# 正则表达式技术分析总结

## （A Short Report about Regular Expression）

**密码学与系统安全研究开发小组GA**

潘金龙

杨剑锋

赵彦博

陈皓

陈宵

**版本：GB-1-VER1.0**

密碼學與系統安全實驗室

www.systemshell.cn

# 目　录

# 一、绪论

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。

# 二、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。

# 三、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。

## 四、 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 |

## 五、 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" |

## 六、正则表达式匹配规则

### 6.1 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。

**6.2 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]

**6.3 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

## 七、几种常见类型正则表达式的构造与分析

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

分析： 整数包括正整数, 零, 和负整数。

表达式应该匹配以下类型数据：…-2,-1,0,1,2…

我开始写的匹配整数的表达式如下：

^[-|+]?\d*$

很明显这个过于简单，当碰到以下类型的整数时将显示出它的拙劣：

-0000,001,-0001... ...

我们并不会这样输入一个整数，因此需要保证存在两位以上时高位不为零

我们调整表达式如下：

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

其中[1-9][0-9]*即可以保证当输入为 2 位数以上时，高位并不为 0。

**7.2 小数：**^(-|)[1-9]*([1-9][0-9]*|0)(\.[0-9][0-9]*[1-9])$

我开始写的匹配小数的正则表达式如下：

^[-|+]?\d*\.?\d*$

当然也是碰到了如上 1.1 中的问题，修改表达式：

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

小数点前面的即为整数位，直接使用 1.1 结果，其中最后一个[1-9]可以保证小数的末尾不为 0。

**7.3 有理数**：^(-|)(([1-9]*([1-9][0-9]*|0))|([1-9]*([1-9][0-9]*|0)(\.[0-9][0-9]*[1-9])))$
我开始写的表达式如下：
ˆ[-|+]?\d*\.?\d*$
显然考虑错误，将有理数等同于小数了。
所有的整数和有限循环小数构成了有理数，所以其正则表达式可以这样写：
ˆ(-|)(([1-9]*([1-9][0-9]*|0))|([1-9]*([1-9][0-9]*|0)(\.[0-9][0-9]*[1-9])))$

**7. 4 日期格式：**
**^(((00[0-9][0-9]|0[1-9][0-9][0-9]|[1-9][0-9][0-9][0-9])-((0[13578]|1[02])-(0[1-9]|[12][0-9]|3[01])|(0[469]|11)-(0[1-9]|[12][0-9]|30)|02-(0[1-9]|[1][0-9]|2[0-8])))|(([0-9]{2}(04|08|[13579][26]|[2468][048]))|((04|08|[13579][26]|[2468][048])00))-((0[13578]|1[02])-(0[1-9]|[12][0-9]|3[01])|(0[469]|11)-(0[1-9]|[12][0-9]|30)|(02-(0[1-9]|[12][0-9])))$**
开始我写的表达式如下：
^\d{4}\-\d{2}\-\d{2}$
显然这个表达式过于简单，在一年中的 1、3、5、7、8、10、12 月份有 31 天，4、6、9、11 月份有 30 天，而 2 月份在闰年的时候有 29 天，其余时间为 28 天。
我们可以这样判断：除 2 月外, 1 到 12 月中日期为 29, 30 天内的都为合法，1，3，5，7，8，10，12 月为 31 天的都为合法, 1 月到 12 月为 28 天的都为合法, 剩下的只要判断当年是否为闰年就行了，是闰年 2 月 29 日就合法，否则非法!
根据以上分析，我们用三个表达式来进行组合。

- 第一个判断 2 月除外的月份

除 2 月外 1 到 12 月都为 29 或 30 天的表达式为
(?:**0?[1,3-9]**|1[0-2])-(?:29|30) **黄色为一个整体**
除 2 月外 1，3，5，7，8，10，12 月都是 31 天, ((?:0?[13578]|1[02])-31)
把两条合并起来就是
(?:(?:0?[1,3-9]|1[0-2])-(?:29|30)|((?:0?[13578]|1[02])-31))

- 第二条是判断 1 到 12 月都是 1-28 天的表达式

(?:0?[1-9]|1[0-2])-(?:0?[1-9]|1\d|2[0-8])
其中，0?[1-9]是表示一个月中的 01-09 号，1/d 表示 10-19 号，2[0-8]表示 20-28 号。

