1. 入门

使用正则表达式hi，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。忽略大小写的情况下它可以匹配 *hi* , *HI* , *Hi* , *hI* 这四种情况中的任意一种。

对于单词中包含连续的上述字符，可以采用的方法：\bhi\b (精确查找hi这个单词)

\b匹配一个隐式位置，它的前一个字符和后一个字符不全是\w。

\w匹配字母或数字或下划线或汉字等

匹配hi直到Lucy：\bhi\b.*\bLucy\b 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

.匹配除了换行符以外的字符　*指定*前边的内容可以连续重复出现任意次

.*连在一起就意味着任意数量的不包含换行的字符

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

\d是个新的元字符，匹配一位数字

也可以写作：0\d{2}-\d{8}

\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

2. 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

\ba\w*\b匹配以字母a开头的单词——先是某个单词开始处(\b)，然后是字母a,然后是任意数量的字母或数字(\w*)，最后是单词结束处(\b)。

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

\b\w{6}\b 匹配刚好6个字母/数字的单词。

元字符^（和数字6在同一个键位上的符号）和$都匹配一个位置，这和\b有点类似。^匹配你要用来查找的字符串的开头，$匹配结尾。这两个代码在验证输入的内容时非常有用，比如一个网站如果要求你填写的QQ号必须为5位到12位数字时，可以使用：^\d{5,12}$，这里的{5,12}和前面介绍过的{2}是类似的，只不过{2}匹配只能不多不少重复2次，{5,12}则是重复的次数不能少于5次，不能多于12次，否则都不匹配。

3. 转义字符

对于元字符本身的匹配，需要使用\.和*。当然，要查找\本身，需要用\\。

例如：unibetter.com匹配unibetter.com，C:\Windows匹配C:\Windows。

4. 重复匹配

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

Windows\d+匹配Windows后面跟1个或更多数字

^\w+匹配一行的第一个单词

5. 字符集合

字符集合(里面的元字符不用转义)：[aeiou]就匹配任何一个英文元音字母，[.?!]匹配标点符号(.或?或!)。

[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）

(?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})。

6. 分枝条件

> |把不同的规则分隔开,符合其中的一种规则即认为匹配(后续规则不再匹配)

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

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

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

7. 分组

> 小括号来指定**子表达式**(也叫做**分组**)，相当于前面的元字符，可以加上匹配符指定重复次数

(\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})。

它也将匹配 *256.300.888.999* 这种不可能存在的IP地址。如果能使用算术比较的话，或许能简单地解决这个问题，但是正则表达式中并不提供关于数学的任何功能，所以只能使用冗长的分组，选择，字符类来描述一个正确的IP地址：((2[0-4]\d|25[0-5]|[01]?\d\d?).){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)。

8. 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**：

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。

9. 后向引用

> 使用小括号指定一个子表达式后，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， \1：代表分组1匹配的文本。

\b(\w+)\b\s+\1\b可以用来匹配重复的单词，像 *go go* , 或者 *kitty kitty* 。这个表达式首先是一个单词，也就是单词开始处和结束处之间的多于一个的字母或数字(\b(\w+)\b)，这个单词会被捕获到编号为1的分组中，然后是1个或几个空白符(\s+)，最后是分组1中捕获的内容（也就是前面匹配的那个单词）(\1)。

你也可以自己指定子表达式的组名。要指定一个子表达式的组名，请使用这样的语法：(?\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**后向**引用这个分组捕获的内容，你可以使用\k,所以上一个例子也可以写成这样：\b(?\w+)\b\s+\k\b。

小括号的其它语法：

表4.常用分组语法

分类	代码/语法	说明
捕获	(exp)	匹配exp,并捕获文本到自动命名的组里
	(?<name>exp)	匹配exp,并捕获文本到名称为name的组里，也可以写成(?'name'exp)
	(?:exp)	匹配exp,不捕获匹配的文本，也不给此分组分配组号
零宽断言	(?=exp)	匹配exp前面的位置
	(?<=exp)	匹配exp后面的位置
	(?!exp)	匹配后面跟的不是exp的位置
	(?<!exp)	匹配前面不是exp的位置
注释	(?#comment)	这种类型的分组不对正则表达式的处理产生任何影响，用于提供注释让人阅读

第三个(?:exp)不会改变正则表达式的处理方式(仅匹配不捕获)，只是这样的组匹配的内容不会像前两种那样被捕获到某个组里面，也不会拥有组号。

零宽断言中的表达式同样是只匹配不捕获。

10. 零宽断言

> 断言用来声明一个应该为真的事实。正则表达式中只有当断言为真时才会继续进行匹配

(?=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。

(?<=\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)。

11. 负向零宽断言详解

(?<![a-z])\d{7}匹配前面不是小写字母的七位数字

\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

(?<=<(\w+)>).*(?=<\/\1>)匹配不包含属性的简单HTML标签内里的内容，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)。

