主] 2011-03-05 11:12 deerchao

@run_with_horse

@krator

谢谢，确实是笔误.现在已经改过来了。 [回复](#) [引用](#) [查看](#)

#755 楼 2011-03-12 04:07 Admonis

谢谢博主!!!!!! 正则终于入门了, (谢谢){10000}+! [回复](#) [引用](#) [查看](#)

#756 楼 2011-03-22 15:29 今月古人

很好，很强大，受教了，感谢 lz 分享这么好的文章 [回复](#) [引用](#) [查看](#)

#757 楼 2011-03-28 17:24 longgege

`((2([0-4]\\d|5[0-5])|1\\d{2}|\\d{2}|\\d)\\.){3}(2([0-4]\\d|5[0-5])|1\\d{2}|\\d{2}|\\d)`

ip 地址那个用这个应该更好吧，呵呵。。

我刚好看到，呵呵。。 [回复](#) [引用](#) [查看](#)

#758 楼 2011-03-28 18:02 longgege

@changbalao

那个正则表达式好像不支持数学计算吧，我用 C 语言写了个，你看有用没？呵呵。。

```
#include<stdio.h>
```

```
#include<string.h>
```

```
int main()
```

```
{
```

```
char *p1, *p2;
```

```

int i;
char str1[2000], str2[2000];
int sum = 0;

printf("请输入源字符串:\n");
scanf("%s", str1);
printf("请输入子字符串:\n");
scanf("%s", str2);
p1 = str1;
p2 = str2;
while(strstr(p1,p2) && *p1) //判断在"遍历"源串之前有无出现子串
{
    sum++; //有的话，次数加 1
    p1=strstr(p1,p2)+1; //取出现位置首地址加 1 形成新的串，再判断
}
printf("第子串在第源串中出现的次数为%d!\n", sum);

return 0;
}

```

[回复](#) [引用](#) [查看](#)

#759 楼 2011-04-09 16:05 老男孩玩编程

四个字：“豁然开朗”，不用再看厚重的教程了，因为多数人接触这东西并不多，经常用的明白了就行。 [回复](#) [引用](#) [查看](#)

#760 楼 2011-04-15 08:35 yakczh

```

$patten=qr(<div[^>]*>[^<>]*(((?'Open'<div[^>]*>)[^<>]*)+((
?'-Open'</div>)[^<>]*)+)*(?(Open)(?!))</div>);

```

```

print $patten;

```

提示报错

```
Sequence (?'-...) not recognized in regex; marked by <-- HERE in
m/<div[^>]*>[^<>]*(((?'Open'<div[^>]*>)[^<>]*)+(((?'- <--
HERE Open'</div>)[^<>]*)+)*(?(Open)(?!))</div>/
```

perl 版本是 v5.10.0 [回复](#) [引用](#) [查看](#)

#761 楼 2011-04-26 20:26 [波波.](#)

楼主真给力 [回复](#) [引用](#) [查看](#)

#762 楼 2011-04-26 20:49 [john23.net](#)

mark [回复](#) [引用](#) [查看](#)

#763 楼 2011-05-14 23:38 [void241](#)

特地来感谢博主的！

感谢博主抽出宝贵的时间写出了这么个好东西，让广大 coder 收益 [回复](#) [引用](#) [查看](#)

#764 楼 2011-05-18 15:25 [p2227](#)

我觉得这篇文章最好注明这里说的是 C# 里面的正则，不同语言下的正则
有差别呢，比如 javascript 的正则的不支持(?<=exp) [回复](#) [引用](#) [查看](#)

#765 楼 2011-05-31 09:07 [Hello! Linux](#) 博客

我还说只有天才才可以 30 分钟入门呢，哈哈。

刚刚读完，感谢 LZ 写了如此好的教程。 [回复](#) [引用](#) [查看](#)

#766 楼 2011-05-31 09:21 [Hello! Linux](#) 博客

@p2227

感觉可能 perl 支持的更好，使用 grep -P 参数试试好了 [回复](#) [引用](#) [查看](#)

#767 楼 2011-07-06 13:45 [忽而今夏](#)

被忽悠了，不过感觉很爽....呵呵 [回复](#) [引用](#) [查看](#)

#768 楼 2011-07-10 18:07 [今天我不乖](#)

博主，依然是

```
((2[0-4]\d|25[0-5]|([01]?[d\d?))\.){3}(2[0-4]\d|25[0-5]|([01]?[d\d?))
```

这里的问题，这个 IP 在遇到

000.000.000.000

0.0.0.0

025.025.025.025

这类的地址的时候，是依然可以匹配成功的~~看来，要想真正严格的匹配一个东西，还真是不容易呢，呵呵~~

依然感谢博主的分享，从里面学到了很多东西~~ [回复](#) [引用](#) [查看](#)

#769 楼 2011-07-12 22:29 **z-lll**

很棒、、、可是我怎么在程序里使用啊？是不是稍微提点，让咱一步到位？

[回复](#) [引用](#) [查看](#)

#770 楼 2011-07-14 18:11 **51qqhe**

很大有实用 [回复](#) [引用](#) [查看](#)

#771 楼 2011-07-22 17:23 **cgme**

UP 灰常实用 园子里就缺这样的好教程 朴实没有那么多花哨的外表 但

确实实用 [回复](#) [引用](#) [查看](#)

#772 楼 2011-07-25 19:14 **hxxhxm**

很实用，看了正则表达式手册，再看这个那是非常的实用！看了这个才会应

用 [回复](#) [引用](#) [查看](#)

#773 楼 2011-07-26 18:15 **我尹熊**

顶一下

标示感谢 好东东哦 [回复](#) [引用](#) [查看](#)

#774 楼 2011-08-02 15:35 **gulunmu**

作者您好！

感谢你带来这么优秀的教程，我跟着里面的例题做的时候有一个问题，就是在写ip 地址的表达式的时候

```
var ip = "sadasdsad11.188.255.109asdsadsadsad";
```

```
var reip
```

```
=/(?:(?:2[0-4]\d|25[0-5]||[01]?[0-9]\d?)\.){3}(?:2[0-4]\d|25[0-5]||[01]?[0-9]\d?)/;
```

```
var rip = ip.match(reip);
```

```
document.write('IP: ' + rip + '<br/>');
```

得到的结果是：IP:11.188.255.109

而

```
var ip = "sadasdsad11.188.255.109asdsadsadsad";
```

```
var reip
```

```
=/((2[0-4]\d|25[0-5]|([01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|([01]?\d\d?)\d?)/;
```

```
var rip = ip.match(reip);
```

```
document.write('IP: ' + rip + '<br/>');
```

得到的结果是: IP: 11.188.255.109,255.,255,109

我想请问一下: '?:'到底起了什么作用呢? [回复](#) [引用](#) [查看](#)

#775 楼 2011-08-02 17:21 gulunmu

我已经明白了, 正则表达式只是返回匹配结果: true 或者 false, 但不返回匹配结果。

这里我进一步想问的是: 如果获得匹配结果 [回复](#) [引用](#) [查看](#)

#776 楼 2011-08-06 11:26 豆浆油油条

顶个, 通俗易懂!! [回复](#) [引用](#) [查看](#)

#777 楼 2011-08-14 00:04 xiaojun123

谢谢 lz 分享, 辛苦了。。 [回复](#) [引用](#) [查看](#)

#778 楼 2011-08-15 14:39 faithyme

很不错的一个教程 很好 [回复](#) [引用](#) [查看](#)

#779 楼 2011-08-18 12:04 diorlv

非常非常有用啊, 最近正在学习, 博主这边文章使我有更深层次的了解, 谢谢楼主在此花费的时间和精力 [回复](#) [引用](#) [查看](#)

#780 楼 2011-08-21 14:50 chuan0326

你好看完入門教學 有點事請教
我有一網址如下

<http://5278.cc/forumdisplay.php?fid=29&filter=0&orderby=replies&page=86>"

target="_blank"><http://5278.cc/forumdisplay.php?fid=29&filter=0&orderby=replies&page=86>

如

[http://5278.cc/forumdisplay.php?fid=29&filter=0&orderby=replies&page=\[1-86\]](http://5278.cc/forumdisplay.php?fid=29&filter=0&orderby=replies&page=[1-86])

&page=[1-86] []為取得 1-86 的頁數

另一種 會隨跳頁 轉換英文任意數

網頁頁面

第 1 頁

<http://www07.eyny.com/forumdisplay.php?fid=17&filter=type&typeid=2&page=3UWMG3B3>

第 2 頁

<http://www07.eyny.com/forumdisplay.php?fid=17&filter=type&typeid=2&page=2CXK8U1R>

&page=2CXK8U1R 這 2CXK8U1R 會隨跳頁 轉換英文任意數

請教如何取任意數 [回復](#) [引用](#) [查看](#)

#781 樓 2011-08-22 18:07 chuan0326

感謝答覆

再次請教

$p=\backslash d+\$;id=\backslash d+\$$

這在正則表達式代表甚麼意思

這行是列表第一頁

<http://www05.eyny.com/forumdisplay.php?fid=17&page=DLBRAFTX>

第一頁

<http://www05.eyny.com/forumdisplay.php?fid=17&page=31K8IHS5>

第二頁

假如這個網頁有 100 頁，我要如何設定

fid=17&&page[1-100]=? 後面英文字均不同，這樣對嗎

上次提到

另一種 會隨跳頁 轉換英文任意數

網頁頁面

第 1 頁

<http://www07.eyny.com/forumdisplay.php?fid=17&filter=type&typeid=2&page=3UWMG3B3>

target="_blank"><http://www07.eyny.com/forumdisplay.php?fid=17&filter=type&typeid=2&page=3UWMG3B3>

第 2 頁

<http://www07.eyny.com/forumdisplay.php?fid=17&filter=type&typeid=2&page=2CXK8U1R>

target="_blank"><http://www07.eyny.com/forumdisplay.php?fid=17&filter=type&typeid=2&page=2CXK8U1R>

正則表達式為

<http://www07.eyny.com/forumdisplay.php?fid=17&filter=type&typeid=2&page=\w+>

正則表達式為何為=\w+

<http://5278.cc/forumdisplay.php?fid=29&filter=0&orderby=replies&page=86>

target="_blank"><http://5278.cc/forumdisplay.php?fid=29&filter=0&orderby=replies&page=86>

[http://5278.cc/forumdisplay.php?fid=29&filter=0&orderby=replies&page=\[1-86\]](http://5278.cc/forumdisplay.php?fid=29&filter=0&orderby=replies&page=[1-86])

<http://5278.cc/forumdisplay.php?fid=29&filter=0&orderby=replies&page=\d+>

正則表達式為何為=\d+

兩個為何不同

如果是數字會隨跳頁 轉換數字任意數

<http://www07.eyny.com/forumdisplay.php?fid=17&filter=type&typeid=2&page=1122112>

<http://www07.eyny.com/forumdisplay.php?fid=17&filter=type&typeid=2&page=4562343>

正則表達式又如何設定呢

以上英文任意數 正則表達式為 `\w+`

那數字為任意數呢?