- 第三条判断年份是否为闰年

我们知道闰年是指可以被 4 整除的年份，因此只要满足当年份的尾数两位可以被 4 整除即可，在 100 之内可以被 4 整除的数字包括：04,08,12,16,20,24,28,32,36,40,44,48,52,56,60,64,68,72,76,80,84,88,92,96;
而能被 100 与 400 整除的四位数则可以表示为
##00 的形式;
将 2 月份的日期 29 号加入即可得到闰年的 2 月份 日期正则表达式如下：
((?:(\d\d(?:0[48]|[2468][048]|[13579][26]))
|(?:0[48]00|[2468][048]00|[13579][26]00))
-0?2-29)
到此可以得到正确的匹配日期的正则表达式了：

^(?:([0-9]{4}-(?:(?:0?[1,3-9]|1[0-2])-(?:29|30)|((?:0?[13578]|1[02])-31)))
|([0-9]{4}-(?:0?[1-9]|1[0-2])-(?:0?[1-9]|1\d|2[0-8]))
|(((?:(\d\d(?:0[48]|[2468][048]|[13579][26]))|(?:0[48]00|[2468][048]00|[13579][26]00)
)-0?2-29)))$
这条表达式可以匹配的日期格式为 2004-02-29,2004-2-29,不匹配 2007-2-29,格式固定为####-##-##,只有输入日期正确了才会匹配。

## 7.5 日期格式：

### HTML 类型的匹配：

以<td>...</td>为例

我开始写的表达式如下：

^<td>(.|\n)*<\/td>$

手册上讲：. 可以匹配除 “\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。但使用^<td>[.|\n]*<\/td>$匹配并不成功。

扩展到一般的 HTML 类型可以这样写：

(?<=<(\w+)>).*(?=<\/\1>)\1 指先前捕获的标签

断言匹配文本前缀查找 HTML 标签 断言要匹配的文本的后缀

- ◆ 注释:正则表达式的零宽断言
- ● 零宽度正预测先行断言(?=exp) ，匹配 exp 前面的位置，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)
- ● 零宽度正回顾后发断言(?<=exp) ，匹配 exp 后面的位置，它断言自身出现的位置的前面能匹配表达式 exp。
- ● 零宽度负预测先行断言(?!exp)，匹配后面跟的不是 exp 的位置，断言此位置的后面不能匹配表达式 exp。
- ● 零宽度正回顾后发断言(?<!exp),匹配前面不是 exp 的位置，断言此位置的前面不能匹配表达式 exp 。

## 7.6 ip 地址匹配：

我开始写的表达式如下：

^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$

显然这个表达式比较简单，但不幸的是它将匹配类似 256.000.888.999 等不可能存在的 ip 地址。

我们知道 ip 地址中每个数字不能大于 255，将其分为三段来表示。分别是 0-199，200-249，250-255。

修改表达式如下：

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

## 7.7 网络地址匹配

网络地址包括 www，ftp，user 等

我写的表达式如下：

^(www|ftp|user){1}(\.|:){1}\w+\.\w+\.*$

考虑的不够全面，当网址中出现‘-’时无法匹配
修改表达式如下：
^(www|ftp|user){1}(\.|:){1}([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?
([\w-]+\.)中的‘-’用来匹配网址中输入的-

### 7.8 sql 关键词的匹配

我们知道所谓的 sql 注入攻击是指用户在输入网址的同时提交一段数据库查询的代码，收集程序及服务器的信息，获得想要得到的数据。

以 select * from table 最为常用，下面这个表达式用来匹配这一类型的数据库查询语句：

^.*\bselect\b.*\bfrom\b.*$

其他关键词例如 insert delete 等可以采用相同的方法进行匹配。

### 7.9 其它与 WEB 相关输入内容的匹配

匹配账号是否合法（限定条件为以字母开头，允许 5-16 字节，允许字母数字下划线）

^[a-zA-Z][a-zA-Z0-9_]{4,15}$

匹配国内电话号码：

\d{3}-\d{8}|\d{4}-\d{7} 匹配形式如 0511-4405222 或 021-87888822

匹配中国邮政编码：

[1-9]\d{5}(?!\d) 中国邮政编码为 6 位数字

匹配身份证：

\d{15}|\d{18} 中国的身份证为 15 位或 18 位

匹配 email 地址：

^\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*$

## 八、ASP,PHP,JSP 等脚本语言使用正则过滤和匹配

前言：如果我们问那些 UNIX 系统的爱好者他们最喜欢什么，答案除了稳定的系统和可以远程启动之外，十有八九的人会提到正则表达式；如果我们再问他们最头痛的是什么，可能除了复杂的进程控制和安装过程之外，还会是正则表达式。那么正则表达式到底是什么？如何才能真正的掌握正则表达式并正确的加以灵活运用？本文将就此展开介绍，希望能够对那些渴望了解和掌握正则表达式的读者有所助益。

