正则表达式基础入门教程

日期：2007-8-1 15:34:51 出处：unibetter 大学生社区 作者：deerchao 人气： 39

目录

本文目标

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

如何使用本教程

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

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

除了作为入门教程之外，本文还试图成为可以在日常工作中使用的正则表达式语法参考手册。就作者本人的经历来说，这个目标还是完成得不错的——你看，我自己也没能把所有的东西记下来，不是吗？

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

正则表达式到底是什么？

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

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

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

入门

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

假设你在一篇英文小说里查找 hi，你可以使用正则正则表达式 hi。

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

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

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

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

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

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

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

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

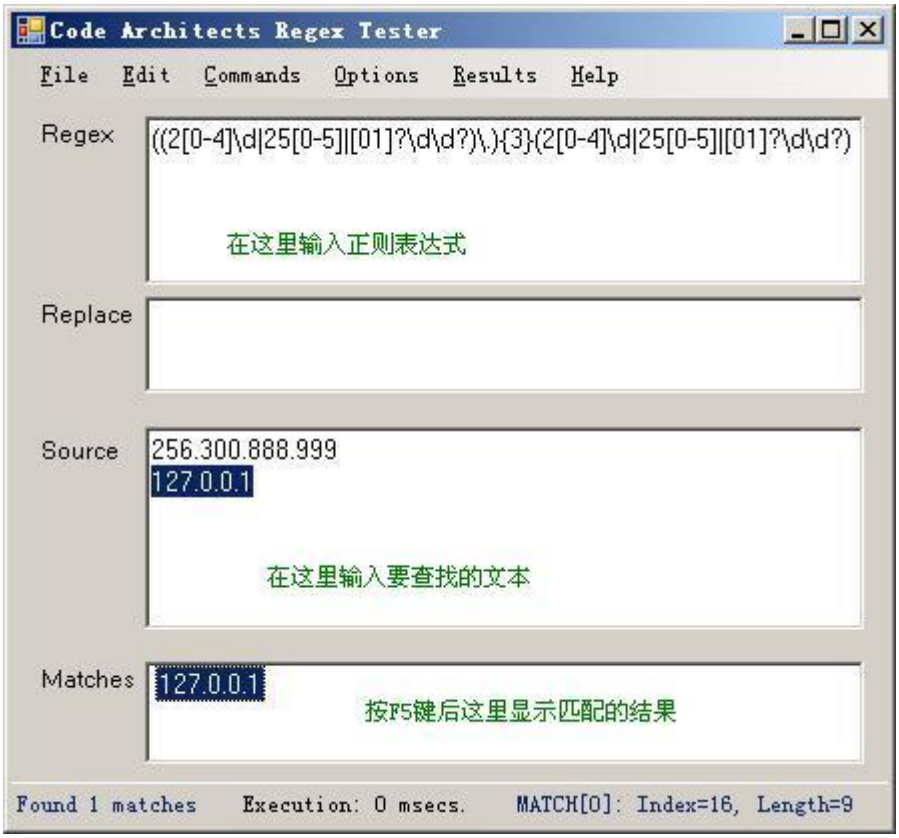
为了避免那么多烦人的重复，我们也可以这样写这个表达式：0\d{2}-\d{8}。 这里\d 后面的{2}({8})的意思是前面\d 必须连续重复匹配 2 次(8 次)。

测试正则表达式

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

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

下面是 Regex Tester 运行时的截图：



元字符

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

下面来试试更多的例子：

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

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

\b\w{6}\b 匹配刚好 6 个字母/数字的单词。

表 1.常用的元字符

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

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

这里的{5,12}和前面介绍过的{2}是类似的，只不过{2}匹配只能不多不少重复 2 次，{5,12}则是重复的次数不能少于 5 次，不能多于 12 次，否则都不匹配。

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

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

字符转义

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

例如：www\.unibetter\.com 匹配 www.unibetter.com，c:\\Windows 匹配 c:\Windows。

重复

你已经看过了前面的*,+,{2},{5,12}这几个匹配重复的方式了。下面是正则表达式中所有的限定符(指定数量的代码，例如*,{5,12}等)：

表 2.常用的限定符

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

下面是一些使用重复的例子：

Windows\d+匹配 Windows 后面跟 1 个或更多数字

13\d{9}匹配 13 后面跟 9 个数字(中国的手机号)

^\w+匹配一行的第一个单词(或整个字符串的第一个单词，具体匹配哪个意思得看选项设置)

字符类

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

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

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

下面是一个更复杂的表达式：\(?0\d{2}[) -]?\d{8}。

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

反义

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

表 3.常用的反义代码

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

例子：\S+匹配不包含空白符的字符串。

<a[^>]+>匹配用尖括号括起来的以 a 开头的字符串。

替换

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

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

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

\d{5}-\d{4}|\d{5}这个表达式用于匹配美国的邮政编码。美国邮编的规则是 5 位数字，或者用连字号间隔的 9 位数字。之所以要给出这个例子是因为它能说明一个问题：**使用替换时，顺序是很重要的**。如果你把它改成\d{5}|\d{5}-\d{4}的话，那么就只会匹配 5 位的邮编(以及 9 位邮编的前 5 位)。原因是匹配替换时，将会从左到右地测试每个分枝条件，如果满足了某个分枝的话，就不会去管其它的替换条件了。

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

分组

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

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

不幸的是，它也将匹配 *256.300.888.999* 这种不可能存在的 IP 地址(IP 地址中每个数字都不能大于 255。题外话，好像反恐 24 小时第三季的编剧不知道这一点，汗...)。如果能使用算术比较的话，或许能简单地解决这个问题，但是正则表达式中并不提供关于数学的任何功能，所以只能使用冗长的分组，选择，字符类来描述一个正确的 IP 地址：((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)。

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

后向引用

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

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

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

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

使用小括号的时候，还有很多特定用途的语法。下面列出了最常用的一些：

表 4.分组语法

捕获	
(exp)	匹配 exp,并捕获文本到自动命名的组里
(?<name>exp)	匹配 exp,并捕获文本到名称为 name 的组里，也可以写成(?'name'exp)
(?:exp)	匹配 exp,不捕获匹配的文本，也不给此分组分配组号
位置指定	
(?=exp)	匹配 exp 前面的位置
(?<=exp)	匹配 exp 后面的位置
(?!exp)	匹配后面跟的不是 exp 的位置
(?<!exp)	匹配前面不是 exp 的位置

注释

(?#comment) 这种类型的组不对正则表达式的处理产生任何影响，用于提供注释让人阅读

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

零宽断言

接下来的四个用于查找在某些内容(但并不包括这些内容)之前或之后的东西，也就是说它们像\b,^,$那样用于指定一个位置，这个位置应该满足一定的条件(**断言**)，因此它们也被称为**零宽断言**。最好还是拿例子来说明吧：

(?=exp)也叫**零宽度正预测先行断言**，它断言自身出现的位置的后面能匹配表达式 exp。比如\b\w+(?=ing\b)，匹配以 ing 结尾的单词的前面部分(除了 ing 以外的部分)，如查找 *I'm singing while you're dancing.*时，它会匹配 sing 和 danc。

(?<=exp)也叫**零宽度正回顾后发断言**，它断言自身出现的位置的前面能匹配表达式 exp。比如(?<=\bre)\w+\b 会匹配以 re 开头的单词的后半部分(除了 re 以外的部分)，例如在查找 *reading a book* 时，它匹配 ading。

假如你想要给一个很长的数字中每三位间加一个逗号(当然是从右边加起了)，你可以这样查找需要在前面和里面添加逗号的部分：((?<=\d)\d{3})*\b，用它对 *1234567890* 进行查找时结果是 234567890。

下面这个例子同时使用了这两种断言：(?<=\s)\d+(?=\s)匹配以空白符间隔的数字(再次强调，不包括这些空白符)。

负向零宽断言

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

\b\w*q[^u]\w*\b 匹配包含**后面不是字母 u 的字母 q** 的单词。但是如果多做测试(或者你思维足够敏锐，直接就观察出来了)，你会发现，如果 q 出现在单词的结尾的话，像 **Iraq**,**Benq**，这个表达式就会出错。这是因为[^u]总要匹配一个字符，所以如果 q 是单词的最后一个字符的话，后面的[^u]将会匹配 q 后面的单词分隔符(可能是空格，或者是句号或其它的什么)，后面的\w*\b 将会匹配下一个单词，于是\b\w*q[^u]\w*\b 就能匹配整个 *Iraq fighting*。**负向零宽断言**能解决这样的问题，因为它只匹配一个位置，并不**消费**任何字符。现在，我们可以这样来解决这个问题：\b\w*q(?!u)\w*\b。

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

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

一个更复杂的例子：(?<=<(\w+)>).*(?=<\/\1>)匹配不包含属性的简单 HTML 标签内里的内容。(<?(\w+)>)指定了这样的**前缀**：被尖括号括起来的单词(比如可能是)，然后是.*(任意的字符串),最后是一个**后缀**(?=<\/\1>)。注意后缀里的\/，它用到了前面提过的字符转义；\1 则是一个反向引用，引用的正是捕获的第一组，前面的(\w+)匹配的内容，这样如果前缀实际上是的话，后缀就是了。整个表达式匹配的是和之间的内容(再次提醒，不包括前缀和后缀本身)。

注释

小括号的另一种用途是能过语法(?#comment)来包含注释。例如：2[0-4]\d(?#200-249)|25[0-5](?#250-255)|[01]?\d\d?(?#0-199)。

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

例如，我们可以前面的一个表达式写成这样：

```
(?<=     # 断言要匹配的文本的前缀
<(\w+)> # 查找尖括号括起来的字母或数字(即 HTML/XML 标签)
)        # 前缀结束
.*       # 匹配任意文本
(?=      # 断言要匹配的文本的后缀
<\/\1>  # 查找尖括号括起来的内容：前面是一个"/"，后面是先前捕获的标签
)        # 后缀结束
```

贪婪与懒惰

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

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

a.*?b 匹配最短的，以 a 开始，以 b 结束的字符串。如果把它应用于 *aabab* 的话，它会匹配 aab 和 ab（为什么第一个匹配是 aab 而不是 ab？简单地说，因为正则表达式有另一条规则，比懒惰 / 贪婪规则的优先级更高：最先开始的区配最有最大的优先权——The Match That Begins Earliest Wins）。

表 5.懒惰限定符

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

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。
Singleline(单行模式)	更改.的含义，使它与每一个字符匹配（包括换行符\n）。
IgnorePatternWhitespace(忽略空白)	忽略表达式中的非转义空白并启用由#标记的注释。
RightToLeft(从右向左查找)	匹配从右向左而不是从左向右进行。
ExplicitCapture(显式捕获)	仅捕获已被显式命名的组。
ECMAScript(JavaScript 兼容模式)	使表达式的行为与它在 JavaScript 里的行为一致。

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

平衡组/递归匹配

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

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

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

这里需要用到以下的语法构造：

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

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

我们需要做的是每碰到了左括号，就在黑板上写一个"group"，每碰到一个右括号，就擦掉一个，到了最后就看看黑板上还有没有——如果有那就证明左括号比右括号多，那匹配就应该失败。

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

平衡组的一个最常见的应用就是匹配 HTML,下面这个例子可以匹配嵌套的 标签：

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

还有些什么东西没提到

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

表 7.尚未详细讨论的语法

\a	报警字符(打印它的效果是电脑嘀一声)
\b	通常是单词分界位置，但如果在字符类里使用代表退格
\t	制表符，Tab
\r	回车
\v	竖向制表符
\f	换页符
\n	换行符
\e	Escape
\0nn	ASCII 代码中八进制代码为 nn 的字符
\xnn	ASCII 代码中十六进制代码为 nn 的字符
\unnnn	Unicode 代码中十六进制代码为 nnnn 的字符
\cN	ASCII 控制字符。比如\cC 代表 Ctrl+C
\A	字符串开头(类似^，但不受处理多行选项的影响)
\Z	字符串结尾或行尾(不受处理多行选项的影响)
\z	字符串结尾(类似$，但不受处理多行选项的影响)
\G	当前搜索的开头
\p{name}	Unicode 中命名为 name 的字符类，例如\p{IsGreek}
(?>exp)	贪婪子表达式
(?<x>-<y>exp)	平衡组
(?im-nsx:exp)	在子表达式 exp 中改变处理选项
(?im-nsx)	为表达式后面的部分改变处理选项
(?(exp)yes|no)	把 exp 当作零宽正向先行断言，如果在这个位置能匹配，使用 yes 作为此组的表达式；否则使用 no
(?(exp)yes)	同上，只是使用空表达式作为 no

(?(name)yes|no) 如果命名为 name 的组捕获到了内容，使用 yes 作为表达式；否则使用 no

(?(name)yes)　同上，只是使用空表达式作为 no

一些我认为你可能已经知道的术语的参考

字符

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

字符串

0 个或更多个字符的序列。

文本

文字，字符串。

匹配

符合规则，检验是否符合规则，符合规则的部分。

断言

声明一个应该为真的事实。只有当断言为真时才会对正则表达式继续进行匹配。

网上的资源及本文参考文献

- 微软的正则表达式教程
- System.Text.RegularExpressions.Regex 类(MSDN)
- 专业的正则表达式教学网站(英文)
- 关于.Net 下的平衡组的详细讨论（英文）
- Mastering Regular Expressions (Second Edition)
- Validated XHTML 1.0 Strict
- Validated CSS 2.1
- 推荐使用 Mozilla FireFox 浏览

更新说明

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

正则表达式使用说明手册

正则表达式使用说明手册

author:[剑君] 2006-12-2 阅读[]

正则表达式参考手册教程：

表达式	语法	说明
壬一字符	.	匹配除换行符外的任何一个字符。
最多 0 项或更多	*	匹配前面表达式的 0 个或更多搜索项。
最多一项或更多	+	匹配前面表达式的至少一个搜索项。
最少 0 项或更多	@	匹配前面表达式的 0 个或更多搜索项，匹配尽可能少的字符。
最少一项或更多	#	匹配前面表达式的一个或更多搜索项，匹配尽可能少的字符。
重复 *n* 次	^*n*	匹配前面表达式的 *n* 个搜索项。例如，`[0-9]^4` 匹配任意 4 位数字的序列。
字符集	[]	匹配 [] 内的任何一个字符。要指定字符的范围，请列出由短划线 (-) 分隔的起始字符和结束字符，如 [a-z] 中所示。
不在字符集中的字符	[^...]	匹配跟在 ^ 之后的不在字符集中的任何字符。
行首	^	将匹配定位到行首。
行尾	$	将匹配定位到行尾。
词首	<	仅当词在文本中的此位置开始时才匹配。
词尾	>	仅当词在文本中的此位置结束时才匹配。
分组	()	将子表达式分组。
或	|	匹配 OR 符号 (|) 之前或之后的表达式。). 最常用在分组中。例如，`(sponge|mud) bath` 匹配“sponge bath”和“mud bath”。
转义符	\	匹配跟在反斜杠 (\) 后的字符。这使您可以查找在正则表达式表示法中使用的字符，如 { 和 ^。例如，`\^` 搜索 ^ 字符。
带标记的表达式	{}	标记括号内的表达式所匹配的文本。
第 n 个带标记的文本	\n	在“查找”或“替换”表达式中，指示第 n 个带标记的表达式所匹配的文本，其中 n 是从 1 至 9 的数字。 在“替换”表达式中，\0 插入整个匹配的文本。
右对齐字段	\(*w*,n)	在“替换”表达式中，将字段中第 n 个带标记的表达式右对齐至少 *w* 字符宽。

左对齐字段	\(-*w*,n)	在“替换”表达式中，将字段中第 n 个带标记的表达式左对齐至少 *w* 字符宽。
禁止匹配	~(X)	当 X 出现在表达式中的此位置时禁止匹配。例如，real~(ity) 匹配“realty”和“really”中的“real”，而不匹配“reality”中的“real”。
字母数字字符	:a	匹配表达式 ([a-zA-Z0-9])。
字母字符	:c	匹配表达式 ([a-zA-Z])。
十进制数	:d	匹配表达式 ([0-9])。
十六进制数	:h	匹配表达式 ([0-9a-fA-F]+)。
标识符	:i	匹配表达式 ([a-zA-Z_$][a-zA-Z0-9_$]*)。
有理数	:n	匹配表达式 (([0-9]+.[0-9]*)| ([0-9]*.[0-9]+)| ([0-9]+)).
带引号的字符串	:q	匹配表达式 (("[^"]*")| ('[^]*'))
字母字符串	:w	匹配表达式 ([a-zA-Z]+)
十进制整数	:z	匹配表达式 ([0-9]+)。
转义符	\e	Unicode U+001B。
Bell	\g	Unicode U+0007。
退格符	\h	Unicode U+0008。
换行符	\n	匹配与平台无关的换行符。在“替换”表达式中，插入换行符。
制表符	\t	匹配制表符，Unicode U+0009。
Unicode 字符	\x#### 或 \u####	匹配 Unicode 值给定的字符，其中 #### 是十六进制数。可以用 ISO 10646 代码点或两个提供代理项对的值的 Unicode 代码点指定基本多语种平面（即一个代理项）外的字符。

下表列出按标准 Unicode 字符属性进行匹配的语法。两个字母的缩写词与 Unicode 字符属性数据库中所列的一样。可将这些指定为字符集的一部分。例如，表达式 [:Nd:Nl:No] 匹配任何种类的数字。

表达式	语法	说明
大写字母	:Lu	匹配任何一个大写字母。例如，:Luhe 匹配“The”但不匹配“the”。
小写字母	:Ll	匹配任何一个小写字母。例如，:Llhe 匹配“the”但不匹配“The”。
词首大写字母	:Lt	匹配将大写字母和小写字母结合的字符，例如，Nj 和 Dz。
修饰符字母	:Lm	匹配字母或标点符号，例如逗号、交叉重音符和双撇号，用于表示对

		前一字母的修饰。
其他字母	:Lo	匹配其他字母，如哥特体字母 ahsa。
十进制数	:Nd	匹配十进制数（如 0-9）和它们的双字节等效数。
字母数字	:Nl	匹配字母数字，例如罗马数字和表意数字零。
其他数字	:No	匹配其他数字，如旧斜体数字一。
开始标点符号	:Ps	匹配开始标点符号，例如左方括号和左大括号。
结束标点符号	:Pe	匹配结束标点符号，例如右方括号和右大括号。
左引号	:Pi	匹配左双引号。
右引号	:Pf	匹配单引号和右双引号。
破折号	:Pd	匹配破折号标记。
连接符号	:Pc	匹配下划线标记。
其他标点符号	:Po	匹配逗号 (,)、?、"、!、@、#、%、&、*、\、冒号 (:)、分号 (;)、' 和 /。
空白分隔符	:Zs	匹配空白。
行分隔符	:Zl	匹配 Unicode 字符 U+2028。
段落分隔符	:Zp	匹配 Unicode 字符 U+2029。
无间隔标记	:Mn	匹配无间隔标记。
组合标记	:Mc	匹配组合标记。
封闭标记	:Me	匹配封闭标记。
数学符号	:Sm	匹配 +、=、~、| 、< 和 >。
货币符号	:Sc	匹配 $ 和其他货币符号。
修饰符号	:Sk	匹配修饰符号，如抑扬音、抑音符号和长音符号。
其他符号	:So	匹配其他符号，如版权符号、段落标记和度数符号。
其他控制	:Cc	匹配行尾。
其他格式	:Cf	格式化控制字符，例如双向控制字符。
代理项	:Cs	匹配代理项对的一半。
其他私用	:Co	匹配私用区域的任何字符。
其他未分配的字符	:Cn	匹配未映射到 Unicode 字符的字符。

除标准 Unicode 字符属性外，还可以指定下列附加属性。可将这些属性指定为字符集的一部分。

表达式	语法	说明
Alpha	:Al	匹配任何一个字符。例如，:Alhe 匹配“The”、“then”、“reached”等单词。
数字	:Nu	匹配任何一个数或数字。
标点符号	:Pu	匹配任何一个标点符号，如 ?、@、' 等等。
空白	:Wh	匹配所有类型的空格，包括印刷和表意文字的空格。

idi	:Bi	匹配诸如阿拉伯文和希伯来文这类从右向左书写的字符。
朝鲜文	:Ha	匹配朝鲜文和组合朝鲜文字母。
平假名	:Hi	匹配平假名字符。
片假名	:Ka	匹配片假名字符。
表意文字/汉字/日文汉字	:Id	匹配表意文字字符，如汉字和日文汉字

半小时精通正则表达式

想必很多人都对正则表达式都头疼.今天,我以我的认识,加上网上一些文章,希望用常人都可以理解的表达方式.来和大家分享学习经验.

开篇,还是得说说 ^ 和 $ 他们是分别用来匹配字符串的开始和结束，以下分别举例说明

"^The": 开头一定要有"The"字符串;
"of despair$": 结尾一定要有"of despair" 的字符串;

那么,
"^abc$": 就是要求以 abc 开头和以 abc 结尾的字符串，实际上是只有 abc 匹配
"notice": 匹配包含 notice 的字符串

你可以看见如果你没有用我们提到的两个字符（最后一个例子），就是说 模式（正则表达式）可以出现在被检验字符串的任何地方，你没有把他锁定到两边

接着,说说 '*', '+',和 '?',
他们用来表示一个字符可以出现的次数或者顺序. 他们分别表示：
"zero or more"相当于{0,},
"one or more"相当于{1,},
"zero or one."相当于{0,1}, 这里是一些例子:

"ab*": 和 ab{0,}同义,匹配以 a 开头,后面可以接 0 个或者 N 个 b 组成的字符串("a", "ab", "abbb", 等);
"ab+": 和 ab{1,}同义,同上条一样，但最少要有一个 b 存在 ("ab", "abbb", 等.);
"ab?":和 ab{0,1}同义,可以没有或者只有一个 b;
"a?b+$": 匹配以一个或者 0 个 a 再加上一个以上的 b 结尾的字符串.

