轻松学习正则表达式

极客学院出版

前言

正则表达式（英文：Regular Expression）在计算机科学中，是指一个用来描述或者匹配一系列符合某个句法规则的字符串的单个字符串。通过本教程的学习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。

适用群体

在程序开发中,难免会遇到需要匹配、查找、替换、判断字符串的情况发生,而这些情况有时又比较复杂,如果用纯编码方式解决,往往会浪费程序员的时间及精力。因此,学习及使用正则表达式,便成了解决这一矛盾的主要手段。

预备知识

本教程是初级教程，学习本教程之前，你需要了解一门或多门编程语言，如 Java、C、JavaScript 等。

鸣谢：http://www.jb51.net/tools/zhengze.html

目录

1

正则表达式基础

什么是正则表达式

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

很可能你使用过 Windows/Dos 下用于文件查找的通配符(wildcard)，也就是 `*` 和 `？` 。如果你想查找某个目录下的所有的 Word 文档的话，你会搜索 `*.doc` 。在这里， `*` 会被解释成任意的字符串。和通配符类似，正则表达式也是用来进行文本匹配的工具，只不过比起通配符，它能更精确地描述你的需求——当然，代价就是更复杂——比如你可以编写一个正则表达式，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)。

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。

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。

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。

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。

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)。

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

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)。

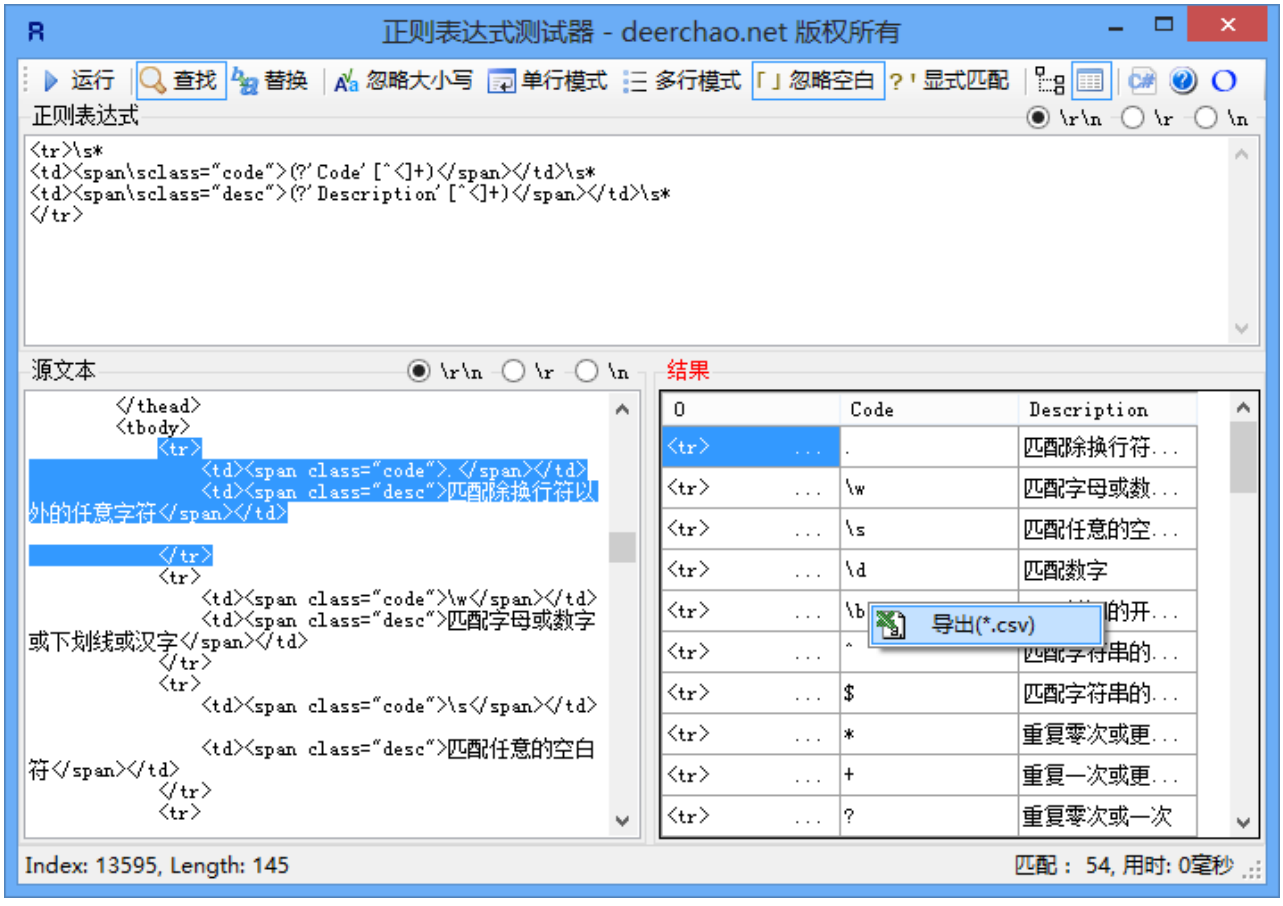
为了避免那么多烦人的重复，我们也可以这样写这个表达式：0\d{2}-\d{8}。这里\d 后面的{2}({8})的意思是前面\d 必须连续重复匹配 2 次(8 次)。

测试正则表达式

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 (http://www.jb51.net/tools/regex_tester/index.htm) 。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。

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)。

\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
.	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）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。

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。

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
*	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。

^\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-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}`)。

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}`)。

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。

IP 地址中每个数字都不能大于 255. 经常有人问我, 01.02.03.04 这样前面带有 0 的数字, 是不是正确的 IP 地址呢? 答案是: 是的, IP 地址里的数字可以包含有前导 0 (leading zeroes).