简单的说，正则表达式是一种可以用于模式匹配和替换的强有力的工具。我们可以在几乎所有的基于 UNIX 系统的工具中找到正则表达式的身影，例如，vi 编辑器，Perl 或 PHP 脚本语言，以及 awk 或 sed shell 程序等。此外，象 JavaScript 这种客户端的脚本语言也提供了对正则表达式的支持。由此可见，正则表达式已经超出了某种语言或某个系统的局限，成为人们广为接受的概念和功能。

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。

举例来说，正则表达式的一个最为普遍的应用就是用于验证用户在线输入的邮件地址的格式是否正确。如果通过正则表达式验证用户邮件地址的格式正确，用户所填写的表单信息将会被正常处理；反之，如果用户输入的邮件地址与正则表达的模式不匹配，将会弹出提示信息，要求用户重新输入正确的邮件地址。由此可见正则表达式在 WEB 应用的逻辑判断中具有举足轻重的作用。

## 8.1 asp 中使用正则表达式过滤 html 标签

在 asp 中使用正则表达式可以提取纯文本的不包含 html 标签的字符，作为摘要显示出来。Html 标签特定都是用<>包含的，正则表达式只要匹配<>就可以了，函数可以过滤 html 标签的作用：

```
Function RemoveHTML(str)  //建立函数
Dim re                                  //建立变量
Set re=New RegExp           //建立正则表达式
re.Pattern=“<.*?>”             //对正则匹配模板赋值
re.IgnoreCase=True //设置是否区分字符大小写
re.Global=True              //设置全局可用性
str=re.Replace(str,"")     //执行搜索
Set re=Nothing
RemoveHTML=str
End Function
```

代码如下：

```
<%
Dim contenstr
contenstr="<a href='http://www.aspxhome.com' title="中国 asp 之家">中国 asp 之家<b>欢迎您</b>！欢迎出来看看，呵呵！ "
'当然这里的 contenstr 在实际使用时换成你的文章内容
'调用过滤函数移除 html 标签
 contenstr=RemoveHTML(contenstr)
contenstr=left(contenstr,100) '截取前 100 个字符
response.write("移除 html 标签后：" & contenstr)
%>
```

## 8.2 Asp 中正则表达式对象常用的方法

- Execute 方法
  描述：对指定的字符串执行正则表达式搜索。
  语法：object.Execute(string)
- Test 方法
  描述:对指定的字符串执行一个正则表达式搜索,并返回一个 Boolean 值指示是否找到匹配的模式。
  语法：object.Test(string)
- Replace 方法
  描述:替换在正则表达式查找中找到的文本。
  语法:object.Replace(string1, string2)

## 8.3 php 中使用正则检查 email 地址格式是否正确的一个 php 源程序：

PHP 有六个函数来处理正则表达式，它们都把一个正则表达式作为它们的第一个参数，列出如下：

ereg: 最常用的正则表达式函数，ereg 允许我们搜索跟一个正则表达式匹配的一个字符串.

ereg_replace: 允许我们搜索跟正则表达式匹配的一个字符串，并用新的字符串代替所有这个表达式出现的地方。

eregi: 和 ereg 几乎是一样效果，不过忽略大小写。

eregi_replace: 和 ereg_replace 有着一样的搜索-替换功能，不过忽略大小写.

split: 允许我们搜索和正则表达式匹配的字符串，并且以字符串集合的方式返回匹配结果.

spliti: split 函数忽略大小写的版本.

为什么使用正则表达式?

如果你不断地建立不同的函数来检查或者操作字符串的一部分，现在你可能要放弃所有的这些函数，取而代之的用正则表达式。如果你对下列的问题都答“是的”，那么你肯定要考虑使用正则表达式了：

你是否正在写一些定制的函数来检查表单数据（比如在电子信箱地址中的一个@，一个点）?

你是否写一些定制的函数，在一个字符串中循环每个字符，如果这个字符匹配了一个特定特征（比如它是大写的，或者它是一个空格），那么就替换它?