12. 注释

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

要包含注释的话，最好是启用“忽略模式里的空白符”选项，这样在编写表达式时能任意的添加空格，Tab，换行，而实际使用时这些都将被忽略。启用这个选项后，在#后面到这一行结束的所有文本都将被当成注释忽略掉。例如，我们可以前面的一个表达式写成这样：

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

13. 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。前面给出的重复匹配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。

a.*?b匹配最短的，以a开始，以b结束的字符串。如果把它应用于 *aabab* 的话，它会匹配aab（第一到第三个字符）和ab（第四到第五个字符）。

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

14. 特殊处理

名称	说明
IgnoreCase(忽略大小写)	匹配时不区分大小写。
Multiline(多行模式)	更改^和$的含义，使它们分别在任意一行的行首和行尾匹配，而不仅仅在整个字符串的开头和结尾匹配。(在此模式下,$的精确含意是:匹配\n之前的位置以及字符串结束前的位置.)
Singleline(单行模式)	更改.的含义，使它与每一个字符匹配（包括换行符\n）。
IgnorePatternWhitespace(忽略空白)	忽略表达式中的非转义空白并启用由#标记的注释。
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。

15. 其他语法

代码/语法	说明
\a	报警字符(打印它的效果是电脑嘀一声)
\b	通常是单词分界位置，但如果在字符类里使用代表退格
\t	制表符，Tab
\r	回车
\v	竖向制表符
\f	换页符
\n	换行符
\e	Escape
\0nn	ASCII代码中八进制代码为nn的字符
\xnn	ASCII代码中十六进制代码为nn的字符
\unnnn	Unicode代码中十六进制代码为nnnn的字符
\cN	ASCII控制字符。比如\cC代表Ctrl+C
\A	字符串开头(类似^，但不受处理多行选项的影响)
\Z	字符串结尾或行尾(不受处理多行选项的影响)
\z	字符串结尾(类似$，但不受处理多行选项的影响)
\G	当前搜索的开头
\p{name}	Unicode中命名为name的字符类，例如\p{IsGreek}
(?>exp)	贪婪子表达式
(?-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

16. 元字符汇总

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"。'\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。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 "\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 "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。

字符	描述
(?! *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。
\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。

字符	描述
n	标识一个八进制转义值或一个后向引用。如果 *n* 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。
nm	标识一个八进制转义值或一个后向引用。如果 *nm* 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 *nm* 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 *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 (?)。

17. 扩展：平衡组和递归匹配

1、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。

2、如 果你不是一个程序员（或者你自称程序员但是不知道堆栈是什么东西），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。

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

我们需要做的是每碰到了左括号，就在压入一个"Open",每碰到一个右括号，就弹出一个，到了最后就看看堆栈是否为空 - - 如果不为空那就证明左括号比右括号多，那匹配就应该失败。正则表达式引擎会进行回溯(放弃最前面或最后面的一些字符)，尽量使整个表达式得到匹配。

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

平衡组的一个最常见的应用就是匹配HTML,下面这个例子可以匹配 嵌套的 标签 ： <div[^>]*>[^<>]*(((?'Open'<div[^>]*>)[^<>]*)+((?'-Open')[^<>]*)+)*(?(Open)(?!)) .

18. C语言运用

标准的C和C++都不支持正则表达式，但有一些函数库可以辅助C/C++程序员完成这一功能，其中最著名的当数Philip Hazel的Perl-Compatible Regular Expression库，许多Linux发行版本都带有这个函数库。

C语言处理正则表达式常用的函数有regcomp()、regexec()、regfree()和regerror()，一般分为三个步骤，如下所示：

- **C语言中使用正则表达式一般分为三步：**

  编译正则表达式 regcomp()匹配正则表达式 regexec()释放正则表达式 regfree()

下边是对三个函数的详细解释

1、int regcomp (regex_t *compiled, const char *pattern, int cflags)
这个函数把指定的正则表达式pattern编译成一种特定的数据格式compiled，这样可以使匹配更有效。函数regexec 会使用这个数据在目标文本串中进行模式匹配。执行成功返回 0 。

参数说明：
①regex_t 是一个结构体数据类型，用来存放编译后的正则表达式，它的成员re_nsub 用来存储正则表达式中的子正则表达式的个数，子正则表达式就是用圆括号包起来的部分表达式。
②pattern 是指向我们写好的正则表达式的指针。
③cflags 有如下4个值或者是它们或运算(|)后的值：
REG_EXTENDED 以功能更加强大的扩展正则表达式的方式进行匹配。
REG_ICASE 匹配字母时忽略大小写。
REG_NOSUB 不用存储匹配后的结果。
REG_NEWLINE 识别换行符，这样'$'就可以从行尾开始匹配，'^'就可以从行的开头开始匹配。