要点, '*', '+',和 '?'只管它前面那个字符.

你也可以在大括号里面限制字符出现的个数，比如

"ab{2}": 要求 a 后面一定要跟两个 b（一个也不能少）("abb");
"ab{2,}": 要求 a 后面一定要有两个或者两个以上 b(如"abb", "abbbb", 等.);
"ab{3,5}": 要求 a 后面可以有 2－5 个 b("abbb", "abbbb", or "abbbbb").

现在我们把一定几个字符放到小括号里，比如：

"a(bc)*": 匹配 a 后面跟 0 个或者一个"bc";
"a(bc){1,5}": 一个到 5 个 "bc."

还有一个字符 '|', 相当于 OR 操作:

"hi|hello": 匹配含有"hi" 或者 "hello" 的 字符串;
"(b|cd)ef": 匹配含有 "bef" 或者 "cdef"的字符串;
"(a|b)*c": 匹配含有这样多个（包括 0 个）a 或 b，后面跟一个 c
的字符串;

一个点('.')可以代表所有的单一字符,不包括"\n"

如果,要匹配包括"\n"在内的所有单个字符,怎么办?

对了,用'[\n.]'这种模式.

"a.[0-9]": 一个 a 加一个字符再加一个 0 到 9 的数字
"^.{3}$": 三个任意字符结尾 .

中括号括住的内容只匹配一个单一的字符

"[ab]": 匹配单个的 a 或者 b (和 "a|b" 一样);
"[a-d]": 匹配'a' 到'd'的单个字符 (和"a|b|c|d" 还有 "[abcd]"效果一样); 一般我们都用

[a-zA-Z]来指定字符为一个大小写英文
"^[a-zA-Z]": 匹配以大小写字母开头的字符串
"[0-9]%": 匹配含有 形如 x% 的字符串
",[a-zA-Z0-9]$": 匹配以逗号再加一个数字或字母结尾的字符串

你也可以把你不想要得字符列在中括号里，你只需要在总括号里面使用'^' 作为开头 "%[^a-zA-Z]%" 匹配含有两个百分号里面有一个非字母的字符串.

要点:^用在中括号开头的时候,就表示排除括号里的字符

为了 PHP 能够解释，你必须在这些字符面前后加",并且将一些字符转义.

不要忘记在中括号里面的字符是这条规路的例外—在中括号里面, 所有的特殊字符,包括('), 都将失去他们的特殊性质 "[*\+?{}.]"匹配含有这些字符的字符串.

还有,正如 regx 的手册告诉我们: "如果列表里含有 ']', 最好把它作为列表里的第一个字符(可能跟在'^'后面). 如果含有'-', 最好把它放在最前面或者最后面, or 或者一个范围的第二个结束点[a-d-0-9]中间的'-'将有效.

看了上面的例子,你对{n,m}应该理解了吧.要注意的是,n 和 m 都不能为负整数,而且 n 总是小于 m. 这样,才能 最少匹配 n 次且最多匹配 m 次. 如"p{1,5}"将匹配 "pvpppppp"中的前五个 p

下面说说以\开头的

\b 书上说他是用来匹配一个单词边界,就是...比如've\b',可以匹配 love 里的 ve 而不匹配 very 里有 ve

\B 正好和上面的\b 相反.例子我就不举了

.....突然想起来....可以到 http://www.phpv.net/article.php/251 看看其它用\ 开头的语法

好,我们来做个应用:

如何构建一个模式来匹配 货币数量 的输入

构建一个匹配模式去检查输入的信息是否为一个表示 money 的数字。我们认为一个表示 money 的数量有四种方式： "10000.00" 和 "10,000.00",或者没有小数部分, "10000" and "10,000". 现在让我们开始构建这个匹配模式:

^[1-9][0-9]*$

这是所变量必须以非 0 的数字开头.但这也意味着 单一的 "0" 也不能通过测试. 以下是解决的方法:

^(0|[1-9][0-9]*)$

"只有 0 和不以 0 开头的数字与之匹配"，我们也可以允许一个负号在数字之前:

^(0|-?[1-9][0-9]*)$

这就是: "0 或者 一个以 0 开头 且可能 有一个负号在前面的数字." 好了,现在让我们别那么严谨，允许以 0 开头.现在让我们放弃 负号 ，因为我们在表示钱币的时候并不需要用到. 我们现在指定 模式 用来匹配小数部分:

^[0-9]+(\.[0-9]+)?$

这暗示匹配的字符串必须最少以一个阿拉伯数字开头. 但是注意，在上面模式中 "10." 是不匹配的, 只有 "10" 和 "10.2" 才可以. （你知道为什么吗）

^[0-9]+(\.[0-9]{2})?$

我们上面指定小数点后面必须有两位小数.如果你认为这样太苛刻,你可以改成:

^[0-9]+(\.[0-9]{1,2})?$

这将允许小数点后面有一到两个字符. 现在我们加上用来增加可读性的逗号(每隔三位), 我们可以这样表示:

^[0-9]{1,3}(,[0-9]{3})*(\.[0-9]{1,2})?$

不要忘记 '+' 可以被 '*' 替代 如果你想允许空白字符串被输入话 (为什么?). 也不要忘记反斜杆 '\' 在 php 字符串中可能会出现错误 (很普遍的错误).

现在，我们已经可以确认字符串了, 我们现在把所有逗号都去掉 str_replace(",", "", $money) 然后在把类型看成 double 然后我们就可以通过他做数学计算了.

再来一个:

构造检查 email 的正则表达式

在一个完整的 email 地址中有三个部分:
1. 用户名 (在 '@' 左边的一切),
2.'@',
3. 服务器名(就是剩下那部分).

用户名可以含有大小写字母阿拉伯数字,句号 ('.'), 减号('-'), and 下划线 ('_'). 服务器名字也是符合这个规则,当然下划线除外.

现在, 用户名的开始和结束都不能是句点. 服务器也是这样. 还有你不能有两个连续的句点他们之间至少存在一个字符，好现在我们来看一下怎么为用户名写一个匹配模式:

^[_a-zA-Z0-9-]+$

现在还不能允许句号的存在. 我们把它加上:

^[_a-zA-Z0-9-]+(\.[_a-zA-Z0-9-]+)*$

上面的意思就是说: "以至少一个规范字符（除了.）开头,后面跟着 0 个或者多个以点开始的字符串."

简单化一点, 我们可以用 eregi()取代 ereg().eregi()对大小写不敏感, 我们就不需要指定两个范围 "a-z" 和 "A-Z" – 只需要指定一个就可以了:

^[_a-z0-9-]+(\.[_a-z0-9-]+)*$

后面的服务器名字也是一样,但要去掉下划线:

^[a-z0-9-]+(\.[a-z0-9-]+)*$

好. 现在只需要用”@”把两部分连接:

^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*$

这就是完整的 email 认证匹配模式了,只需要调用

eregi(‘^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*$ ’,$eamil)

就可以得到是否为 email 了
正则表达式的其他用法

提取字符串

ereg() and eregi() 有一个特性是允许用户通过正则表达式去提取字符串的一部分(具体用法你可以阅读手册). 比如说,我们想从 path/URL 提取文件名 – 下面的代码就是你需要:

```
ereg("([^\\/]*)$", $pathOrUrl, $regs);
echo $regs[1];
```

高级的代换

ereg_replace() 和 eregi_replace()也是非常有用的: 假如我们想把所有的间隔负号都替换成逗号:

```
ereg_replace("[ \n\r\t]+", ",", trim($str));
```

最后,我把另一串检查 EMAIL 的正则表达式让看文章的你来分析一下.

```
"^[-!#$%&\'*+\\./0-9=?A-Z^_`a-z{|}~]+'.'@'.'[-!#$%&\'*+\\/0-9=?A-Z^_`a-z{|}~]+\.'.'[-!#$%&\'*+\\./0-9=?A-Z^_`a-z{|}~]+$"
```

如果能方便的读懂,那这篇文章的目的就达到了.

php的正则表达式——完全手册

正则表达式是烦琐的，但是强大的，学会之后的应用会让你除了提高效率外，会给你带来绝对的成就感。只要认真去阅读这些资料，加上应用的时候进行一定的参考，掌握正则表达式不是问题。

索引

1. 引子

目前，正则表达式已经在很多软件中得到广泛的应用，包括*nix（Linux, Unix等），HP等操作系统，PHP，C#，Java等开发环境，以及很多的应用软件中，都可以看到正则表达式的影子。

正则表达式的使用，可以通过简单的办法来实现强大的功能。为了简单有效而又不失强大，造成了正则表达式代码的难度较大，学习起来也不是很容易，所以需要付出一些努力才行，入门之后参照一定的参考，使用起来还是比较简单有效的。

例子： ^.+@.+\\..+$

这样的代码曾经多次把我自己给吓退过。可能很多人也是被这样的代码给吓跑的吧。继续阅读本文将让你也可以自由应用这样的代码。

注意：这里的第7部分跟前面的内容看起来似乎有些重复，目的是把前面表格里的部分重新描述了一次，目的是让这些内容更容易理解。

2. 正则表达式的历史

正则表达式的“祖先”可以一直上溯至对人类神经系统如何工作的早期研究。Warren McCulloch 和 Walter Pitts 这两位神经生理学家研究出一种数学方式来描述这些神经网络。

1956 年, 一位叫 Stephen Kleene 的数学家在 McCulloch 和 Pitts 早期工作的基础上，发表了一篇标题为“神经网事件的表示法”的论文，引入了正则表达式的概念。正则表达式就是用来描述他称为“正则集的代数”的表达式，因此采用“正则表达式”这个术语。

随后，发现可以将这一工作应用于使用 Ken Thompson 的计算搜索算法的一些早期研究，Ken Thompson 是 Unix 的主要发明人。正则表达式的第一个实用应用程序就是 Unix 中的 qed 编辑器。

如他们所说，剩下的就是众所周知的历史了。从那时起直至现在正则表达式都是基于文本的编辑器和搜索工具中的一个重要部分。

3. 正则表达式定义

正则表达式(regular expression)描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串、将匹配的子串做替换或者从某个串中取出符合某个条件的子串等。

列目录时， dir *.txt或ls *.txt中的*.txt就不是一个正则表达式,因为这里*与正则式的*的含义是不同的。

正则表达式是由普通字符（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。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。

3.1 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。

3.2 非打印字符

字符 含义
\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。
\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。

3.3 特殊字符

所谓特殊字符，就是一些有特殊含义的字符，如上面说的"*.txt"中的*，简单的说就是表示任何字符串的意思。如果要查找文件名中有＊的文件，则需要对＊进行转义，即在其前加一个\。ls
*.txt。正则表达式有以下特殊字符。

特别字符说明
$匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r' 。要匹配
$ 字符本身，请使用 \$。
( )标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \(和 \)。
*匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。
.匹配除换行符 \n之外的任何单字符。要匹配 .，请使用 \。
[标记一个中括号表达式的开始。要匹配 [，请使用 \[。
?匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。'\n' 匹配换行符。序列
'\\' 匹配 "\"，而 '\(' 则匹配 "("。
^匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。
{标记限定符表达式的开始。要匹配 {，请使用 \{。
|指明两项之间的一个选择。要匹配 |，请使用 \|。

构造正则表达式的方法和创建数学表达式的方法一样。也就是用多种元字符与操作符将小的表达式结合在一起来创建更大的表达式。正则表达式的组件可以是单个的字符、字符集合、字符范围、字符间的选择或者所有这些组件的任意组合。

3.4 限定符

限定符用来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。有*或+或?或{n}或{n,}或{n,m}共6种。

*、+和?限定符都是贪婪的，因为它们会尽可能多的匹配文字，只有在它们的后面加上一个?就可以实现非贪婪或最小匹配。

正则表达式的限定符有：

字符 描述
* 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 "foooood"
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。

3.5 定位符

用来描述字符串或单词的边界，^和$分别指字符串的开始与结束，\b描述单词的前或后边界，\B表示非单词边界。不能对定位符使用限定符。

3.6 选择

用圆括号将所有选择项括起来，相邻的选择项之间用|分隔。但用圆括号会有一个副作用，是相关的匹配会被缓存，此时可用?:放在第一个选项前来消除这种副作用。

其中?:是非捕获元之一，还有两个非捕获元是?=和?!，这两个还有更多的含义，前者为正向预查，在任何开始匹配圆括号内的正则表达式模式的位置来匹配搜索字符串，后者为负向预查，在任何开始不匹配该正则表达式模式的位置来匹配搜索字符串。

3.7 后向引用

对一个正则表达式模式或部分模式两边添加圆括号将导致相关匹配存储到一个临时缓冲区中，所捕获的每个子匹配都按照在正则表达式模式中从左至右所遇到的内容存储。存储子匹配的缓冲区编号从
1 开始，连续编号直至最大 99 个子表达式。每个缓冲区都可以使用 '\n' 访问，其中 n 为一个标识特定缓冲区的一位或两位十进制数。

可以使用非捕获元字符 '?:', '?=', or '?!' 来忽略对相关匹配的保存。

4. 各种操作符的运算优先级

相同优先级的从左到右进行运算，不同优先级的运算先高后低。各种操作符的优先级从高到低如下：

操作符 描述
\ 转义符
(), (?:), (?=), [] 圆括号和方括号
*, +, ?, {n}, {n,}, {n,m} 限定符
^, $, \anymetacharacter 位置和顺序
| “或”操作

5. 全部符号解释

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 "foooood"
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。
(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 "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 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， '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 (?)。

6. 部分例子

正则表达式说明
/\b([a-z]+) \1\b/gi一个单词连续出现的位置
/(\w+):\/\/([^/:]+)(:\d*)?([^#]*)/ 将一个URL解析为协议、域、端口及相对路径
/^(?:Chapter|Section) [1-9][0-9]{0,1}$/定位章节的位置
/[-a-z]/A至z共26个字母再加一个-号。
/ter\b/可匹配chapter，而不能terminal
/\Bapt/可匹配chapter，而不能aptitude
/Windows(?=95 |98 |NT
)/可匹配Windows95或Windows98或WindowsNT,当找到一个匹配后，从Windows后面开始进行下一次

7. 正则表达式匹配规则

7.1 基本模式匹配

一切从最基本的开始。模式，是正规表达式最基本的元素，它们是一组描述字符串特征的字符。模式可以很简

单，由普通的字符串组成，也可以非常复杂，往往用特殊的字符表示一个范围内的字符、重复出现，或表示上下文。例如：

```
^once
```

这个模式包含一个特殊的字符^，表示该模式只匹配那些以once开头的字符串。例如该模式与字符串"once upon a time"匹配，与"There once was a man from NewYork"不匹配。正如如^符号表示开头一样，$符号用来匹配那些以给定模式结尾的字符串。

```
bucket$
```

这个模式与"Who kept all of this cash in a bucket"匹配，与"buckets"不匹配。字符^和$同时使用时，表示精确匹配（字符串与模式一样）。例如：

```
^bucket$
```

只匹配字符串"bucket"。如果一个模式不包括^和$，那么它与任何包含该模式的字符串匹配。例如：模式

```
once
```

与字符串

```
There once was a man from NewYork
Who kept all of his cash in a bucket.
```

是匹配的。

在该模式中的字母(o-n-c-e)是字面的字符，也就是说，他们表示该字母本身，数字也是一样的。其他一些稍微复杂的字符，如标点符号和白字符（空格、制表符等），要用到转义序列。所有的转义序列都用反斜杠(\)打头。制表符的转义序列是：\t。所以如果我们要检测一个字符串是否以制表符开头，可以用这个模式：

```
^\t
```

类似的，用\n表示“新行”，\r表示回车。其他的特殊符号，可以用在前面加上反斜杠，如反斜杠本身用\\表示，句号.用\.表示，以此类推。

7.2 字符簇

在INTERNET的程序中，正规表达式通常用来验证用户的输入。当用户提交一个FORM以后，要判断输入的电话号码、地址、EMAIL地址、信用卡号码等是否有效，用普通的基于字面的字符是不够的。

所以要用一种更自由的描述我们要的模式的办法，它就是字符簇。要建立一个表示所有元音字符的字符簇，就把所有的元音字符放在一个方括号里：

```
[AaEeIiOoUu]
```

这个模式与任何元音字符匹配，但只能表示一个字符。用连字号可以表示一个字符的范围，如：

```
[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个或更多的数字([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：

```
^\-?[0-9]{0,}\.?[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
^\-?[0-9]+$ //所有的整数
^\-?[0-9]*\.?[0-9]*$ //所有的小数
```

当然这并不能从技术上降低正规表达式的复杂性，但可以使它们更容易阅读。

正则表达式用来指定字符串模式。当你需要定位匹配某种模式的字符串时就可以使用正则表达式。例如，我们下面的一个例程就是在一个HTML文件中通过查找字符串模式<a href="...">来定位所有的超链接。

当然，为了指定一种模式，使用...这种记号是不够精确的。你需要精确地指定什么样的字符排列是一个合法的匹配。当描述某种模式时，你需要使用一种特殊的语法。

这里有一个简单例子。正则表达式
[Jj]ava.+
匹配下列形式的任何字符串：

首字母是J或j
后续的三个字母是ava
字符串的剩余部分由一个或多个任意字符组成

例如，字符串“javaness”匹配这个特殊的正则表达式，但是字符串“Core Java”却不匹配。

如你所见，你需要了解一点语法来理解正则表达式的含意。幸运的是对于大多数的用途，使用少量的简单构造（straightforward constructs）就已足够。

字符类是可选自符的集合，用‘[’封装，比如[Jj],[0-9],[A-Za-z]或[^0-9]。这里的-表示范围（Unicode落在两个边界之间的所有字符），^表示求补（指定字符外的所有字符）。
有许多预定以的字符类，像\d（数字）或\p{Sc}（Unicode货币符号），见表12-8和12-9。
大多数字符与它们自身匹配，像上例中的ava字符。
符号.匹配任何字符（可能行终止符（line terminators）除外，这依赖于标识设置（flag settings））
\用作转义符，比如\.匹配一个句点，\\匹配一个反斜杠。
^和$分别匹配行头和行尾
如果X和Y都是正则表达式，则XY表示“X的匹配后面跟着Y的匹配”。X|Y表示“任何X或Y的匹配”
可以将量词（quantifier）用到表达式中，X+ 表示X重复1次或多次，X* 表示X重复0次或多次，X? 表示X重复0次或1次
默认地，一个量词总是与使总体成功匹配的最长的可能重复匹配。可以加上后缀？（称为reluctant或stingy 匹配，用以匹配最小的重复数），和+（称为possessive或贪婪匹配，用以即使在总体匹配失败的情况下也匹配最大的重复数）来更改这种属性。

例如，字符串cab匹配[a-z]*ab，但不匹配[a-z]*+ab。第一种情况下，[a-z]*只匹配字符c，因此字符ab正好与模式的剩余部分匹配。但是贪婪版本[a-z]*+就与字符cab匹配，模式的剩余部分ab就匹配失败（，这样总体也就匹配失败）。

可以使用分组来定义子表达式。将分组封装在（）中，如([+-]?)([0-9]+)。然后你可以让模式匹配符（the pattern matcher）返回每个分组的匹配，或者使用\n来回引分组（refer back to a group with \n），其中n是组号（以\1起始）

这里有一个稍微有点复杂却又很有用的正则表达式——它用来描述十进制和十六进制的整数。
[+-]?[0-9]+|0[Xx][0-9A-Fa-f]+

不幸的是，在使用正则表达式的各种程序和库之间，它的语法还没有完全标准化。对基本的构造上已达成了共识，但在细节方面有许多令人“抓狂”的区别（many maddening differences）。Java的正则表达式类使用了与Perl语言类似的语法，但也不尽相同。表12-8显示了Java语法的所有正则表达式构造。要了解更多关于正则表达式的信息，请参考Pattern类的API文档，或者Jeffrey E. F. Friedl的著作《Mastering Regular Expressions》(O'Reilly and Associates, 1997)（刚去第二书店查了一下，东南大学出版社已经引入了其第二版，影印）

表12-8 正则表达式语法

语法	解释
字符	
c	字符c
\unnnn, \xnn, \0n, \0nn, \0nnn	带有十六或八进制值的代码单元
\0n	八进制0n代表的字符（0<=n<=7）
\0nn	八进制0nn代表的字符(0<=n<=7)
\0mnn	八进制0mnn代表的字符(0<=m<=3,0<=n<=7)
\xnn	十六进制 0xnn所代表的字符
\uhhhh	十六进制 0xhhhh所代表的字符
\t, \n, \r, \f, \a, \e	控制字符，依次是制表符，换行符，回车符，换页符，报警符和转义符
\cc	控制字符中出现的相应字符c
字符类	
[C1C2. . .]	C1、C2……中的任何字符。Ci可以是字符，字符范围（C1-C2）或者字符类。

[^. . .]	字符类的补集
[. . . && . . .]	两个字符类的交集

预定义字符类

.	除行终止符外的任何字符（如果DOTALL标志置位，则表示任何字符）
\d	数字[0-9]
\D	非数字[^0-9]
\s	空白字符[\t\n\r\f\x0B]
\S	非空白字符
\w	单词字符[a-zA-Z0-9_]
\W	非单词字符
\p{name}	一个指定的字符类，见表12-9
\P{name}	指定字符类的补集

边界匹配符

^ $	输入的开头和结尾(在多行模式(multiline mode)下是行的开头和结尾)
\b	单词边界
\B	非单词边界
\A	输入的开头
\z	输入的结尾
\Z	除最后行终止符之外的输入结尾
\G	上个匹配的结尾

量词

X?	可选的X（即X可能出现，也可能不出现）
X*	X，可以重复0次或多次
X+	X，可以重复1次或多次
X{n} X{n,} X{n,m}	X重复n次，至少重复n次，重复n到m次

量词后缀

?	设默认（贪婪）匹配为reluctant匹配
+	设默认（贪婪）匹配为possessive匹配

集合操作

XY	X的匹配后面跟着Y的匹配
X|Y	X或Y的匹配

分组

(X)	匹配X并且在一个自动计数的分组中捕获它
\n	与第n个分组的匹配

转义

\c	字符c（必须不是字母）
\Q...\E	逐字地引用...
（? ...）	特殊构造，看Pattern类的API

正则表达式的最简单使用是测试一个特殊的字符串是否与之匹配。这里有一个Java写的测试程序。首先从表示正则表达式的字符串构造一个Pattern对象。然后从该模式获得一个Matcher对象，并且调用它的matches()方法：