另只取圖片所在頁面，不取讀者回復頁

<http://www05.eyny.com/viewthread.php?tid=5779425&extra=page%3DDLBRFTX>

這是讀者回復頁{沒有圖片}

<http://www05.eyny.com/viewthread.php?tid=5779425&extra=page%3DDLBRFTX&page=2>

要如何設定只取第一頁(有圖片的頁面)，不取回覆頁

以上如何用正則表達式實現

謝謝 感恩 回复 引用 查看

#782 楼 2011-08-22 22:49 **gaohong**

`(?=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

->

`(?=exp)` 匹配后面跟的是 **exp** 的位置

`(?<=exp)` 匹配前面是 **exp** 的位置

这样说明 是不是更好一些, 不易引起误解 回复 引用 查看

#783 楼 2011-08-23 16:48 **卖紫英的 NPC**

我被忽悠了，我以为我已经笨的无可救药了。。。。。

回复 引用 查看

评论共 8 页: [上一页](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#)

注册用户登录后才能发表评论, 请 [登录](#) 或 [注册](#), [返回博客园首页](#)。

最新 IT 新闻:

- [微软: 搭载 Tango 系统手机很便宜](#)
 - [苹果令所有厂商绝望 浮躁硅谷不相信坚忍图成](#)
 - [传统手机短信末路? 替代服务众多 费用更低廉](#)
 - [Fedora 16 Alpha 发布](#)
 - [十款短命科技产品 苹果微软谷歌榜上有名](#)
- » [更多新闻...](#)

最新知识库文章:

- [Hack, Everything!](#)
 - [我是如何从煤矿工成为程序员的](#)
 - [能大大提升工作效率和时间效率的 9 个重要习惯](#)
 - [高效的面试方式: 结对编程](#)
 - [如何学习一门新的语言](#)
- » [更多知识库文章...](#)



[China-pub 2011 秋季教材巡展](#)

[China-Pub 计算机绝版图书按需印刷服务](#)

网站导航: [网站首页](#) [知识库](#) [IT 新闻](#) [我的园子](#) [闪存](#) [程序员招聘](#) [博问](#)

Powered by:

[博客园](#)

Copyright © deerchao

[^,] 除了逗号之外的任何字符

`[^,]*` 0 或者多个非逗号字符

`\([^\,]*\)` 将这些非逗号字符标记为`\1`，这样可以在之后的替换模式表达式中引用它

`\([^\,]*\)`，我们必须找到 0 或者多个非逗号字符后面跟着一个逗号，并且非逗号字符那部分要标记出来以备后用。

匹配中文字符的正则表达式： `[\u4e00-\u9fa5]`

评注：匹配中文还真是个头疼的事，有了这个表达式就好办了哦
获取日期正则表达式：`\d{4}[年|\-|\.] \d{\1-\12} [月|\-|\.] \d{\1-\31} 日?`

评注：可用来匹配大多数年月日信息。

匹配双字节字符(包括汉字在内)：`[\x00-\xff]`

评注：可以用来计算字符串的长度（一个双字节字符长度计 2，ASCII 字符计 1）

匹配空白行的正则表达式：`\n\s*\r`

评注：可以用来删除空白行

匹配 HTML 标记的正则表达式：`<(\S*?)[^>]*>.*?</>|<.*? />`

评注：网上流传的版本太糟糕，上面这个也仅仅能匹配部分，对于复杂的嵌套标记依旧无能为力

匹配首尾空白字符的正则表达式：`^\s*|\s*$`

评注：可以用来删除行首行尾的空白字符(包括空格、制表符、换页符等等)，非常有用的表达式

匹配 Email 地址的正则表达式：
`\w+([-+.]\w+)*@\w+([-.\]\w+)*\.\w+([-.\]\w+)*`

评注：表单验证时很实用

匹配网址 URL 的正则表达式：`[a-zA-z]+://[^\s]*`

评注：网上流传的版本功能很有限，上面这个基本可以满足需求

匹配帐号是否合法(字母开头，允许 5-16 字节，允许字母数字下划线)：
`^[a-zA-Z][a-zA-Z0-9_]{4,15}$`

评注：表单验证时很实用

匹配国内电话号码：`\d{4}-\d{7}|\d{3}-\d{8}`

评注：匹配形式如 0511 - 4405222 或 021 - 87888822

匹配腾讯 QQ 号：`[1-9][0-9]\{4,\}`

评注：腾讯 QQ 号从 1000 0 开始

匹配中国邮政编码：`[1-9]\d{5}(?! \d)`

评注：中国邮政编码为 6 位数字

匹配身份证：`\d{17}[\d|X]|\d{15}`

评注：中国的身份证为 15 位或 18 位

匹配 ip 地址：
`((2[0-4]\d|25[0-5]| [01]?\d\d?)\.) {3} (2[0-4]\d|25[0-5]| [01]?\d\d?)`
)。

评注：提取 ip 地址时有用

匹配特定数字：

`^[1-9]\d*$` //匹配正整数

`^- [1-9]\d*$` //匹配负整数

```
^-?[1-9]\d*$ //匹配整数
^[1-9]\d*|0$ //匹配非负整数（正整数 + 0）
^-[1-9]\d*|0$ //匹配非正整数（负整数 + 0）
^[1-9]\d*\.\d*|0\.\d*[1-9]\d*$ //匹配正浮点数
^-([1-9]\d*\.\d*|0\.\d*[1-9]\d*)$ //匹配负浮点数
^-?([1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.\d*|0)$ //匹配浮点数
^[1-9]\d*\.\d*|0\.\d*[1-9]\d*|0?\.\d*|0$ //匹配非负浮点数（正浮点数 + 0）
```

```
^-(-([1-9]\d*\.\d*|0\.\d*[1-9]\d*))|0?\.\d*|0$ //匹配非正浮点数（负浮点数 + 0）
```

评注：处理大量数据时有用，具体应用时注意修正

匹配特定字符串：

```
^[A-Za-z]+$ //匹配由 26 个英文字母组成的字符串
^[A-Z]+$ //匹配由 26 个英文字母的大写组成的字符串
^[a-z]+$ //匹配由 26 个英文字母的小写组成的字符串
^[A-Za-z0-9]+$ //匹配由数字和 26 个英文字母组成的字符串
^\w+$ //匹配由数字、26 个英文字母或者下划线组成的字符串
```

评注：最基本也是最常用的一些表达式

[a-z] //匹配所有的小写字母
[A-Z] //匹配所有的大写字母
[a-zA-Z] //匹配所有的字母
[0-9] //匹配所有的数字
[0-9\.\-] //匹配所有的数字，句号和减号
[\f\r\t\n] //匹配所有的白字符

同样的，这些也只表示一个字符，这是一个非常重要的。如果要匹配一个由一个小写字母和一位数字组成的字符串，比如“z2”、“t6”或“g7”，但不是“ab2”、“r2d3”或“b52”的话，用这个模式：

```
^[a-z][0-9]$
```

尽管[a-z]代表 26 个字母的范围，但在这里它只能与第一个字符是小写字母的字符串匹配。

前面曾经提到^表示字符串的开头，但它还有另外一个含义。当在一组方括号里使用^是，它表示“非”或“排除”的意思，常常用来剔除某个字符。还用前面的例子，我们要求第一个字符不能是数字：

```
^[^0-9][0-9]$
```

这个模式与“&5”、“g7”及“-2”是匹配的，但与“12”、“66”是不匹配的。下面是几个排除特定字符的例子：

```
[^a-z] //除了小写字母以外的所有字符  
[^\(\)\^] //除了(\)(\^)-之外的所有字符  
[^\\"'] //除了双引号(")和单引号(')之外的所有字符
```

特殊字符“.”（点，句号）在正规表达式中用来表示除了“新行”之外的所有字符。所以模式“^.5\$”与任何两个字符的、以数字 5 结尾和以其他非“新行”字符开头的字符串匹配。模式“.”可以匹配任何字符串，除了空串和只包括一个“新行”的字符串。

PHP 的正规表达式有一些内置的通用字符簇，列表如下：

字符簇	含义
[:alpha:]	任何字母
[:digit:]	任何数字
[:alnum:]	任何字母和数字
[:space:]	任何白字符
[:upper:]	任何大写字母
[:lower:]	任何小写字母
[:punct:]	任何标点符号
[:xdigit:]	任何 16 进制的数字，相当于[0-9a-fA-F]

7.3 确定重复出现

到现在为止，你已经知道如何去匹配一个字母或数字，但更多的情况下，可能要匹配一个单词或一组数字。

一个单词有若干个字母组成，一组数字有若干个单数组成。跟在字符或字符簇后面的花括号({})用来确定前面的内容的重复出现的次数。

字符簇 含义

`^[a-zA-Z_]$` 所有的字母和下划线

`^[[:alpha:]]{3}$` 所有的 3 个字母的单词

`^a$` 字母 a

`^a{4}$` aaaa

`^a{2,4}$` aa, aaa 或 aaaa

`^a{1,3}$` a, aa 或 aaa

`^a{2,}$` 包含多于两个 a 的字符串

`^a{2,}` 如: aardvark 和 aaab, 但 apple 不行

`a{2,}` 如: baad 和 aaa, 但 Nantucket 不行

`\t{2}` 两个制表符

`.{2}` 所有的两个字符

这些例子描述了花括号的三种不同的用法。一个数字, {x} 的意思是“前面的字符或字符簇只出现 x 次”；一个数字加逗号, {x,} 的意思是“前面的内容出现 x 或更多的次数”；两个用逗号分隔的数字, {x, y} 表示“前面的内容至少出现 x 次, 但不超过 y 次”。我们可以把模式扩展到更多的单词或数字：

`^[a-zA-Z0-9_]{1,}$` //所有包含一个以上的字母、数字或下划线的字符串

`^[0-9]{1,}$` //所有的正数

`^\-{0,1}[0-9]{1,}$` //所有的整数

`^\-{0,1}[0-9]{0,}\.{0,1}[0-9]{0,}$` //所有的小数

最后一个例子不太好理解，是吗？这么看吧：与所有以一个可选的负号(`^\-{0,1}`)开头(`^`)、跟着 0 个或多个的数字(`[0-9]{0,}`)、和一个可选的小数点(`\.{0,1}`)再跟上 0 个或多个数字(`[0-9]{0,}`)，并且没有其他任何东西(`$`)。下面你将知道能够使用的更为简单的方法。

特殊字符“?”与 {0,1} 是相等的，它们都代表着：“0 个或 1 个前面的内容”或“前面的内容是可选的”。所以刚才的例子可以简化为：

`^\-{0,1}[0-9]{0,}\.{0,1}[0-9]{0,}$`

特殊字符“*”与 {0,} 是相等的，它们都代表着“0 个或多个前面的内容”。最后，字符“+”与 {1,} 是相等的，表示“1 个或多个前面的内容”，所以上面的 4 个例子可以写成：

`^[a-zA-Z0-9_]+$` //所有包含一个以上的字母、数字或下划线的字符串

`^[0-9]+$` //所有的正数

`^\-{0,1}[0-9]+$` //所有的整数

`^\-{0,1}[0-9]*\.[0-9]*$` //所有的小数

当然这并不能从技术上降低正则表达式的复杂性，但可以使它们更容易阅读。

正则表达式基础知识

一个正则表达式就是由普通字符（例如字符 **a** 到 **z**）以及特殊字符（称为元字符）组成的文字模式。该模式描述在查找文字主体时待匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。如：

JScript	VBScript	匹配
/^[\t]*\$/	"^[\t]*\$"	匹配一个空白行。
/\d{2}-\d{5}/	"\d{2}-\d{5}"	验证一个 ID 号码是否由一个 2 位数字，一个连字符以及一个 5 位数字组成。
/<(.*?)>.*<\1>/	"<(.*?)>.*<\1>"	匹配一个 HTML 标记。

下表是元字符及其在正则表达式上下文中的行为的一个完整列表：

字符	描述
\	将下一个字符标记为一个特殊字符、或一个原义字符、或一个 向后引用、或一个八进制转义符。例如，'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\' 匹配 "\" 而 \"(\" 则匹配 "("。
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。
\$	匹配输入字符串的结束位置。如果设置了 RegExp 对象的 Multiline 属性，\$ 也匹配 '\n' 或 '\r' 之前的位置。
*	匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。* 等价于 {0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 中的 "do" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配 n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooooo" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中 n <= m。最少匹配 n 次且最多匹配 m 次。例如，

	"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。
?	当该字符紧跟在任何一个其他限制符 (*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串 "oooo", 'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。
.	匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。
(<i>pattern</i>)	匹配 <i>pattern</i> 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在 VBScript 中使用 SubMatches 集合，在 JScript 中则使用 \$0...\$9 属性。要匹配圆括号字符，请使用 '\(' 或 '\)'。
(?: <i>pattern</i>)	匹配 <i>pattern</i> 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用 "或" 字符 () 来组合一个模式的各个部分是很有用。例如，'industr(?:y ies) 就是一个比 'industry industries' 更简略的表达式。
(?= <i>pattern</i>)	正向预查，在任何匹配 <i>pattern</i> 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，'Windows (=?95 98 NT 2000)' 能匹配 "Windows 2000" 中的 "Windows"，但不能匹配 "Windows 3.1" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
(?! <i>pattern</i>)	负向预查，在任何不匹配 <i>pattern</i> 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如'Windows (?!95 98 NT 2000)' 能匹配 "Windows 3.1" 中的 "Windows"，但不能匹配 "Windows 2000" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始
x y	匹配 x 或 y。例如，'z food' 能匹配 "z" 或 "food"。'(z f)ood' 则匹配 "zood" 或 "food"。
[xyz]	字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。
[^xyz]	负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的 'p'。

[a-z]	字符范围。匹配指定范围内的任意字符。例如, '[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符。
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如, '[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符。
\b	匹配一个单词边界, 也就是指单词和空格间的位置。例如, 'er\b' 可以匹配"never" 中的 'er', 但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er', 但不能匹配 "never" 中的 'er'。
\cx	匹配由 <i>x</i> 指明的控制字符。例如, \cM 匹配一个 Control-M 或回车符。 <i>x</i> 的值必须为 A-Z 或 a-z 之一。否则, 将 <i>c</i> 视为一个原义的 'c' 字符。
\d	匹配一个数字字符。等价于 [0-9]。
\D	匹配一个非数字字符。等价于 [^0-9]。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符, 包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。
\w	匹配包括下划线的任何单词字符。等价于 '[A-Za-z0-9_]'。
\W	匹配任何非单词字符。等价于 '[^A-Za-z0-9_]'。
\xn	匹配 <i>n</i> , 其中 <i>n</i> 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如, '\x41' 匹配 "A"。'\x041' 则等价于 '\x04' & "1"。正则表达式中可以使用 ASCII 编码。.
\num	匹配 <i>num</i> , 其中 <i>num</i> 是一个正整数。对所获取的匹配的引用。例如, '(.)\1' 匹配两个连续的相同字符。
\n	标识一个八进制转义值或一个向后引用。如果 \n 之前至少 <i>n</i> 个获取的子表达式, 则 <i>n</i> 为向后引用。否则, 如果 <i>n</i> 为八进制数字 (0-7), 则 <i>n</i> 为一个八进制转义值。

