**PCRE正则表达式的定义：**

用于描述字符排列和匹配模式的一种语法规则。它主要用于字符串的模式分割、匹配、查找及替换操作。

[**PHP**](http://lib.csdn.net/base/php)**中的正则函数：**

PHP中有两套正则函数，两者功能差不多，分别为：

一套是由PCRE（Perl Compatible Regular Expression）库提供的。使用“preg\_”为前缀命名的函数；

一套由POSIX（Portable Operating System Interface of Unix ）扩展提供的。使用以“ereg\_”为前缀命名的函数；（POSIX的正则函数库，自PCRE 5.3以后，就不在推荐使用，从PCRE6以后，就将被移除）

由于POSIX正则即将推出历史舞台，并且PCRE和perl的形式差不多，更利于我们在perl和PCRE之间切换，所以这里重点介绍PCRE正则的使用。

**PCRE正则表达式**

PCRE全称为Perl Compatible Regular Expression，意思是Perl兼容正则表达式。

在PCRE中，通常将模式表达式（即正则表达式）包含在两个反斜线“/”之间，如“/apple/”。

正则中重要的几个概念有：元字符、转义、模式单元（重复）、反义、引用和断言，这些概念都可以在文章[1]中轻松的理解和掌握。

**常用的元字符(Meta-character)：**

元字符     说明

\A       匹配字符串串首的原子

\Z       匹配字符串串尾的原子

\b       匹配单词的边界     /\bis/   匹配头为is的字符串   /is\b/   匹配尾为is的字符串   /\bis\b/ 定界

\B       匹配除单词边界之外的任意字符   /\Bis/   匹配单词“This”中的“is”

\d       匹配一个数字；等价于[0-9]

\D       匹配除数字以外任何一个字符；等价于[^0-9]

\w       匹配一个英文字母、数字或下划线；等价于[0-9a-zA-Z\_]

\W       匹配除英文字母、数字和下划线以外任何一个字符；等价于[^0-9a-zA-Z\_]

\s      匹配一个空白字符；等价于[\f\t\v]

\S      匹配除空白字符以外任何一个字符；等价于[^\f\t\v]

\f      匹配一个换页符等价于 \x0c 或 \cL

匹配一个换行符；等价于 \x0a 或 \cJ

匹配一个回车符等价于\x0d 或 \cM

\t     匹配一个制表符；等价于 \x09\或\cl

\v     匹配一个垂直制表符；等价于\x0b或\ck

\oNN   匹配一个八进制数字

\xNN   匹配一个十六进制数字

\cC    匹配一个控制字符

**模式修正符（Pattern Modifiers）：**

模式修正符在忽略大小写、匹配多行中使用特别多，掌握了这一个修正符，往往能解决我们遇到的很多问题。

i     －可同时匹配大小写字母

M     －将字符串视为多行

S     －将字符串视为单行，换行符做普通字符看待，使“.”匹配任何字符

X     －模式中的空白忽略不计

U     －匹配到最近的字符串

e     －将替换的字符串作为表达使用

格式：/apple/i匹配“apple”或“Apple”等，忽略大小写。     /i

**PCRE的模式单元：**

//1 提取第一位的属性

/^\d{2} ([\W])\d{2}\\1\d{4}$匹配“12-31-2006”、“09/27/1996”、“86 01 4321”等字符串。但上述正则表达式不匹配“12/34-5678”的格式。这是因为模式“[\W]”的结果“/”已经被存储。下个位置“\1”引用时，其匹配模式也是字符“/”。

当不需要存储匹配结果时使用非存储模式单元“（？：）”

例如/(?:a|b|c)(D|E|F)\\1g/ 将匹配“aEEg”。在一些正则表达式中，使用非存储模式单元是必要的。否则，需要改变其后引用的顺序。上例还可以写成/（a|b|c） (C|E|F)\2g/。