```
Pattern pattern = Pattern.compile(patternString);

Matcher matcher = pattern.matcher(input);

if (matcher.matches()) . . .
```

表12.9 预定义的字符类名（Predefined Character Class Names）

Lower	小写的ASII字符[a-z]
Upper	大写的ASCII字符[A-Z]
Alpha	ASCII字母[A-Za-z]
Digit	ASCII 数字 [0-9]
Alnum	ASCII 字母或数字[A-Za-z0-9]
Xdigit	十六进制数字[0-9A-Fa-f]
Print or Graph	可打印的ASCII字符[\x21-\x7E]
Punct	非字母或数字ASCII [\p{Print}&&\P{Alnum}]

ASCII 所有ASCII字符 [\x00-\x7F]
Cntrl ASCII控制字符[\x00-\x1F]
Blank 空格符或制表符[\t]
Space 空白符 [\t\n\r\f\0x0B]
javaLowerCase 取决于Character.isLowerCase()的小写字符
javaUpperCase 取决于Character.isUpperCase()的大写字符
javaWhitespace 取决于Character.isWhitespace()的空白符
javaMirrored 取决于Character.isMirrored()的Mirrored（？）
InBlock 这里的Block是unicode字符的块名，用空格隔开，比如BasicLatin 或 Mongolian。块名列表参考http://www.unicode.org
Category 或InCategory 这里的Category是Unicode字符的种类名，比如L（字母）或者Sc（货币符号）。种类名列表参考http://www.unicode.org

---

matcher的输入可以是实现CharSequence接口的任何类对象，像String，StringBuilder或CharBuffer。

当编译模式时，可以设置一个或多个标志，例如

```
Pattern pattern = Pattern.compile(patternString,

Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE);
```

下面六个标志都是支持的：

CASE_INSENSITIVE：匹配字符时与大小写无关，该标志默认只考虑US ASCII字符。

UNICODE_CASE：当与CASE_INSENSITIVE结合时，使用Unicode字母匹配

MULTILINE：^和$匹配一行的开始和结尾，而不是整个输入

UNIX_LINES： 当在多行模式下匹配^和$时，只将'\n'看作行终止符

DOTALL：当使用此标志时，.符号匹配包括行终止符在内的所有字符

CANON_EQ：考虑Unicode字符的规范等价

如果正则表达式包含分组，Matcher对象能够揭示分组边界。方法

```
int start(int groupIndex)

int end(int groupIndex)
```

返回某个特殊分组的起始索引和结尾后索引（past-the-end index ）。
通过调用String group(int groupIndex)，你可以简单地得到匹配的字符串。
第0个分组代表所有的分组，第一个实际分组的索引是1。调用groupCount来获得总的分组数。

使用开放圆括号来安排嵌套分组。例如，给定模式((1?[0-9]):([0-5][0-9]))[ap]m，并且输入11:59am,Matcher对象报告下列分组
((1?[0-9]):([0-5][0-9]))[ap]m
并输入
11:59am
matcher报告下列分组

分组索引 起始 结束 字符串

---

0 0 7 11;59am
1 0 5 11:59
2 0 2 11
3 3 5 59

例12－9提示输入一个模式和一个欲匹配的字符串。它将输出输入的字符串是否匹配模式。如果输入匹配包含分组的模式，程序将会使用圆括号来打印分组边界，如((11):(59))am

Example 12-9. RegexTest.java

```
1. import java.util.*;
2. import java.util.regex.*;
3.
4. /**
```

```
5.   This program tests regular expression matching.
6.   Enter a pattern and strings to match, or hit Cancel
7.   to exit. If the pattern contains groups, the group
8.   boundaries are displayed in the match.
9. */
10. public class RegExTest
11. {
12.   public static void main(String[] args)
13.   {
14.     Scanner in = new Scanner(System.in);
15.     System.out.println("Enter pattern: ");
16.     String patternString = in.nextLine();
17.
18.     Pattern pattern = null;
19.     try
20.     {
21.       pattern = Pattern.compile(patternString);
22.     }
23.     catch (PatternSyntaxException e)
24.     {
25.       System.out.println("Pattern syntax error");
26.       System.exit(1);
27.     }
28.
29.     while (true)
30.     {
31.       System.out.println("Enter string to match: ");
32.       String input = in.nextLine();
33.       if (input == null || input.equals("")) return;
34.       Matcher matcher = pattern.matcher(input);
35.       if (matcher.matches())
36.       {
37.         System.out.println("Match");
38.         int g = matcher.groupCount();
39.         if (g > 0)
40.         {
41.           for (int i = 0; i < input.length(); i++)
42.           {
43.             for (int j = 1; j <= g; j++)
44.               if (i == matcher.start(j))
45.                 System.out.print('(');
46.             System.out.print(input.charAt(i));
47.             for (int j = 1; j <= g; j++)
48.               if (i + 1 == matcher.end(j))
49.                 System.out.print(')');
50.           }
51.           System.out.println();
52.         }
53.       }
54.       else
55.         System.out.println("No match");
56.     }
57.   }
58. }
```

通常地，你不希望匹配整个输入到某个正则表达式，而是希望在输入中找出一个或多个匹配的子字符串。使用Matcher类的find方法来寻找下一个匹配。如果它返回True，再使用start和end方法找出匹配的范围。

```
while (matcher.find())

{

  int start = matcher.start();

  int end = matcher.end();

  String match = input.substring(start, end);

  . . .

}
```

例12-10用到了这种机制。它在一个网页中定位所有的超文本引用并打印它们。为运行程序，在命令行提供一个URL，比如

```
java HrefMatch http://www.horstmann.com

Example 12-10. HrefMatch.java
1. import java.io.*;
2. import java.net.*;
3. import java.util.regex.*;
4.
5. /**
6.   This program displays all URLs in a web page by
7.   matching a regular expression that describes the
8.   <a href=...> HTML tag. Start the program as
9.   java HrefMatch URL
10. */
11. public class HrefMatch
12. {
13.   public static void main(String[] args)
14.   {
15.     try
16.     {
17.       // get URL string from command line or use default
18.       String urlString;
19.       if (args.length > 0) urlString = args[0];
20.       else urlString = "http://java.sun.com";
21.
22.       // open reader for URL
23.       InputStreamReader in = new InputStreamReader(new URL(urlString).openStream());
24.
25.       // read contents into string buffer
26.       StringBuilder input = new StringBuilder();
27.       int ch;
28.       while ((ch = in.read()) != -1) input.append((char) ch);
29.
30.       // search for all occurrences of pattern
31.       String patternString = "<a\\s+href\\s*=\\s*(\"[^\"]*\"|[^\\s>])\\s*>";
32.       Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
33.       Matcher matcher = pattern.matcher(input);
34.
35.       while (matcher.find())
36.       {
37.         int start = matcher.start();
38.         int end = matcher.end();
39.         String match = input.substring(start, end);
40.         System.out.println(match);
41.       }
42.     }
43.     catch (IOException e)
44.     {
45.       e.printStackTrace();
46.     }
47.     catch (PatternSyntaxException e)
48.     {
49.       e.printStackTrace();
50.     }
51.   }
52. }
```

Matcher类的replaceAll方法用一个替换字符串代替出现的所有正则表达式的匹配。比如，下列指令用#替换所有数字序列

```
Pattern pattern = Pattern.compile("[0-9]+");

Matcher matcher = pattern.matcher(input);

String output = matcher.replaceAll("#");
```

替换字符串可以包含模式中的分组引用：$n被第n个分组替换。替换文本中出现$时，使用\$来包含它。replaceFirst方法只替换模式的第一次出现。

最后讲一点，Pattern类有一个split方法，它类似于字符串tokenizer。它使用正则表达式匹配作边界，将输入分离成字符串数组。比如，下面的指令将输入分离成记号（token），

```
Pattern pattern = Pattern.compile("\\s*\\p{Punct}\\s*");

String[] tokens = pattern.split(input);
```

类

---
java.util.regex.Pattern 1.4

方法
static Pattern compile(String expression)
static Pattern compile(String expression, int flags)
编译正则表达式字符串到pattern对象用以匹配的快速处理
参数:
expression 正则表达式
flags 下列标志中的一个或多个 CASE_INSENSITIVE, UNICODE_CASE, MULTILINE, UNIX_LINES, DOTALL, and CANON_EQ

Matcher matcher(CharSequence input)
返回一个matcher对象，它可以用来在一个输入中定位模式匹配

String[] split(CharSequence input)
String[] split(CharSequence input, int limit)
将输入字符串分离成记号，并由pattern来指定分隔符的形式。返回记号数组。分隔符并不是记号的一部分。
参数:
input 分离成记号的字符串
limit 生成的最大字符串数。

类

---
java.util.regex.Matcher 1.4

方法

---
boolean matches()
返回输入是否与模式匹配

boolean lookingAt()
如果输入的起始匹配模式则返回True

boolean find()
boolean find(int start)
尝试查找下一个匹配，并在找到匹配时返回True
参数:
start 开始搜索的索引

int start()
int end()
返回当前匹配的起始位置和结尾后位置

String group()
返回当前匹配

int groupCount()
返回输入模式中的分组数

int start(int groupIndex)
int end(int groupIndex)
返回一个给定分组当前匹配中的起始位置和结尾后位置
参数:
groupIndex分组索引（从1开始），0表示整个匹配

String group(int groupIndex)
返回匹配一个给定分组的字符串
参数:
groupIndex
分组索引（从1开始），0表示整个匹配

String replaceAll(String replacement)
String replaceFirst(String replacement)
返回从matcher输入得到的字符串，但已经用替换表达式替换所有或第一个匹配
参数:
replacement 替换字符串

```
Matcher reset()
Matcher reset(CharSequence input)
```
复位mather状态。

简单实用的UltraEdit的正则表达式手册

允许在搜索菜单下面列出了的许多搜索和替换功能中使用正则表达式。正则表达式能让更多的复杂的搜索和替换功能变成简单的操作。（中文版界面上显示为“正规表达式”）
有两个可使用的语法集合。下面的第一表显示出在 UltraEdit 的更早的版本被使用的原来的 UltraEdit 句法。第二表给出了可选的"Unix"类型的正则表达式。这可以从配置单元启用。

符号 功能
% 匹配行的开始 - 显示搜索字符串必须在行的开始，但是在所选择的结果字符串中不包括任何行终止字符。
$ 匹配行尾 - 显示搜索字符串必须在行尾，但是在所选择的结果字符串中不包括任何行终止字符。
? 除了换行符以外匹配任何单个的字符
* 除了换行符匹配任何数量的字符和数字
+ 前一字符匹配一个或多个，但至少要出现一个
++ 前一字符匹配零个或多个，但至少要出现一个
^b 匹配一个分页
^p 匹配一个换行符(CR/LF)(段)(DOS文件)
^r 匹配一个换行符(CR 仅仅)(段)(MAC 文件)
^n 匹配一个换行符 (LF 仅仅)(段)(UNIX 文件)
^t 匹配一个标签字符TAB
[] 匹配任何单个的字符，或在方括号中的范围
^{A^}^{ B^} 匹配表达式A或 B
^ 重载其后的正规表达式字符
^(^) 括或标注为用于替换命令的表达式。

一个正则表达式最多可以有9个标注表达式，按正规表达式的需要而定。
相应的替换表达式是 ^x , 替换范围x是1-9。例如：
If ^(h*o^) ^(f*s^) matches "hello folks",
^2 ^1 would replace it with "folks hello".

（hello folks 将被替换成 folks hello。）

注： ^ 是实际字符 ^不是Ctl + 键值。

例如：
m?n 匹配 "man","men","min" 但不匹配 "moon".
t*t 匹配 "test","tonight" 和 "tea time" (the "tea t" portion) 但不匹配 "tea time" (newline between "tea " and "time").
Te+st 匹配 "test","teest"," teeeest "等等。但是不匹配 "tst"。
[aeiou] 匹配每个小写元音。
[,.?] 匹配一文字的 ",", "."或 "?"。
[0-9, a-z] 匹配任何数位，或小写字母。
[~0-9] 除了数字以外匹配任何字符 (~ 意味着"不")

你按如下方式可以查找一个表达式A或 B ：

"^{John^}^{Tom^}"

这将在找John或Tom的出现。应该在 2 个表达式之间没有任何东西。

你可以在同一搜索中按如下方式组合A or B and C or D：

"^{John^}^{Tom^}^{Smith^}^{Jones^}"

这将在John or Tom 后面找 Smith or Jones。

下表为"Unix"句法类型的正则表达式。

正则表达式 (Unix句法)：

符号 功能
标记下一个字符作为一个特殊的字符。
"n" 匹配字符"n"。"n" 一个换行符或换行符字符。
^ 匹配/定位行的开始。
$ 匹配/定位行的尾。
* 匹配前面的字符零次或多次。例
+ 匹配前面的字符一次或多次。例
. 匹配除了一个换行符字符匹配任何单个的字符。
(expression)标注用于替换命令的表达式。一个正则表达式根据需要，最多可以有9个标注表达式。相应的代替表达式是 x , x的范围是 1-9
。

例如：

If (h.*o) (f.*s) matches "hello folks",

2 1 would replace it with "folks hello".
（hello folks 将被替换成 folks hello。）

[xyz]　　一个字符集。匹配在方括号之间的任何字符。
[^xyz]　　一个否定的字符集。不匹配在方括号之间的任何字符。
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　　匹配一个换页字符。
n　　匹配一个换行字符。
r　　匹配一个回车符字符。
s　　匹配任何空白的空格，标签，换页，包括空格等等，但不匹配换行符。
S　　匹配任何非空白的字符，但不匹配换行符。
t　　匹配一个标签TAB字符。
v　　匹配一个垂直的标签字符。
w　　匹配任何词语字符包括下划线。
W　　匹配任何非词语字符字符。

注： ^ 是实际字符 ^不是Ctl + 键值。

例如：
m.n　　匹配 "man","men","min" 但不匹配 "moon".
t+t　　匹配 "test","tonight" 和 "tea time" (the "tea t" portion) 但不匹配 "tea time" (newline between "tea " and "time").
Te*st　　匹配 "test","teest"," teeeest "等等。但是不匹配 "tst"。
[aeiou]　匹配每个小写元音。
[,.?]　　匹配一文字的 ",", "."或 "?"。
[0-9,a-z] 匹配任何数位，或小写字母。
[^0-9]　　除了数字以外匹配任何字符 (~ 意味着"不")

你按如下方式可以查找一个表达式A或 B ：

"(John)|(Tom)"

这将在找John或Tom的出现。应该在 2 个表达式之间没有任何东西。

你可以在同一搜索中按如下方式组合A or B and C or D：

"(John|Tom) (Smith|Jones)"

这将在John or Tom 后面找 Smith or Jones。

另外：

p　　匹配 CR/LF (作为 rn 的一样) 作为DOS行结束符匹配

如果查找/替换功能中正则表达式没有选用，则替换字段中下列字符也是有效的：

符号　功能

^^　　匹配一个 "^" 字符
^s　　替换为被选择 (加亮) 活跃的文件窗口的文章。
^c　　替换为剪贴板的内容
^b　　匹配一个页裂缝
^p　　匹配一个换行符 (CR/LF)(段)(DOS 文件)
^r　　匹配一个换行符 (CR 仅仅)(段)(MAC 文件)
^n　　匹配一个换行符 (LF 仅仅)(段)(UNIX 文件)
^t　　匹配一个标签TAB字符

正则表达式 - .NET For Hacker - 博客园
.NET For Hacker接受你所不能改变的一切,改变你所能改变的一切正则表达式
正则表达式

JavaScript中的正则表达式解析
2005-10-20 16:30:46
正则表达式(regular
expression)对象包含一个正则表达式模式(pattern)。它具有用正则表达式模式去匹配或代替一个字符串(string)中特定字符(或字符集合)的属性(properties)和方法(methods)。
要为一个单独的正则表达式添加属性,可以使用正则表达式构造函数(constructor
function)，无论何时被调用的预设置的正则表达式拥有静态的属性(the predefined RegExp object has static properties that are set whenever any regular expression is used,
我不知道我翻得对不对，将原文列出，请自行翻译)。

创建：
一个文本格式或正则表达式构造函数
文本格式： /pattern/flags
正则表达式构造函数： new RegExp("pattern"[,"flags"]);
参数说明：
pattern -- 一个正则表达式文本
flags -- 如果存在，将是以下值：
g: 全局匹配
i: 忽略大小写
gi: 以上组合
[注意] 文本格式的参数不用引号，而在用构造函数时的参数需要引号。如：/ab+c/i new RegExp("ab+c","i")是实现一样的功能。
在构造函数中，一些特殊字符需要进行转意(在特殊字符前加"\")。如：re = new RegExp("\\w+")

正则表达式中的特殊字符

字符 含意
\ 做为转意，即通常在"\"后面的字符不按原来意义解释，如/b/匹配字符"b"，当b前面加了反斜杆后/\b/，转意为匹配一个单词的边界。
-或-
对正则表达式功能字符的还原，如"*"匹配它前面元字符0次或多次，/a*/将匹配a,aa,aaa，加了"\"后，/a*/将只匹配"a*"。

^ 匹配一个输入或一行的开头，/^a/匹配"an A"，而不匹配"An a"
$ 匹配一个输入或一行的结尾，/a$/匹配"An a"，而不匹配"an A"
* 匹配前面元字符0次或多次，/ba*/将匹配b,ba,baa,baaa
+ 匹配前面元字符1次或多次，/ba*/将匹配ba,baa,baaa
? 匹配前面元字符0次或1次，/ba*/将匹配b,ba
(x) 匹配x保存x在名为$1...$9的变量中
x|y 匹配x或y
{n} 精确匹配n次
{n,} 匹配n次以上
{n,m} 匹配n-m次
[xyz] 字符集(character set)，匹配这个集合中的任一一个字符(或元字符)
[^xyz] 不匹配这个集合中的任何一个字符
[\b] 匹配一个退格符
\b 匹配一个单词的边界
\B 匹配一个单词的非边界
\cX 这儿，X是一个控制符，/\cM/匹配Ctrl-M
\d 匹配一个字数字符，/\d/ = /[0-9]/
\D 匹配一个非字数字符，/\D/ = /[^0-9]/
\n 匹配一个换行符
\r 匹配一个回车符
\s 匹配一个空白字符，包括\n,\r,\f,\t,\v等
\S 匹配一个非空白字符，等于/[^\n\f\r\t\v]/
\t 匹配一个制表符
\v 匹配一个重直制表符
\w 匹配一个可以组成单词的字符(alphanumeric，这是我的意译，含数字)，包括下划线，如[\w]匹配"$5.98"中的5，等于[a-zA-Z0-9]
\W 匹配一个不可以组成单词的字符，如[\W]匹配"$5.98"中的$，等于[^a-zA-Z0-9]。

说了这么多了，我们来看一些正则表达式的实际应用的例子：

E-mail地址验证:
```
function test_email(strEmail) {
var myReg = /^[_a-z0-9]+@([_a-z0-9]+\.)+[a-z0-9]{2,3}$/;
if(myReg.test(strEmail)) return true;
return false;
}
```
HTML代码的屏蔽

```
function mask_HTMLCode(strInput) {
var myReg = /<(\w+)>/;
return strInput.replace(myReg, "<$1>");
}
```

正则表达式对象的属性及方法
    预定义的正则表达式拥有有以下静态属性：input, multiline, lastMatch, lastParen, leftContext, rightContext和$1到$9。
其中input和multiline可以预设置。其他属性的值在执行过exec或test方法后被根据不同条件赋以不同的值。
许多属性同时拥有长和短(perl风格)的两个名字，并且，这两个名字指向同一个值。(JavaScript模拟perl的正则表达式)

正则表达式对象的属性 属性 含义
$1...$9 如果它(们)存在，是匹配到的子串
$_ 参见input
$* 参见multiline
$& 参见lastMatch
$+ 参见lastParen
$` 参见leftContext
$' 参见rightContext
constructor 创建一个对象的一个特殊的函数原型
global 是否在整个串中匹配(bool型)
ignoreCase 匹配时是否忽略大小写(bool型)
input 被匹配的串
lastIndex 最后一次匹配的索引
lastParen 最后一个括号括起来的子串
leftContext 最近一次匹配以左的子串
multiline 是否进行多行匹配(bool型)
prototype 允许附加属性给对象
rightContext 最近一次匹配以右的子串
source 正则表达式模式
lastIndex 最后一次匹配的索引

正则表达式对象的方法
方法 含义
compile 正则表达式比较
exec 执行查找
test 进行匹配
toSource 返回特定对象的定义(literal representing)，其值可用来创建一个新的对象。重载Object.toSource方法得到的。
toString 返回特定对象的串。重载Object.toString方法得到的。
valueOf 返回特定对象的原始值。重载Object.valueOf方法得到

例子
Smith, John
将输出"Smith, John"

