**[正则表达式基础知识](http://www.cnblogs.com/fineboy/archive/2005/09/08/232410.html)**

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

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

下表是元字符及其在正则表达式上下文中的行为的一个完整列表：

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

下面看几个例子：  
"^The"：表示所有以"The"开始的字符串（"There"，"The cat"等）；   
"of despair$"：表示所以以"of despair"结尾的字符串；   
"^abc$"：表示开始和结尾都是"abc"的字符串——呵呵，只有"abc"自己了；   
"notice"：表示任何包含"notice"的字符串。   
  
'\*'，'+'和'?'这三个符号，表示一个或一序列字符重复出现的次数。它们分别表示“没有或   
更多”，“一次或更多”还有“没有或一次”。下面是几个例子：

"ab\*"：表示一个字符串有一个a后面跟着零个或若干个b。（"a", "ab", "abbb",……）；   
"ab+"：表示一个字符串有一个a后面跟着至少一个b或者更多；   
"ab?"：表示一个字符串有一个a后面跟着零个或者一个b；   
"a?b+$"：表示在字符串的末尾有零个或一个a跟着一个或几个b。

也可以使用范围，用大括号括起，用以表示重复次数的范围。

"ab{2}"：表示一个字符串有一个a跟着2个b（"abb"）；   
"ab{2,}"：表示一个字符串有一个a跟着至少2个b；   
"ab{3,5}"：表示一个字符串有一个a跟着3到5个b。

请注意，你必须指定范围的下限（如："{0,2}"而不是"{,2}"）。还有，你可能注意到了，'\*'，'+'和   
'?'相当于"{0,}"，"{1,}"和"{0,1}"。   
还有一个'¦'，表示“或”操作：

"hi¦hello"：表示一个字符串里有"hi"或者"hello"；   
"(b¦cd)ef"：表示"bef"或"cdef"；   
"(a¦b)\*c"：表示一串"a""b"混合的字符串后面跟一个"c"；

'.'可以替代任何字符：

"a.[0-9]"：表示一个字符串有一个"a"后面跟着一个任意字符和一个数字；   
"^.{3}$"：表示有任意三个字符的字符串（长度为3个字符）；

方括号表示某些字符允许在一个字符串中的某一特定位置出现：

"[ab]"：表示一个字符串有一个"a"或"b"（相当于"a¦b"）；   
"[a-d]"：表示一个字符串包含小写的'a'到'd'中的一个（相当于"a¦b¦c¦d"或者"[abcd]"）；   
"^[a-zA-Z]"：表示一个以字母开头的字符串；   
"[0-9]%"：表示一个百分号前有一位的数字；   
",[a-zA-Z0-9]$"：表示一个字符串以一个逗号后面跟着一个字母或数字结束。

你也可以在方括号里用'^'表示不希望出现的字符，'^'应在方括号里的第一位。（如："%[^a-zA-Z]%"表   
示两个百分号中不应该出现字母）。

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

请注意在方括号中，不需要转义字符

[**全面剖析C#正则表达式**](http://www.cnblogs.com/fineboy/archive/2005/09/09/233088.html)

        到目前为止，许多的编程语言和工具都包含对正则表达式的支持，当然.NET也不例外，.NET基础类库中包含有一个名称空间和一系列可以充分发挥规则表达式威力的类。   
        正则表达式的知识可能是不少编程人员最烦恼的事儿了。如果你还没有规则表达式方面的知识的话，建议从正则表达式的基础知识入手。前参见 [**正则表达式语法**](http://fineboy.cnblogs.com/archive/2005/09/08/232410.html)。   
  
        下面就来研究C#中的正则表达式，C#中的正则表达式包含在.NET基础雷库的一个名称空间下，这个名称空间就是System.Text.RegularExpressions。该名称空间包括8个类，1个枚举，1个委托。他们分别是：  
                     Capture: 包含一次匹配的结果；   
                     CaptureCollection: Capture的序列；   
                     Group: 一次组记录的结果，由Capture继承而来；   
                     GroupCollection：表示捕获组的集合  
                     Match: 一次表达式的匹配结果，由Group继承而来；   
                     MatchCollection: Match的一个序列；   
                     MatchEvaluator: 执行替换操作时使用的委托；   
                     Regex：编译后的表达式的实例。   
                     RegexCompilationInfo：提供编译器用于将正则表达式编译为独立程序集的信息  
                     RegexOptions 提供用于设置正则表达式的枚举值  
Regex类中还包含一些静态的方法：   
                    Escape: 对字符串中的regex中的转义符进行转义；   
                    IsMatch: 如果表达式在字符串中匹配，该方法返回一个布尔值；   
                    Match: 返回Match的实例；   
                    Matches: 返回一系列的Match的方法；   
                    Replace: 用替换字符串替换匹配的表达式；   
                    Split: 返回一系列由表达式决定的字符串；   
                    Unescape:不对字符串中的转义字符转义。  
  