\2. int regexec (regex_t *compiled, char *string, size_t nmatch, regmatch_t matchptr [], int eflags)
当我们编译好正则表达式后，就可以用regexec 匹配我们的目标文本串了，如果在编译正则表达式的时候没有指定cflags的参数为REG_NEWLINE，则默认情况下是忽略换行符的，也就是把整个文本串当作一个字符串处理。执行成功返回 0 。
regmatch_t 是一个结构体数据类型，在regex.h中定义：

```
typedef struct
{
  regoff_t rm_so;
  regoff_t rm_eo;
} regmatch_t;
```

成员rm_so 存放匹配文本串在目标串中的开始位置，rm_eo 存放结束位置。通常我们以数组的形式定义一组这样的结构。因为往往我们的正则表达式中还包含子正则表达式。数组0单元存放主正则表达式位置，后边的单元依次存放子正则表达式位置。

参数说明：
①compiled 是已经用regcomp函数编译好的正则表达式。
②string 是目标文本串。
③nmatch 是regmatch_t结构体数组的长度。
④matchptr regmatch_t类型的结构体数组，存放匹配文本串的位置信息。
⑤eflags 有两个值
REG_NOTBOL 按我的理解是如果指定了这个值，那么'^'就不会从我们的目标串开始匹配。总之我到现在还不是很明白这个参数的意义；
REG_NOTEOL 和上边那个作用差不多，不过这个指定结束end of line。

\3. void regfree (regex_t *compiled)
当我们使用完编译好的正则表达式后，或者要重新编译其他正则表达式的时候，我们可以用这个函数清空compiled指向的regex_t结构体的内容，请记住，如果是重新编译的话，一定要先清空regex_t结构体。

\4. size_t regerror (int errcode, regex_t *compiled, char *buffer, size_t length)
当执行regcomp 或者regexec 产生错误的时候，就可以调用这个函数而返回一个包含错误信息的字符串。

参数说明：
①errcode 是由regcomp 和 regexec 函数返回的错误代号。
②compiled 是已经用regcomp函数编译好的正则表达式，这个值可以为NULL。
③buffer 指向用来存放错误信息的字符串的内存空间。
④length 指明buffer的长度，如果这个错误信息的长度大于这个值，则regerror 函数会自动截断超出的字符串，但他仍然会返回完整的字符串的长度。所以我们可以用如下的方法先得到错误字符串的长度。

size_t length = regerror (errcode, compiled, NULL, 0);

```c
//下面的程序负责从命令行获取正则表达式，然后将其运用于从标准输入得到的每行数据，并打印出匹配结果。
#include <stdio.h>
#include <sys/types.h>
#include <regex.h>

/* 取子串的函数 */
static char* substr(const char*str,
unsigned start, unsigned end)
{
  unsigned n = end - start;
  static char stbuf[256];
  strncpy(stbuf, str + start, n);
  stbuf[n] = 0;
  return stbuf;
}

/* 主程序 */
int main(int argc, char** argv)
{
  char * pattern;
  int x, z, lno = 0, cflags = 0;
  char ebuf[128], lbuf[256];
  regex_t reg;
  regmatch_t pm[10];
  const size_t nmatch = 10;
  /* 编译正则表达式*/
  pattern = argv[1];
  z = regcomp(?, pattern, cflags);
  if (z != 0){
    regerror(z, ?, ebuf, sizeof(ebuf));
    fprintf(stderr, "%s: pattern '%s' \n",ebuf, pattern);
    return 1;
  }
  /* 逐行处理输入的数据 */
  while(fgets(lbuf, sizeof(lbuf), stdin))
  {
    ++lno;
    if ((z = strlen(lbuf)) > 0 && lbuf[z-1] == '\n')
    lbuf[z - 1] = 0;
    /* 对每一行应用正则表达式进行匹配 */
    z = regexec(?, lbuf, nmatch, pm, 0);
```

```
      if (z == REG_NOMATCH) continue;
      else if (z != 0) {
        regerror(z, ?, ebuf, sizeof(ebuf));
        fprintf(stderr, "%s: regcom('%s')\n", ebuf, lbuf);
        return 2;
      }
      /* 输出处理结果 */
      for (x = 0; x < nmatch && pm[x].rm_so != -1; ++ x)
      {
        if (!x) printf("%04d: %s\n", lno, lbuf);
        printf(" $%d='%s'\n", x, substr(lbuf, pm[x].rm_so, pm[x].rm_eo));
      }
    }
    /* 释放正则表达式 */
    regfree(?);
    return 0;
  }
```

```
执行下面的命令可以编译并执行该程序:
# gcc regexp.c -o regexp
# ./regexp 'regex[a-z]*' < regexp.c
0003: #include <regex.h>
$0='regex'
0027: regex_t reg;
$0='regex'
0054: z = regexec(?, lbuf, nmatch, pm, 0);
$0='regexec'
```