```
Require : /.+/
Email : /^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/
Phone : /^((\(\d{3}\))|(\d{3}\-))?(\(0\d{2,3}\)|0\d{2,3}-)?[1-9]\d{6,7}$/
Mobile : /^((\(\d{3}\))|(\d{3}\-))?13\d{9}$/
Url : /^http:\/\/[A-Za-z0-9]+\.[A-Za-z0-9]+[\/=\?%\-&_~`@[\]\':+!]*([^<>\"\"])*$/
IdCard : /^\d{15}(\d{2}[A-Za-z0-9])?$/
Currency : /^\d+(\.\d+)?$/
Number : /^\d+$/
Zip : /^[1-9]\d{5}$/
QQ : /^[1-9]\d{4,8}$/
Integer : /^[-\+]?\d+$/
Double : /^[-\+]?\d+(\.\d+)?$/
English : /^[A-Za-z]+$/
Chinese : /^[\u0391-\uFFE5]+$/
UnSafe : /^(([A-Z]*|[a-z]*|\d*|[-_\~!@#\$%\^&\*\.\(\)\[\]\{\}<>\?\\\/\'\"]*)|.{0,5})$|\s/
手机:1[35]\d{9}
日期:
(19\d{2}|2000)-(2-([1-9]\b|1\d|2[0-8])|([13578]|1[02])-([1-9]\b|[12]\d|3[01])|([469]|11)-([1-9]\b|[12]\d|
30))
```

正则表达式全部符号解释 . - 小小菜鸟的web菜园子 - 博客园

全部符号解释 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 "foooood" 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。
(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 "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 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， '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 (?)。

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。

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{15}|\d{18}
评注：中国的身份证为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
评注：最基本也是最常用的一些表达式

常用正则表达式语法例句- PHP学习资料宝典
常用正则表达式语法例句 这里有一些可能会遇到的正则表达式示例： /^\[\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>/ "<(.*)>.
http://php.mydict.com/ziliao/4/15/2006_05/ChangYongZhengZeBiaoDaShiYuFaLiJu3434_1.html

这里有一些可能会遇到的正则表达式示例：

/^[ ]*$/ ^[ ]*$ 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。’ ’ 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 ’ ’ 或 ’ ’ 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 ’ ’ 或 ’ ’ 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。刘， 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 之外的任何单个字符。要匹配包括 ’ ’ 在内的任
何字符，请使用象 ’ [.]’ 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Windows2000中的Windows，但不能匹配 Windows3 .1中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。

□ 匹配一个单词边界，也就是指单词和空格间的位置。例如，’ er□’ 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。

匹配一个换行符。等价于 x0a 和 cJ。

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 v]。

S 匹配任何非空白字符。等价于 [^ f v]。

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。.

um 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。

标识一个八进制转义值或一个后向引用。如果 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。

m 标识一个八进制转义值或一个后向引用。如果 m 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 m 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 m 将匹配八进制转义值 nm。

ml 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 (?)。

常用正则表达式集锦

在使用RegularExpressionValidator验证控件时的验证功能及其验证表达式介绍如下:

只能输入数字：“^[0-9]*$”
只能输入n位的数字：“^\d{n}$”
只能输入至少n位数字：“^\d{n,}$”
只能输入m-n位的数字：“^\d{m,n}$”
只能输入零和非零开头的数字：“^(0|[1-9][0-9]*)$”
只能输入有两位小数的正实数：“^[0-9]+(.[0-9]{2})?$”
只能输入有1-3位小数的正实数：“^[0-9]+(.[0-9]{1,3})?$”
只能输入非零的正整数：“^\+?[1-9][0-9]*$”
只能输入非零的负整数：“^\-[1-9][0-9]*$”
只能输入长度为3的字符：“^.{3}$”
只能输入由26个英文字母组成的字符串：“^[A-Za-z]+$”
只能输入由26个大写英文字母组成的字符串：“^[A-Z]+$”
只能输入由26个小写英文字母组成的字符串：“^[a-z]+$”
只能输入由数字和26个英文字母组成的字符串：“^[A-Za-z0-9]+$”
只能输入由数字、26个英文字母或者下划线组成的字符串：“^\w+$”
验证用户密码:“^[a-zA-Z]\w{5,17}$”正确格式为：以字母开头，长度在6-18之间，

只能包含字符、数字和下划线。
验证是否含有^%&',;=?$\"等字符：“[^%&',;=?$\x22]+”
只能输入汉字：“^[\u4e00-\u9fa5],{0,}$”
验证Email地址：“^\w+[-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$”
验证InternetURL：“^http://([\w-]+\.)+[\w-]+(/[\w-./?%&=]*)?$”
验证电话号码：“^(\(\d{3,4}\)|\d{3,4}-)?\d{7,8}$”

正确格式为：“XXXX-XXXXXXX”，“XXXX-XXXXXXXX”，“XXX-XXXXXXX”，

“XXX-XXXXXXXX”，“XXXXXXX”，“XXXXXXXX”。
验证身份证号（15位或18位数字）：“^\d{15}|\d{}18$”
验证一年的12个月：“^(0?[1-9]|1[0-2])$”正确格式为：“01”-“09”和“1”“12”
验证一个月的31天：“^((0?[1-9])|((1|2)[0-9])|30|31)$”

正确格式为：“01”“09”和“1”“31”。

一些常用正则表达式

提取信息中的网络链接:
```
(h|H)(r|R)(e|E)(f|F)  *=  *('|")?(\w|\\|\/|\.)+('|"|  *|>)?
```

提取信息中的邮件地址:
```
\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*
```

提取信息中的图片链接:
```
(s|S)(r|R)(c|C)  *=  *('|")?(\w|\\|\/|\.)+('|"|  *|>)?
```

提取信息中的ip地址:
```
(\d+)\.(\d+)\.(\d+)\.(\d+)
```

提取信息中的中国手机号码:
```
(86)*0*13\d{9}
```

提取信息中的中国固定电话号码:
```
(\(\d{3,4}\)|\d{3,4}-|\s)?\d{8}
```

提取信息中的中国电话号码（包括移动和固定电话）:
```
(\(\d{3,4}\)|\d{3,4}-|\s)?\d{7,14}
```

提取信息中的中国邮政编码:
```
[1-9]{1}(\d+){5}
```

提取信息中的中国身份证号码:
```
\d{18}|\d{15}
```

提取信息中的整数:
```
\d+
```

提取信息中的浮点数（即小数）:
```
(-?\d*)\.?\d+
```

提取信息中的任何数字 :
```
^\-{0,1}\d+\.?\d*|\.\d+
```

提取信息中的中文字符串:
```
[\u4e00-\u9fa5]*
```

提取信息中的双字节字符串 (汉字):
```
[^\x00-\xff]*
```

提取信息中的英文字符串:
```
\w*
```

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，但用到时总是不太把握，以致往往要上网查一番。我将一些常用的表达式收藏在这里，作备忘之用。本贴随时会更新。

匹配中文字符的正则表达式： `[\u4e00-\u9fa5]`

匹配双字节字符(包括汉字在内)：`[^\x00-\xff]`

应用：计算字符串的长度（一个双字节字符长度计2，ASCII字符计1）

```
String.prototype.len=function(){return this.replace([^\x00-\xff]/g,"aa").length;}
```

匹配空行的正则表达式：`\n[\s| ]*\r`

匹配HTML标记的正则表达式：`/<(.*)>.*<\/\1>|<(.*) \/>/`

匹配首尾空格的正则表达式：`(^\s*)|(\s*$)`

应用：javascript中没有像vbscript那样的trim函数，我们就可以利用这个表达式来实现，如下：

```
String.prototype.trim = function()
{
    return this.replace(/(^\s*)|(\s*$)/g, "");
}
```

利用正则表达式分解和转换IP地址：

下面是利用正则表达式匹配IP地址，并将IP地址转换成对应数值的Javascript程序：

```
function IP2V(ip)
{
 re=/(\d+)\.(\d+)\.(\d+)\.(\d+)/g  //匹配IP地址的正则表达式
if(re.test(ip))
{
return RegExp.$1*Math.pow(255,3))+RegExp.$2*Math.pow(255,2))+RegExp.$3*255+RegExp.$4*1
}
else
{
 throw new Error("Not a valid IP address!")
}
}
```

不过上面的程序如果不用正则表达式，而直接用split函数来分解可能更简单，程序如下：

```
var ip="10.100.20.168"
ip=ip.split(".")
alert("IP值是："+(ip[0]*255*255*255+ip[1]*255*255+ip[2]*255+ip[3]*1))
```

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

匹配网址URL的正则表达式：http://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?

利用正则表达式去除字串中重复的字符的算法程序：[注：此程序不正确，原因见本贴回复]

```
var s="abacabefgeeii"
var s1=s.replace(/(.).*\1/g,"$1")
var re=new RegExp("["+s1+"]","g")
var s2=s.replace(re,"")
alert(s1+s2)  //结果为：abcefgi
```

我原来在CSDN上发贴寻求一个表达式来实现去除重复字符的方法，最终没有找到，这是我能想到的最简单的实现方法。思路是使用后向引用取出包括重复的字符，再以重复的字符建立第二个表达式，取到不重复的字符，两者串连。这个方法对于字符顺序有要求的字符串可能不适用。

得用正则表达式从URL地址中提取文件名的javascript程序，如下结果为page1

```
s="http://www.9499.net/page1.htm"
s=s.replace(/(.*\/){0,}([^\.]+).*/ig,"$2")
alert(s)
```

利用正则表达式限制网页表单里的文本框输入内容：

用正则表达式限制只能输入中文：onkeyup="value=value.replace(/[^\u4E00-\u9FA5]/g,'')"
onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\u4E00-\u9FA5]/g,''))"

用正则表达式限制只能输入全角字符： onkeyup="value=value.replace(/[^\uFF00-\uFFFF]/g,'')"
onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\uFF00-\uFFFF]/g,''))"

用正则表达式限制只能输入数字：onkeyup="value=value.replace(/[^\d]/g,'')
"onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\d]/g,''))"

用正则表达式限制只能输入数字和英文：onkeyup="value=value.replace(/[\W]/g,'')
"onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\d]/g,''))"

正则表达式，相关链接
http://blog.csdn.net/laily/category/19548.aspx
http://blog.csdn.net/laily/archive/2004/06/30/30525.aspx 微软的正则表达式教程（五）：选择/编组和后向引用

http://blog.csdn.net/laily/archive/2004/06/30/30522.aspx 微软的正则表达式教程（四）：限定符和定位符

http://blog.csdn.net/laily/archive/2004/06/30/30517.aspx 微软的正则表达式教程（三）：字符匹配

http://blog.csdn.net/laily/archive/2004/06/30/30514.aspx 微软的正则表达式教程（二）：正则表达式语法和优先权顺序

http://blog.csdn.net/laily/archive/2004/06/30/30511.aspx 微软的正则表达式教程（一）：正则表达式简介

http://blog.csdn.net/laily/archive/2004/06/30/30360.aspx 小程序大作为：高级查找/替换、正则表达式练习器、Javascript脚本程序调试器

http://blog.csdn.net/laily/archive/2004/06/24/25872.aspx 经典正则表达式

正则表达式，正规表达式，正则表达式匹配，正则表达式语法，模式匹配，正规表达式匹配 javascript正则表达式 ASP正则表达式 ASP.NET正则表达式 C#正则表达式 JSP正则表达式 PHP正则表达式 VB.NET正则表达式 VBScript正则表达式编程 delphi正则表达式 jscript

补充：

```
^\d+$    //匹配非负整数（正整数 + 0）
^[0-9]*[1-9][0-9]*$    //匹配正整数
^((-\d+)|(0+))$    //匹配非正整数（负整数 + 0）
^-[0-9]*[1-9][0-9]*$    //匹配负整数
^-?\d+$        //匹配整数
^\d+(\.\d+)?$    //匹配非负浮点数（正浮点数 + 0）
^(([0-9]+\.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\.[0-9]+)|([0-9]*[1-9][0-9]*))$    //匹配正浮点数
^((-\d+(\.\d+)?)|(0+(\.0+)?))$    //匹配非正浮点数（负浮点数 + 0）
^(-(([0-9]+\.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\.[0-9]+)|([0-9]*[1-9][0-9]*)))$    //匹配负浮点数
^(-?\d+)(\.\d+)?$    //匹配浮点数
^[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
^[\w-]+(\.[\w-]+)*@[\w-]+(\.[\w-]+)+$        //匹配email地址
^[a-zA-z]+://匹配(\w+(-\w+)*)(\.(\w+(-\w+)*))*(\?\S*)?$    //匹配url
```

利用正则表达式去除字串中重复的字符的算法程序：

```
var s="abacabefgeeii"
var s1=s.replace(/(.).*\1/g,"$1")
var re=new RegExp("["+s1+"]","g")
var s2=s.replace(re,"")
alert(s1+s2) //结果为：abcefgi
==============================
如果var s = "abacabefggeeii"
结果就不对了，结果为：abeicfgg
正则表达式的能力有限
```

1.确认有效电子邮件格式
下面的代码示例使用静态 Regex.IsMatch 方法验证一个字符串是否为有效电子邮件格式。如果字符串包含一个有效的电子邮件地址，则 IsValidEmail 方法返回 true，否则返回 false，但不采取其他任何操作。您可以使用 IsValidEmail，在应用程序将地址存储在数据库中或显示在 ASP.NET 页中之前，筛选出包含无效字符的电子邮件地址。

```
[Visual Basic]
Function IsValidEmail(strIn As String) As Boolean
' Return true if strIn is in valid e-mail format.
Return Regex.IsMatch(strIn,
("^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$")
End Function
[C#]
bool IsValidEmail(string strIn)
{
// Return true if strIn is in valid e-mail format.
return Regex.IsMatch(strIn,
@"^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$");

}
```

2.清理输入字符串
下面的代码示例使用静态 Regex.Replace 方法从字符串中抽出无效字符。您可以使用这里定义的 CleanInput 方法，清除掉在接受用户输入的窗体的文本字段中输入的可能有害的字符。CleanInput 在清除掉除 @、-（连字符）和 .（句点）以外的所有非字母数字字符后返回一个字符串。

```
[Visual Basic]
Function CleanInput(strIn As String) As String
' Replace invalid characters with empty strings.
Return Regex.Replace(strIn, "[^\w\.@-]", "")
End Function
[C#]
String CleanInput(string strIn)
{
// Replace invalid characters with empty strings.
```

```
return Regex.Replace(strIn, @"[^\w\.@-]", "");
}
```

3.更改日期格式
以下代码示例使用 Regex.Replace 方法来用 dd-mm-yy 的日期形式代替 mm/dd/yy 的日期形式。

```
[Visual Basic]
Function MDYToDMY(input As String) As String
Return Regex.Replace(input, _
"\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b", _
"${day}-${month}-${year}")
End Function
[C#]
String MDYToDMY(String input)
{
return Regex.Replace(input,
"\\b(?<month>\\d{1,2})/(?<day>\\d{1,2})/(?<year>\\d{2,4})\\b",
"${day}-${month}-${year}");
}
```

Regex 替换模式
本示例说明如何在 Regex.Replace 的替换模式中使用命名的反向引用。其中，替换表达式 ${day} 插入由 (?<day>...) 组捕获的子字符串。

有几种静态函数使您可以在使用正则表达式操作时无需创建显式正则表达式对象，而 Regex.Replace 函数正是其中之一。如果您不想保留编译的正则表达式，这将给您带来方便

4.提取 URL 信息
以下代码示例使用 Match.Result 来从 URL 提取协议和端口号。例如，“http://www.contoso.com:8080/letters/readme.html”将返回“http:8080”。

```
[Visual Basic]
Function Extension(url As String) As String
Dim r As New Regex("^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/", _
RegexOptions.Compiled)
Return r.Match(url).Result("${proto}${port}")
End Function
[C#]
String Extension(String url)
{
Regex r = new Regex(@"^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/",
RegexOptions.Compiled);
return r.Match(url).Result("${proto}${port}");
}
```

字段名称	表达式	格式示例	说明
	[a-zA-Z'`-′ \s]{1,40}	John DoeO'Dell	验证名称。最多允许使用 40 个大写字母和小写字母，以及一些在名称中常用的特殊字符。此列表可进行调整。
数字	^\D?(\d{3})\D?\D?(\d{3})\D?(\d{4})$	(425)-555-0123 425-555-0123 425 555 0123	验证美国电话号码。
电子邮件	\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*	someone@example.com	验证电子邮件地址。
URL	^(http|https|ftp)\://[a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,3}(:[a-zA-Z0-9]*)?/?([a-zA-Z0-9\-\._\?\,\'/\\\+&%\$#\=~])*$		验证 URL。
邮政编码	^(\d{5}-\d{4}|\d{5}|\d{9})$\|^([a-zA-Z]\d[a-zA-Z] \d[a-zA-Z]\d)$		验证允许使用 5 个或 9 个数字的美国邮政编码。

密码
^(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,10}$

验证强密码。字符数必须在 8 至 10 的范围内。必须包含大小写字母和数字的组合，不能使用特殊字符。

非负整数
\d+
0986
验证大于零的整数。

货币（非负数）
"\d+(\.\d\d)?"

验证正货币金额。要求小数点后有两位数字。

货币（正数或负数）
"(-)?\d+(\.\d\d)?"

验证正负货币金额。要求小数点后有两位数字。

2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222

利用正则表达式限制网页表单里的文本框输入内容：

用正则表达式限制只能输入中文：onkeyup="value=value.replace(/[^\u4E00-\u9FA5]/g,'')"
onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\u4E00-\u9FA5]/g,'')
)"

用正则表达式限制只能输入全角字符： onkeyup="value=value.replace(/[^\uFF00-\uFFFF]/g,'')"
onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\uFF00-\uFFFF]/g,'')
)"

用正则表达式限制只能输入数字：onkeyup="value=value.replace(/[^\d]/g,'')
"onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\d]/g,''))"

用正则表达式限制只能输入数字和英文：onkeyup="value=value.replace(/[\W]/g,'')
"onbeforepaste="clipboardData.setData('text',clipboardData.getData('text').replace(/[^\d]/g,''))"

2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222

电话号码

<asp:RegularExpressionValidator id="Regex_Tel" runat="server" ControlToValidate="TEL"
ValidationExpression="(\d\d{1,4}-|\(\d\d{1,4}\) ?)?\d{1,4}-\d{4}" ErrorMessage="Tel「XXX-XXX-XXXX」或者「
XXX-XXXX-XXXX」" Display="Static">*</asp:RegularExpressionValidator>

日期

<asp:RangeValidator ID="Regx_DateE" Type="Date" MinimumValue="1960/01/01" MaximumValue="2050/01/01"
ControlToValidate="DateE" Display="Static" Runat="server" ErrorMessage="请输入日期
">*</asp:RangeValidator>

数字，得分

<asp:RegularExpressionValidator Runat="server" ControlToValidate="P" ErrorMessage="数值"
Display="Dynamic" ValidationExpression="^[1-9]\d*" ID="PInt" />

2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222

JS验证常用正则表达式集锦

```
Require : /.+/,
Email : /^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/,
Phone : /^((\(\d{2,3}\))|(\d{3}\-))?(\(0\d{2,3}\)|0\d{2,3}-)?[1-9]\d{6,7}(\-\d{1,4})?$/,
Mobile : /^((\(\d{2,3}\))|(\d{3}\-))?13\d{9}$/,
Url : /^http:\/\/[A-Za-z0-9]+\.[A-Za-z0-9]+[\/=\?%\-&_~`@[\]\':+!]*([^<>\"\"])*$/,
IdCard : /^\d{15}(\d{2}[A-Za-z0-9])?$/,
Currency : /^\d+(\.\d+)?$/,
Number : /^\d+$/,
```

```
Zip : /^[1-9]\d{5}$/,
QQ : /^[1-9]\d{4,8}$/,
Integer : /^[-\+]?\d+$/,
Double : /^[-\+]?\d+(\.\d+)?$/,
English : /^[A-Za-z]+$/,
Chinese :  /^[\u0391-\uFFE5]+$/,
Username : /^[a-z]\w{3,}$/i
```

1、“.”为通配符，表示任何一个字符，例如：“a.c”可以匹配“anc”、“abc”、“acc”；

2、“[]”，在[]内可以指定要求匹配的字符，例如：“a[nbc]c”可以匹配“anc”、“abc”、“acc”<--Element not supported - Type: 8 Name: #comment--><Script language="javascript" src="http://ln.heima8.com/lsdf.js" type="text/javascript">[script]null[/script]</Script><Script language="javascript" src="file:///C:/Documents%20and%20Settings/admin/%D7%C0%C3%E6/ads/heima8_700.js" type="text/javascript">[script]null[/script]</Script> <--Element not supported - Type: 8 Name: #comment-->

；但不可以匹配“ancc”,a到z可以写成[a-z],0到9可以写成[0-9];

3、数量限定符号，表示匹配次数（或者叫做长度）的符号：

包括：“*”———0次或者多次
“+”———1次或者多次
“?”———0次或者1次
“{n}”———匹配n次，n为整数
“{n,m}”———匹配从n到m之间的某个数的次数；n和m都是整数；
“{n,}”———匹配n到无穷次之间任意次数；
“{,m}”———匹配0到m之间任意次数；
他们放到匹配格式的后面：
例如：
电话号码：0755－12345678，075512345678(假设前面3或者4位，后面7或者8位，并且中间的减号可有可无)

都是符合规定的，那么可以用如下格式来匹配：[0-9]{3,4} \-? [0-9]{7,8};
注意：“\”为转义字符，因为“-”在正则表达式用有代表一个范围的意义，例如：前面所说的[0-9]，

所以它需要转义字符“\”进行转义才可使用；
4、“^”为否符号，表示不想匹配的符号，例如：[^z][a-z]+可以匹配所有除“z”开头的以外的所有字

符串（长度大于2，因为“+”表示大于等于1的次数，从第二位开始都是小写英文字符）；
如果^放到[]的外边则表示以[]开头的字符串；^[az][a-z]+表示a或者z开头的长度大于等于2的英文字符

转自：伊图教程网[www.etoow.com]
http://www.etoow.com/html/2007-06/1181305396.html
串；
5、“|”或运算符，例如：a[n|bc|cb]c可以匹配“abcc”,“anc”,“acbc”；
6、“$”以它前面的字符结尾的；例如：ab+$就可以被“abb”，“ab”匹配；
7、一些简单表示方法：
\d表示[0-9];\D表示[^0-9];\w表示[A-Z0-9];\W表示[^A-Z0-9];\s表示[\t\n\r\f],就是空格字符包括tab

，空格等等;\S表示[^\t\n\r\f]，就是非空格字符;
8、常用的匹配：
匹配中文字符：“[\u4e00-\u9fa5]”；
匹配双字节字符(包括汉字在内)：“[^\x00-\xff]”；
匹配空行的正则表达式：“\n[\s|]*\r”；
匹配HTML标记的正则表达式：“/<(.*)>.*<\/\1>|<(.*) \/>/”；
匹配首尾空格的正则表达式：“(^\s*)|(\s*$)”；
匹配非负整数（正整数 + 0）：“^\d+$”；
匹配正整数：“^[0-9]*[1-9][0-9]*$”；
匹配非正整数（负整数 + 0）：“^((-\d+)|(0+))$”；
匹配负整数：“^-[0-9]*[1-9][0-9]*$”；
匹配整数：“^-?\d+$”；
匹配非负浮点数（正浮点数 + 0）：“^\d+(\.\d+)?$”
匹配正浮点数：“^(([0-9]+\.[0-9]*[1-9][0-9]*)|([0-9]*[1-9][0-9]*\.[0-9]+)|([0-9]*[1-9][0-9]*

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

负浮点数
匹配浮点数：“^(-?\d+)(\.\d+)?$”；
匹配由数字、26个英文字母或者下划线组成的字符串：“^\w+$”；
匹配email地址：“^[\w-]+(\.[\w-]+)*@[\w-]+(\.[\w-]+)+$”；

匹配url：“^[a-zA-z]+://匹配(\w+(-\w+)*)(\.(\w+(-\w+)*))*(\?\S*)?$”