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。

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, 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

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：

`(?=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。

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。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
*?	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。
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

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 a a> 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

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>`。

详细语法

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
(?<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

速查手册

常用正则表达式

说明：正则表达式通常用于两种任务：1.验证，2.搜索/替换。用于验证时，通常需要在前后分别加上 `^` 和 `$`，以匹配整个待验证字符串；搜索/替换时是否加上此限定则根据搜索的要求而定，此外，也有可能要在前后加上\b而不是 `^` 和 `$` 。此表所列的常用正则表达式，除个别外均未在前后加上任何限定，请根据需要，自行处理。

常用正则表达式

正则表达式用于字符串处理、表单验证等场合，实用高效。现将一些常用的表达式收集于此，以备不时之需。

用户名：/^[a-z0-9_-]{3,16}$/

密码：/^[a-z0-9_-]{6,18}$/

十六进制值：/^#?([a-f0-9]{6}|[a-f0-9]{3})$/

电子邮箱：/^([a-z0-9_.-]+)@([\da-z.-]+).([a-z.]{2,6})$/

URL：/^(https?:\/\/)?([\da-z.-]+).([a-z.]{2,6})([\/\w .-]*)\/?$/

IP 地址：/^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?).){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/

HTML 标签：/^<([a-z]+)([^<]+)*(?:>(.)<\/\1>|\s+\/>)$/

Unicode 编码中的汉字范围：/^[u4e00-u9fa5],{0,}$/

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

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*?)[^>]*>.*?</\1>|<.*? />
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{3}-\d{8}|\d{4}-\d{7}

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10000开始

匹配中国大陆邮政编码：[1-9]\d{5}(?!\d)

评注：中国大陆邮政编码为6位数字

匹配身份证：\d{15}|\d{18}

评注：中国大陆的身份证为 15 位或 18 位

匹配 ip 地址：\d+.\d+.\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?\.0+|0)$　　//匹配浮点数

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

^(-([1-9]\d*\.\d*|0\.\d*[1-9]\d*))|0?\.0+|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
```

表达式全集

正则表达式有多种不同的风格。下表是在PCRE中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“fooood”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，请使用像“[.\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JScript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“Windows2000”中的“Windows”，但不能匹配“Windows3.1”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“Windows3.1”中的“Windows”，但不能匹配“Windows2000”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[\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向后引用。如果\n之前至少n个获取的子表达式，则n为向后引用。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向后引用。如果\nm之前至少有nm个获得子表达式，则nm为向后引用。如果\nm之前至少有n个获取，则n为一个后跟文字m的向后引用。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（？）。

以下是以 PHP 的语法所写的示例

验证字符串是否只含数字与英文，字符串长度并在 4~16 个字符之间

```
<?php
$str = 'a1234';
if (preg_match("^[a-zA-Z0-9]{4,16}$", $str)) {
echo "验证成功";
} else {
```