下面介绍他们的用途：  
        先看一个简单的匹配例子，我们首先从使用Regex、Match类的简单表达式开始学习。　Match m = Regex.Match("abracadabra", "(a|b|r)+"); 我们现在有了一个可以用于测试的Match类的实例，例如：if (m.Success){}，如果想使用匹配的字符串，可以把它转换成一个字符串： 　　MesaageBox.Show("Match="+m.ToString()); 这个例子可以得到如下的输出: Match=abra。这就是匹配的字符串了。  
  
        Regex 类表示只读正则表达式类。它还包含各种静态方法（在下面的实例中将逐一介绍），允许在不显式创建其他类的实例的情况下使用其他正则表达式类。  
  
        以下代码示例创建了 Regex 类的实例并在初始化对象时定义一个简单的正则表达式。声明一个Regex对象变量：Regex objAlphaPatt;，接着创建Regex对象的一个实例，并定义其规则：objAlphaPatt=new Regex("[^a-zA-Z]");  
  
        IsMatch方法指示 Regex 构造函数中指定的正则表达式在输入字符串中是否找到匹配项。这是我们使用C#正则表达式时最常用的方法之一。下面的例子说明了IsMatch方法的使用：  
if( !objAlphaPatt.IsMatch("testisMatchMethod"))  
 lblMsg.Text = "匹配成功";  
else  
 lblMsg.Text = "匹配不成功";  
这段代码执行的结果是“匹配成功”  
if( ! objAlphaPatt.IsMatch("testisMatchMethod7654298"))  
 lblMsg.Text = "匹配成功";  
else  
 lblMsg.Text = "匹配不成功";  
这段代码执行的结果是“匹配不成功”  
  
         Escape方法表示把转义字符作为字符本身使用，而不再具有转义作用，最小的元字符集（\、\*、+、?、|、{、[、(、)、^、$、.、# 和空白）。Replace方法则是用指定的替换字符串替换由正则表达式定义的字符模式的所有匹配项。看下面的例子，还是使用上面定义的Regex对象：objAlphaPatt.Replace("this [test] \*\* replace and escape" ,Regex.Escape("()"));他的返回结果是：this\(\)\(\)test\(\)\(\)\(\)\(\)\(\)replace\(\)and\(\)escape，如果不是Escape的话，则返回结果是：this()()test()()()()()replace()and()escape，Unescape 反转由 Escape 执行的转换，但是，Escape 无法完全反转 Unescape。  
  
        Split方法是把由正则表达式匹配项定义的位置将输入字符串拆分为一个子字符串数组。例如：  
Regex r = new Regex("-"); // Split on hyphens.  
string[] s = r.Split("first-second-third");  
for(int i=0;i<s.Length;i++)  
{  
 Response.Write(s[i]+"<br>");  
}

执行的结果是：  
First  
Second  
Third

        看上去和String的Split方法一样，但string的Split方法在由正则表达式而不是一组字符确定的分隔符处拆分字符串。  
  
        Match方法是在输入字符串中搜索正则表达式的匹配项，并Regex 类的 Match 方法返回 Match 对象，Match 类表示正则表达式匹配操作的结果。下面的例子演示Match方法的使用，并利用Match对象的Group属性返回Group对象：  
  
string text = @"public string testMatchObj string s string  match ";  
string pat = @"(\w+)\s+(string)";  
// Compile the regular expression.  
Regex r = new Regex(pat, RegexOptions.IgnoreCase);  
// Match the regular expression pattern against a text string.  
Match m = r.Match(text);  
int matchCount = 0;  
while (m.Success)   
{  
 Response.Write("Match"+ (++matchCount) + "<br>");  
 for (int i = 1; i <= 2; i++)   
 {  
  Group g = m.Groups[i];  
  Response.Write("Group"+i+"='" + g + "'"  + "<br>");  
  CaptureCollection cc = g.Captures;  
  for (int j = 0; j < cc.Count; j++)   
  {  
   Capture c = cc[j];  
   Response.Write("Capture"+j+"='" + c + "', Position="+c.Index + "<br>");  
  }  
 }  
 m = m.NextMatch();  
}

该事例运行结果是：  
Match1  
Group1='public'  
Capture0='public', Position=0  
Group2='string'  
Capture0='string', Position=7  
Match2  
Group1='testMatchObj'  
Capture0='testMatchObj', Position=14  
Group2='string'  
Capture0='string', Position=27  
Match3  
Group1='s'  
Capture0='s', Position=34  
Group2='string'  
Capture0='string', Position=36  
  
        MatchCollection 类表示成功的非重叠匹配的只读的集合，MatchCollection 的实例是由 Regex.Matches 属性返回的，下面的实例说明了通过在输入字符串中找到所有与Regex中指定的匹配并填充 MatchCollection。  
  
MatchCollection mc;  
Regex r = new Regex("match");   
mc = r.Matches("matchcollectionregexmatchs");  
for (int i = 0; i < mc.Count; i++)   
{  
 Response.Write( mc[i].Value + " POS:" + mc[i].Index.ToString() + "<br>");  
}  
该实例运行的结果是：  
match POS:0  
match POS:20