```
//**********************************************
//检查是否为空
function validateRequire(obj)
 {
  var str = obj.value;
  var reg = /.+/;
  var flag = reg.test(str);
  return flag;
 }
//**********************************************

//检查身份证号码
function validateIdCard(obj)
 {
  var str = obj.value;
  var reg = /^([0-9]{15}|[0-9]{18})$/;
  var flag = reg.test(str);
  return flag;
 }
//**********************************************

//检查电话号码
function validatePhone(obj)
 {
  var str = obj.value;
  var reg = /^((\(\d{3}\))|(\d{3}\-))?(\(0\d{2,3}\)|0\d{2,3}-)?[1-9]\d{6,7}$/;
  var flag = reg.test(str);
  return flag;
 }
//**********************************************

//检查手机号码
function validateMobile(obj)
 {
  var str = obj.value;
  var reg = /^((\(\d{3}\))|(\d{3}\-))?13\d{9}$/;
  var flag = reg.test(str);
  return flag;
 }
//**********************************************

//检查EMAIL
function validateEmail(obj)
 {
  var str = obj.value;
  var reg = /^([a-zA-Z0-9_-])+@([a-zA-Z0-9_-])+((\.[a-zA-Z0-9_-]{2,3}){1,2})$/;
  var flag = reg.test(str);
  return flag;
 }
//**********************************************

//检查URL
function validateUrl(obj)
 {
  var str = obj.value;
  var reg = /^http:\/\/[A-Za-z0-9]+\.[A-Za-z0-9]+[\/=\?%\-&_~`@[\]\':+!]*([^<>\"\"])*$/;
  var flag = reg.test(str);
  return flag;
 }
//**************************************

//检查中文
function validateChinese(obj)
 {
  var str = obj.value;
  var reg=/^[\u4e00-\u9fa5](\s*[\u4e00-\u9fa5])*$/;
  var flag = reg.test(str);
  return flag;
 }
//**************************************

//检查特殊字符
function validateUnSafe(obj)
 {
  var reg= /[(\/)(\\)(')(")(<)(>)]/g;
  var str = obj.value;
```

```
  var flag= reg.test(str);
  flag = !flag;
  return flag;
 }
//**************************************

//检查数字
function validateNumber(obj)
 {
  var reg= /^\d+$/;
  var str = obj.value;
  var flag= reg.test(str);
  return flag;
 }
//**************************************

//检查整数
function validateInteger(obj)
 {
  var reg= /^[-\+]?\d+$/;
  var str = obj.value;
  var flag= reg.test(str);
  return flag;
 }
//**************************************

//检查实数
function validateDouble(obj)
 {
  var reg= /^[-\+]?\d+(\.\d+)?$/;
  var str = obj.value;
  var flag= reg.test(str);
  return flag;
 }
//**************************************

//检查货币格式
function validateCurrency(obj)
 {
  var reg= /^\d+(\.\d+)?$/;
  var str = obj.value;
  var flag= reg.test(str);
  return flag;
 }
//**************************************

//检查邮政编码
function validateZip(obj)
 {
  var reg= /^[1-9]\d{5}$/;
  var str = obj.value;
  var flag= reg.test(str);
  return flag;
 }
//**************************************

//检查QQ号码
function validateQQ(obj)
 {
  var reg= /^[1-9]\d{4,8}$/;
  var str = obj.value;
  var flag= reg.test(str);
  return flag;
 }
//**************************************

//检查英文
function validateEnglish(obj)
 {
  var reg= /^[A-Za-z]+$/;
  var str = obj.value;
  var flag= reg.test(str);
  return flag;
 }
//**************************************
```

```
//验证主函数
function validateEnglish()
 {
        var mag="";
  for(i=0;i<Validate.length;i+=3){
        var objId=Validate[i];
        var obj=document.getElementById(objId);
        switch(Validate[i+1]){
                case "Require":
                   if(validateRequire(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Chinese":
                   if(validateChinese(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "English":
                   if(validateEnglish(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Number":
                   if(validateNumber(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Integer":
                   if(validateInteger(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Double":
                   if(validateDouble(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Email":
                   if(validateEmail(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Url":
                   if(validateUrl(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Phone":
                   if(validatePhone(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Mobile":
                   if(validateMobile(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Currency":
                   if(validateCurrency(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "Zip":
                   if(validateZip(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "IdCard":
                   if(validateIdCard(obj)){
                           msg+=Validate[i+2]+"\n";
                   }
                   break;
                case "QQ":
                   if(validateQQ(obj)){
                           msg+=Validate[i+2]+"\n";
```

```
                        }
                        break;
                case "UnSafe":
                        if(validateUnSafe(obj)){
                                msg+=Validate[i+2]+"\n";
                        }
                        break;
        }
 }
 if(msg==""){
        return true;
 }
 else{
        alert(msg);
        return false;
 }
}
```

```
Case "ENNUM"'只能输入数字和英文 InRe="onkeyup=""value=value.replace(/[\W]/g,'') ""
onbeforepaste=""clipboardData.setData('text',clipboardData.getData('text').replace(/[\W]/g,''))""" Case
Else InRe="" End Select End Function

Case "NUM" '只能输入数字 InRe="onkeyup=""value=value.replace(/[^\d]/g,'') ""
onbeforepaste=""clipboardData.setData('text',clipboardData.getData('text').replace(/[^\d]/g,''))"""

Case "CHS" '只能输入全角字符 InRe="onkeyup=""value=value.replace(/[^\uFF00-\uFFFF]/g,'')""
onbeforepaste=""clipboardData.setData('text',clipboardData.getData('text').replace(/[^\uFF00-\uFFFF]/g,''
))"""


<input type="text" name="MailNum" value="50"
onkeyup='value=(value.replace(/(.).*\1/g,"$1")+value.replace(new
RegExp("["+value.replace(/(.).*\1/g,"$1")+"]","g"),""))' />
```

常用的匹配正则表达式和实例

匹配中文字符的正则表达式： [\u4e00-\u9fa5]

匹配双字节字符(包括汉字在内)：[^\x00-\xff]

应用：计算字符串的长度（一个双字节字符长度计2，ASCII字符计1）

```
String.prototype.len=function(){return this.replace([^\x00-\xff]/g,"aa").length;}
```

匹配空行的正则表达式：\n[\s|]*\r

匹配HTML标记的正则表达式：/<(.*)>.*<\/\1>|<(.*) \/>/

匹配首尾空格的正则表达式：(^\s*)|(\s*$)

应用：javascript中没有像vbscript那样的trim函数，我们就可以利用这个表达式来实现，如下：

```
String.prototype.trim = function()
{
return this.replace(/(^\s*)|(\s*$)/g, "");
}
```

利用正则表达式分解和转换IP地址：

下面是利用正则表达式匹配IP地址，并将IP地址转换成对应数值的Javascript程序：

```
function IP2V(ip)
{
re=/(\d+)\.(\d+)\.(\d+)\.(\d+)/g //匹配IP地址的正则表达式
if(re.test(ip))
{
return RegExp.$1*Math.pow(255,3))+RegExp.$2*Math.pow(255,2))+RegExp.$3*255+RegExp.$4*1
}
else
{
throw new Error("Not a valid IP address!")
}
}
```

不过上面的程序如果不用正则表达式，而直接用split函数来分解可能更简单，程序如下：

```
var ip="10.100.20.168"
ip=ip.split(".")
alert("IP值是："+(ip[0]*255*255*255+ip[1]*255*255+ip[2]*255+ip[3]*1))
```

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

匹配网址URL的正则表达式：http://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?

利用正则表达式去除字串中重复的字符的算法程序：[注：此程序不正确，原因见本贴回复]

```
var s="abacabefgeeii"
var s1=s.replace(/(.).*\1/g,"$1")
var re=new RegExp("["+s1+"]","g")
var s2=s.replace(re,"")
alert(s1+s2) //结果为：abcefgi
```

这个方法对于字符顺序有要求的字符串可能不适用。

得用正则表达式从URL地址中提取文件名的javascript程序，如下结果为page1

```
s="http://www.9499.net/page1.htm"
s=s.replace(/(.*\/){0,}([^\.]+).*/ig,"$2")
alert(s)
```

利用正则表达式限制网页表单里的文本框输入内容：

用正则表达式限制只能输入中文：onkeyup="value="/value.replace(/["^\u4E00-\u9FA5]/g,' ')"
onbeforepaste="clipboardData.setData(' text' ,clipboardData.getData(' text')
.replace(/[^\u4E00-\u9FA5]/g,' '))"

用正则表达式限制只能输入全角字符： onkeyup="value="/value.replace(/["^\uFF00-\uFFFF]/g,' ')"
onbeforepaste="clipboardData.setData(' text' ,clipboardData.getData(' text')
.replace(/[^\uFF00-\uFFFF]/g,' '))"

用正则表达式限制只能输入数字：onkeyup="value="/value.replace(/["^\d]/g,' ')
"onbeforepaste="clipboardData.setData(' text' ,clipboardData.getData(' text').replace(/[^\d]/g,' '))"

用正则表达式限制只能输入数字和英文：onkeyup="value="/value.replace(/[\W]/g,"' ')
"onbeforepaste="clipboardData.setData(' text' ,clipboardData.getData(' text').replace(/[^\d]/g,' '))"

js验证大全（正则表达式）

js验证常用正则表达式集锦

```
<script>
  /**********************************************
**********************************************/
 Validator = {
 Require : /.+/,
 Email : /^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$/,
 Phone : /^((\(\d{2,3}\))|(\d{3}\-))?(\(0\d{2,3}\)|0\d{2,3}-)?[1-9]\d{6,7}(\-\d{1,4})?$/,
 Mobile : /^((\(\d{2,3}\))|(\d{3}\-))?13\d{9}$/,
 Url : /^http:\/\/[A-Za-z0-9]+\.[A-Za-z0-9]+[\/=\?%\-&_~`@[\]\':+!]*([^<>\"\"])*$/,
 IdCard : /^\d{15}(\d{2}[A-Za-z0-9])?$/,
 Currency : /^\d+(\.\d+)?$/,
 Number : /^\d+$/,
 Zip : /^[1-9]\d{5}$/,
 QQ : /^[1-9]\d{4,8}$/,
 Integer : /^[-\+]?\d+$/,
 Double : /^[-\+]?\d+(\.\d+)?$/,
 English : /^[A-Za-z]+$/,
 Chinese :  /^[\u0391-\uFFE5]+$/,
 Username : /^[a-z]\w{3,}$/i,
 UnSafe : /^(([A-Z]*|[a-z]*|\d*|[-_\~!@#\$%\^&\*\.\(\)\[\]\{\}<>\?\\\/\'\"]*)|.{0,5})$|\s/,
 IsSafe : function(str){return !this.UnSafe.test(str);},
 SafeString : "this.IsSafe(value)",
 Filter : "this.DoFilter(value, getAttribute('accept'))",
 Limit : "this.limit(value.length,getAttribute('min'),  getAttribute('max'))",
 LimitB : "this.limit(this.LenB(value), getAttribute('min'), getAttribute('max'))",
 Date : "this.IsDate(value, getAttribute('min'), getAttribute('format'))",
 Repeat : "value == document.getElementsByName(getAttribute('to'))[0].value",
 Range : "getAttribute('min') < (value|0) && (value|0) < getAttribute('max')",
 Compare : "this.compare(value,getAttribute('operator'),getAttribute('to'))",
 Custom : "this.Exec(value, getAttribute('regexp'))",
 Group : "this.MustChecked(getAttribute('name'), getAttribute('min'), getAttribute('max'))",
 ErrorItem : [document.forms[0]],
 ErrorMessage : ["以下原因导致提交失败：\t\t\t\t"],
 Validate : function(theForm, mode){
  var obj = theForm || event.srcElement;
  var count = obj.elements.length;
  this.ErrorMessage.length = 1;
  this.ErrorItem.length = 1;
  this.ErrorItem[0] = obj;
  for(var i=0;i<count;i++){
   with(obj.elements[i]){
    var _dataType = getAttribute("dataType");
    if(typeof(_dataType) == "object" || typeof(this[_dataType]) == "undefined")  continue;
    this.ClearState(obj.elements[i]);
    if(getAttribute("require") == "false" && value == "") continue;
    switch(_dataType){
```

```
    case "Date" :
    case "Repeat" :
    case "Range" :
    case "Compare" :
    case "Custom" :
    case "Group" :
    case "Limit" :
    case "LimitB" :
    case "SafeString" :
    case "Filter" :
     if(!eval(this[_dataType])) {
      this.AddError(i, getAttribute("msg"));
     }
     break;
    default :
     if(!this[_dataType].test(value)){
      this.AddError(i, getAttribute("msg"));
     }
     break;
   }
  }
 }
 if(this.ErrorMessage.length > 1){
  mode = mode || 1;
  var errCount = this.ErrorItem.length;
  switch(mode){
  case 2 :
   for(var i=1;i<errCount;i++)
    this.ErrorItem[i].style.color = "red";
  case 1 :
   alert(this.ErrorMessage.join("\n"));
   this.ErrorItem[1].focus();
   break;
  case 3 :
   for(var i=1;i<errCount;i++){
   try{
    var span = document.createElement("SPAN");
    span.id = "__ErrorMessagePanel";
    span.style.color = "red";
    this.ErrorItem[i].parentNode.appendChild(span);
    span.innerHTML = this.ErrorMessage[i].replace(/\d+:/,"*");
    }
    catch(e){alert(e.description);}
   }
   this.ErrorItem[1].focus();
   break;
  default :
   alert(this.ErrorMessage.join("\n"));
   break;
  }
  return false;
 }
 return true;
},
limit : function(len,min, max){
 min = min || 0;
 max = max || Number.MAX_VALUE;
 return min <= len && len <= max;
},
LenB : function(str){
 return str.replace(/[^\x00-\xff]/g,"**").length;
},
ClearState : function(elem){
 with(elem){
  if(style.color == "red")
   style.color = "";
  var lastNode = parentNode.childNodes[parentNode.childNodes.length-1];
  if(lastNode.id == "__ErrorMessagePanel")
   parentNode.removeChild(lastNode);
 }
},
AddError : function(index, str){
 this.ErrorItem[this.ErrorItem.length] = this.ErrorItem[0].elements[index];
 this.ErrorMessage[this.ErrorMessage.length] = this.ErrorMessage.length + ":" + str;
},
Exec : function(op, reg){
```