<code>\nm</code>	标识一个八进制转义值或一个向后引用。如果 <code>\nm</code> 之前至少有 <code>nm</code> 个获得子表达式，则 <code>nm</code> 为向后引用。如果 <code>\nm</code> 之前至少有 <code>n</code> 个获取，则 <code>n</code> 为一个后跟文字 <code>m</code> 的向后引用。如果前面的条件都不满足，若 <code>n</code> 和 <code>m</code> 均为八进制数字 (0-7)，则 <code>\nm</code> 将匹配八进制转义值 <code>nm</code> 。
<code>\nml</code>	如果 <code>n</code> 为八进制数字 (0-3)，且 <code>m</code> 和 <code>l</code> 均为八进制数字 (0-7)，则匹配八进制转义值 <code>nml</code> 。
<code>\un</code>	匹配 <code>n</code> ，其中 <code>n</code> 是一个用四个十六进制数字表示的 Unicode 字符。例如， <code>\u00A9</code> 匹配版权符号 (©)。

下面看几个例子：

`"^The"`：表示所有以"The"开始的字符串（"There"，"The cat"等）；

`"of despair$"`：表示所以以"of despair"结尾的字符串；

`"^abc$"`：表示开始和结尾都是"abc"的字符串——呵呵，只有"abc"自己了；

`"notice"`：表示任何包含"notice"的字符串。

`'*'`，`'+'`和`'?'`这三个符号，表示一个或一序列字符重复出现的次数。它们分别表示“没有或

更多”，“一次或更多”还有“没有或一次”。下面是几个例子：

`"ab*"`：表示一个字符串有一个 `a` 后面跟着零个或若干个 `b`。（`"a"`，`"ab"`，`"abbb"`，.....）；

`"ab+"`：表示一个字符串有一个 `a` 后面跟着至少一个 `b` 或者更多；

`"ab?"`：表示一个字符串有一个 `a` 后面跟着零个或者一个 `b`；

`"a?b+$"`：表示在字符串的末尾有零个或一个 `a` 跟着一个或几个 `b`。

也可以使用范围，用大括号括起，用以表示重复次数的范围。

`"ab{2}"`：表示一个字符串有一个 `a` 跟着 2 个 `b`（`"abb"`）；

`"ab{2,}"`：表示一个字符串有一个 `a` 跟着至少 2 个 `b`；

`"ab{3,5}"`：表示一个字符串有一个 `a` 跟着 3 到 5 个 `b`。

请注意，你必须指定范围的下限（如：`"{0,2}"`而不是`"{,2}"`）。还有，你可能注意到了，`'*'`，`'+'`和

'?'相当于"`{0,}`", "`{1,}`"和"`{0,1}`"。

还有一个'|', 表示“或”操作:

"`hi|hello`": 表示一个字符串里有"`hi`"或者"`hello`";

"`(b|cd)ef`": 表示"`bef`"或"`cdef`";

"`(a|b)*c`": 表示一串"`a`"和"`b`"混合的字符串后面跟一个"`c`";

'.'可以替代任何字符:

"`a.[0-9]`": 表示一个字符串有一个"`a`"后面跟着一个任意字符和一个数字;

"`^. {3}$`": 表示有任意三个字符的字符串 (长度为 3 个字符);

方括号表示某些字符允许在一个字符串中的某一特定位置出现:

"`[ab]`": 表示一个字符串有一个"`a`"或"`b`" (相当于"`a|b`");

"`[a-d]`": 表示一个字符串包含小写的'`a`'到'`d`'中的一个 (相当于"`a|b|c|d`"或者"`[abcd]`");

"`^[a-zA-Z]`": 表示一个以字母开头的字符串;

"`[0-9]%`": 表示一个百分号前有一位的数字;

"`,[a-zA-Z0-9]$"`: 表示一个字符串以一个逗号后面跟着一个字母或数字结束。

你也可以在方括号里用'^'表示不希望出现的字符, '^'应在方括号里的第一位。(如:

"`%[^a-zA-Z]%`"表

示两个百分号中不应该出现字母)。

为了逐字表达, 必须在"`^.$()*!~*+?{\\"`这些字符前加上转移字符'\'。

请注意在方括号中, 不需要转义字符。

正则表达式语法一个正则表达式就是由普通字符 (例如字符 `a` 到 `z`) 以及特殊字符 (称为元字符) 组成的文字模式。该模式描述在查找文字主体时待匹配的一个或多个字符串。正则表达式作为一个模板, 将某个字符模式与所搜索的字符串进行匹配。这里有一些可能会遇到

的正则表达式示例：**JScript VBScript** 匹配 `/^\[\t]*$/` `"^\[\t]*$"` 匹配一个空白行。
`\d{2}-\d{5}/` `"\d{2}-\d{5}"` 验证一个 ID 号码是否由一个 2 位数字，一个连字符以及一个 5 位数字组成。
`/<(.*?)>.*</\1>/` `"<(.*?)>.*</\1>"` 匹配一个 HTML 标记。下表是元字符及其在正则表达式上下文中的行为的一个完整列表：字符 描述 \ 将下一个字符标记为一个特殊字符、或一个原义字符、或一个 后向引用、或一个八进制转义符。例如，`'n'` 匹配字符 `"n"`。
`'\n'` 匹配一个换行符。序列 `'\\'` 匹配 `"\ ..."`

一个正则表达式就是由普通字符（例如字符 `a` 到 `z`）以及特殊字符（称为元字符）组成的文字模式。该模式描述在查找文字主体时待匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。

这里有一些可能会遇到的正则表达式示例：

**Visual Basic VBScript 匹配
Scripting Edition**

`/^\[\t]*$/` `"^\[\t]*$"` 匹配一个空白行。

`/\d{2}-\d{5}/` `"\d{2}-\d{5}"` 验证一个 ID 号码是否由一个 2 位字，一个连字符以及一个 5 位数字组成。

`/<(.*?)>.*</\1>/` `"<(.*?)>.*</\1>"` 匹配一个 HTML 标记。

下表是元字符及其在正则表达式上下文中的行为的一个完整列表：

字符 描述

将下一个字符标记为一个特殊字符、或一个原义字符、或一个 后向引用、或一个八进制转义符。例如，`'n'` 匹配字符 `"n"`。`'\n'` 匹配一个换行符。序列 `''` 匹配 `""` 而 `"("` 则匹配 `"("`。

`^` 匹配输入字符串的开始位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，`^` 也匹配 `'n'` 或 `'r'` 之后的位置。

`$` 匹配输入字符串的结束位置。如果设置了 **RegExp** 对象的 **Multiline** 属性，`$` 也匹配 `'n'` 或 `'r'` 之前的位置。

* 匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。 * 等价于 {0,}。

+ 匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。

? 匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 中的 "do" 。? 等价于 {0,1}。

{n} n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。

{n,} n 是一个非负整数。至少匹配 n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooooo" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。

{n,m} m 和 n 均为非负整数，其中 n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "foooooo" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格

? 当该字符紧跟在任何一个其他限制符 (*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串 "oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。

. 匹配除 "n" 之外的任何单个字符。要匹配包括 'n' 在内的任何字符，请使用象 '[.n]' 的模式。

(pattern) 匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在 VBScript 中使用 SubMatches 集合，在 Visual Basic Scripting Edition 中则使用 \$0...\$9 属性。要匹配圆括号字符，请使用 '(' 或 ')'。

(?:pattern) 匹配 pattern 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用 "或" 字符 (|) 来组合一个模式的各个部分是很有用。例如，'industr(?:y|ies)' 就是一个比 'industry|industries' 更简略的表达式。

(?=pattern) 正向预查，在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，'Windows (?!95|98|NT|2000)' 能匹配 "Windows 2000" 中的 "Windows"，但不能匹配 "Windows 3.1" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹

配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。

(?!pattern) 负向预查，在任何不匹配 **Negative lookahead matches the search string at any point where a string not matching pattern** 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如 'Windows (?!95|98|NT|2000)' 能匹配 "Windows 3.1" 中的 "Windows"，但不能匹配 "Windows 2000" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始

x|y 匹配 x 或 y。例如， 'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。

[xyz] 字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。

[^xyz] 负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的 'p'。

[a-z] 字符范围。匹配指定范围内的任意字符。例如， '[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符。

[^a-z] 负值字符范围。匹配任何不在指定范围内的任意字符。例如， '[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符。

b 匹配一个单词边界，也就是指单词和空格间的位置。例如， 'erb' 可以匹配 "never" 中的 'er'，但不能匹配 "verb" 中的 'er'。

B 匹配非单词边界。'erB' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。

cx 匹配由 x 指明的控制字符。例如， cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。

d 匹配一个数字字符。等价于 [0-9]。

D 匹配一个非数字字符。等价于 [^0-9]。

f 匹配一个换页符。等价于 x0c 和 cL。

n 匹配一个换行符。等价于 **x0a** 和 **cJ**。

r 匹配一个回车符。等价于 **x0d** 和 **cM**。

s 匹配任何空白字符，包括空格、制表符、换页符等等。等价于 **[fnrtv]**。

S 匹配任何非空白字符。等价于 **[^ fnrtv]**。

t 匹配一个制表符。等价于 **x09** 和 **cI**。

v 匹配一个垂直制表符。等价于 **x0b** 和 **cK**。

w 匹配包括下划线的任何单词字符。等价于 **'[A-Za-z0-9_]'**。

W 匹配任何非单词字符。等价于 **'[^A-Za-z0-9_]'**。

xn 匹配 **n**，其中 **n** 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，**'x41'** 匹配 **"A"**。**'x041'** 则等价于 **'x04' & "1"**。正则表达式中可以使用 **ASCII** 编码。.




num 匹配 **num**，其中 **num** 是一个正整数。对所获取的匹配的引用。例如，**'(.)1'** 匹配两个连续的相同字符。

n 标识一个八进制转义值或一个后向引用。如果 **n** 之前至少 **n** 个获取的子表达式，则 **n** 为后向引用。否则，如果 **n** 为八进制数字 (0-7)，则 **n** 为一个八进制转义值。

nm 标识一个八进制转义值或一个后向引用。如果 **nm** 之前至少有 **is preceded by at least nm** 个获取子表达式，则 **nm** 为后向引用。如果 **nm** 之前至少有 **n** 个获取，则 **n** 为一个后跟文字 **m** 的后向引用。如果前面的条件都不满足，若 **n** 和 **m** 均为八进制数字 (0-7)，则 **nm** 将匹配八进制转义值 **nm**。

nml 如果 **n** 为八进制数字 (0-3)，且 **m** 和 **l** 均为八进制数字 (0-7)，则匹配八进制转义值 **nml**。

un 匹配 **n**，其中 **n** 是一个用四个十六进制数字表示的 **Unicode** 字符。例如，**u00A9** 匹配版权符号 (?)。




校验登录名：只能输入 5-20 个以字母开头、可带数字、“_”、“.” 的字串
Java 代码   

```

1. function isRegisterUserName(s)
2. {
3. var patrn=/^[a-zA-Z]{1}([a-zA-Z0-9]|[_]){4,19}$/;
4. if (!patrn.exec(s)) return false
5. return true
6. }

```

//校验用户姓名：只能输入 1-30 个以字母开头的字符串

Java 代码   

```

1. function isTrueName(s)
2. {
3. var patrn=/^[a-zA-Z]{1,30}$/;
4. if (!patrn.exec(s)) return false
5. return true
6. }
7. }}
8.
9. //校验密码：只能输入 6-20 个字母、数字、下划线
10. <PRE class=java name="code">function isPasswd(s)
11. {
12. var patrn=/^(\\w){6,20}$/;
13. if (!patrn.exec(s)) return false
14. return true
15. }
16. </PRE>
17. <BR>
18. <BR> //校验普通电话、传真号码：可以 “+” 开头，除数字外，可含有 “-”
19. <BR> <PRE class=java name="code">function isTel(s)
20. {
21. //var patrn=/^[+]{0,1}(\\d){1,3}[ ]?([-]?\\d){1,12})+$/;
22. var patrn=/^[+]{0,1}(\\d){1,3}[ ]?([-]?((\\d)|[ ]){1,12})+$/;
23. if (!patrn.exec(s)) return false
24. return true
25. }
26. </PRE>
27. <BR>
28. <BR> //校验手机号码：必须以数字开头，除数字外，可含有 “-”
29. <BR> <PRE class=java name="code">function isMobil(s)
30. {
31. var patrn=/^[+]{0,1}(\\d){1,3}[ ]?([-]?((\\d)|[ ]){1,12})+$/;
32. if (!patrn.exec(s)) return false