除了是令人不舒服的字符串检查和操作方法，如果没有有效率地写代码，上述的两条也会使你的程序慢下来。你是否更倾向于用下面的代码检查一个电子信箱地址呢:

```
<?php
function validateEmail($email)
{
$hasAtSymbol = strpos($email, "@");
$hasDot = strpos($email, ".");
if($hasAtSymbol && $hasDot)
return true;
else
return false;
}
echo validateEmail("mitchell@devarticles.com");
?>
```

... 或者使用下面的代码:

```
<?php
function validateEmail($email)
{
return ereg("^[a-zA-Z]+@[a-zA-Z]+\.[a-zA-Z]+$", $email);
}
echo validateEmail("mitchell@devarticles.com");
?>
```

可以肯定的是，第一个函数比较容易，而且看起来结构也不错。但是如果我们用

上面的下一个版本的 email 地址检查函数不是更容易吗？
上面展示的第二个函数只用了正则表达式，包括了对 ereg 函数的一个调用。Ereg 函数返回 true 或者 false,来声明它的字符串参数是否和正则表达式相匹配。
很多编程者避开正则表达式，只因为它们（在一些情况下）比其它的文本处理方法更慢。正则表达式可能慢的原因是因为它们涉及把字符串在内存中拷贝和粘贴，因为正则表达式的每一个新的部分都对应匹配一个字符串。但是，从我对正则表达式的经验来说，除非你在文本中几百个行运行一个复杂的正则表达式，否则性能上的缺陷都可以忽略不计，当把正则表达式作为输入数据检查工具时，也很少出现这种情况。

## 8.4 PHP 的 Perl 兼容正则表达式提供的多个函数

1、preg_match ：
函数格式：int preg_match(string pattern, string subject, array [matches]);
这个函数会在 string 中使用 pattern 表达式来匹配，如果给定了[regs]，就会将 string 记录到[regs][0]中，[regs][1]代表使用括号“（）”记录下来的第一个字符串，[regs][2]代表记录下来的第二个字符串，以此类推。preg 如果在 string 中找到了匹配的 pattern，就会返回“true”，否则返回“false”。
2、preg_replace ：
函数格式：mixed preg_replace(mixed pattern, mixed replacement, mixed subject);
这个函数会使用将 string 中符合表达式 pattern 的字符串全部替换为表达式 replacement。如果 replacement 中需要包含 pattern 的部分字符，则可以使用“()”来记录，在 replacement 中只是需要用“\\1”来读取。
3、preg_split ：
函数格式：array preg_split(string pattern, string subject, int [limit]);
这个函数和函数 split 一样，区别仅在与 split 可以使用简单正则表达式来分割匹配的字符串，而 preg_split 使用完全的 Perl 兼容正则表达式。第三个参数 limit 代表允许返回多少个符合条件的值。
4、preg_grep ：
函数格式：array preg_grep(string patern , array input);
这个函数和 preg_match 功能基本上，不过 preg_grep 可以将给定的数组 input 中的所有元素匹配，返回一个新的数组。

## 8.5 jsp 中使用正则表达式检测 email 地址有效性

**源程序如下：**

```
<script language=\'JScript\'>
function validateEmail(emailStr)
{
 var re=/^[\\w.-]+@([0-9a-z][\\w-]+\\.)+[a-z]{2,3}$/i;
  if(re.test(emailStr))
 {
  alert(\"有效 email 地址!\");
  return true;
 }
```

```
 else
 {
  alert(\"无效 email 地址!\");
  return false;
 }
}
</script>
```

- 在 JavaScript 中,正则表达式是由一个 RegExp 对象表示的. RegExp 对象常用方法如下：
  - rgExp.compile(pattern,[flags])
    将 rgExp 转化为内部格式，以加快匹配的执行，这对于大量模式一致的匹配更有效
  - rgExp.exec(str)
    按照 rgExp 的匹配模式对 str 字符串进行匹配查找,当 rgExp 对象中设定了全局搜索模式
  - rgExp.test(str)
    返回布尔值来反映被查找的目标字符串 str 中是否存在符合匹配的模式。该方法不改变 RegExp 的属性