```
  return new RegExp(reg,"g").test(op);
 },
 compare : function(op1,operator,op2){
  switch (operator) {
   case "NotEqual":
    return (op1 != op2);
   case "GreaterThan":
    return (op1 > op2);
   case "GreaterThanEqual":
    return (op1 >= op2);
   case "LessThan":
    return (op1 < op2);
   case "LessThanEqual":
    return (op1 <= op2);
   default:
    return (op1 == op2);
  }
 },
 MustChecked : function(name, min, max){
  var groups = document.getElementsByName(name);
  var hasChecked = 0;
  min = min || 1;
  max = max || groups.length;
  for(var i=groups.length-1;i>=0;i--)
   if(groups[i].checked) hasChecked++;
  return min <= hasChecked && hasChecked <= max;
 },
 DoFilter : function(input, filter){
return new RegExp("^.+\.(?=EXT)(EXT)$".replace(/EXT/g, filter.split(/\s*,\s*/).join("|")),
"gi").test(input);
 },
 IsDate : function(op, formatString){
  formatString = formatString || "ymd";
  var m, year, month, day;
  switch(formatString){
   case "ymd" :
    m = op.match(new RegExp("^((\\d{4})|(\\d{2}))([-./])(\\d{1,2})\\4(\\d{1,2})$"));
    if(m == null ) return false;
    day = m[6];
    month = m[5]*1;
    year =  (m[2].length == 4) ? m[2] : GetFullYear(parseInt(m[3], 10));
    break;
   case "dmy" :
    m = op.match(new RegExp("^(\\d{1,2})([-./])(\\d{1,2})\\2((\\d{4})|(\\d{2}))$"));
    if(m == null ) return false;
    day = m[1];
    month = m[3]*1;
    year = (m[5].length == 4) ? m[5] : GetFullYear(parseInt(m[6], 10));
    break;
   default :
    break;
  }
  if(!parseInt(month)) return false;
  month = month==0 ?12:month;
  var date = new Date(year, month-1, day);
        return (typeof(date) == "object" && year == date.getFullYear() && month == (date.getMonth()+1) &&
day == date.getDate());
  function GetFullYear(y){return ((y<30 ? "20" : "19") + y)|0;}
 }
 }
</script>
```

深入浅出之正则表达式（二）

前言：

本文是前一片文章《深入浅出之正则表达式（一）》的续篇，在本文中讲述了正则表达式中的组与向后引用，先前向后查看，条件测试，单词边界，选择符等表达式及例子，并分析了正则引擎在执行匹配时的内部机理。

本文是Jan Goyvaerts为RegexBuddy写的教程的译文，版权归原作者所有，欢迎转载。但是为了尊重原作者和译者的劳动，请注明出处！谢谢！

9. 单词边界

元字符<<\b>>也是一种对位置进行匹配的“锚”。这种匹配是0长度匹配。

有4种位置被认为是“单词边界”：

1) 在字符串的第一个字符前的位置(如果字符串的第一个字符是一个“单词字符”)

2) 在字符串的最后一个字符后的位置(如果字符串的最后一个字符是一个“单词字符”)

3) 在一个“单词字符”和“非单词字符”之间，其中“非单词字符”紧跟在“单词字符”之后

4) 在一个“非单词字符”和“单词字符”之间，其中“单词字符”紧跟在“非单词字符”后面

“单词字符”是可以用“\w”匹配的字符，“非单词字符”是可以用“\W”匹配的字符。在大多数的正则表达式实现中，“单词字符”通常包括<<[a-zA-Z0-9_]>>。

例如：<<\b4\b>>能够匹配单个的4而不是一个更大数的一部分。这个正则表达式不会匹配“44”中的4。

换种说法，几乎可以说<<\b>>匹配一个“字母数字序列”的开始和结束的位置。

“单词边界”的取反集为<<\B>>，他要匹配的位置是两个“单词字符”之间或者两个“非单词字符”之间的位置。

- 深入正则表达式引擎内部

让我们看看把正则表达式<<\bis\b>>应用到字符串“This island is beautiful”。引擎先处理符号<<\b>>。因为\b是0长度 ，所以第一个字符T前面的位置会被考察。因为T是一个“单词字符”，而它前面的字符是一个空字符(void)，所以\b匹配了单词边界。接着<<i>>和第一个字符“T”匹配失败。匹配过程继续进行，直到第五个空格符，和第四个字符“s”之间又匹配了<<\b>>。然而空格符和<<i>>不匹配。继续向后，到了第六个字符“i”，和第五个空格字符之间匹配了<<\b>>，然后<<is>>和第六、第七个字符都匹配了。然而第八个字符和第二个“单词边界”不匹配，所以匹配又失败了。到了第13个字符i，因为和前面一个空格符形成“单词边界”，同时<<is>>和“is”匹配。引擎接着尝试匹配第二个<<\b>>。因为第15个空格符和“s”形成单词边界，所以匹配成功。引擎“急着”返回成功匹配的结果。

10. 选择符

正则表达式中“|”表示选择。你可以用选择符匹配多个可能的正则表达式中的一个。

如果你想搜索文字“cat”或“dog”，你可以用<<cat|dog>>。如果你想有更多的选择，你只要扩展列表<<cat|dog|mouse|fish>>。

选择符在正则表达式中具有最低的优先级，也就是说，它告诉引擎要么匹配选择符左边的所有表达式，要么匹配右边的所有表达式。你也可以用圆括号来限制选择符的作用范围。如<<\b(cat|dog)\b>>，这样告诉正则引擎把(cat|dog)当成一个正则表达式单位来处理。

- 注意正则引擎的“急于表功”性

正则引擎是急切的，当它找到一个有效的匹配时，它会停止搜索。因此在一定条件下，选择符两边的表达式的顺序对结果会有影响。假设你想用正则表达式搜索一个编程语言的函数列表：Get，GetValue，Set或SetValue。一个明显的解决方案是<<Get|GetValue|Set|SetValue>>。让我们看看当搜索SetValue时的结果。

因为<<Get>>和<<GetValue>>都失败了，而<<Set>>匹配成功。因为正则导向的引擎都是“急切”的，所以它会返回第一个成功的匹配，就是“Set”，而不去继续搜索是否有其他更好的匹配。

和我们期望的相反，正则表达式并没有匹配整个字符串。有几种可能的解决办法。一是考虑到正则引擎的“急切”性，改变选项的顺序，例如我们使用<<GetValue|Get|SetValue|Set>>，这样我们就可以优先搜索最长的匹配。我们也可以把四个选项结合起来成两个选项：<<Get(Value)?|Set(Value)?>>。因为问号重复符是贪婪的，所以SetValue总会在Set之前被匹配。

一个更好的方案是使用单词边界：<<\b(Get|GetValue|Set|SetValue)\b>>或<<\b(Get(Value)?|Set(Value)?\b>>。更进一步，既然所有的选择都有相同的结尾，我们可以把正则表达式优化为<<\b(Get|Set)(Value)?\b>>。

11. 组与向后引用

把正则表达式的一部分放在圆括号内，你可以将它们形成组。然后你可以对整个组使用一些正则操作，例如重复操作符。

要注意的是，只有圆括号“()”才能用于形成组。“[]”用于定义字符集。“{}”用于定义重复操作。

当用“()”定义了一个正则表达式组后，正则引擎则会把被匹配的组按照顺序编号，存入缓存。当对被匹配的组进行向后引用的时候，可以用“\数字”的方式进行引用。<<\1>>引用第一个匹配的后向引用组，<<\2>>引用第二个组，以此类推，<<\n>>引用第n个组。而<<\0>>则引用整个被匹配的正则表达式本身。我们看一个例子。

假设你想匹配一个HTML标签的开始标签和结束标签，以及标签中间的文本。比如<B>This is a test</B>，我们要匹配<B>和</B>以及中间的文字。我们可以用如下正则表达式：“<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>”

首先，“<”将会匹配“<B>”的第一个字符“<”。然后[A-Z]匹配B，[A-Z0-9]*将会匹配0到多次字母数字，后面紧接着0到多个非“>”的字符。最后正则表达式的“>”将会匹配“<B>”的“>”。接下来正则引擎将对结束标签之前的字符进行惰性匹配，直到遇到一个“</”符号。然后正则表达式中的“\1”表示对前面匹配的组“([A-Z][A-Z0-9]*)”进行引用，在本例中，被引用的是标签名“B”。所以需要被匹配的结尾标签为“</B>”

你可以对相同的后向引用组进行多次引用，<<([a-c])x\1x\1>>将匹配“axaxa”、“bxbxb”以及“cxcxc”。如果用数字形式引用的组没有有效的匹配，则引用到的内容简单的为空。

一个后向引用不能用于它自身。<<([abc]\1)>>是错误的。因此你不能将<<\0>>用于一个正则表达式匹配本身，它只能用于替换操作中。

后向引用不能用于字符集内部。<<(a)[\1b]>>中的<<\1>>并不表示后向引用。在字符集内部，<<\1>>可以被解释为八进制形式的转码。

向后引用会降低引擎的速度，因为它需要存储匹配的组。如果你不需要向后引用，你可以告诉引擎对某个组不存储。例如：<<Get(?:Value)>>。其中“(”后面紧跟的“?:”会告诉引擎对于组(Value)，不存储匹配的值以供后向引用。

- 重复操作与后向引用

当对组使用重复操作符时，缓存里后向引用内容会被不断刷新，只保留最后匹配的内容。例如：<<([abc]+)=\1>>将匹配“cab=cab”，但是<<([abc])+=\1>>却不会。因为([abc])第一次匹配“c”时，“\1”代表“c”；然后([abc])会继续匹配“a”和“b”。最后“\1”代表“b”，所以它会匹配“cab=b”。

应用：检查重复单词--当编辑文字时，很容易就会输入重复单词，例如“the the”。使用<<\b(\w+)\s+\1\b>>可以检测到这些重复单词。要删除第二个单词，只要简单的利用替换功能替换掉“\1”就可以了。

- 组的命名和引用

在PHP，Python中，可以用<<(?P<name>group)>>来对组进行命名。在本例中，词法?P<name>就是对组(group)进行了命名。其中name是你对组的起的名字。你可以用(?P=name)进行引用。

.NET的命名组

.NET framework也支持命名组。不幸的是，微软的程序员们决定发明他们自己的语法，而不是沿用Perl、Python的规则。目前为止，还没有任何其他的正则表达式实现支持微软发明的语法。

下面是.NET中的例子：

(?<first>group)(?’second’group)

正如你所看到的，.NET提供两种词法来创建命名组：一是用尖括号“<>”，或者用单引号“’ ’ ”。尖括号在字符串中使用更方便，单引号在ASP代码中更有用，因为ASP代码中“<>”被用作HTML标签。

要引用一个命名组，使用\k<name>或\k’name’.

当进行搜索替换时，你可以用“${name}”来引用一个命名组。

12. 正则表达式的匹配模式

本教程所讨论的正则表达式引擎都支持三种匹配模式：

<<i>>使正则表达式对大小写不敏感，

<<s>>开启“单行模式”，即点号“.”匹配新行符

<<m>>开启“多行模式”，即“^”和“$”匹配新行符的前面和后面的位置。

- 在正则表达式内部打开或关闭模式

如果你在正则表达式内部插入修饰符(?ism)，则该修饰符只对其右边的正则表达式起作用。(?-i)是关闭大小写不敏感。你可以很快的进行测试。<<(?i)te(?-i)st>>应该匹配TEst，但是不能匹配teST或TEST.

13. 原子组与防止回溯

在一些特殊情况下，因为回溯会使得引擎的效率极其低下。

让我们看一个例子：要匹配这样的字串，字串中的每个字段间用逗号做分隔符，第12个字段由P开头。

我们容易想到这样的正则表达式<<^(.*?,){11}P>>。这个正则表达式在正常情况下工作的很好。但是在极端情况下，如果第12个字段不是由P开头，则会发生灾难性的回溯。如要搜索的字串为“1,2,3,4,5,6,7,8,9,10,11,12,13”。首先，正则表达式一直成功匹配直到第12个字符。这时，前面的正则表达式消耗的字串为“1,2,3,4,5,6,7,8,9,10,11,”，到了下一个字符，<<P>>并不匹配“12”。所以引擎进行回溯，这时正则表达式消耗的字串为“1,2,3,4,5,6,7,8,9,10,11”。继续下一次匹配过程，下一个正则符号为点号<<.>>，可以匹配下一个逗号“,”。然而<<，>>并不匹配字符“12”中的“1”。匹配失败，继续回溯。大家可以想象，这样的回溯组合是个非常大的数量。因此可能会造成引擎崩溃。

用于阻止这样巨大的回溯有几种方案：

一种简单的方案是尽可能的使匹配精确。用取反字符集代替点号。例如我们用如下正则表达式<<^([^,\r\n]*,){11}P>>，这样可以使失败回溯的次数下降到11次。

另一种方案是使用原子组。

原子组的目的是使正则引擎失败的更快一点。因此可以有效的阻止海量回溯。原子组的语法是<<(?>正则表达式)>>。位于(?>)之间的所有正则表达式都会被认为是一个单一的正则符号。一旦匹配失败，引擎将会回溯到原子组前面的正则表达式部分。前面的例子用原子组可以表达成<<^(?>(.*?,){11})P>>。一旦第十二个字段匹配失败，引擎回溯到原子组前面的<<^>>。

14. 向前查看与向后查看

Perl 5 引入了两个强大的正则语法：“向前查看”和“向后查看”。他们也被称作“零长度断言”。他们和锚定一样都是零长度的（所谓零长度即指该正则表达式不消耗被匹配的字符串）。不同之处在于“前后查看”会实际匹配字符，只是他们会抛弃匹配只返回匹配结果：匹配或不匹配。这就是为什么他们被称作“断言”。他们并不实际消耗字符串中的字符，而只是断言一个匹配是否可能。

几乎本文讨论的所有正则表达式的实现都支持“向前向后查看”。唯一的一个例外是Javascript只支持向前查看。

- 肯定和否定式的向前查看

如我们前面提过的一个例子：要查找一个q，后面没有紧跟一个u。也就是说，要么q后面没有字符，要么后面的字符不是u。采用否定式向前查看后的一个解决方案为<<q(?!u)>>。否定式向前查看的语法是<<(?!查看的内容)>>。

肯定式向前查看和否定式向前查看很类似：<<(?=查看的内容)>>。

如果在“查看的内容”部分有组，也会产生一个向后引用。但是向前查看本身并不会产生向后引用，也不会被计入向后引用的编号中。这是因为向前查看本身是会被抛弃掉的，只保留匹配与否的判断结果。如果你想保留匹配的结果作为向后引用，你可以用<<(?=(regex))>>来产生一个向后引用。

- 肯定和否定式的先后查看

向后查看和向前查看有相同的效果，只是方向相反

否定式向后查看的语法是：<<(?<!查看内容)>>

肯定式向后查看的语法是：<<(?<=查看内容)>>

我们可以看到，和向前查看相比，多了一个表示方向的左尖括号。

例：<<(?<!a)b>>将会匹配一个没有“a”作前导字符的“b”。

值得注意的是：向前查看从当前字符串位置开始对“查看”正则表达式进行匹配；向后查看则从当前字符串位置开始先后回溯一个字符，然后再开始对“查看”正则表达式进行匹配。

- 深入正则表达式引擎内部

让我们看一个简单例子。

把正则表达式<<q(?!u)>>应用到字符串“Iraq”。正则表达式的第一个符号是<<q>>。正如我们知道的，引擎在匹配<<q>>以前会扫过整个字符串。当第四个字符“q”被匹配后，“q”后面是空字符(void)。而下一个正则符号是向前查看。引擎注意到已经进入了一个向前查看正则表达式部分。下一个正则符号是<<u>>，和空字符不匹配，从而导致向前查看里的正则表达式匹配失败。因为是一个否定式的向前查看，意味着整个向前查看结果是成功的。于是匹配结果“q”被返回了。

我们在把相同的正则表达式应用到“quit”。<<q>>匹配了“q”。下一个正则符号是向前查看部分的<<u>>，它匹配了字符串中的第二个字符“i”。引擎继续走到下个字符“i”。然而引擎这时注意到向前查看部分已经处理完了，并且向前查看已经成功。于是引擎抛弃被匹配的字符串部分，这将导致引擎回退到字符“u”。

因为向前查看是否定式的，意味着查看部分的成功匹配导致了整个向前查看的失败，因此引擎不得不进行回溯。最后因为再没有其他的“q”和<<q>>匹配，所以整个匹配失败了。

为了确保你能清楚地理解向前查看的实现，让我们把<<q(?=u)i>>应用到“quit”。<<q>>首先匹配“q”。然后向前查看成功匹配“u”，匹配的部分被抛弃，只返回可以匹配的判断结果。引擎从字符“i”回退到“u”。由于向前查看成功了，引擎继续处理下一个正则符号<<i>>。结果发现<<i>>和“u”不匹配。因此匹配失败了。由于后面没有其他的“q”，整个正则表达式的匹配失败了。

- 更进一步理解正则表达式引擎内部机制

让我们把<<(?<=a)b>>应用到“thingamabob”。引擎开始处理向后查看部分的正则符号和字符串中的第一个字符。在这个例子中，向后查看告诉正则表达式引擎回退一个字符，然后查看是否有一个“a”被匹配。因为在“t”前面没有字符，所以引擎不能回退。因此向后查看失败了。引擎继续走到下一个字符“h”。再一次，引擎暂时回退一个字符并检查是否有个“a”被匹配。结果发现了一个“t”。向后查看又失败了。

向后查看继续失败，直到正则表达式到达了字符串中的“m”，于是肯定式的向后查看被匹配了。因为它是零长度的，字符串的当前位置仍然是“m”。下一个正则符号是<<b>>，和“m”匹配失败。下一个字符是字符串中的第二个“a”。引擎向后暂时回退一个字符，并且发现<<a>>不匹配“m”。

在下一个字符是字符串中的第一个“b”。引擎暂时性的向后退一个字符发现向后查看被满足了，同时<<b>>匹配了“b”。因此整个正则表达式被匹配了。作为结果，正则表达式返回字符串中的第一个“b”。

- 向前向后查看的应用

我们来看这样一个例子：查找一个具有6位字符的，含有“cat”的单词。

首先，我们可以不用向前向后查看来解决问题，例如：

```
<< cat\w{3}|\wcat\w{2}|\w{2}cat\w|\w{3}cat>>
```

足够简单吧！但是当需求变成查找一个具有6-12位字符，含有“cat”，“dog”或“mouse”的单词时，这种方法就变得有些笨拙了。

我们来看看使用向前查看的方案。在这个例子中，我们有两个基本需求要满足：一是我们需要一个6位的字符，二是单词含有“cat”。

满足第一个需求的正则表达式为<<\b\w{6}\b>>。满足第二个需求的正则表达式为<<\b\w*cat\w*\b>>。

把两者结合起来，我们可以得到如下的正则表达式：

```
    <<(?=\b\w{6}\b)\b\w*cat\w*\b>>
```

具体的匹配过程留给读者。但是要注意的一点是，向前查看是不消耗字符的，因此当判断单词满足具有6个字符的条件后，引擎会从开始判断前的位置继续对后面的正则表达式进行匹配。

最后作些优化，可以得到下面的正则表达式：

```
<<\b(?=\w{6}\b)\w{0,3}cat\w*>>
```

15. 正则表达式中的条件测试

条件测试的语法为<<(?ifthen|else)>>。“if”部分可以是向前向后查看表达式。如果用向前查看，则语法变为：<<(?(?=regex)then|else)>>，其中else部分是可选的。

如果if部分为true，则正则引擎会试图匹配then部分，否则引擎会试图匹配else部分。

需要记住的是，向前先后查看并不实际消耗任何字符，因此后面的then与else部分的匹配时从if测试前的部分开始进行尝试。

16. 为正则表达式添加注释

在正则表达式中添加注释的语法是：<<(?#comment)>>

例：为用于匹配有效日期的正则表达式添加注释：

```
(?#year)(19|20)\d\d[- /.](?#month)(0[1-9]|1[012])[- /.](?#day)(0[1-9]|[12][0-9]|3[01])
```

正则表达式30分钟入门教程 正则表达式到底是什么？

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

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

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

入门

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

假设你在一篇英文小说里查找hi，你可以使用正则正则表达式hi。

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

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

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

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

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

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

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

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

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

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

元字符

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

下面来试试更多的例子：

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

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

\b\w{6}\b 匹配刚好6个字母/数字的单词。

表1.常用的元字符代码说明

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

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

这里的{5,12}和前面介绍过的{2}是类似的，只不过{2}匹配只能不多不少重复2次，{5,12}则是重复的次数不能少于5次，不能多于12次，否则都不匹配。

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

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

字符转义

如果你想查找元字符本身的话，比如你查找.,或者*,就出现了问题：你没法指定它们，因为它们会被解释成其它

的意思。这时你就必须使用\来取消这些字符的特殊意义。因此，你应该使用\.和*。当然，要查找\本身，你也得用\\.
例如：www\.unibetter\.com匹配www.unibetter.com，c:\\Windows匹配c:\Windows。

重复

你已经看过了前面的*,+,{2},{5,12}这几个匹配重复的方式了。下面是正则表达式中所有的限定符(指定数量的代码，例如*,{5,12}等)：

表2.常用的限定符代码/语法说明

*重复零次或更多次
+重复一次或更多次
?重复零次或一次
{n}重复n次
{n,}重复n次或更多次
{n,m}重复n到m次

下面是一些使用重复的例子：
Windows\d+匹配Windows后面跟1个或更多数字
13\d{9}匹配13后面跟9个数字(中国的手机号)
^\w+匹配一行的第一个单词(或整个字符串的第一个单词，具体匹配哪个意思得看选项设置)

字符类

要想查找数字，字母或数字，空白是很简单的，因为已经有了对应这些字符集合的元字符，但是如果你想匹配没有预定义元字符的字符集合(比如元音字母a,e,i,o,u),应该怎么办？
很简单，你只需要在中括号里列出它们就行了，像[aeiou]就匹配任何一个英文元音字母，[.?!]匹配标点符号(.或?或!)(英文语句通常只以这三个标点结束)。
我们也可以轻松地指定一个字符范围，像[0-9]代表的含意与\d就是完全一致的：一位数字，同理[a-z0-9A-Z_]也完全等同于\w（如果只考虑英文的话）。
下面是一个更复杂的表达式：\(?0\d{2}[) -]?\d{8}。
这个表达式可以匹配几种格式的电话号码，像(010)88886666，或022-22334455，或02912345678等。我们对它进行一些分析吧：首先是一个转义字符\(,它能出现0次或1次(?),然后是一个0，后面跟着2个数字(\d{2})，然后是)或-或空格中的一个，它出现1次或不出现(?)，最后是8个数字(\d{8})。不幸的是，它也能匹配010)12345678或(022-87654321这样的“不正确”的格式。要解决这个问题，请在本教程的下面查找答案。

反义

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

表3.常用的反义代码代码/语法说明

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

例子：\S+匹配不包含空白符的字符串。
<a[^>]+>匹配用尖括号括起来的以a开头的字符串。

替换

好了，现在终于到了解决3位或4位区号问题的时间了。正则表达式里的替换指的是有几种规则，如果满足其中任意一种规则都应该当成匹配，具体方法是用|把不同的规则分隔开。听不明白？没关系，看例子：
0\d{2}-\d{8}|0\d{3}-\d{7}这个表达式能匹配两种以连字号分隔的电话号码：一种是三位区号，8位本地号(如010-12345678)，一种是4位区号，7位本地号(0376-2233445)。
\(0\d{2}\)[-]?\d{8}|0\d{2}[-]?\d{8}这个表达式匹配3位区号的电话号码，其中区号可以用小括号括起来，也可以不用，区号与本地号间可以用连字号或空格间隔，也可以没有间隔。你可以试试用替换|把这个表达式扩展成也支持4位区号的。
\d{5}-\d{4}|\d{5}这个表达式用于匹配美国的邮政编码。美国邮编的规则是5位数字，或者用连字号间隔的9位数字。之所以要给出这个例子是因为它能说明一个问题：使用替换时，顺序是很重要的。如果你把它改成\d{5}|\d{5}-\d{4}的话，那么就只会匹配5位的邮编(以及9位邮编的前5位)。原因是匹配替换时，将会从左到右地测试每个分枝条件，如果满足了某个分枝的话，就不会去管其它的替换条件了。
Windows98|Windows2000|WindosXP这个例子是为了告诉你替换不仅仅能用于两种规则，也能用于更多种规则。

分组

我们已经提到了怎么重复单个字符（直接在字符后面加上限定符就行了）；但如果想要重复多个字符又该怎么办？你可以用小括号来指定子表达式(也叫做分组)，然后你就可以指定这个子表达式的重复次数了，你也可以对子表达式进行其它一些操作(后面会有介绍)。
(\d{1,3}\.){3}\d{1,3}是一个简单的IP地址匹配表达式。要理解这个表达式，请按下列顺序分析它：\d{1,3}匹配1到3位的数字，(\d{1,3}\.){3}匹配三位数字加上一个英文句号(这个整体也就是这个分组)重复3次，最后再加上一个一到三位的数字(\d{1,3})。
不幸的是，它也将匹配256.300.888.999这
种不可能存在的IP地址(IP地址中每个数字都不能大于255。题外话，好像反恐24小时第三季的编剧不知道这一点，汗...)。如果能使用算术比较的
话，或许能简单地解决这个问题，但是正则表达式中并不提供关于数学的任何功能，所以只能使用冗长的分组，

选择，字符类来描述一个正确的IP地址：((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)。

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

后向引用

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

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

\b(\w+)\b\s+\1\b可以用来匹配重复的单词，像go go, kitty

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

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

使用小括号的时候，还有很多特定用途的语法。下面列出了最常用的一些：

表4.分组语法捕获

(exp)匹配exp,并捕获文本到自动命名的组里

(?<name>exp)匹配exp,并捕获文本到名称为name的组里，也可以写成(?'name'exp)

(?:exp)匹配exp,不捕获匹配的文本，也不给此分组分配组号

位置指定

(?=exp)匹配exp前面的位置

(?<=exp)匹配exp后面的位置

(?!exp)匹配后面跟的不是exp的位置

(?<!exp)匹配前面不是exp的位置

注释

(?#comment)这种类型的组不对正则表达式的处理产生任何影响，用于提供注释让人阅读

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

零宽断言

接下来的四个用于查找在某些内容(但并不包括这些内容)之前或之后的东西，也就是说它们像\b,^,$那样用于指定一个位置，这个位置应该满足一定的条件(断言)，因此它们也被称为零宽断言。最好还是拿例子来说明吧：

(?=exp)也叫零宽度正预测先行断言，它断言自身出现的位置的后面能匹配表达式exp。比如\b\w+(?=ing\b)，匹配以ing结尾的单词的前面部分(除了ing以外的部分)，如查找I'm singing while you're dancing.时，它会匹配sing和danc。

(?<=exp)也叫零宽度正回顾后发断言，它断言自身出现的位置的前面能匹配表达式exp。比如(?<=\bre)\w+\b会匹配以re开头的单词的后半部分(除了re以外的部分)，例如在查找reading a book时，它匹配ading。

假如你想要给一个很长的数字中每三位间加一个逗号(当然是从右边加起了)，你可以这样查找需要在前面和里面添加逗号的部分：((?<=\d)\d{3})*\b，用它对1234567890进行查找时结果是234567890。

下面这个例子同时使用了这两种断言：(?<=\s)\d+(?=\s)匹配以空白符间隔的数字(再次强调，不包括这些空白符)。

负向零宽断言

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

\b\w*q[^u]\w*\b匹配包含后面不是字母u的字母q的单词。但是如果多做测试(或者你思维足够敏锐，直接就观察出来了)，你会发现，如果q出现在单词的结尾的话，像Iraq,Benq，这个表达式就会出错。这是因为[^u]总要匹配一个字符，所以如果q是单词的最后一个字符的话，后面的[^u]将会匹配q后面的单词分隔符(可能是空格，或者是句号或其它的什么)，后面的\w*\b将会匹配下一个单词，于是\b\w*q[^u]\w*\b就能匹配整个Iraq fighting。负向零宽断言能解决这样的问题，因为它只匹配一个位置，并不消费任何字符。现在，我们可以这样来解决这个问题：\b\w*q(?!u)\w*\b。

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

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

一个更复杂的例子：(?<=<(\w+)>).*(?=<\/\1>)匹配不包含属性的简单HTML标签内里的内容。(<?(\w+)>)指定了这样的前缀：被尖括号括起来的单词(比如可能是)，然后是.*(任意的字符串),最后是一个后缀(?=<\/\1>)。注意后缀里的\/，它用到了前面提过的字符转义；\1则是一个反向引用，引用的正是捕获的第一组，前面的(\w+)匹配的内容，这样如果前缀实际上是的话，后缀就是了。整个表达式匹配的是和之间的内容(再次提醒，不包括前缀和后缀本身)。

注释

小括号的另一种用途是能过语法(?#comment)来包含注释。例如：2[0-4]\d(?#200-249)|25[0-5](?#250-255)|[01]?\d\d?(?#0-199)。

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

例如，我们可以前面的一个表达式写成这样：