```
echo "验证失败";
}
?>
```

如何写出高效率的正则表达式

如果纯粹是为了挑战自己的正则水平，用来实现一些特效（例如使用正则表达式计算质数、解线性方程），效率不是问题；如果所写的正则表达式只是为了满足一两次、几十次的运行，优化与否区别也不太大。但是，如果所写的正则表达式会百万次、千万次地运行，效率就是很大的问题了。我这里总结了几条提升正则表达式运行效率的经验（工作中学到的，看书学来的，自己的体会），贴在这里。如果您有其它的经验而这里没有提及，欢迎赐教。

为行文方便，先定义两个概念。

误匹配：指正则表达式所匹配的内容范围超出了所需要范围，有些文本明明不符合要求，但是被所写的正则式“击中了”。例如，如果使用 `\d{11}` 来匹配 11 位的手机号，`\d{11}` 不单能匹配正确的手机号，它还会匹配 98765432100 这样的明显不是手机号的字符串。我们把这样的匹配称之为误匹配。

漏匹配：指正则表达式所匹配的内容所规定的范围太狭窄，有些文本确实是所需要的，但是所写的正则没有将这种情况囊括在内。例如，使用\d{18}来匹配 18 位的身份证号码，就会漏掉结尾是字母 X 的情况。

写出一条正则表达式，既可能只出现误匹配（条件写得极宽松，其范围大于目标文本），也可能只出现漏匹配（只描述了目标文本中多种情况种的一种），还可能既有误匹配又有漏匹配。例如，使用 `\w+\.com` 来匹配 `.com` 结尾的域名，既会误匹配 abc_.com 这样的字串（合法的域名中不含下划线，`\w` 包含了下划线这种情况），又会漏掉 `ab-c.com` 这样的域名（合法域名中可以含中划线，但是 `\w` 不匹配中划线）。

精准的正则表达式意味着既无误匹配且无漏匹配。当然，现实中存在这样的情况：只能看到有限数量的文本，根据这些文本写规则，但是这些规则将会用到海量的文本中。这种情况下，尽可能地（如果不是完全地）消除误匹配以及漏匹配，并提升运行效率，就是我们的目标。本文所提出的经验，主要是针对这种情况。

掌握语法细节。正则表达式在各种语言中，其语法大致相同，细节各有千秋。明确所使用语言的正则的语法的细节，是写出正确、高效正则表达式的基础。例如，perl 中与 \w 等效的匹配范围是 `[a-zA-Z0-9_]`；perl 正则式不支持肯定逆序环视中使用可变的重复（variable repetition inside lookbehind，例如 `(?<=.*)abc` ），但是 .Net 语法是支持这一特性的；又如，JavaScript 连逆序环视 （Lookbehind,如(?<=ab)c） 都不支持，而 Perl 和 Python 是支持的。《精通正则表达式》第3章《正则表达式的特性和流派概览》明确地列出了各大派系正则的异同，这篇文章也简要地列出了几种常用语言、工具中正则的比较。对于具体使用者而言，至少应该详细了解正在使用的那种工作语言里正则的语法细节。

先粗后精，****先加后减。使用正则表达式语法对于目标文本进行描述和界定，可以像画素描一样，先大致勾勒出框架，再逐步在局步实现细节。仍举刚才的手机号的例子，先界定 `\d{11}`，总不会错；再细化为 `1[358]\d{9}`，就向前迈了一大步（至于第二位是不是 3、5、8，这里无意深究，只举这样一个例子，说明逐步细化的

过程）。这样做的目的是先消除漏匹配（刚开始先尽可能多地匹配，做加法），然后再一点一点地消除误匹配（做减法）。这样有先有后，在考虑时才不易出错，从而向“不误不漏”这个目标迈进。

留有余地。所能看到的文本sample是有限的，而待匹配检验的文本是海量的，暂时不可见的。对于这样的情况，在写正则表达式时要跳出所能见到的文本的圈子，开拓思路，作出“战略性前瞻”。例如，经常收到这样的垃圾短信：“发*票”、“发#漂”。如果要写规则屏蔽这样烦人的垃圾短信，不但要能写出可以匹配当前文本的正则表达式 发*# (?:票|漂)，还要能够想到 发.(?:票|漂|飘)之类可能出现的“变种”。这在具体的领域或许会有针对性的规则，不多言。这样做的目的是消除漏匹配，延长正则表达式的生命周期。

明确。具体说来，就是谨慎用点号这样的元字符，尽可能不用星号和加号这样的任意量词。只要能确定范围的，例如\w，就不要用点号；只要能够预测重复次数的，就不要用任意量词。例如，写析取twitter消息的脚本，假设一条消息的 xml 正文部分结构是…且正文中无尖括号，那么 [^<]{1,480} 这种写法的思路要好 于.* ，原因有二：一是使用 [^<] ，它保证了文本的范围不会超出下一个小于号所在的位置；二是明确长度范围， {1,480} ，其依据是一条 twitter 消息大致能的字符长度范围。当然，480 这个长度是否正确还可推敲，但是这种思路是值得借鉴的。说得狠一点，“滥用点号、星号和加号是不环保、不负责任的做法”。

不要让稻草压死骆驼。每使用一个普通括号()而不是非捕获型括号 (?:…) ，就会保留一部分内存等着你再次访问。这样的正则表达式、无限次地运行次数，无异于一根根稻草的堆加，终于能将骆驼压死。养成合理使用 (?:…) 括号的习惯。

宁简勿繁。将一条复杂的正则表达式拆分为两条或多条简单的正则表达式，编程难度会降低，运行效率会提升。例如用来消除行首和行尾空白字符的正则表达式 s/^\s+|\s+$//g ;，其运行效率理论上要低于 s/^\s+//g ; s/\s+$//g ;。这个例子出自《精通正则表达式》第五章，书中对它的评论是“它几乎总是最快的，而且显然最容易理解”。既快又容易理解，何乐而不为？工作中我们还有其它的理由要将 C==(A|B) 这样的正则表达式拆为 A 和 B 两条表达式分别执行。例如，虽然 A 和 B 这两种情况只要有一种能够击中所需要的文本模式就会成功匹配，但是如果只要有一条子表达式（例如 A）会产生误匹配，那么不论其它的子表达式（例如 B）效率如何之高，范围如何精准，C 的总体精准度也会因A而受到影响。

巧妙定位。有时候，我们需要匹配的 the，是作为单词的 the（两边有空格），而不是作为单词一部分的 t-h-e 的有序排列（例如 together 中的 the）。在适当的时候用上^，$，\b 等等定位锚点，能有效提升找到成功匹配、淘汰不成功匹配的效率。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/regex/