## 九、正则表达式在其他语言中的应用

### 9.1 VC 中的正则表达式

正则表达式在 VC 中的应用主要工作是先配置 VC6.0，使它编译使时候能找到正则库。

把 vc6 下的所有 lib 和 dll 文件拷贝到 Visual Studio 安装目录下的 VC98\boostRex（boostRex 是我自己建的）

然后打开 vc6.0，选择“Tools->Options->Directories->Include files”，加入一行“D:\BOOST”

选择“Tools->Options->Directories->Library file”，加入一行“C:\PROGRAM FILES\MICROSOFT VISUAL STUDIO\VC98\BOOSTREX”（彩色部分是我的 Visual Studio 安装目录）

配置也 OK 了！

然后是编写程序测试

SDK 下的测试：

```
#include "stdafx.h"
#include <cstdlib>
#include <stdlib.h>
#include <boost/regex.hpp>
#include <string>
#include <iostream>
using namespace std;
using namespace boost;
regex expression("^select ([a-zA-Z]*) from ([a-zA-Z]*)");
int main(int argc, char* argv[])
{
std::string in;
```

```
cmatch what;
cout << "enter test string" << endl;
getline(cin,in);
if(regex_match(in.c_str(), what, expression))
{
for(int i=0;i<what.size();i++)
cout<<"str :"<<what.str()<<endl;
}
else
{
cout<<"Error Input"<<endl;
}
return 0;
}
```

输入: select name from table

输出: str:select name from table

str:name

str:table

MFC 下的测试（有几个地方要注意，下面有提示）：

新建一个对话框的 MFC 工程，

加入头文件

#include <boost/regex.hpp>

在按钮鼠标单击事件响应函数中加入

```
boost::regex expression("^select ([a-zA-Z]*) from ([a-zA-Z]*)");
CString in = "select gm from tab";
CString sRet;
boost::cmatch what;
if(boost::regex_match(LPCSTR(in), what, expression))//CString 转 string
{
   for(int i=0;i<what.size();i++){
      sRet = (what.str()).c_str();//string 转 CString
      MessageBox(sRet);
   }
}
else
{
MessageBox("Error Input");
}
```

输出的结果跟上面一样。

**9.2 VB 中的正则表达式**

VB 中正则表达式主要有三个对象：RegExp、MatchCollection、Match。

1. RegExp　这是 VB 使用正则表达式匹配模式的主要对象了。其提供的属

性用于设置那些用来比较的传递给 RegExp 实例的字符串的模式。 其提供的方法以确定字符串是否与正则表达式的特定模式相匹配。

属性：

Pattern：一个字符串，用来定义正则表达式。

IgnoreCase：一个布尔值属性，指示是否必须对一个字符串中的所有可能的匹配进行正则表达式测试。这是 MS 的解释，有点费解，实际使用中的实例是，如果 True，则忽略英文字母大小的匹配，False 对大小写进行匹配。

Global：设置一个布尔值或返回一个布尔值，该布尔值指示一个模式是必须匹配整个搜索字符串中的所有搜索项还是只匹配第一个搜索项。

MultiLine：这个 MS 没有介绍。查了一下资料，设置一个布尔值或返回一个布尔值，是否在串的多行中搜索。如果允许匹配多行文本，则 multiline 为 true，如果搜索必须在换行时停止，则为 false 。

方法：

Execute：返回一个 MatchCollection 对象，该对象包含每个成功匹配的 Match 对象。

Replace：MS 没有介绍，这是返回一个将匹配字符替换为指定字符的字符串。

Test：返回一个布尔值，该值指示正则表达式是否与字符串成功匹配。

2. MatchCollection 是集合对象，包含有关匹配字符串的信息，该对象包含每个成功匹配的 Match 对象。

属性

Count：匹配对象的总数。

Item：匹配对象的索引。

3. Match 是成功匹配的对象。

属性：

FirstIndex：匹配对象所匹配字符串的起始位置。

Length：匹配对象所匹配字符串的字符长度。

SubMatches：匹配对象所匹配结果的子项。

Value：匹配对象所匹配的值。

**1、RegExp 的 Test 方法：**

```
Function bTest(ByVal s As String, ByVal p As String) As Boolean
    Dim re As RegExp
    Set re = New RegExp
    re.IgnoreCase = False    '设置是否匹配大小写
    re.Pattern = p
    bTest = re.Test(s)
  End Function
  Private Sub Command1_Click()
    Dim s As String
    Dim p As String
    s = "我的邮箱: test@163.com 。欢迎致电！ "
    '测试字符串中是否包含数字：
    p = "\d+"
    MsgBox bTest(s, p)
```

```
    '测试字符串中是否全是由数字组成：
    p = "^\d+$"
    MsgBox bTest(s, p)
    '测试字符串中是否有大写字母：
    p = "[A-Z]+"
    MsgBox bTest(s, p)
  End Sub
```