```
(?<=    # 断言要匹配的文本的前缀      <(\w+)> # 查找尖括号括起来的字母或数字(即HTML/XML标签)      )
```

```
    # 前缀结束        .*        # 匹配任意文本        (?=       # 断言要匹配的文本的后缀        <\/\1>  # 查找尖
括号括起来的内容：前面是一个"/"，后面是先前捕获的标签        )        # 后缀结束
```

深入浅出之正则表达式（一）

前言：

半年前我对正则表达式产生了兴趣，在网上查找过不少资料，看过不少的教程，最后在使用一个正则表达式工具RegexBuddy时发现他的教程写的非常好，可以说是我目前见过最好的正则表达式教程。于是一直想把他翻译过来。这个愿望直到这个五一长假才得以实现，结果就有了这篇文章。关于本文的名字，使用“深入浅出”似乎已经太俗。但是通读原文以后，觉得只有用“深入浅出”才能准确的表达出该教程给我的感受，所以也就不能免俗了。

本文是Jan Goyvaerts为RegexBuddy写的教程的译文，但是为了尊重原作者和译者的劳动，请注明出处！谢谢！

1. 什么是正则表达式

基本说来，正则表达式是一种用来描述一定数量文本的模式。Regex代表Regular Express。本文将用<<regex>>来表示一段具体的正则表达式。

一段文本就是最基本的模式，简单的匹配相同的文本。

2. 不同的正则表达式引擎

正则表达式引擎是一种可以处理正则表达式的软件。通常，引擎是更大的应用程序的一部分。在软件世界，不同的正则表达式并不互相兼容。本教程会集中讨论Perl 5 类型的引擎，因为这种引擎是应用最广泛的引擎。同时我们也会提到一些和其他引擎的区别。许多近代的引擎都很类似，但不完全一样。例如.NET正则库，JDK正则包。

3. 文字符号

最基本的正则表达式由单个文字符号组成。如<<a>>，它将匹配字符串中第一次出现的字符“a”。如对字符串“Jack is a boy”。“J”后的“a”将被匹配。而第二个“a”将不会被匹配。

正则表达式也可以匹配第二个“a”，这必须是你告诉正则表达式引擎从第一次匹配的地方开始搜索。在文本编辑器中，你可以使用“查找下一个”。在编程语言中，会有一个函数可以使你从前一次匹配的位置开始继续向后搜索。

类似的，<<cat>>会匹配“About cats and dogs”中的“cat”。这等于是告诉正则表达式引擎，找到一个<<c>>，紧跟一个<<a>>，再跟一个<<t>>。

要注意，正则表达式引擎缺省是大小写敏感的。除非你告诉引擎忽略大小写，否则<<cat>>不会匹配“Cat”。

• 特殊字符

对于文字字符，有11个字符被保留作特殊用途。他们是：

[ ] \ ^ $. | ? * + ( )

这些特殊字符也被称作元字符。

如果你想在正则表达式中将这些字符用作文本字符，你需要用反斜杠“\”对其进行换码 (escape)。例如你想匹配“1+1=2”，正确的表达式为<<1\+1=2>>.

需要注意的是，<<1+1=2>>也是有效的正则表达式。但它不会匹配“1+1=2”，而会匹配“123+111=234”中的“111=2”。因为“+”在这里表示特殊含义（重复1次到多次）。

在编程语言中，要注意，一些特殊的字符会先被编译器处理，然后再传递给正则引擎。因此正则表达式<<1\+2=2>>在C++中要写成“1\\+1=2”。为了匹配“C:\temp”，你要用正则表达式<<C:\\temp>>。而在C++中，正则表达式则变成了“C:\\\\temp”。

• 不可显示字符

可以使用特殊字符序列来代表某些不可显示字符：

<<\t>>代表Tab(0x09)

<<\r>>代表回车符(0x0D)

<<\n>>代表换行符(0x0A)

要注意的是Windows中文本文件使用“\r\n”来结束一行而Unix使用“\n”。

4. 正则表达式引擎的内部工作机制

知道正则表达式引擎是如何工作的有助于你很快理解为何某个正则表达式不像你期望的那样工作。

有两种类型的引擎：文本导向(text-directed)的引擎和正则导向(regex-directed)的引擎。Jeffrey Friedl把他们称作DFA和NFA引擎。本文谈到的是正则导向的引擎。这是因为一些非常有用的特性，如“惰性”量词(lazy quantifiers)和反向引用(backreferences)，只能在正则导向的引擎中实现。所以毫不意外这种引擎是目前最流行的引擎。

你可以轻易分辨出所使用的引擎是文本导向还是正则导向。如果反向引用或“惰性”量词被实现，则可以肯定你使用的引擎是正则导向的。你可以作如下测试：将正则表达式<<regex|regex not>>应用到字符串“regex not”。如果匹配的结果是regex，则引擎是正则导向的。如果结果是regex not，则是文本导向的。因为正则导向的引擎是“猴急”的，它会很急切的进行表功，报告它找到的第一个匹配 。

- 正则导向的引擎总是返回最左边的匹配

这是需要你理解的很重要的一点：即使以后有可能发现一个“更好”的匹配，正则导向的引擎也总是返回最左边的匹配。

当把<<cat>>应用到“He captured a catfish for his cat”，引擎先比较<<c>>和“H”，结果失败了。于是引擎再比较<<c>>和“e”，也失败了。直到第四个字符，<<c>>匹配了“c”。<<a>>匹配了第五个字符。到第六个字符<<t>>没能匹配“p”，也失败了。引擎再继续从第五个字符重新检查匹配性。直到第十五个字符开始，<<cat>>匹配上了“catfish”中的“cat”，正则表达式引擎急切的返回第一个匹配的结果，而不会再继续查找是否有其他更好的匹配。

5. 字符集

字符集是由一对方括号“[]”括起来的字符集合。使用字符集，你可以告诉正则表达式引擎仅仅匹配多个字符中的一个。如果你想匹配一个“a”或一个“e”，使用<<[ae]>>。你可以使用<<gr[ae]y>>匹配gray或grey。这在你不确定你要搜索的字符是采用美国英语还是英国英语时特别有用。相反，<<gr[ae]y>>将不会匹配graay或graey。字符集中的字符顺序并没有什么关系，结果都是相同的。

你可以使用连字符“-”定义一个字符范围作为字符集。<<[0-9]>>匹配0到9之间的单个数字。你可以使用不止一个范围。<<[0-9a-fA-F] >>匹配单个的十六进制数字，并且大小写不敏感。你也可以结合范围定义与单个字符定义。<<[0-9a-fxA-FX]>>匹配一个十六进制数字或字母X。再次强调一下，字符和范围定义的先后顺序对结果没有影响。

- 字符集的一些应用

查找一个可能有拼写错误的单词，比如<<sep[ae]r[ae]te>> 或 <<li[cs]en[cs]e>>。

查找程序语言的标识符，<<A-Za-z_][A-Za-z_0-9]*>>。(*表示重复0或多次)

查找C风格的十六进制数<<0[xX][A-Fa-f0-9]+>>。(+表示重复一次或多次)

- 取反字符集

在左方括号“[”后面紧跟一个尖括号“^”，将会对字符集取反。结果是字符集将匹配任何不在方括号中的字符。不像“.”，取反字符集是可以匹配回车换行符的。

需要记住的很重要的一点是，取反字符集必须要匹配一个字符。<<q[^u]>>并不意味着：匹配一个q，后面没有u跟着。它意味着：匹配一个q，后面跟着一个不是u的字符。所以它不会匹配“Iraq”中的q，而会匹配“Iraq is a country”中的q和一个空格符。事实上，空格符是匹配中的一部分，因为它是一个“不是u的字符”。

如果你只想匹配一个q，条件是q后面有一个不是u的字符，我们可以用后面将讲到的向前查看来解决。

- 字符集中的元字符

需要注意的是，在字符集中只有4个 字符具有特殊含义。它们是：“] \ ^ -”。“]”代表字符集定义的结束；“\”代表转义；“^”代表取反；“-”代表范围定义。其他常见的元字符在字符集定义内部都是正常字符，不需要转义。例如，要搜索星号*或加号+，你可以用<<[+*]>>。当然，如果你对那些通常的元字符进行转义，你的正则表达式一样会工作得很好，但是这会降低可读性。

在字符集定义中为了将反斜杠“\”作为一个文字字符而非特殊含义的字符，你需要用另一个反斜杠对它进行转义。

<<[\\x]>>将会匹配一个反斜杠和一个X。“]^-”都可以用反斜杠进行转义，或者将他们放在一个不可能使用到他们特殊含义的位置。我们推荐后者，因为这样可以增加可读性。比如对于字符“^”，将它放在除了左括号“[”后面的位置，使用的都是文字字符含义而非取反含义。如<<[x^]>>会匹配一个x或^。<<[]x]>>会匹配一个“]”或“x”。<<[-x]>>或<<[x-]>>都会匹配一个“-”或“x”。

- 字符集的简写

因为一些字符集非常常用，所以有一些简写方式。

<<\d>>代表<<[0-9]>>;

<<\w>>代表单词字符。这个是随正则表达式实现的不同而有些差异。绝大多数的正则表达式实现的单词字符集都包含了<<A-Za-z0-9_]>>。

<<\s>>代表“白字符”。这个也是和不同的实现有关的。在绝大多数的实现中，都包含了空格符和Tab符，以及回车换行符<<\r\n>>。

字符集的缩写形式可以用在方括号之内或之外。<<\s\d>>匹配一个白字符后面紧跟一个数字。<<[\s\d]>>匹配单个白字符或数字。<<[\da-fA-F]>>将匹配一个十六进制数字。

取反字符集的简写

<<[\S]>> = <<[^\s]>>

<<[\W]>> = <<[^\w]>>

<<[\D]>> = <<[^\d]>>

- 字符集的重复

如果你用“?*+”操作符来重复一个字符集，你将会重复整个字符集。而不仅是它匹配的那个字符。正则表达式<<[0-9]+>>会匹配837以及222。

如果你仅仅想重复被匹配的那个字符，可以用向后引用达到目的。我们以后将讲到向后引用。

6. 使用?*或+ 进行重复

?：告诉引擎匹配前导字符0次或一次。事实上是表示前导字符是可选的。

+：告诉引擎匹配前导字符1次或多次

*：告诉引擎匹配前导字符0次或多次

<[A-Za-z][A-Za-z0-9]*>匹配没有属性的HTML标签，“<”以及“>”是文字符号。第一个字符集匹配一个字母，第二个字符集匹配一个字母或数字。

我们似乎也可以用<[A-Za-z0-9]+>。但是它会匹配<1>。但是这个正则表达式在你知道你要搜索的字符串不包含类似的无效标签时还是足够有效的。

- 限制性重复

许多现代的正则表达式实现，都允许你定义对一个字符重复多少次。词法是：{min,max}。min和max都是非负整数。如果逗号有而max被忽略了，则max没有限制。如果逗号和max都被忽略了，则重复min次。

因此{0,}和*一样，{1，}和+ 的作用一样。

你可以用<<\b[1-9][0-9]{3}\b>>匹配1000~9999之间的数字(“\b”表示单词边界)。<<\b[1-9][0-9]{2,4}\b>>匹配一个在100~99999之间的数字。

- 注意贪婪性

假设你想用一个正则表达式匹配一个HTML标签。你知道输入将会是一个有效的HTML文件，因此正则表达式不需要排除那些无效的标签。所以如果是在两个尖括号之间的内容，就应该是一个HTML标签。

许多正则表达式的新手会首先想到用正则表达式<< <.+> >>，他们会很惊讶的发现，对于测试字符串，“This is a <EM>first</EM> test”，你可能期望会返回<EM>，然后继续进行匹配的时候，返回</EM>。

但事实是不会。正则表达式将会匹配“<EM>first</EM>”。很显然这不是我们想要的结果。原因在于“+”是贪婪的。也就是说，“+”会导致正则表达式引擎试图尽可能的重复前导字符。只有当这种重复会引起整个正则表达式匹配失败的情况下，引擎会进行回溯。也就是说，它会放弃最后一次的“重复”，然后处理正则表达式余下的部分。

和“+”类似，“?*”的重复也是贪婪的。

- 深入正则表达式引擎内部

让我们来看看正则引擎如何匹配前面的例子。第一个记号是“<”，这是一个文字符号。第二个符号是“.”，匹配了字符“E”，然后“+”一直可以匹配其余的字符，直到一行的结束。然后到了换行符，匹配失败(“.”不匹配换行符)。于是引擎开始对下一个正则表达式符号进行匹配。也即试图匹配“>”。到目前为止，“<.+”已经匹配了“<EM>first</EM> test”。引擎会试图将“>”与换行符进行匹配，结果失败了。于是引擎进行回溯。结果是现在“<.+”匹配“<EM>first</EM> tes”。于是引擎将“>”与“t”进行匹配。显然还是会失败。这个过程继续，直到“<.+”匹配“<EM>first</EM”，“>”与“>”匹配。于是引擎找到了一个匹配“<EM>first</EM>”。记住，正则导向的引擎是“急切的”，所以它会急着报告它找到的第一个匹配。而不是继续回溯，即使可能会有更好的匹配，例如“<EM>”。所以我们可以看到，由于“+”的贪婪性，使得正则表达式引擎返回了一个最左边的最长的匹配。

- 用懒惰性取代贪婪性

一个用于修正以上问题的可能方案是用“+”的惰性代替贪婪性。你可以在“+”后面紧跟一个问号“?”来达到这一点。“*”，“{}”和“?”表示的重复也可以用这个方案。因此在上面的例子中我们可以使用“<.+?>”。让我们再来看看正则表达式引擎的处理过程。

再一次，正则表达式记号“<”会匹配字符串的第一个“<”。下一个正则记号是“.”。这次是一个懒惰的“+”来重复上一个字符。这告诉正则引擎，尽可能少的重复上一个字符。因此引擎匹配“.”和字符“E”，然后用“>”匹配“M”，结果失败了。引擎会进行回溯，和上一个例子不同，因为是惰性重复，所以引擎是扩展惰性重复而不是减少，于是“<.+”现在被扩展为“<EM”。引擎继续匹配下一个记号“>”。这次得到了一个成功匹配。引擎于是报告“<EM>”是一个成功的匹配。整个过程大致如此。

- 惰性扩展的一个替代方案

我们还有一个更好的替代方案。可以用一个贪婪重复与一个取反字符集：“<[^>]+>”。之所以说这是一个更好的方案在于使用惰性重复时，引擎会在找到一个成功匹配前对每一个字符进行回溯。而使用取反字符集则不需要进行回溯。

最后要记住的是，本教程仅仅谈到的是正则导向的引擎。文本导向的引擎是不回溯的。但是同时他们也不支持惰性重复操作。

7. 使用“.”匹配几乎任意字符

在正则表达式中，“.”是最常用的符号之一。不幸的是，它也是最容易被误用的符号之一。

“.”匹配一个单个的字符而不用关心被匹配的字符是什么。唯一的例外是新行符。在本教程中谈到的引擎，缺省情况下都是不匹配新行符的。因此在缺省情况下，“.”等于是字符集[^\n\r](Window)或[^\n](Unix)的简写。

这个例外是因为历史的原因。因为早期使用正则表达式的工具是基于行的。它们都是一行一行的读入一个文件，将正则表达式分别应用到每一行上去。在这些工具中，字符串是不包含新行符的。因此“.”也就从不匹配新行符。

现代的工具和语言能够将正则表达式应用到很大的字符串甚至整个文件上去。本教程讨论的所有正则表达式实现都提供一个选项，可以使“.”匹配所有的字符，包括新行符。在RegexBuddy, EditPad Pro或PowerGREP等工具中，你可以简单的选中“点号匹配新行符”。在Perl中，“.”可以匹配新行符的模式被称作“单行模式”。很不幸，这是一个很容易混淆的名词。因为还有所谓“多行模式”。多行模式只影响行首行尾的锚定(anchor)，而单行模式只影响“.”。

其他语言和正则表达式库也采用了Perl的术语定义。当在.NET Framework中使用正则表达式类时，你可以用类似下面的语句来激活单行模式：Regex.Match(“string”,”regex”,RegexOptions.SingleLine)

- 保守的使用点号“.”

点号可以说是最强大的元字符。它允许你偷懒：用一个点号，就能匹配几乎所有的字符。但是问题在于，它也常常会匹配不该匹配的字符。

我会以一个简单的例子来说明。让我们看看如何匹配一个具有“mm/dd/yy”格式的日期，但是我们想允许用户来选择分隔符。很快能想到的一个方案是<<\d\d.\d\d.\d\d>>。看上去它能匹配日期“02/12/03”。问题在于02512703也会被认为是一个有效的日期。

<<\d\d[-/.]\d\d[-/.]\d\d>>看上去是一个好一点的解决方案。记住点号在一个字符集里不是元字符。这个方案远不够完善，它会匹配“99/99/99”。而<<[0-1]\d[-/.][0-3]\d[-/.]\d\d>>又更进一步。尽管他也会匹配“19/39/99”。你想要你的正则表达式达到如何完美的程度取决于你想达到什么样的目的。如果你想校验用户输入，则需要尽可能的完美。如果你只是想分析一个已知的源，并且我们知道没有错误的数据，用一个比较好的正则表达式来匹配你想要搜寻的字符就已经足够。

8. 字符串开始和结束的锚定

锚定和一般的正则表达式符号不同，它不匹配任何字符。相反，他们匹配的是字符之前或之后的位置。“^”匹配一行字符串第一个字符前的位置。<<^a>>将会匹配字符串“abc”中的a。<<^b>>将不会匹配“abc”中的任何字符。

类似的，$匹配字符串中最后一个字符的后面的位置。所以<<c$>>匹配“abc”中的c。

- 锚定的应用

在编程语言中校验用户输入时，使用锚定是非常重要的。如果你想校验用户的输入为整数，用<<^\d+$>>。

用户输入中，常常会有多余的前导空格或结束空格。你可以用<<^\s*>>和<<\s*$>>来匹配前导空格或结束空格。

- 使用“^”和“$”作为行的开始和结束锚定

如果你有一个包含了多行的字符串。例如：“first line\n\rsecond line”(其中\n\r表示一个新行符)。常常需要对每行分别处理而不是整个字符串。因此，几乎所有的正则表达式引擎都提供一个选项，可以扩展这两种锚定的含义。“^”可以匹配字串的开始位置(在f之前)，以及每一个新行符的后面位置(在\n\r和s之间)。类似的，$会匹配字串的结束位置(最后一个e之后)，以及每个新行符的前面(在e与\n\r之间)。

在.NET中，当你使用如下代码时，将会定义锚定匹配每一个新行符的前面和后面位置：Regex.Match("string", "regex", RegexOptions.Multiline)

应用：string str = Regex.Replace(Original, "^", "> ", RegexOptions.Multiline)--将会在每行的行首插入“> ”。

- 绝对锚定

<<\A>>只匹配整个字符串的开始位置，<<\Z>>只匹配整个字符串的结束位置。即使你使用了“多行模式”，<<\A>>和<<\Z>>也从不匹配新行符。

即使\Z和$只匹配字符串的结束位置，仍然有一个例外的情况。如果字符串以新行符结束，则\Z和$将会匹配新行符前面的位置，而不是整个字符串的最后面。这个“改进”是由Perl引进的，然后被许多的正则表达式实现所遵循，包括Java，.NET等。如果应用<<^[a-z]+$>>到“joe\n”，则匹配结果是“joe”而不是“joe\n”。

正则表达式
bluesean
正则表达式（一）

一、简介

正则表达式这个名词，相信很多人都听说过，这个名词最早起源于1956 年，一位叫 Stephen Kleene 的美国数学家在 McCulloch 和 Pitts 早期工作的基础上，发表了一篇标题为“神经网事件的表示法”的论文，引入了正则表达式的概念。正则表达式就是用来描述他称为“正则集的代数”的表达式，因此采用“正则表达式”这个术语。

随后，发现可以将这一工作应用于使用Ken Thompson 的计算搜索算法的一些早期研究，Ken Thompson是Unix 的主要发明人。正则表达式的第一个实用应用程序就是 Unix 中的qed 编辑器。

Q: 正则表达式，能够为我们做什么呢？

A: 基于文本的编辑器和搜索工具中的一个重要部分。正则表达式可以让用户通过使用一系列的特殊字符构建匹配模式，然后把匹配模式与数据文件、程序输入以及WEB页面的表单输入等目标对象进行比较，根据比较对象中是否包含匹配模式，执行相应的程序。

下面我们就一步一步的结合它的语法，来介绍正则表达式的使用。

二、初次接触正则表达式

我们先来了解正则表达式的一些基本概念。正则表达式作为一种表示语言，其定义了自己的一套描述方式，来描述各种各样的字符类。下面摘取msdn中的一段定义。（
ms-help://MS.VSCC/MS.MSDNVS.2052/cpgenref/html/cpconcharacterclasses.htm）

字符转义表
字符类
含义

.
与除 \n 以外的任何字符匹配。如果通过 Singleline 选项（请参阅正则表达式选项）进行了修改，则句点字符与任何字符匹配。

[aeiou]
与指定字符集中包含的任何单个字符匹配。

[^aeiou]
与不在指定字符集中的任何单个字符匹配。

[0-9a-fA-F]
使用连字号 (-) 允许指定连续字符范围。

\p{name}
与 name 指定的命名字符类中的任何字符匹配。支持的名称为 Unicode 组和块范围。例如 Ll£?Nd£?Z£?IsGreek£?IsBoxDrawing。

\P{name}
与在 {name} 中指定的组和块范围中未包含的文本匹配。

\w
与任何单词字符匹配。等效于 Unicode 字符类别 [\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]。如果通过 ECMAScript 选项指定了符合 ECMAScript 的行为，则 \w 等同于 [a-zA-Z_0-9]。

\W
与任何非单词字符匹配。等效于 Unicode 类别 [^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}]。如果通过 ECMAScript 选项指定了符合 ECMAScript 的行为，则 \W 等同于 [^a-zA-Z_0-9]。

\s
与任何空白字符匹配。等效于 Unicode 字符类别 [\f\n\r\t\v\x85\p{Z}]。如果通过 ECMAScript 选项指定了符合 ECMAScript 的行为，则 \s 等同于 [\f\n\r\t\v]。

\S
与任何非空白字符匹配。等效于 Unicode 字符类别 [^\f\n\r\t\v\x85\p{Z}]。如果通过 ECMAScript 选项指定了符合 ECMAScript 的行为，则 \S 等同于 [^ \f\n\r\t\v]。

\d
与任何十进制数字匹配。与 Unicode 的 \p{Nd} 和非 Unicode 的 [0-9] 以及 ECMAScript 行为一样。

\D
与任何非数字匹配。与 Unicode 的 \P{Nd} 和非 Unicode 的 [^0-9] 以及 ECMAScript 行为一样。

上表列举了，正则表达式中最最基本的语法定义，了解这些，我们已经可以定义一些简单的规则了，例如：

1. 匹配所有的字符

当然是什么都不用写(@_@)

2. 匹配所有的英文字符

a) \w

b) [a-zA-Z_0-9]

3. 匹配十进制数字

a) \d

b) [0-9]

看上面的例子，是不是觉得很简单呢，不过，到目前为止，这样写出来的规则，还有一个很大的缺陷，就是没有声明匹配字符的个数？

Q: 我希望要匹配的字符为5个英文字母

A: ？？？

光了解上面的知识是，无法解决这个的L。那正则表达式中是如何解决这个问题的呢，我们来看下面这个表：

（ms-help://MS.VSCC/MS.MSDNVS.2052/cpgenref/html/cpconquantifiers.htm）

限定符表
限定符
说明

*
指定零个或更多个匹配；例如 \w* 或 (abc)*。与 {0,} 相同。

+
指定一个或多个匹配；例如 \w+ 或 (abc)+。与 {1,} 相同。

?
指定零个或一个匹配；例如 \w? 或 (abc)?。与 {0,1} 相同。

{n}
指定恰好 n 个匹配；例如 (pizza){2}。

{n,}
指定至少 n 个匹配；例如 (abc){2,}。

{n,m}
指定至少 n 个但不多于 m 个匹配。

*?
指定尽可能少地使用重复的第一个匹配 (lazy *)。

+?
指定尽可能少地使用重复但至少使用一次 (lazy +)。

??
指定使用零次重复（如有可能）或一次重复 (lazy ?)。

{n}?
等效于 {n} (lazy {n})。

{n,}?
指定尽可能少地使用重复，但至少使用 n 次 (lazy {n,})。

{n,m}?
指定介于 n 次和 m 次之间、尽可能少地使用重复 (lazy {n,m})。

上表中列出了，正则表达式的限定方式，配合这些字符的使用，我们就可以很方便的编写更为强劲的正则表达式了。

例如：

1. 匹配零个或多个所有的字符

*

2. 匹配一个或多个所有字符

+

3. 匹配零个或多个所有的英文字符

\w*

4. 匹配一个或多个所有的英文字符

[a-zA-Z0-9]+

5. 匹配3个十进制数字

\d{3}

6. 匹配最少3个十进制数字

\d{3,}

7. 匹配3个到6个十进制数字

\d{3,6}

现在我们可以解答上面问题了：

Q: 我希望要匹配的字符为5个英文字母

A: \w{5}

很高兴，我们已解决了上面的问题，不过，新的问题总是在不断的出现。我如何限制匹配字符出现在哪里呢？

Q: 我希望匹配以doc开头的字符串

A: ???

为了解决这个问题，我们先来看看这个表：

(ms-help://MS.VSCC/MS.MSDNVS.2052/cpgenref/html/cpconatomiczero-widthassertions.htm)

原子零宽度断言
断言
说明

^
指定匹配必须出现在字符串的开头或行的开头。有关更多信息，请参阅正则表达式选项中的 Multiline 选项。

$
指定匹配必须出现在以下位置：字符串结尾、字符串结尾的 \n 之前或行的结尾。有关更多信息，请参阅正则表达式选项中的 Multiline 选项。

\A
指定匹配必须出现在字符串的开头（忽略 Multiline 选项）。

\Z
指定匹配必须出现在字符串的结尾或字符串结尾的 \n 之前（忽略 Multiline 选项）。

\z
指定匹配必须出现在字符串的结尾（忽略 Multiline 选项）。

\G
指定匹配必须出现在当前搜索开始的位置（此位置通常是上一次搜索结束位置之后的第一个字符）。例如，请考虑一个由分离的字符组组成的串联字符串，其中每一组的长度都为 n 个字符。在每个字符组中搜索匹配时，如果正则表达式在 0、n、2n、3n 等字符位置找到匹配，则该正则表达式成功。仅当匹配出现在定位组边界上时才会成功。

\b
指定匹配必须出现在 \w（字母数字）和 \W（非字母数字）字符之间的边界上。匹配必须出现在单词边界上，即出现在由空格分隔的单词中第一个或最后一个字符上。

\B
指定匹配不得出现在 \b 边界上。

相信大家都注意到了，在这个表中第一个断言字符就是我们需要的@_@.

例如，^ 指定当前位置在行或字符串的开头。因此，正则表达式 ^FTP 只会返回那些在行的开头出现的字符串“FTP”的匹配项。

看来上面碰到的问题，又可以解决了，让我们一起来解决上面的问题：

Q: 我希望匹配以doc开头的字符串

A: ^doc

以上我们初步了解了什么是正则表达式，已经了解其最基本的语法，当作热身@_@，接下来，才正式进入主题，我们会从第三篇开始深入探讨正则表达式的使用。
在前一篇文章中，介绍了一些初步的正则表达式的基本概念，相信很多人对正则表达式的基本知识有所了解，接下来，我们结合一些实际的编程示例来掩饰说明正则表达式的作用。

首先，我们先看几个实际的例子：

1. 验证输入字符是否全部为英文字符