**PCRE正则表达式函数：**

|  |
| --- |
| preg\_match() 和preg\_match\_all() preg\_quote() preg\_split() preg\_grep() preg\_replace() |

函数的具体使用，我们可以通过PCRE手册来找到，下面分享一些平时积累的正则表达式：

**匹配action属性**

|  |
| --- |
| $str = ''; $match = ''; preg\_match\_all('/\s+action=\"(?!http:)(.\*?)\"\s/', $str, $match); print\_r($match); |

**在正则中使用回调函数**

|  |
| --- |
| function callback\_replace() { $url = 'http://esfang.house.sina.com.cn'; $str = ''; $str = preg\_replace ( '/(?<=\saction=\")(?!http:)(.\*?)(?=\"\s)/e', 'search(\$url, \\1)', $str );  echo $str; }  function search($url, $match){ return $url . '/' . $match; } |

**带断言的正则匹配**

|  |
| --- |
| $match = ''; $str = 'xxxxxx.com.cn bold font paragraph text  '; preg\_match\_all ( '/(?<=<(\w{1})>).\*(?=<\/\1>)/', $str, $match ); echo "匹配没有属性的HTML标签中的内容："; print\_r ( $match ); |

**替换HTML源码中的地址**

|  |
| --- |
| $form\_html = preg\_replace ( '/(?<=\saction=\"|\ssrc=\"|\shref=\")(?!http:|[**JavaScript**](http://lib.csdn.net/base/javascript))(.\*?)(?=\"\s)/e', 'add\_url(\$url, \'\\1\')', $form\_html ); |

最后，正则工具虽然强大，但是从效率和编写时间上来讲，有的时候可能没有explode来的更直接，对于一些紧急或者要求不高的任务，简单、粗暴的方法也许更好。

 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**](http://lib.csdn.net/base/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号或组名。要指定一个子表达式的组名，请使用这样的语法：(?<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，只是为了提供让人阅读注释我们已经讨论了前两种语法。第三个(?:exp)不会改变正则表达式的处理方式，只是这样的组匹配的内容不会像前两种那样被捕获到某个组里面。

位置指定

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，也就是说它们用于指定一个位置，就像\b,^,$那样，因此它们也被称为零宽断言。最好还是拿例子来说明吧：

(?=exp)也叫零宽先行断言，它匹配文本中的某些位置，这些位置的后面能匹配给定的后缀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)，如果在查找I'm singing while you're dancing.时，它会匹配sing和danc。

(?<=exp)也叫零宽后行断言，它匹配文本中的某些位置，这些位置的前面能给定的前缀匹配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。请仔细分析这个表达式，它可能不像你第一眼看出来的那么简单。

下面这个例子同时使用了前缀和后缀：(?<=\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)。

负向位置指定

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**](http://lib.csdn.net/base/softwaretest)(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]总是匹配一个字符，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。负向位置指定能解决这样的问题，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。

零宽负向先行断言(?!exp)，只会匹配后缀exp不存在的位置。\d{3}(?!\d)匹配三位数字，而且这三位数字的后面不能是数字。

同理，我们可以用(?<!exp),零宽负向后行断言来查找前缀exp不存在的位置：(?<![a-z])\d{7}匹配前面不是小写字母的七位数字(实验时发现错误？注意你的“区分大小写”先项是否选中)。

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<b>)，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<b>的话，后缀就是</b>了。整个表达式匹配的是<b>和</b>之间的内容(再次提醒，不包括前缀和后缀本身)。

贪婪与懒惰

当正则表达式中包含能接受重复的限定符(指定数量的代码，例如\*,{5,12}等)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，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

平衡组

如果想要匹配可嵌套的层次性结构的话，就得使用平衡组了。举个例子吧，如何把“xx <aa <bbb> <bbb> aa> yy”这样的字符串里，最长的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（或再写一个）"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（为了能看得更清楚一点，我用了(?'group')的语法）：

<                                                        #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