**2、RegExp 的 Replace 方法：**

```
Function StrReplace(s As String, p As String, r As String) As String
    Dim re As RegExp
    Set re = New RegExp
    re.IgnoreCase = True
    re.Global = True
    re.Pattern = p
    StrReplace = re.Replace(s, r)
  End Function
  Private Sub Command2_Click()
    Dim s As String    '字符串
    Dim p As String    '正则表达式
    Dim r As String    '要替换的字符串
      '以下代码是替换邮箱地址
    s = "我的 E-mail: Test@163.com 。欢迎致电!"
    p = "w+@w+.w+"
    r = "E_Mail@sohu.net"
    s = StrReplace(s, p, r)
    Debug.Print s
    '结果：我的 E-mail: E_Mail@sohu.net 。欢迎致电!
  End Sub
```

**3、Match 的 SubMatches 属性：**

```
Private Sub Command3_Click()
    Dim re As RegExp
    Dim mh As Match
    Dim mhs As MatchCollection
    Dim inpStr As String
    inpStr = "我的 E-mail: lucky@163.com 。欢迎致电!"
    Set re = New RegExp
    re.Pattern = "(w+)@(w+).(w+)"       '同样是匹配地址，注意和上例的不同
    Set mhs = re.Execute(inpStr)
    Set mh = mhs(0)                     '只有一个匹配
    Debug.Print "电子邮件地址是： " & mh.Value            '这里是匹配的内容
    Debug.Print "用户名是：       " & mh.SubMatches(0)  '第一个括号中的内容
    Debug.Print "邮箱是：         " & mh.SubMatches(1)  '第二个括号中的内容
```

```
    Debug.Print "域名是：                " & mh.SubMatches(2) '第三个括号中的内容
End Sub
```

### 9.3 java 中的正则表达式

有许多源代码开放的正则表达式库可供 Java 程序员使用，而且它们中的许多支持 Perl 5 兼容的正则表达式语法。最常用的是 Jakarta-ORO 正则表达式库，它是最全面的正则表达式 API 之一，而且它与 Perl 5 正则表达式完全兼容。另外，它也是优化得最好的 API 之一。

**▲ PatternCompiler 对象**

首先，创建一个 Perl5Compiler 类的实例，并把它赋值给 PatternCompiler 接口对象。Perl5Compiler 是 PatternCompiler 接口的一个实现，允许你把正则表达式编译成用来匹配的 Pattern 对象。

```
PatternCompiler compiler=new Perl5Compiler();
```

**▲ Pattern 对象**

要把正则表达式编译成 Pattern 对象，调用 compiler 对象的 compile()方法，并在调用参数中指定正则表达式。例如，你可以按照下面这种方式编译正则表达式“t[aeio]n”：

```
Pattern pattern=null;
try {
  pattern=compiler.compile("t[aeio]n");
} catch (MalformedPatternException e) {
  e.printStackTrace();
}
```

默认情况下，编译器创建一个大小写敏感的模式（pattern）。因此，上面代码编译得到的模式只匹配“tin”、“tan”、 “ten”和“ton”，但不匹配“Tin”和“taN”。要创建一个大小写不敏感的模式，你应该在调用编译器的时候指定一个额外的参数：

```
pattern=compiler.compile("t[aeio]n",Perl5Compiler.CASE_INSENSITIVE_MASK);
```

创建好 Pattern 对象之后，你就可以通过 PatternMatcher 类用该 Pattern 对象进行模式匹配。

**▲ PatternMatcher 对象**

PatternMatcher 对象根据 Pattern 对象和字符串进行匹配检查。你要实例化一个 Perl5Matcher 类并把结果赋值给 PatternMatcher 接口。Perl5Matcher 类是 PatternMatcher 接口的一个实现，它根据 Perl 5 正则表达式语法进行模式匹配：

```
PatternMatcher matcher=new Perl5Matcher();
```