```

```

33.return true
34.}
35.</PRE>
36.<BR>
37.<BR>//校验邮政编码
38.<BR><PRE class=java name="code">function isPostalCode(s)
39.{
40.//var patrn=/^[a-zA-Z0-9]{3,12}$/;
41.var patrn=/^[a-zA-Z0-9]{3,12}$/;
42.if (!patrn.exec(s)) return false
43.return true
44.}
45.</PRE>
46.<BR>
47.<BR>//校验搜索关键字
48.<BR><PRE class=java name="code">function isSearch(s)
49.{
50.var
    patrn=/^[^~!@#%&*()+=|\\\[\\]\{\}\:;'\.,.<?/>{1}[^~!@#%&*()+=|\\\[\\]\{\}\:;'\.,.<?/>{0,19}$/;
51.[\\]\{\}\:;'\.,.<?/>{0,19}$/;
52.if (!patrn.exec(s)) return false
53.return true
54.}
55.
56.function isIP(s) //by zergling
57.{
58.var patrn=/^[0-9.]{1,20}$/;
59.if (!patrn.exec(s)) return false
60.return true
61.}
62.</PRE>
63.<BR>
64.<BR><SPAN style="FONT-SIZE: 18pt">正则表达式</SPAN>
65.<BR><PRE class=java name="code">"^\d+$" //非负整数（正整数 +
    0）
66."^[0-9]*[1-9][0-9]*$" //正整数
67."^((-\\d+)|(0+))$" //非正整数（负整数 + 0）
68."^-[0-9]*[1-9][0-9]*$" //负整数
69."^-?\\d+$" //整数
70."^\\d+(\\.\\d+)?$" //非负浮点数（正浮点数 + 0）
71."^(([0-9]+\\. [0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\\. [0-9]+)|([
    0-9]*[1-9][0-9]*))$"
72.//正浮点数

```

73. `"^((-\\d+(\\.\\d+)?)|(0+(\\.0+)?))$"` //非正浮点数(负浮点数 + 0)
74. `"^(-([0-9]+\\. [0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\\. [0-9]+)|([0-9]*[1-9][0-9]*))$"`
75. //负浮点数
76. `"^(-?\\d+)(\\.\\d+)?$"` //浮点数
77. `"^[A-Za-z]+$"` //由 26 个英文字母组成的字符串
78. `"^[A-Z]+$"` //由 26 个英文字母的大写组成的字符串
79. `"^[a-z]+$"` //由 26 个英文字母的小写组成的字符串
80. `"^[A-Za-z0-9]+$"` //由数字和 26 个英文字母组成的字符串
81. `"^[\\w]+$"` //由数字、26 个英文字母或者下划线组成的字符串
82. `"^[\\w-]+(\\. [\\w-]+)*@[\\w-]+(\\. [\\w-]+)+$"` //email 地址
83. `"^[a-zA-Z]+://(\\w+(-\\w+)*)(\\. (\\w+(-\\w+)*))*((\\?\\S*)?)$"`
//url
84. `"^[A-Za-z0-9_]*$"`
85. </PRE>
86.

87.
正则表达式使用详解
88.

89.
简介
90.

91.
简单的说,正则表达式是一种可以用于模式匹配和替换的强有力的工具。其作用如下:
92.
测试字符串的某个模式。例如,可以对一个输入字符串进行测试,看
在该字符串是否存在一个电话号码模式或一个信用卡号码模式。这称为数
据有效性验证。
93.
替换文本。可以在文档中使用一个正则表达式来标识特定文字,然后
可以全部将其删除,或者替换为别的文字。
94.
根据模式匹配从字符串中提取一个子字符串。可以用来在文本或输入
字段中查找特定文字。
95.

96.
基本语法
97.

98.
在对正则表达式的功能和作用有了初步的了解之后,我们就来具体看
一下正则表达式的语法格式。
99.

100.
正则表达式的形式一般如下:
101.

102.
/love/ 其中位于 "/" 定界符之间的部分就是将要在目标
对象中进行匹配的模式。用户只要把希望查找匹配对象的模式内容放入
"/" 定界符之间即可。为了能够使用户更加灵活的定制模式内容,正则
表达式提供了专门的“元字符”。所谓元字符就是指那些在正则表达式中

具有特殊意义的专用字符，可以用来规定其前导字符（即位于元字符前面的字符）在目标对象中的出现模式。

103.
较为常用的元字符包括：“+”，“*”，以及“?”。
104.

105.
“+”元字符规定其前导字符必须在目标对象中连续出现一次或多次。
106.

107.
“*”元字符规定其前导字符必须在目标对象中出现零次或连续多次。
108.

109.
“?”元字符规定其前导对象必须在目标对象中连续出现零次或一次。
110.

111.
下面，就让我们来看一下正则表达式元字符的具体应用。
112.

113.
/fo+/ 因为上述正则表达式中包含“+”元字符，表示可以与目标对象中的“fool”，“fo”，或者“football”等在字母f后面连续出现一个或多个字母o的字符串相匹配。
114.

115.
/eg*/ 因为上述正则表达式中包含“*”元字符，表示可以与目标对象中的“easy”，“ego”，或者“egg”等在字母e后面连续出现零个或多个字母g的字符串相匹配。
116.

117.
/Wil?/ 因为上述正则表达式中包含“?”元字符，表示可以与目标对象中的“Win”，或者“Wilson”，等在字母i后面连续出现零个或一个字母l的字符串相匹配。
118.

119.
有时候不知道要匹配多少字符。为了能适应这种不确定性，正则表达式支持限定符的概念。这些限定符可以指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。
120.

121.
{n} n 是一个非负整数。匹配确定的 n 次。例如，‘o{2}’不能匹配“Bob”中的‘o’，但是能匹配“food”中的两个 o。
122.

123.
{n,} n 是一个非负整数。至少匹配 n 次。例如，‘o{2,}’不能匹配“Bob”中的‘o’，但能匹配“fooooood”中的所有 o。‘o{1,}’等价于‘o+’。‘o{0,}’则等价于‘o*’。
124.

125.
{n,m} m 和 n 均为非负整数，其中 n <= m。最少匹配 n 次且最多匹配 m 次。例如，“o{1,3}”将匹配“fooooood”中的前三个 o。‘o{0,1}’等价于‘o?’。请注意在逗号和两个数之间不能有空格。
126.

127.
除了元字符之外，用户还可以精确指定模式在匹配对象中出现的频率。例如，/jim {2,6}/ 上述正则表达式规定字符 m 可以在匹配对象

中连续出现 2-6 次，因此，上述正则表达式可以同 jimmy 或 jimmmmmmy 等字符串相匹配。

128.
在对如何使用正则表达式有了初步了解之后，我们来看一下其它几个重要的元字符的使用方式。
129.
<PRE class=java name="code">\s: 用于匹配单个空格符，包括 tab 键和换行符；
130. \S: 用于匹配除单个空格符之外的所有字符；
131. \d: 用于匹配从 0 到 9 的数字；
132. \w: 用于匹配字母，数字或下划线字符；
133. \W: 用于匹配所有与 \w 不匹配的字符；
134. . : 用于匹配除换行符之外的所有字符。
135. </PRE>
136.
（说明：我们可以把 \s 和 \S 以及 \w 和 \W 看作互为逆运算）
137.
下面，我们就通过实例看一下如何在正则表达式中使用上述元字符。
138.
/\s+/ 上述正则表达式可以用于匹配目标对象中的一个或多个空格字符。
139.
/\d000/ 如果我们手中有一份复杂的财务报表，那么我们可以通过上述正则表达式轻而易举的查找到所有总额达千元的款项。
140.
除了我们以上所介绍的元字符之外，正则表达式中还具有另外一种较为独特的专用字符，即定位符。定位符用于规定匹配模式在目标对象中的出现位置。较为常用的定位符包括：“^”，“\$”，“\b”以及“\B”。
141.
<PRE class=java name="code">“^”定位符规定匹配模式必须出现在目标字符串的开头
142. “\$”定位符规定匹配模式必须出现在目标对象的结尾
143. “\b”定位符规定匹配模式必须出现在目标字符串的开头或结尾的两个边界之一
144. “\B”定位符则规定匹配对象必须位于目标字符串的开头和结尾两个边界之内，
145. 即匹配对象既不能作为目标字符串的开头，也不能作为目标字符串的结尾。
146. </PRE>
147.
同样，我们也可以把“^”和“\$”以及“\b”和“\B”看作是互为逆运算的两组定位符。举例来说：/^hell/ 因为上述正则表达式中包含“^”定位符，所以可以与目标对象中以“hell”，“hello”或“hellhound”开头的字符串相匹配。/ar\$/ 因为上述正则表达式中包含“\$”定位符，所以可以与目标对象中以“car”，“bar”或“ar”结尾的字符串相匹配。/\bbom/ 因为上述正则表达式模式以“\b”定位符开头，所以可以与目标对象中以“bomb”，或“bom”开头的字符串相匹配。/man\b/ 因为上述正则表达式模式以“\b”定位符结尾，所以可以与目标对象中以“human”，“woman”或“man”结尾的字符串相匹配。

148.
为了能够方便用户更加灵活的设定匹配模式，正则表达式允许使用者在匹配模式中指定某一个范围而不局限于具体的字符。例如：
149.
<PRE class=java name="code">/[A-Z]/ 上述正则表达式将会与从 A 到 Z 范围内任何一个大写字母相匹配。
150. /[a-z]/ 上述正则表达式将会与从 a 到 z 范围内任何一个小写字母相匹配。
151. /[0-9]/ 上述正则表达式将会与从 0 到 9 范围内任何一个数字相匹配。
152. /([a-z][A-Z][0-9]+)/ 上述正则表达式将会与任何由字母和数字组成的字符串，如 “aB0” 等相匹配。
153. </PRE>
154.
这里需要提醒用户注意的一点就是可以在正则表达式中使用 “()” 把字符串组合在一起。“()” 符号包含的内容必须同时出现在目标对象中。因此，上述正则表达式将无法与诸如 “abc” 等的字符串匹配，因为 “abc” 中的最后一个字符为字母而非数字。
155.
如果我们在正则表达式中实现类似编程逻辑中的“或”运算，在多个不同的模式中任选一个进行匹配的话，可以使用管道符 “|”。例如：/to|too|2/ 上述正则表达式将会与目标对象中的 “to”，“too”，或 “2” 相匹配。
156.
正则表达式中还有一个较为常用的运算符，即否定符 “[^]”。与我们前文所介绍的定位符 “^” 不同，否定符 “[^]” 规定目标对象中不能存在模式中所规定的字符串。例如：/[^A-C]/ 上述字符串将会与目标对象中除 A，B，和 C 之外的任何字符相匹配。一般来说，当 “^” 出现在 “[^]” 内时就被视做否定运算符；而当 “^” 位于 “[^]” 之外，或没有 “[^]” 时，则应当被视做定位符。
157.
最后，当用户需要在正则表达式的模式中加入元字符，并查找其匹配对象时，可以使用转义符 “\”。例如：/Th*/ 上述正则表达式将会与目标对象中的 “Th*” 而非 “The” 等相匹配。
158.
在构造正则表达式之后，就可以象数学表达式一样来求值，也就是说，可以从左至右并按照一个优先级顺序来求值。优先级如下：
159.
<PRE class=java name="code">1. \ 转义符
160. 2. (), (?:), (?:=), [] 圆括号和方括号
161. 3. *, +, ?, {n}, {n,}, {n,m} 限定符
162. 4. ^, \$, \anymetacharacter 位置和顺序
163. 5. | “或” 操作
164. </PRE>
165.

166.
使用实例
167.
在 JavaScript 1.2 中带有功能强大的 RegExp() 对象，可以用来进行正则表达式的匹配操作。其中的 test() 方法可以检验目标对象中是否包含匹配模式，并相应的返回 true 或 false。
168.
我们可以使用 JavaScript 编写以下脚本，验证用户输入的邮件地址的有效性。
169.
<PRE class=java name="code"><html>

```

170. <head>
171.     <script language="Javascript1.2">
172.         <!-- start hiding
173.         function verifyAddress(obj)
174.         {
175.             var email = obj.email.value;
176.             var pattern =
177. / ^ ([a-zA-Z0-9_-])+@([a-zA-Z0-9_-])+(\. [a-zA-Z0-9_-])+ /;
178.             flag = pattern.test(email);
179.             if(flag)
180.             {
181.                 alert( "Your email address is correct!" );
182.                 return true;
183.             }
184.             else
185.             {
186.                 alert( "Please try again!" );
187.                 return false;
188.             }
189.         }
190.         // stop hiding -->
191.     </script>
192. </head>
193. <body>
194.     <form onSubmit="return verifyAddress(this);">
195.         <input name="email" type="text">
196.         <input type="submit">
197.     </form>
198. </body>
199. </html>
200. </PRE>
201. <BR>
202. <BR><SPAN style="FONT-SIZE: 18pt">正则表达式对象</SPAN>
203. <BR>本对象包含正则表达式模式以及表明如何应用模式的标志。
204. <BR><PRE class=java name="code">语法 1 re = /pattern/[flags]
205. 语法 2 re = new RegExp("pattern",["flags"])
206. </PRE>
207. <BR>参数
208. <BR>re
209. <BR>必选项。将要赋值为正则表达式模式的变量名。
210. <BR>
211. <BR>Pattern
212. <BR>必选项。要使用的正则表达式模式。如果使用语法 1, 用 "/" 字
    符分隔模式。如果用法 2, 用引号将模式引起来。

```

213.