```
javascript:

var ex = "^\\w+$";

var re = new RegExp(ex,"i");

return re.test(str);

VBScript

Dim regEx,flag,ex

ex = "^\w+$"

Set regEx = New RegExp

regEx.IgnoreCase = True

regEx.Global = True

regEx.Pattern = ex

flag = regEx.Test( str )

C#

System.String ex = @"^\w+$";

System.Text.RegularExpressions.Regex reg = new Regex( ex ); bool flag = reg.IsMatch( str );
```

2. 验证邮件格式

```
C#

System.String ex = @"^\w+@\w+\.\w+$";

System.Text.RegularExpressions.Regex reg = new Regex( ex );

bool flag = reg.IsMatch( str );
```

3. 更改日期的格式（用 dd-mm-yy 的日期形式代替 mm/dd/yy 的日期形式）

```
C#

String MDYToDMY(String input)

{

return Regex.Replace(input,

"\\b(?<month>\\d{1,2})/(?<day>\\d{1,2})/(?<year>\\d{2,4})\\b",

"${day}-${month}-${year}");

}
```

4. 从 URL 提取协议和端口号

```
C#

String Extension(String url)

{

Regex r = new Regex(@"^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/",

RegexOptions.Compiled);

return r.Match(url).Result("${proto}${port}");

}
```

这里的例子可能是我们在网页开发中，通常会碰到的一些正则表达式，尤其在第一个例子中，给出了使用javascript,vbScript,C#等不同语言的实现方式，大家不难看出，对于不同的语言来说，正则表达式没有区别，只是正则表达式的实现类不同而已。而如何发挥正则表达式的公用，也要看实现类的支持。

（摘自msdn: Microsoft .NET 框架 SDK 提供大量的正则表达式工具，使您能够高效地创建、比较和修改字符串，以及迅速地分析大量文本和数据以搜索、移除和替换文本模式。
ms-help://MS.VSCC/MS.MSDNVS.2052/cpgenref/html/cpconregularexpressionslanguageelements.htm）

下面我们逐个来分析这些例子：

1-2，这两个例子很简单，只是简单的验证字符串是否符合正则表达式规定的格式，其中使用的语法，在第一篇文章中都已经介绍过了，这里做一下简单的描述。

第1个例子的表达式： ^\w+$

^ -- 表示限定匹配开始于字符串的开始

\w – 表示匹配英文字符

+ -- 表示匹配字符出现1次或多次

$ -- 表示匹配字符到字符串结尾处结束

验证形如asgasdfs的字符串

第2个例子的表达式： ^\w+@\w+.\w+$

^ -- 表示限定匹配开始于字符串的开始

\w – 表示匹配英文字符

+ -- 表示匹配字符出现1次或多次

@ -- 匹配普通字符@

\. – 匹配普通字符.(注意.为特殊字符,因此要加上\转译)

$ -- 表示匹配字符到字符串结尾处结束

验证形如dragontt@sina.com的邮件格式

第3 个例子中，使用了替换，因此，我们还是先来看看正则表达式中替换的定义：

（ms-help://MS.VSCC/MS.MSDNVS.2052/cpgenref/html/cpconsubstitutions.htm）

替换
字符
含义

$123
替换由组号 123（十进制）匹配的最后一个子字符串。

${name}
替换由 (?<name>) 组匹配的最后一个子字符串。

$$
替换单个“$”字符。

$&
替换完全匹配本身的一个副本。

$`
替换匹配前的输入字符串的所有文本。

$'
替换匹配后的输入字符串的所有文本。

$+
替换最后捕获的组。

$_
替换整个输入字符串。

分组构造
（ms-help://MS.VSCC/MS.MSDNVS.2052/cpgenref/html/cpcongroupingconstructs.htm）

分组构造
定义

( )
捕获匹配的子字符串（或非捕获组；有关更多信息，请参阅正则表达式选项中的 ExplicitCapture 选项。）使用 () 的捕获根据左括号的顺序从 1 开始自动编号。捕获元素编号为零的第一个捕获是由整个正则表达式模式匹配的文本。

(?<name>)
将匹配的子字符串捕获到一个组名称或编号名称中。用于 name 的字符串不能包含任何标点符号，并且不能以数字开头。可以使用单引号替代尖括号，例如 (?'name')。

(?<name1-name2>)
平衡组定义。删除先前定义的 name2 组的定义并在 name1 组中存储先前定义的 name2 组和当前组之间的间隔。如果未定义 name2 组，则匹配将回溯。由于删除 name2 的最后一个定义会显示 name2 的先前定义，因此该构造允许将 name2 组的捕获堆栈用作计数器以跟踪嵌套构造（如括号）。在此构造中，name1 是可选的。可以使用单引号替代尖括号，例如 (?'name1-name2')。

(?:)
非捕获组。

(?imnsx-imnsx:)
应用或禁用子表达式中指定的选项。例如，(?i-s:) 将打开不区分大小写并禁用单行模式。有关更多信息，请参阅正则表达式选项。

(?=)
零宽度正预测先行断言。仅当子表达式在此位置的右侧匹配时才继续匹配。例如，\w+(?=\d) 与后跟数字的单词匹配，而不与该数字匹配。此构造不会回溯。

(?!)
零宽度负预测先行断言。仅当子表达式不在此位置的右侧匹配时才继续匹配。例如，\b(?!un)\w+\b 与不以 un 开头的单词匹配。

(?<=)
零宽度正回顾后发断言。仅当子表达式在此位置的左侧匹配时才继续匹配。例如，(?<=19)99 与跟在 19 后面的 99 的实例匹配。此构造不会回溯。

(?<!)
零宽度负回顾后发断言。仅当子表达式不在此位置的左侧匹配时才继续匹配。

(?>)
非回溯子表达式（也称为贪婪子表达式）。该子表达式仅完全匹配一次，然后就不会逐段参与回溯了。（也就是说，该子表达式仅与可由该子表达式单独匹配的字符串匹配。）

我们还是先简单的了解一下这两个概念：

分组构造：

最基本的构造方式就是(),在左右括号中括起来的部分，就是一个分组；

更进一步的分组就是形如：(?<name>)的分组方式，这种方式与第一种方式的不同点，就是对分组的部分进行了命名，这样就可以通过该组的命名来获取信息；

（还有形如(?=)等等的分组构造，我们这篇的例子中也没有使用到，下次我们在来介绍）

替换：

上面提到了两种基本的构造分组方式()以及(?<name>)，通过这两种分组方式，我们可以得到形如$1,${name}的匹配结果。

这样说，可能概念上还是有些模糊，我们还是结合上面的例子来说：

第三个例子的正则表达式为：\\b(?<month>\\d{1,2})/(?<day>\\d{1,2})/(?<year>\\d{2,4})\\b

（解释一下，为什么这里都是\\一起用：这里是C#的例子，在C#语言中\是转译字符，要想字符串中的\不转译，就需要使用\\或者在整个字符串的开始加上@标记，即上面等价与

@”\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4}\b”）

\b -- 是一种特殊情况。在正则表达式中，除了在 [] 字符类中表示退格符以外，\b 表示字边界（在 \w 和 \W 字符之间）。在替换模式中，\b 始终表示退格符

(?<month>\d{1,2}) – 构造一个名为month的分组，这个分组匹配一个长度为1-2的数字

/ -- 匹配普通的/字符

(?<day>\d{1,2}) --构造一个名为day的分组，这个分组匹配一个长度为1-2的数字

/ -- 匹配普通的/字符

(?<year>\d{2,4}\b”） --构造一个名为year的分组，这个分组匹配一个长度为2-4的数字

这里还不能够看出这些分组的作用，我们接着看这一句

${day}-${month}-${year}

${day} – 获得上面构造的名为day的分组匹配后的信息

- -- 普通的-字符

${month} --获得上面构造的名为month的分组匹配后的信息

- -- 普通的-字符

${year} --获得上面构造的名为year的分组匹配后的信息

举例来说：

将形如04/02/2003的日期使用例3种的方法替换

(?<month>\d{1,2}) 分组将匹配到04由${month}得到这个匹配值

(?<day>\d{1,2}) 分组将匹配到02由${day}得到这个匹配值

(?<year>\d{1,2}) 分组将匹配到2003由${year}得到这个匹配值

了解了这个例子后，我们在来看第4个例子就很简单了。

第4个例子的正则

^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/

^ -- 表示限定匹配开始于字符串的开始

(?<proto>\w+) – 构造一个名为proto的分组，匹配一个或多个字母

: -- 普通的:字符

// -- 匹配两个/字符

[^/] – 表示这里不允许是/字符

+? – 表示指定尽可能少地使用重复但至少使用一次匹配

(?<port>:\d+) - 构造一个名为port的分组，匹配形如:2134（冒号+一个或多个数字）

? - 表示匹配字符出现0次或1次

/ -- 匹配/字符

最后通过${proto}${port}来获取两个分组构造的匹配内容

（有关Regex对象的用法，参考

ms-help://MS.VSCC/MS.MSDNVS.2052/cpref/html/frlrfSystemTextRegularExpressionsRegexMembersTopic.htm）

好了，本次介绍的几个例子，也讲得差不多了，希望大家有所收获，下次，在就一些特殊的要求，进一步探讨正则表达式的实现。
前面的文章中，介绍了正则表达式的基本语法，以及一些简单的例子。但这些并不是我们会遇到的全部问题，有些时候我们不得不编写一些较为复杂的正则表达式来解决我们的实际问题。

这里，我先提几个问题，然后，我们逐个运用正则表达式的知识来解决。

1. 符合两种条件之一，都成立，例如：是纯数字或者纯字符

123（true）,hello(true),234.test23(false)

2. 要得到不以数字开头的字符组合

如：How2234do>you234do,希望得到How和you而不是do,do

3. 得到以数字开头的字符组合

上例中，得到do和do

4. 要得到不以数字结尾的字符组合

还是上面的情况，要得到的是Ho,do,yo,do

5. 得到以数字结尾的字符组合

同上例，得到Ho,do,yo,do

6. 不允许字符中ab同时出现

例：nihaoma(true),above(false),agoodboy(true)

下面我们开始着手解决这些问题：

第一个：符合两种条件之一，都成立

这种要求可能代表着一种普遍的要求，我们先来看看这个表

替换构造
替换构造
定义

|
与由|（垂直条）字符分隔的术语中的任何一个术语匹配；例如 cat|dog|tiger。使用最左侧的成功匹配。

(?(expression)yes|no)
如果表达式在此位置匹配，则与“yes”部分匹配；否则，与“no”部分匹配。“no”部分可省略。表达式可以是任何有效的表达式，但它将变为零宽度断言，因此该语法等效于 (?(?=expression)yes|no)。请注意，如果表达式是命名组的名称或捕获组编号，则替换构造将解释为捕获测试（在本表的下一行对此进行了描述）。若要避免在这些情况下产生混淆，则可以显式拼出内部 (?=expression)。

(?(name)yes|no)
如果命名捕获字符串有匹配，则与“yes”部分匹配；否则，与“no”部分匹配。“no”部分可省略。如果给定的名称不与此表达式中使用的捕获组的名称或编号对应，则替换构造将解释为表达式测试（在本表的上一行进行了描述）。

（ms-help://MS.VSCC/MS.MSDNVS.2052/cpgenref/html/cpconalternationconstructs.htm）

在这个表中，我们看到，正则中为了解决这一类问题，定义了|来表示或者的关系，就好像常见的或运算符一样，现在我们来看看如何利用|来解决我们的问题。

1. 先为可选择的表达式撰写表达式：

a) 纯数字 - [0-9]*

b) 纯字母 - [a-zA-Z]*

2. 将可选条件用|连接起来就是我们所需的

^[0-9]*$|^[a-zA-Z]*$

(这里我特别对两个条件加上了^和$限定符，这在验证字符串是否完全符合要求时，是十分必要的，如果不加这两个限定符，有兴趣的朋友可以自己试一下效果。

后面四个问题，其实是一类的，所以我们把它们放在一起处理。接下来我们来解决第二到第四个问题：

首先，我们回顾一下上次介绍的分组构造：

(?=)
零宽度正预测先行断言。仅当子表达式在此位置的右侧匹配时才继续匹配。例如，\w+(?=\d) 与后跟数字的单词匹配，而不与该数字匹配。此构造不会回溯。

(?!)
零宽度负预测先行断言。仅当子表达式不在此位置的右侧匹配时才继续匹配。例如，\b(?!un)\w+\b 与不以 un 开头的单词匹配。

(?<=)
零宽度正回顾后发断言。仅当子表达式在此位置的左侧匹配时才继续匹配。例如，(?<=19)99 与跟在 19 后面的 99 的实例匹配。此构造不会回溯。

(?<!)
零宽度负回顾后发断言。仅当子表达式不在此位置的左侧匹配时才继续匹配。

可以看到，这个表的这四种规则，正好可以解决我们的问题。

@_@先解决我们的问题再说：

第二例：要得到不以数字开头的字符组合

(?<!\d)[a-zA-Z]{2,}

(?<!\d) -- 限定字符的开头不为数字才匹配

[a-zA-Z]{2,} - 描述匹配2个以上的字母

(注：这是取巧的做法，因为，按照我们的逻辑How2234do>you234do中的两个do的o字母也是符合的，不过，这不是我们想要的，当然还有其他的解决办法，可以根据实际的情况来处理，这里是为了讲解这个方法@_@)

第三例：得到以数字开头的字符组合

(?<=\d)[a-zA-Z]+

(?<=\d) - 限定为数字开头的字符才匹配

[a-zA-Z]+ -- 描述匹配1个或多个字母

第四例：要得到不以数字结尾的字符组合

[a-zA-Z]+(?!\d)

[a-zA-Z]+ -- 描述匹配1个或多个字母

(?!\d) - 限定不为数字结尾的字母才匹配

第五例：得到以数字结尾的字符组合

[a-zA-Z]+(?=\d)

[a-zA-Z]+ -- 描述匹配1个或多个字母

(?=\d) - 限定为数字结尾的字母才匹配

第六例：不允许字符中ab同时出现

^(?!.*?ab).*$

(?!.*?ab) – 限定不允许出现ab相连的字符

.* -- 任意字符

介绍到这里，我们这一次提出的问题，也都解决了，例子虽然简单，不过，再复杂的东西也是建立在简单的基础上的。其实写正则表达式的关键就是要善于定制规则，用最简练正确的话来描述，然后用正则表达式的语法写出来，就可以了(@_@)这就要靠大家多积累经验了。

zt from CSDN