小结：对那些需要进行复杂数据处理的程序来说，正则表达式无疑是一个非常有用的工具。本文重点在于阐述如何在C语言中利用正则表达式来简化字符串处理，以便在数据处理方面能够获得与Perl语言类似的灵活性。

19. php语言运用

PHP中嵌入正则表达式常用的函数有四个：

1、preg_match()： preg_match() 函数用于进行正则表达式匹配，成功返回 1 ，否则返回 0 。

语法：int preg_match(string pattern, string subject [, array matches])

参数	说明
pattern	正则表达式
subject	需要匹配检索的对象
matches	可选，存储匹配结果的数组， $matches[0] 将包含与整个模式匹配的文本，$matches[1] 将包含与第一个捕获的括号中的子模式所匹配的文本，以此类推

例子 1：

```php
<?php
   if(preg_match("/php/i", "PHP is the web scripting language of choice.", $matches))
   {
      print "A match was found:". $matches[0];
   }
   else
   {
      print "A match was not found.";
   }
?>
```

浏览器输出：

```
A match was found: PHP
```

在该例子中，由于使用了 i 修正符，因此会不区分大小写去文本中匹配 php 。

提示：preg_match() 第一次匹配成功后就会停止匹配，如果要实现全部结果的匹配，即搜索到subject结尾处，则需使用 preg_match_all() 函数。

例子 2 ，从一个 URL 中取得主机域名：

```php
<?php
   // 从 URL 中取得主机名
   preg_match("/^(http://)?([^/]+)/i","http://www.5idev.com/index.html", $matches);
   $host = $matches[2]; // 从主机名中取得后面两段
   preg_match("/[^./]+.[^./]+$/", $host, $matches);
   echo "域名为: {$matches[0]}";
?>
```

浏览器输出：

```
域名为: 5idev.com
```

2、preg_match_all()：preg_match_all() 函数用于进行正则表达式全局匹配，成功返回整个模式匹配的次数（可能为零），如果出错返回 FALSE 。

语法：int preg_match_all(string pattern, string subject, array matches [, int flags])

参数	说明
pattern	正则表达式
subject	需要匹配检索的对象
matches	存储匹配结果的数组
flags	可选，指定匹配结果放入 matches 中的顺序，可供选择的标记有： PREG_PATTERN_ORDER：默认，对结果排序使 $matches[0] 为全部模式匹配的数组，$matches[1] 为第一个括号中的子模式所匹配的字符串组成的数组，以此类推 PREG_SET_ORDER：对结果排序使 $matches[0] 为第一组匹配项的数组，$matches[1] 为第二组匹配项的数组，以此类推PREG_OFFSET_CAPTURE：如果设定本标记，对每个出现的匹配结果也同时返回其附属的字符串偏移量

下面的例子演示了将文本中所有

标签内的关键字（php）显示为红色。

```
<?php
   $str = "<pre>学习php是一件快乐的事。</pre><pre>所有的phper需要共同努力！</pre>";
   $kw = "php"; preg_match_all('/<pre>([sS]*?)</pre>/',$str,$mat);
   for($i=0;$i<count($mat[0]);$i++)
   {
      $mat[0][$i] = $mat[1][$i];
      $mat[0][$i] = str_replace($kw, '<span
style="color:#ff0000">'.$kw.'</span>', $mat[0][$i]);
      $str = str_replace($mat[1][$i], $mat[0][$i], $str);
   }
   echo $str;
?>
```

3、preg_replace()：字符串比对解析并取代。

语法: mixed preg_replace(mixed pattern, mixed replacement, mixed subject);

返回值: 混合类型资料

内容说明: 本函数以 pattern 的规则来解析比对字符串 subject，欲取而代之的字符串为参数 replacement。返回值为混合类型资料，为取代后的字符串结果。

范例：下例返回值为 $startDate = 6/19/1969

$patterns = array("/(19|20\d{2})-(\d{1,2})-(\d{1,2})/", "/^\s*{(\w+)}\s*=/");

$replace = array("\\3/\\4/\\1", "$\\1 =");

print preg_replace($patterns, $replace, "{startDate} = 1969-6-19");

4、preg_split()：将字符串依指定的规则切开。

语法: array preg_split(string pattern, string subject, int [limit]);

返回值: 数组

内容说明: 本函数可将字符串依指定的规则分开。切开后的返回值为数组变量。参数 pattern 为指定的规则字符串、参数 subject 则为待处理的字符串、参数 limit 可省略，表示欲处理的最多合乎值。