214.
Flags
215.
可选项。如果使用语法 2 要用引号将 flag 引起来。标志可以组合使用，可用的有：
216.
<PRE class=java name="code">g （全文查找出现的所有 pattern）
217. i （忽略大小写）
218. m （多行查找）
219. </PRE>
220.

221.
示例
222.
下面的示例创建一个包含正则表达式模式及相关标志的对象（re），向您演示正则表达式对象的用法。在本例中，作为结果的正则表达式对象又用于 match 方法中：
223.
<PRE class=java name="code">function MatchDemo()
224. {
225. var r, re; // 声明变量。
226. var s = "The rain in Spain falls mainly in the plain";
227. re = new RegExp("ain", "g"); // 创建正则表达式对象。
228. r = s.match(re); // 在字符串 s 中查找匹配。
229. return(r);
230. }
231. </PRE>
232.

233.
返回值： ain,ain,ain,ain\\
234.
属性 lastIndex 属性 | source 属性\\
235.
方法 compile 方法 | exec 方法 | test 方法\\
236.
要求 版本 3\\
237.
请参阅 RegExp 对象 | 正则表达式语法 | String 对象\\
238.

239.
exec 方法
240.
用正则表达式模式在字符串中运行查找，并返回包含该查找结果的一个数组。
241.
RegExp.exec(str)
242.

243.
参数
244.

245.
RegExp
246.
必选项。包含正则表达式模式和可用标志的正则表达式对象。
247.

248.
str
249.
必选项。要在其中执行查找的 String 对象或字符串文字。
250.

251.
说明\\

252.
如果 exec 方法没有找到匹配, 则它返回 null。如果它找到匹配, 则 exec 方法返回一个数组, 并且更新全局 RegExp 对象的属性, 以反映匹配结果。数组的 0 元素包含了完整的匹配, 而第 1 到 n 元素中包含的是匹配中出现的任意一个子匹配。这相当于没有设置全局标志 (g) 的 match 方法。
253.
如果为正则表达式设置了全局标志, exec 从以 lastIndex 的值指示的位置开始查找。如果没有设置全局标志, exec 忽略 lastIndex 的值, 从字符串的起始位置开始搜索。
254.

255.
exec 方法返回的数组有三个属性, 分别是 input、index 和 lastIndex。Input 属性包含了整个被查找的字符串。Index 属性中包含了整个被查找字符串中被匹配的子字符串的位置。LastIndex 属性中包含了匹配中最后一个字符的下一个位置。
256.

257.
示例\\
258.
下面的例子举例说明了 exec 方法的用法:
259.
<PRE class=java name="code">function RegExpTest()
260. {
261. var ver = Number(ScriptEngineMajorVersion() + "." +
ScriptEngineMinorVersion())
262. if (ver >= 5.5) { // 测试 JScript 的版本。
263. var src = "The rain in Spain falls mainly in the plain.";
264. var re = /\w+/g; // 创建正则表达式模式。
265. var arr;
266. while ((arr = re.exec(src)) != null)
267. document.write(arr.index + "-" + arr.lastIndex + arr + "\t");
268. }
269. else {
270. alert("请使用 JScript 的更新版本");
271. }
272. }
273. </PRE>
274.

275.
返回值: 0-3The 4-8rain 9-11in 12-17Spain 18-23falls
24-30mainly 31-33in 34-37the 38-43plain
276.

277.
test 方法\\
278.
返回一个 Boolean 值, 它指出在被查找的字符串中是否存在模式。
279.
regexp.test(str)
280.

281.
参数\\
282.
regexp
283.
必选项。包含正则表达式模式或可用标志的正则表达式对象。

284.

285.
str
286.
必选项。要在其上测试查找的字符串。
287.

288.
说明
289.
test 方法检查在字符串中是否存在一个模式，如果存在则返回 true，否则就返回 false。
290.
全局 RegExp 对象的属性不由 test 方法来修改。
291.

292.
示例
293.
下面的例子举例说明了 test 方法的用法：
294.
<PRE class=java name="code">function TestDemo(re, s)
295. {
296. var s1; // 声明变量。
297. // 检查字符串是否存在正则表达式。
298. if (re.test(s)) // 测试是否存在。
299. s1 = " contains "; // s 包含模式。
300. else
301. s1 = " does not contain "; // s 不包含模式。
302. return("'' " + s + "'' " + s1 + "'' " + re.source + "'' "); // 返回
字符串。
303. }
304. </PRE>
305.

306.
函数调用：document.write (TestDemo(/ain+/, "The rain in
Spain falls mainly in the plain."));
307.

308.
返回值：'The rain in Spain falls mainly in the plain.'
contains 'ain+'
309.

310.
match 方法
311.
使用正则表达式模式对字符串执行查找，并将包含查找的结果
作为数组返回。\\
312.
stringObj.match(rgExp)
313.

314.
参数\\
315.
stringObj
316.
必选项。对其进行查找的 String 对象或字符串文字。
317.

318.
rgExp
319.
必选项。为包含正则表达式模式和可用标志的正则表达式对象。
也可以是包含正则表达式模式和可用标志的变量名或字符串文字。
320.

321.
说明\\

322.
如果 match 方法没有找到匹配，返回 null。如果找到匹配返回一个数组并且更新全局 RegExp 对象的属性以反映匹配结果。

323.
match 方法返回的数组有三个属性：input、index 和 lastIndex。Input 属性包含整个的被查找字符串。Index 属性包含了在整个被查找字符串中匹配的子字符串的位置。LastIndex 属性包含了最后一次匹配中最后一个字符的下一个位置。

324.
如果没有设置全局标志 (g)，数组的 0 元素包含整个匹配，而第 1 到 n 元素包含了匹配中曾出现过的任一个子匹配。这相当于没有设置全局标志的 exec 方法。如果设置了全局标志，元素 0 到 n 中包含所有匹配。

325.

326.
示例\\

327.
下面的示例演示了 match 方法的用法：

328.
<PRE class=java name="code">function MatchDemo()
329. {
330. var r, re; // 声明变量。
331. var s = "The rain in Spain falls mainly in the plain";
332. re = /ain/i; // 创建正则表达式模式。
333. r = s.match(re); // 尝试匹配搜索字符串。
334. return(r); // 返回第一次出现 "ain" 的地方。
335. }
336. </PRE>

337.
返回值：ain

338.

339.
本示例说明带 g 标志设置的 match 方法的用法。

340.
<PRE class=java name="code">function MatchDemo()
341. {
342. var r, re; // 声明变量。
343. var s = "The rain in Spain falls mainly in the plain";
344. re = /ain/ig; // 创建正则表达式模式。
345. r = s.match(re); // 尝试去匹配搜索字符串。
346. return(r); // 返回的数组包含了所有 "ain"
347. // 出现的四个匹配。
348. }
349. </PRE>

350.
返回值：ain, ain, ain, ain

351.

352.
上面几行代码演示了字符串文字的 match 方法的用法。

353.
<PRE class=java name="code">var r, re = "Spain";
354. r = "The rain in Spain".replace(re, "Canada");
355. return r;
356. </PRE>

357.
返回值：The rain in Canada

358.

359.
search 方法
360.
返回与正则表达式查找内容匹配的的第一个子字符串的位置。
361.

362.
stringObj.search(rgExp)
363.

364.
参数\\
365.
stringObj
366.
必选项。要在其上进行搜索的 String 对象或字符串文字。
367.

368.
rgExp
369.
必选项。包含正则表达式模式和可用标志的正则表达式对象。
370.

371.
说明
372.

373.
search 方法指明是否存在相应的匹配。如果找到一个匹配，
search 方法将返回一个整数值，指明这个匹配距离字符串开始的偏移位置。如果没有找到匹配，则返回 -1。
374.

375.
示例\\
376.
下面的示例演示了 search 方法的用法。
377.
<PRE class=java name="code">function SearchDemo()
378. {
379. var r, re; // 声明变量。
380. var s = "The rain in Spain falls mainly in the plain.";
381. re = /falls/i; // 创建正则表达式模式。
382. r = s.search(re); // 查找字符串。
383. return(r); // 返回 Boolean 结果。
384. }
385. </PRE>
386.
返回值: 18
387.

388.

389.
正则表达式语法
390.
一个正则表达式就是由普通字符（例如字符 a 到 z）以及特殊
字符（称为元字符）组成的文字模式。该模式描述在查找文字主体时待匹
配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所
搜索的字符串进行匹配。
391.

392.
这里有一些可能会遇到的正则表达式示例：
393.
<PRE class=java name="code">JScript VBScript 匹配
394. /^[\ \t]*\$/ ""^[\ \t]*\$" 匹配一个空白行。
395. /\d{2}-\d{5}/ "\d{2}-\d{5}" 验证一个 ID 号码是否由一个 2 位数
字，一个连字符以及一个 5 位数字组成。
396. /<(.*).*<\/\1>/ "<(.*).*<\/\1>" 匹配一个 HTML 标记。

397. </PRE>
398.

399.
下表是元字符及其在正则表达式上下文中的行为的一个完整列表:
400.

401.
字符 描述
402.
\ 将下一个字符标记为一个特殊字符、或一个原义字符、或一个后向引用、或一个八进制转义符。例如, 'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\' 匹配 "\" 而 "\(" 则匹配 "("。
403.

404.
^ 匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性, ^ 也匹配 '\n' 或 '\r' 之后的位置。
405.

406.
\$ 匹配输入字符串的结束位置。如果设置了 RegExp 对象的 Multiline 属性, \$ 也匹配 '\n' 或 '\r' 之前的位置。
407.

408.
* 匹配前面的子表达式零次或多次。例如, zo* 能匹配 "z" 以及 "zoo"。* 等价于 {0,}。
409.

410.
+ 匹配前面的子表达式一次或多次。例如, 'zo+' 能匹配 "zo" 以及 "zoo", 但不能匹配 "z"。+ 等价于 {1,}。
411.

412.
? 匹配前面的子表达式零次或一次。例如, "do(es)?" 可以匹配 "do" 或 "does" 中的 "do"。? 等价于 {0,1}。
413.

414.
{n} n 是一个非负整数。匹配确定的 n 次。例如, 'o{2}' 不能匹配 "Bob" 中的 'o', 但是能匹配 "food" 中的两个 o。
415.

416.
{n,} n 是一个非负整数。至少匹配 n 次。例如, 'o{2,}' 不能匹配 "Bob" 中的 'o', 但能匹配 "foooooo" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
417.

418.
{n,m} m 和 n 均为非负整数, 其中 n <= m。最少匹配 n 次且最多匹配 m 次。例如, "o{1,3}" 将匹配 "foooooo" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。
419.

420.
? 当该字符紧跟在任何一个其他限制符 (*, +, ?, {n}, {n,}, {n,m}) 后面时, 匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串, 而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如, 对于字符串 "oooo", 'o+?' 将匹配单个 "o", 而 'o+' 将匹配所有 'o'。
421.

422.
. 匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符, 请使用象 '[.\n]' 的模式。

423.
(pattern) 匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到, 在 VBScript 中使用 SubMatches 集合, 在 JScript 中则使用 \$0...\$9 属性。要匹配圆括号字符, 请使用 '\(' 或 '\)'。
424.

425.
(?:pattern) 匹配 pattern 但不获取匹配结果, 也就是说这是一个非获取匹配, 不进行存储供以后使用。这在使用“或”字符 (|) 来组合一个模式的各个部分是很有用。例如, 'industr(?:y|ies)' 就是一个比 'industry|industries' 更简略的表达式。
426.

427.
(?=pattern) 正向预查, 在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配, 也就是说, 该匹配不需要获取供以后使用。例如, 'Windows (?!95|98|NT|2000)' 能匹配“Windows 2000”中的“Windows”, 但不能匹配“Windows 3.1”中的“Windows”。预查不消耗字符, 也就是说, 在一个匹配发生后, 在最后一次匹配之后立即开始下一次匹配的搜索, 而不是从包含预查的字符之后开始。
428.

