正则表达式由一些普通[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6)和一些[元字符](https://baike.baidu.com/item/%E5%85%83%E5%AD%97%E7%AC%A6)（metacharacters）组成。普通字符包括大小写的字母和数字，而元字符则具有特殊的含义，我们下面会给予解释。

在最简单的情况下，一个正则表达式看上去就是一个普通的查找串。例如，正则表达式"testing"中没有包含任何元字符，它可以匹配"testing"和"testing123"等字符串，但是不能匹配"Testing"。

要想真正的用好正则表达式，正确的理解元字符是最重要的事情。下表列出了所有的元字符和对它们的一个简短的描述。

|  |  |
| --- | --- |
| 元字符 | 描述 |
| \ | 将下一个字符标记符、或一个向后引用、或一个八进制转义符。例如，“\\n”匹配\n。“\n”匹配换行符。序列“\\”匹配“\”而“\(”则匹配“(”。即相当于多种编程语言中都有的“转义字符”的概念。 |
| ^ | 匹配输入字行首。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。 |
| $ | 匹配输入行尾。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任意次。例如，zo\*能匹配“z”，也能匹配“zo”以及“zoo”。\*等价于{0,}。 |
| + | 匹配前面的子表达式一次或多次(大于等于1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”。?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为一组，后三个o为一组。“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地匹配所搜索的字符串，而默认的贪婪模式则尽可能多地匹配所搜索的字符串。例如，对于字符串“oooo”，“o+”将尽可能多地匹配“o”，得到结果[“oooo”]，而“o+?”将尽可能少地匹配“o”，得到结果 ['o', 'o', 'o', 'o'] |
| .点 | 匹配除“\n”和"\r"之外的任何单个字符。要匹配包括“\n”和"\r"在内的任何字符，请使用像“[\s\S]”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) | 非获取匹配，匹配pattern但不获取匹配结果，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时很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。 |
| (?=pattern) | 非获取匹配，正向肯定预查，在任何匹配pattern的字符串开始处匹配查找字符串，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) | 非获取匹配，正向否定预查，在任何不匹配pattern的字符串开始处匹配查找字符串，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。 |
| (?<=pattern) | 非获取匹配，反向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。  \*python的正则表达式没有完全按照正则表达式规范实现，所以一些高级特性建议使用其他语言如java、scala等 |
| (?<!patte\_n) | 非获取匹配，反向否定预查，与正向否定预查类似，只是方向相反。例如“(?<!95|98|NT|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。  \*python的正则表达式没有完全按照正则表达式规范实现，所以一些高级特性建议使用其他语言如java、scala等 |
| x|y | 匹配x或y。例如，“z|food”能匹配“z”或“food”(此处请谨慎)。“[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“plin”任一字符。 |
| [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。  注意:只有连字符在字符组内部时,并且出现在两个字符之间时,才能表示字符的范围; 如果出字符组的开头,则只能表示连字符本身. |
| [^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 | 匹配一个单词的边界，也就是指单词和空格间的位置（即正则表达式的“匹配”有两种概念，一种是匹配字符，一种是匹配位置，这里的\b就是匹配位置的）。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”；“\b1\_”可以匹配“1\_23”中的“1\_”，但不能匹配“21\_3”中的“1\_”。 |
| \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]。grep 要加上-P，perl正则支持 |
| \D | 匹配一个非数字字符。等价于[^0-9]。grep要加上-P，perl正则支持 |
| \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 | 匹配任何不可见字符，包括空格、制表符、换页符等等。等价于[ \f\n\r\t\v]。 |
| \S | 匹配任何可见字符。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。类似但不等价于“[A-Za-z0-9\_]”，这里的"单词"字符使用Unicode字符集。 |
| \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（0-7），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（&copy;）。 |
| \p{P} | 小写 p 是 property 的意思，表示 Unicode 属性，用于 Unicode 正表达式的前缀。中括号内的“P”表示Unicode 字符集七个字符属性之一：标点字符。  其他六个属性：  L：字母；  M：标记符号（一般不会单独出现）；  Z：分隔符（比如空格、换行等）；  S：符号（比如数学符号、货币符号等）；  N：数字（比如阿拉伯数字、罗马数字等）；  C：其他字符。  *\*注：此语法部分语言不支持，例：javascript。* |
| \<  \> | 匹配词（word）的开始（\<）和结束（\>）。例如正则表达式\<the\>能够匹配字符串"for the wise"中的"the"，但是不能匹配字符串"otherwise"中的"the"。注意：这个元字符不是所有的软件都支持的。 |
| ( ) | 将( 和 ) 之间的表达式定义为“组”（group），并且将匹配这个表达式的字符保存到一个临时区域（一个正则表达式中最多可以保存9个），它们可以用 \1 到\9 的符号来引用。 |
| | | 将两个匹配条件进行逻辑“或”（Or）运算。例如正则表达式(him|her) 匹配"it belongs to him"和"it belongs to her"，但是不能匹配"it belongs to them."。注意：这个元字符不是所有的软件都支持的。 |