使用 PatternMatcher 对象，你可以用多个方法进行匹配操作，这些方法的第一

个参数都是需要根据正则表达式进行匹配的字符串：
· boolean matches(String input, Pattern pattern)：当输入字符串和正则表达式要精确匹配时使用。换句话说，正则表达式必须完整地描述输入字符串。
· boolean matchesPrefix(String input, Pattern pattern)：当正则表达式匹配输入字符串起始部分时使用。
· boolean contains(String input, Pattern pattern)：当正则表达式要匹配输入字符串的一部分时使用（即，它必须是一个子串）。
另外，在上面三个方法调用中，你还可以用 PatternMatcherInput 对象作为参数替代 String 对象；这时，你可以从字符串中最后一次匹配的位置开始继续进行匹配。当字符串可能有多个子串匹配给定的正则表达式时，用 PatternMatcherInput 对象作为参数就很有用了。用 PatternMatcherInput 对象作为参数替代 String 时，上述三个方法的语法如下：
• boolean matches(PatternMatcherInput input, Pattern pattern)
· boolean matchesPrefix(PatternMatcherInput input, Pattern pattern)
· boolean contains(PatternMatcherInput input, Pattern pattern)

## 9.4 DELPHI 中的正则表达式

在 Delphi 中使用正则表达式
方法一 使用微软ScriptControl控件
1. 编写一个脚本文件(test.vbs)，里面包含要使用的正则表达式函数

```
function GetUrlFile(Url)
    Set RegObject = New RegExp
    With RegObject
    .Pattern = "\w+\.\w+(?!.)"
    .IgnoreCase = True
    .Global = True
    End With
    Set matchs =    RegObject.Execute(Url)
    If matchs.Count > 0 Then
       For Each mach in matchs
          GetUrlFile=mach.value
       Next
    End If
    Set RegObject = nothing
end function
```

2. 下载最新版的"Microsoft(r) Windows(r) Script"
你可以在以下地址找到下载 http://computer.lqinfo.net.cn/down/soft.asp?id=302
3．安装Microsoft(r) Windows(r) Script
Visual Basic(r) Script Edition (VBScript.) Version 5.6,
JScript(r) Version 5.6, Windows Script Components,
Windows Script Host 5.6,
Windows Script Runtime Version 5.6.将被安装到你的系统中
4 .在Delphi中导入MsScript.ocx ,生成TScriptControl控件
5.使用以下代码调用TScriptControl

```
procedure TForm1.Button2Click(Sender: TObject);
var
a: OleVariant;
begin
memo2.Lines.LoadFromFile('test.vbs');
ScriptControl1.Language := 'Vbscript';
ScriptControl1.AddCode(string(memo2.Text));
a := VarArrayCreate([0, 0], varVariant);
a[0] := 'http://www.xolor.cn/page1.htm';
memo1.Lines.Add(CallFunction('GetUrlFile', a));
end;
function TForm1.CallFunction(const FunctionName: string;
const Params: oleVariant): OleVariant;
var
Sarray: PSafeArray;
begin
try
// 转化为安全数组
Sarray := PSafeArray(TVarData(Params).VArray);
// 调用函数
Result := ScriptControl1.Run(FunctionName, Sarray);
except
on E: Exception do
begin
end;
end;
end;
```

方法二 使用微软 RegExp

1. 下载并安装最新版的"Microsoft(r) Windows(r) Script"

2. RegExp 包含在 vbscript.dll 中所以我们必须先注册 regsvr32 vbscript.dll
注(安装了 Ie5 后默认已经包含该控件)

3.在 Delphi 中引入"Microsoft VBScript Regular Expressions"
主菜单->Project->Import type library->在列表中选择"Microsoft VBScript Regular Expressions"
生成 TRegExp 控件

4.使用以下代码调用 TRegExp 控件

```
procedure TForm1.Button1Click(Sender: TObject);
var
machs: IMatchCollection;
Matchs: Match;
submatch: ISubMatches;
i, j: integer;
begin
```

```
RegExp1.Global := true;
RegExp1.Pattern := '\w+\.\w+(?!.)';
RegExp1.IgnoreCase := true;
machs := RegExp1.Execute('http://www.xcolor.cn/dd/page1.htm') as
IMatchCollection;
for i := 0 to machs.Count - 1 do
begin
Matchs := machs.Item[i] as Match;
submatch := Matchs.SubMatches as ISubMatches;
memo1.Lines.Add(matchs.Value);
//for j:=0 to submatch.Count -1 do
// memo1.Lines.Add(submatch.Item[j])
end;
end;
```

**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)