429.
(?!pattern) 负向预查, 在任何不匹配 Negative lookahead matches the search string at any point where a string not matching pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配, 也就是说, 该匹配不需要获取供以后使用。例如 'Windows (?!95|98|NT|2000)' 能匹配“Windows 3.1”中的“Windows”, 但不能匹配“Windows 2000”中的“Windows”。预查不消耗字符, 也就是说, 在一个匹配发生后, 在最后一次匹配之后立即开始下一次匹配的搜索, 而不是从包含预查的字符之后开始
430.

431.
x|y 匹配 x 或 y。例如, 'z|food' 能匹配“z”或“food”。'(z|f)ood' 则匹配“zood”或“food”。
432.

433.
[xyz] 字符集合。匹配所包含的任意一个字符。例如, '[abc]' 可以匹配“plain”中的'a'。
434.

435.
[^xyz] 负值字符集合。匹配未包含的任意字符。例如, '[^abc]' 可以匹配“plain”中的'p'。
436.

437.
[a-z] 字符范围。匹配指定范围内的任意字符。例如, '[a-z]' 可以匹配'a'到'z'范围内的任意小写字母字符。
438.

439.
[^a-z] 负值字符范围。匹配任何不在指定范围内的任意字符。例如, '[^a-z]' 可以匹配任何不在'a'到'z'范围内的任意字符。
440.

441.
\b 匹配一个单词边界, 也就是指单词和空格间的位置。例如, 'er\b' 可以匹配“never”中的'er', 但不能匹配“verb”中的'er'。
442.

443.
\B 匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
444.

445.
\cx 匹配由 x 指明的控制字符。例如，\cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
446.

447.
\d 匹配一个数字字符。等价于 [0-9]。
448.

449.
\D 匹配一个非数字字符。等价于 [^0-9]。
450.

451.
\f 匹配一个换页符。等价于 \x0c 和 \cL。
452.

453.
\n 匹配一个换行符。等价于 \x0a 和 \cJ。
454.

455.
\r 匹配一个回车符。等价于 \x0d 和 \cM。
456.

457.
\s 匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
458.

459.
\S 匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
460.

461.
\t 匹配一个制表符。等价于 \x09 和 \cI。
462.

463.
\v 匹配一个垂直制表符。等价于 \x0b 和 \cK。
464.

465.
\w 匹配包括下划线的任何单词字符。等价于 '[A-Za-z0-9_]'
466.

467.
\W 匹配任何非单词字符。等价于 '[^A-Za-z0-9_]'
468.

469.
\xn 匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，'\x41' 匹配 "A"。'\x041' 则等价于 '\x04' & "1"。正则表达式中可以使用 ASCII 编码。。
470.

471.
\num 匹配 num，其中 num 是一个正整数。对所获取的匹配的引用。例如，'(.)\1' 匹配两个连续的相同字符。
472.

473.
\n 标识一个八进制转义值或一个后向引用。如果 \n 之前至少 n 个获取的子表达式，则 n 为后向引用。否则，如果 n 为八进制数字 (0-7)，则 n 为一个八进制转义值。
474.

475.
\nm 标识一个八进制转义值或一个后向引用。如果 \nm 之前至少有 is preceded by at least nm 个获取得子表达式，则 nm 为后向引用。如果 \nm 之前至少有 n 个获取，则 n 为一个后跟文字 m 的后向引

用。如果前面的条件都不满足,若 n 和 m 均为八进制数字 (0-7),则 $\backslash nm$ 将匹配八进制转义值 nm 。

- 476. `
`
- 477. `
\nml` 如果 n 为八进制数字 (0-3), 且 m 和 l 均为八进制数字 (0-7), 则匹配八进制转义值 nml 。
- 478. `
`
- 479. `
\un` 匹配 n , 其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如, `\u00A9` 匹配版权符号 (?)。
- 480. `
`
- 481. `
`
- 482. `
优先权顺序`
- 483. `
`在构造正则表达式之后, 就可以象数学表达式一样来求值, 也就是说, 可以从左至右并按照一个优先权顺序来求值。
- 484. `
`
- 485. `
`下表从最高优先级到最低优先级列出各种正则表达式操作符的优先权顺序:

486.	<code>
<PRE class=java name="code"></code> 操作符 描述
487.	<code>\</code> 转义符
488.	<code>()</code> , <code>(?:)</code> , <code>(?=)</code> , <code>[]</code> 圆括号和方括号
489.	<code>*</code> , <code>+</code> , <code>?</code> , <code>{n}</code> , <code>{n,}</code> , <code>{n,m}</code> 限定符
490.	<code>^</code> , <code>\$</code> , <code>\anymetacharacter</code> 位置和顺序
491.	<code> </code> “或” 操作
492.	<code></PRE></code>
- 493. `
`
- 494. `
`普通字符
- 495. `
`
- 496. `
`普通字符由所有那些未显式指定为元字符的打印和非打印字符组成。这包括所有的大写和小写字母字符, 所有数字, 所有标点符号以及一些符号。
- 497. `
`
- 498. `
`最简单的正则表达式是一个单独的普通字符, 可以匹配所搜索字符串中的该字符本身。例如, 单字符模式 `'A'` 可以匹配所搜索字符串中任何位置出现的字母 `'A'`。这里有一些单字符正则表达式模式的示例:

499.	<code>
<PRE class=java name="code">/a/</code>
500.	<code>/7/</code>
501.	<code>/M/</code>
502.	<code></PRE></code>
- 503. `
`等价的 VBScript 单字符正则表达式为:
- 504. `
<PRE class=java name="code">"a"`
- 505. `"7"`
- 506. `"M"`
- 507. `</PRE>`

508.
可以将多个单字符组合在一起得到一个较大的表达式。例如，下面的 JScript 正则表达式不是别的，就是通过组合单字符表达式 'a'、'7' 以及 'M' 所创建出来的一个表达式。
509.

510.
/a7M/
511.
等价的 VBScript 表达式为：
512.

513.
"a7M"
514.
请注意这里没有连接操作符。所需要做的就是将一个字符放在了另一个字符后面。
515.

516.

^ The caret (^) tells the regular expression that the character must not match the characters to follow.

比如要匹配所有除了 a 或 b 的字符, 字符类可以这么写 [^ab]

^ 定位符规定匹配模式必须出现在目标字符串的开头

那是否说 ^ 在 [] 里面 就表示**排除(负向类)** , 在 [] 前面 就表示要**在开头** ?

\$ 定位符规定匹配模式必须出现在目标对象的**结尾**

- 范围类, 如要匹配 a 到 z 的所有字母, 字符类可以这么写 [a-z]

{n} 出现多少次

/^[0-9]{7}\$/ 匹配的就是一个仅包含 7 个数字的字符串

7.1 RegExp 对象可以有一个或两个参数.

```
var reCat = new RegExp ("cat") //这个表达式只会匹配字符串中出现的第一个"cat"
```

```
var reCat = new RegExp ("cat","g") //匹配所有出现的"cat"
```

正则表达式字面量

```
var reCat = /cat/gi ; 注意字面量不需要放在引号里
```

判断某个字符串是否匹配指定的模式

`RegExp.test(str)` 如果给定的字符串匹配这个模式, 返回 `true` , 否则返回 `false` . `reCat.test("cat")` //返回 `true`

`RegExp.exec(str)` 返回一个数组 , 数组中第一个元素是匹配字符串, 其他是反向引用 . 如果没有找到匹配, 返回 `null`

返回的数组有一个 `index` 属性, 这个属性的值是匹配字符串中第一个字符在原字符串中的下标

```
var toMarch6 = "1 3";
var regExp6 = /(\d+)\s*(\d+)/;
var result = regExp6.exec(toMarch6);
dwr(result); //1 3,1,3
dwr(RegExp.$1 + " " + RegExp.$2); //1 3
```

当正则表达式加上了参数 `g` , 这个正则表达式会从正则表达式对象的

`lastIndex` 属性指定的位置开始查找, 如果找到一个匹配,

会将 `lastIndex` 属性设置为匹配字符串后面一个位置的下标.

```
var toMarch7 = "1 3 4 5";
var regExp7 = /(\d+)\s*(\d+)/g;
var result7
while((result7= regExp7.exec(toMarch7)) !=null){
dwr(result7); //分别为 1 3,1,3 和 4 5,4,5

dwr(regExp7.lastIndex); //分别为 3 和 7
}
```

`String.match(reCat)` 返回一个包含在字符串中的**所有匹配** 的**数组**

```
var toMatch = "a bat, a cat, a fAt baT, a faT cat";
```

```
var reAt = /at/gi; //如果不加参数 g, 返回的数组只会包含一个匹配元素
```

```
var arrMatches = toMatch.match(reAt);
```

返回一个数组 `["at", "at", "At", "aT", "aT", "at"]`

`String.search(reCat)` 与 `indexOf` 类似, 返回在字符串中出现的**第一个** 匹配

的**位置**, 全局匹配表达式 `g` 在这里**不起作用** `toMatch.search(reAt);` //输

出 3

`String.replace(matchStr, replaceStr)`

用第二个参数**替换** 某个子串(第一个参数)的**所有匹配** . 第一个参数 既可以是匹配的**字符串**, 也可以是用于匹配的一个**正则表达式**, 返回是替换后的整个字符串

```
var sToChange = "The sky is red";
```

```
var reRed = /red/;
```

```
sToChange.replace(reRed, "blue" ); //输出 The sky is blue
```

`String.split(reCat)` 将字符串**分割** 成子串, 作为**数组** 返回

```
var sColor = "red, blue, yellow";
```

```
var reExp = /\,/;
```

```
var arrs = sColor.split(reExp); //得到一个数组
```

```
["red", "blue", "yellow" ]
```

 注意**逗号** 在正则表达式有

特殊含义, 这边需要**转义**

7.2 简单模式(元字符, 字符类, 量词)

元字符 11 个 `() [] {} \ ^ $ | ? * + .` . 要匹配字符串中的元字符, 需要**转义** `\/`

注意 `var reMark = new RegExp("\\^");` 当正则表达式以**非字面量的形式** 表示时, 所有的反斜杠 `"\"` 都要用两个反斜杠 `"\\"` 来替换.

因为 javascript 字符串解析器会按照翻译 `\n` 的方式尝试翻译 `\?`. 为了保证不会出现这个问题, 在元字符的前面加上两个反斜杠,

我们称之为**双重转义**. (不太懂这个解释, 我的理解是 `\` 本身也是元字符, 先要对它转义得到 `"\"`, 然后再用这个 `"\"` 对接下来的元字符转义)

预定义的特殊字符	
\t	制表符
\n	换行符
\r	回车符
\f	换页符
\a	Alert 字符
\e	Escape 字符
\cX	与 X 相应的控制字符
\b	回退字符
\v	垂直制表符
\o	空字符

字符类 将一些字符放入方括号中。

1, 简单类

```
var toMatch = "a bat, a cat, a fAt baT, a faT cat";
var reg = /[bcf]at/gi ;

var arrs = toMatch.match(reg); 返回的数组[“bat”, “Cat”,
“fAt”, “baT”, “faT”, “cat”]
```

2, 负向类 可以指定要排除的字符，^ 要在[] 里面

匹配除了 a 和 b 以外的所有字符, 那么这个字符类为`[^ab]`。脱字符`^`告诉正则表达式字符不能匹配后面跟着的字符
只想获得包含 at 但不能以 b 或 c 开头的字符, `/[^bc]at/gi`

3, 范围类

`[a-z]` 匹配所有小写字母

结合负向类可以排除给定范围内的所有字符，例如要排除字符 1~4, 可以使用类`[^1-4]`

4, 组合类

`[a-z1-9\n]`

5, 预定义类

代码 等同于 匹配

. `[^\n\r]` 除了换行回车意外的任意字符

`\d` `[0-9]` 数字

`\D` `[^0-9]` 非数字

`\s` `[\t\n\r\x0B\f]` 空白字符

`\S` `[^\t\n\r\x0B\f]` 非空白字符

`\w` `[a-zA-Z_0-9]` 单词字符(所有字母, 数字和下划线)

`\W` `[^a-zA-Z_0-9]` 非单词字符

6, 量词

用于指定某个特定模式出现的次数

`?` 出现 0 次或 1 次

`*` 出现 0 次或多次(任意次)

`+` 出现 1 次或多次(至少出现一次)

`{n}` 一定出现 n 次

`{n,m}` 至少出现 n 次但不超过 m 次

`{n,}` 至少出现 n 次

贪婪的, 惰性的, 支配性的量词

贪婪量词 先看整个字符串是否匹配, 如果没有发现匹配, 它去掉该字符串中最后一个字符, 并再次尝试. 重复这个过程直到发现匹配或者字符串不剩任何字符.

结合下面会见到的两个例子, 实际过程可能是这样的:

第一步按上面的描述执行, 如果字符串不剩任何字符还是没有找到一个匹配, 那么删除字符串的第一个字符, 重复第一步.

惰性量词 先看字符串中第一个字符是否匹配. 如果单独这个字符还不够, 就读入下一个字符, 组成两个字符的字符串. 如果还是没有发现匹配, 惰性量词继续从字符串中添加字符直到发现匹配或者整个字符串都检查过也没有匹配. 与贪婪量词的工作方式正好相反.

支配量词 只尝试匹配整个字符串. 如果整个字符串不能产生匹配, 不做进一步尝试. (IE 不支持, Mozilla 把支配量词当做贪婪的)

贪婪 惰性 支配 描述

`?? ?+` 零次或一次出现

`* *? *+` 零次或多次出现

+ +? ++ 一次或多次出现

{n} {n}? {n}+ 恰好 n 次出现

{n,m} {n,m}? {n,m}+ 至少 n 次之多 m 次出现

{n,} {n,}? {n,}+ 至少 n 次出现

7.3 复杂模式

1, 分组

分组是通过一系列括号包围一系列字符, 字符类以及量词来使用的.

`/(dog){2}/g` 匹配 dogdog

`/[bd]ad?*/` 匹配 ba, da, bad, dad 等

去掉开头结尾的空白字符

```
String.prototype.trim = function() {
```

//注意, 分组里面.?一定要是惰性的, 不然这个分组会把最后的空白字符也*

匹配进去

```
var reExtraStr = /^\\s+(.\\*?)\\s+$/ ;  
return this.replace(reExtraStr, "$1");  
}
```

2, 反向引用

在表达式计算完成之后, 每个分组都被存放在一个特殊的地方以备将来使用. 这些存储在分组中的特殊值, 我们称之为反向引用 (backreference).

反向引用是按照从左到右遇到的左括号字符的顺序进行创建和编号的.

反向引用的几种不同使用方法:

** 使用正则表达式对象的 test(), exec() 方法后, 反向引用的值可以从 RegExp 对象的构造函数中获得*

```
var toMarch = "#12345";  
var regExp = /#(\\d*)/;  
regExp.exec(toMarch);  
dwr(RegExp.$1);
```

** 还可以直接在定义分组的表达式中包含反向引用, 这可以通过使用特殊转义序*

列如 \\1 , \\2 等实现.

```
var toMarch2 = "dogdog";
```

```

var regExp2 = /(dog)\1/;
dwr(regExp2.test(toMarch2));
* 反向引用可以用在 String 对象的 replace() 方法中
var toMarch3 = "123123123 3211231231";
var regExp3 = /(\d+)\s*(\d+)/;

var sNew = toMarch3.replace(regExp3, "$2 $1 "); //注意 replace
不改变原来的字符串, 而是返回一个替换后的新字符串
dwr(sNew);

```

3, 候选

用一个管道符(|), 它放在两个单独的模式之间.

```

var toMarch4 = "dog";
var toMarch5 = "cat";
var regExp4 = /dog|cat/;
dwr(regExp4.test(toMarch4)); //true
dwr(regExp4.test(toMarch5)); //true

```

OR 模式的一种通常用法是从用户输入中删除不合适的单词.

```

var userInput = "badWord1asdasdandBadWord2";
var toMarch6 = /badword1|badword2/gi;
var newStr = userInput.replace(toMarch6, function(march) {
return march.replace(/./g, "*");
});
dwr(newStr); //*****asdand*****

```

`string.replace(regexp, replacement)`

`replacement` 既可以是一个替换的字符串, 也可以是一个 function

如果是 function 的情况, 这个 function 会为每一个匹配执行一次, 这个 function 的返回值作为最终替换的字符串.

传给 function 的第一个参数是第一个匹配的字符串;

第二个参数是匹配字符串在原始字符串中的位置;

第三个参数是原始字符串本身.

4, 非捕获性分组

创建反向引用的分组称为捕获性分组, 非捕获性分组不会创建反向引用.

在较长的正则表达式中存储反向引用会降低匹配的速度.

要创建非捕获性分组, 只要在左括号后面紧跟一个问号和冒号.

```

var str = "#123456";
var regE = /#(?: 123456)/;
regE.test(str);
dwr(RegExp.$1); //""

```

//去除所有的 HTML 标签

```
String.prototype.skipHTML = function(){  
var regExp = /<(?:.|\s)*?>/g ;  
return this.replace(regExp, "");  
}
```

5, 前瞻

表示当某个特定的字符分组出现在另一个字符串之前时才去捕获它.

前瞻分**正向前瞻** 和**负向前瞻** , 正向前瞻检查的是接下来出现的是不是某个特定的字符集. 而负向前瞻则是检查接下来的不应该出现的特定字符集.

正向前瞻需要将模式放在(=? 和) 之间, 注意这不是分组, 虽然它也用到括号.

负向前瞻需要将模式放在(!= 和) 之间.

```
var toMarch1 = "bedroom";  
var toMarch2 = "bedding";  
var bedReg = /(bed(=?room))/;  
dwr(bedReg.test(toMarch1)); //true
```

dwr(bedReg.exec(toMarch1)); //bed,bed 因此这个正则表达式返回的

第一个匹配是bed, 而不是 bedroom, 但是它只会去匹配后面跟着 room 的 bed,

有点搞

```
dwr(RegExp.$1); //bed  
dwr(bedReg.test(toMarch2)); //false
```

6, 边界

边界用于正则表达式中表示模式的位置.

^ 行开头

\$ 行结尾

\b 单词的边界

\B 非单词的边界

查找一个出现在行尾的单词:

```
var toMarch3 = "Important word is the last one.";  
var regExp3 = /(\w+)\.$/ ; //这边结合上面对贪婪量词的解释, 有点不  
明白为何能匹配 one?  
regExp3.test(toMarch3);  
dwr(RegExp.$1); //one
```

查找一个出现在行首的单词:

```
var toMarch4 = "Important word is the last one.";

var regExp4 = /^(\w+)/ ; //或者 var regExp4 = /^(.+) \b/;

regExp4.test(toMarch4);

dwr(RegExp.$1);
```

抽取出所有的单词

```
var toMarch5 = "First Second Third Fourth Fifth Sixth";

var regExp5 = /\b(\S+)\b/g ; //或者 /\b(\S+)\b/g 和 /(\w+)/g

var sArr = toMarch5.match(regExp5);

dwr(sArr); //First Second Third Fourth Fifth Sixth
```

注意这边如果用 /(\w+)/g 是不行的, 这样得到的是一个一个的字母

F,i,r,s,t,S...

=====

与上面疑问类似的一个问题

例子代码, 如下:

```
str = "abbb1234abbbbaabbbbaabbb1234";

re = /. *bbb/g;

alert(str.match(re)); //结果为 abbb1234abbbbaabbbbaabbb

re = /. *?bbb/g;

alert(str.match(re)); //结果为 abbb,1234abbb,aabbb,aaabbb

re = /a *bbb/g;

alert(str.match(re)); //结果为 abbb,abbb,aabbb,aaabbb

re = /a *?bbb/g;

alert(str.match(re)); //结果为 abbb,abbb,aabbb,aaabbb
```

对于第一、第二和第四个打印结果容易理解:

第一个 str.match(re), 贪婪量词先匹配整个字串, 若不匹配去掉一个尾字符, 继续匹配;

第二个 str.match(re), 惰性量词从第一个字符开始递加去匹配, 直到匹配成功, 清空字串, 从下一个字符继续匹配。

第四个 str.match(re), 同第二个。

但第三个就不知道如何解释, 如果按照第一个的方式去理解:

先匹配整个字符串，发现不匹配，去掉尾字符，继续匹配...到最后，结果应该是 abbb;
而其结果却为 abbb, abbb, aabbb, aaabbb

以下为论坛解释

对于第三个正则，就是这样来执行的：

首先清楚了是用了简单量词(*), 而我们知道了*是贪婪量词：

贪婪量词执行过程。正好楼主所说的那样。“先匹配整体，若不匹配则去掉尾字符继续匹配，直到成功或者结束”

这样说应说只能得到第一被匹配的对象。

javascript 中的 match 返回的是所有匹配。

对于要返回所有匹配。

它还有第二个步：**就是匹配成功后，从最近的一个匹配后的下一个字符开始重新贪婪模式匹配。** 重新执行它的步骤；

例：

```
str = "abbb1234abbbbaabbbbaabbb1234";  
re = /a*bbb/g;  
alert(str.match(re));
```

它的执行过程：

第一步：首先整个字符串("abbb1234abbbbaabbbbaabbb1234")匹配, 发现匹配不成功，

接着。删除最后一个字符("4"), 成了("abbb1234abbbbaabbbbaabbb123"), 这样依次执行下去；

执行...最后，发现("abbb")可以被匹配了..所以生成第一个匹配值。

但在这个 match 方法中是返回所有匹配。所以..

第二步：从最近的一个匹配（这里就是第一次匹配了）后的下一个字符开始重新贪婪模式匹配. 得到字符串是 ("1234abbbbaabbbbaabbb1234"), 然后。就按第一步执行。。

执行完第一步后。

然后就从最近一次（这里就是第二次匹配了）

....后面的过程就是重复一二步了。。

但第二步时若继续按正则/a*bbb/g 去匹配 "1234abbbbaabbbbaabbb1234" 的话，应该是匹配不到才对吧？

怎么匹配不到呢。。

正则表达式执行的时候。首先得找到前导字符(a)，a 是一个普通字符。普通字符，搜索的顺序为从左到右。。

所以搜索 "1234abbbbaabbbbaabbb1234" 字符串时，

得先匹配出 a 字符来“abbbaabbbbaaabb1234”，而解析器又发现了 a 后面是一个贪婪字符。就按贪婪模式去匹配(从右到左)

注意： /a*bbb/g 用到了全局匹配，以上分析的症结所在可能就是因为一个

"g"

7, 多行模式

一下代码中的正则表达式想要匹配行末的一个单词. 它只会匹配最后的 Sixth, 但实际上这个字符串包含两个换行符, 因此, Second, Fourth 也应该匹配出来, 因此引入了多行模式

```
var toMarch6 = "First Second\nThird Fourth\nFifth Sixth";
var regExp6 = /(\w+)$/g;
var sArr6 = toMarch6.match(regExp6);
dwr(sArr6);
```

要引入多行模式, 需要在正则表达式后面添加 **m** 选项, 这会让\$边界匹配换行符(\n) 和字符串真正的结尾.

```
var regExp6 = /(\w+)$/gm;
```

判断日期的正则表达式:

```
function isValidDate(s) {
var reDate =
/((?:[1-9]|0[1-9]|12)[0-9]|3[01])\/(?:[1-9]|0[1-9]|1[0-2])
)\.\/(?:19\d{2}|20\d{2})/;
return reDate.test(s);
}
```

首页 | 常用正则表达式 | 正则表达式测试工具 | 网络信息采集服务

正则表达式 30 分钟入门教程

版本: v2.32 (2011-8-17) 作者: [deerchao](#) 转载请注明[来源](#)

目录

[跳过目录](#)

1. 本文目标

2. [如何使用本教程](#)
3. [正则表达式到底是什么东西？](#)
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. [分枝条件](#)
11. [反义](#)
12. [分组](#)
13. [后向引用](#)
14. [零宽断言](#)
15. [负向零宽断言](#)
16. [注释](#)
17. [贪婪与懒惰](#)
18. [处理选项](#)
19. [平衡组/递归匹配](#)
20. [还有些什么东西没提到](#)
21. [联系作者](#)
22. [网上的资源及本文参考文献](#)
23. [更新纪录](#)

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——除非你是超人 :)

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？

清除格式 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) 对其进行匹配的源字符串 对正则表达式或其中一部分的说明

隐藏边注 本文右边有一些注释，主要是用来提供一些相关信息，或者给没有程序员背景的读者解释一些基本概念，通常可以忽略。

正则表达式到底是什么东西？

字符是计算机软件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**是 0 个或更多个字符的序列。**文本**也就是文字，字符串。说某个字符串**匹配**某个正则表达式，通常是指这个字符串里有一部分（或几部分分别）能满足表达式给出的条件。

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）。

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**。

这几乎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，标点符号或者换行来分隔的，但是 **\b** 并不匹配这些单词分隔字符中的任何一个，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使用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。

换行符就是 **'\n'**，ASCII 编码为 10 (十六进制 0x0A) 的字符。

如果同时使用其它元字符，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** 是个新的元字符，匹配一位数字(0, 或 1, 或 2, 或……)。**-** 不是元字符，只匹配它本身——连字符(或者减号，或者中横线，或者随你怎么称呼它)。

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

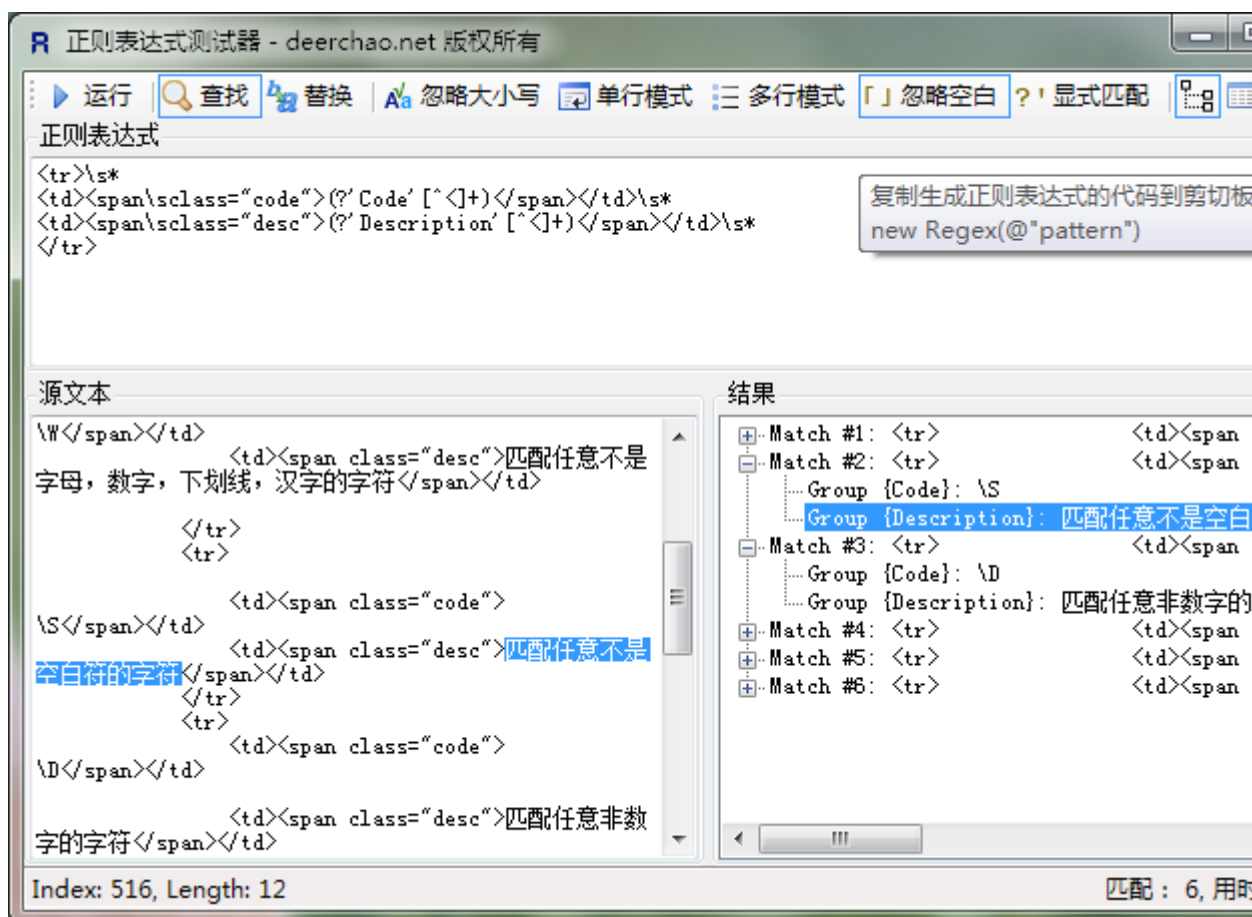
其它可用的测试工具：

- [RegexBuddy](#)
- [Javascript 正则表达式在线测试工具](#)

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，所以找一种工具对正则表达式进行测试是很有必要的。

不同的环境下正则表达式的一些细节是不相同的，本教程介绍的是微软 .Net Framework 4.0 下正则表达式的行为，所以，我向你推荐我编写的 .Net 下的工具 [正则表达式测试器](#)。请参考该页面的说明来安装和运行该软件。

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`. 正则表达式里还有更多的元字符, 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等。

对中文/汉字的特殊处理是由 .Net 提供的正则表达式引擎支持的, 其它环境下的具体情况请查看相关文档。

下面来看看更多的例子:

`\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: 就是不少于一个的连续的 `\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 个或多个连续的数字。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 个字符的单词。

表 1.常用的元字符	
代码	说明
<code>.</code>	匹配除换行符以外的任意字符
<code>\w</code>	匹配字母或数字或下划线或汉字
<code>\s</code>	匹配任意的空白符
<code>\d</code>	匹配数字
<code>\b</code>	匹配单词的开始或结束
<code>^</code>	匹配字符串的开始
<code>\$</code>	匹配字符串的结束

正则表达式引擎通常会提供一个“测试指定的字符串是否匹配一个正则表达式”的方法，如 JavaScript 里的 `RegExp.test()` 方法或 .NET 里的 `Regex.IsMatch()` 方法。这里的匹配是指是字符串里有没有符合表达式规则的部分。如果不使用`^`和`$`的话，对于`\d{5,12}`而言，使用这样的方法就只能保证字符串里包含 5 到 12 连续位数字，而不是整个字符串就是 5 到 12 位数字。

元字符`^`（和数字 6 在同一个键位上的符号）和`$`都匹配一个位置，这和`\b`有点类似。`^`匹配你要用来查找的字符串的开头，`$`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：你没办法指定它们，因为它们会被解释成别的意思。这时你就得使用`\`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：`deerchao\\.net` 匹配 deerchao.net，`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
<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 <code>n</code> 次
<code>{n,}</code>	重复 <code>n</code> 次或更多次
<code>{n,m}</code>	重复 <code>n</code> 到 <code>m</code> 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

`^\\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，你只需要在方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!)。

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-zA-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}`。

“(”和“)”也是元字符，后面的[分组节](#)里会提到，所以在这里需要使用[转义](#)。

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})。

分枝条件

不幸的是，刚才那个表达式也能匹配010)12345678或(022-87654321这样的“不正确”的格式。要解决这个问题，我们需要用到[分枝条件](#)。正则表达式里的[分枝条件](#)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。你可以试试用分枝条件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：**使用分枝条件时，要注意各个条件的顺序**。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)。原因是匹配分枝条件时，将会从左到右地测试每个条件，如果满足了某个分枝的话，就不会去再管其它的条件了。

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}`)。

IP 地址中每个数字都不能大于 255，大家千万不要被《24》第三季的编剧给忽悠了……

不幸的是，它也将匹配 256.300.888.999 这种不可能存在的 IP 地址。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它的意义。

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

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，以分组的左括号标志，第一个出现的分组的组号为 1，第二个为 2，以此类推。

呃……其实，组号分配还不像我刚说得那么简单：

- 分组 0 对应整个正则表达式
- 实际上组号分配过程是要从左向右扫描两遍的：第一遍只给未命名组分配，第二遍只给命名组分配——因此所有命名组的组号都大于未命名的组号
- 你可以使用 `(?:exp)` 这样的语法来剥夺一个分组对组号分配的参与权。

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`，或者 `kitty kitty`。这个表达式首先是一个单词，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)`，这个单词会被捕获到编号为 1 的分组中，然后是**1 个或几个空白符** `(\s+)`，最后是**分组 1 中捕获的内容（也就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+)`（或者把尖括号换成单引号也行：`(?'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.常用分组语法		
分类	代码/语法	说明
捕获	<code>(exp)</code>	匹配 <code>exp</code> ，并捕获文本到自动命名的组里
	<code>(?<name>exp)</code>	匹配 <code>exp</code> ，并捕获文本到名称为 <code>name</code> 的组里，也可以写成 <code>(?'name'exp)</code>
	<code>(?:exp)</code>	匹配 <code>exp</code> ，不捕获匹配的文本，也不给此分组分配组号
零宽断言	<code>(?=exp)</code>	匹配 <code>exp</code> 前面的位置
	<code>(?<=exp)</code>	匹配 <code>exp</code> 后面的位置
	<code>(?!exp)</code>	匹配后面跟的不是 <code>exp</code> 的位置
	<code>(?<!=exp)</code>	匹配前面不是 <code>exp</code> 的位置
注释	<code>(?#comment)</code>	这种类型的分组不对正则表达式的处理产生任何影响，用于提供注释

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，也不会拥有组号。“我为什么会想要这样做？”——好问题，你觉得为什么呢？

零宽断言

地球人，是不是觉得这些术语名称太复杂，太难记了？我也有同感。知道有这么一种东西就行了，它叫什么，随它去吧！人若无名，便可专心练剑；物若无名，便可随意取舍……

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(即断言)，因此它们也被称为**零宽断言**。最好还是拿例子来说明吧：

断言用来声明一个应该为真的事实。正则表达式中只有当断言为真时才会继续进行匹配。

`(?=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)`, 零宽度负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。

请详细分析表达式 `(?<=<(\w+)>).*?(?=<\/\1>)`, 这个表达式最能表现零宽断言的真正用途。

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通过语法 `(?#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：

```
(?<=      # 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。以这个表达式为例：
`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（第一到第三个字符）和 ab（第四到第五个字符）。

为什么第一个匹配是 aab（第一到第三个字符）而不是 ab（第二到第三个字符）？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	
代码/语法	说明

表 5.懒惰限定符	
代码/语法	说明
*?	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

在 C#中，你可以使用 `Regex(String, RegexOptions)` 构造函数来设置正则表达式的处理选项。如：`Regex regex = new Regex(@"\ba\w{6}\b", RegexOptions.IgnoreCase);`

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。
ExplicitCapture(显式捕获)	仅捕获已被显式命名的组。

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

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(Stack)**
- (?-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（或者你自称程序员但是不知道堆栈是什么东西），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，就在压入一个"Open"，每碰到一个右括号，就弹出一个，到了最后就看看堆栈是否为空——如果不为空那就证明左括号比右括号多，那匹配就应该失败。正则表达式引擎会进行回溯(放弃最前面或最后面的一些字符)，尽量使整个表达式得到匹配。

```
<                                     #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<div>标签:

```
<div[<]*[>]*(((?'Open' <div[<]*)[>]*+((?'-Open' </div>)[<]*)+)*(? (Open) (?!))</div>.
```

还有什么东西没提到

上边已经描述了构造正则表达式的大量元素, 但是还有很多没有提到的东西。下面是一些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 .net 下正则表达式详细的文档。

这里的介绍很简略, 如果你需要更详细的信息, 而又没有在电脑上安装 MSDN Library, 可以查看[关于正则表达式语言元素的 MSDN 在线文档](#)。

表 7.尚未详细讨论的语法

代码/语法	说明
\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
\Onn	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

表 7.尚未详细讨论的语法	
代码/语法	说明
(?:<x>-<y>exp)	平衡组
(?:im-nsx:exp)	在子表达式 <code>exp</code> 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 <code>exp</code> 当作零宽正向先行断言，如果在这个位置能匹配，使用 <code>yes</code> 作为此组的表达式；否则使用 <code>no</code>
(?:exp)yes	同上，只是使用空表达式作为 <code>no</code>
(?:name)yes no	如果命名为 <code>name</code> 的组捕获到了内容，使用 <code>yes</code> 作为表达式；否则使用 <code>no</code>
(?:name)yes	同上，只是使用空表达式作为 <code>no</code>

联系作者

好吧, 我承认, 我骗了你, 读到这里你肯定花了不少 30 分钟. 相信我, 这是我的错, 而不是因为你太笨. 我之所以说“30 分钟”, 是为了让你有信心, 有耐心继续下去. 既然你看到了这里, 那证明我的阴谋成功了. 被忽悠的感觉很爽吧?

要投诉我, 或者觉得我其实可以忽悠得更高明, 或者有任何其它问题, 欢迎来[我的博客](#)让我知道.

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 \(Second Edition\)](#)

更新纪录

1. 2006-3-27 第一版
2. 2006-10-12 第二版
 - 修正了几个细节上的错误和不准确的地方
 - 增加了对处理中文时的一些说明
 - 更改了几个术语的翻译 (采用了 MSDN 的翻译方式)
 - 增加了平衡组的介绍
 - 放弃了对 The Regulator 的介绍, 改用 Regex Tester
3. 2007-3-12 V2.1
 - 修正了几个小的错误
 - 增加了对处理选项(RegexOptions)的介绍
4. 2007-5-28 V2.2
 - 重新组织了对零宽断言的介绍

- 删除了几个不太合适的示例，添加了几个实用的示例
- 其它一些微小的更改
- 5. 2007-8-3 V2.21
 - 修改了几处文字错误
 - 修改/添加了对\$, \b 的精确说明
 - 承认了作者是个骗子
 - 给 RegexTester 添加了 Singleline 选项的相关功能
- 6. 2008-4-13 v2.3
 - 调整了部分章节的次序
 - 修改了页面布局，删除了专门的参考节
 - 针对读者的反馈，调整了部分内容
- 7. 2009-4-11 v2.31
 - 修改了几处文字错误
 - 添加了一些注释说明
 - 调整了一些措词
- 8. 2011-8-17 v2.32
 - 更改了工具介绍，换用自行开发的正则表达式测试器

Validated XHTML 1.0 Strict Validated CSS 2.1