最简单的[元字符](https://baike.baidu.com/item/%E5%85%83%E5%AD%97%E7%AC%A6)是点，它能够匹配任何单个字符（注意不包括换行符）。假定有个文件test.txt包含以下几行内容：

he is arat

he is in a rut

the food is Rotten

I like root beer

我们可以使用grep命令来测试我们的正则表达式，grep命令[使用正则表达式](https://baike.baidu.com/item/%E4%BD%BF%E7%94%A8%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F)去尝试匹配指定文件的每一行，并将至少有一处匹配表达式的所有行显示出来。命令

*grep r.t test.txt*

在test.txt文件中的每一行中搜索正则表达式r.t，并打印输出匹配的行。正则表达式r.t匹配一个r接着任何一个字符再接着一个t。所以它将匹配文件中的rat和rut，而不能匹配Rotten中的Rot，因为正则表达式是大小写敏感的。要想同时匹配大写和小写字母，应该使用[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6)区间[元字符](https://baike.baidu.com/item/%E5%85%83%E5%AD%97%E7%AC%A6)（方括号）。正则表达式[Rr]能够同时匹配R和r。所以，要想匹配一个大写或者小写的r接着任何一个字符再接着一个t就要使用这个表达式：[Rr].t。

要想匹配行首的字符要使用抑扬字符（^）——有时也被叫做插入符。例如，想找到text.txt中行首"he"打头的行，你可能会先用简单表达式he，但是这会匹配第三行的the，所以要[使用正则表达式](https://baike.baidu.com/item/%E4%BD%BF%E7%94%A8%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F)^he，它只匹配在行首出现的he。

有时候指定“除了×××都匹配”会比较容易达到目的，当抑扬字符（^）出方括号中时，它表示“排除”，例如要匹配he ，但是排除前面是t or s的情形（也就是the和she），可以使用：[^st]he。

可以使用方括号来指定多个字符区间。例如正则表达式[A-Za-z]匹配任何字母，包括大写和小写的；正则表达式[A-Za-z][A-Za-z]\* 匹配一个字母后面接着0或者多个字母（大写或者小写）。当然我们也可以用[元字符](https://baike.baidu.com/item/%E5%85%83%E5%AD%97%E7%AC%A6)+做到同样的事情，也就是：[A-Za-z]+ ，和[A-Za-z][A-Za-z]\*完全等价。但是要注意元字符+ 并不是所有支持正则表达式的程序都支持的。关于这一点可以参考后面的正则表达式语法支持情况。

要指定特定数量的匹配，要使用大括号（注意并不是所有扩展正则表达式的实现都支持大括号。此外，根据具体的实现，您可能需要先使用反斜杠对其进行转义。）。想匹配所有10和100的实例而排除1和 1000，可以使用：10\{1,2\} 或 10{1, 2}，这个正则表达式匹配数字1后面跟着1或者2个0的模式。在这个元字符的使用中一个有用的变化是忽略第二个数字，例如正则表达式0\{3,\} 或 0{3,} 将匹配至少3个连续的0。[2] 

**例1**

将所有方法foo(a,b,c)的实例改为foo(b,a,c)。这里a、b和c可以是任何提供给方法foo()的参数。也就是说我们要实现这样的转换：

之前 之后

foo(10,7,2) foo(7,10,2)

foo(x+13,y-2,10) foo(y-2,x+13,10)

foo( bar(8), x+y+z, 5) foo( x+y+z, bar(8), 5)

下面这条替换命令能够实现这一方法：

*:%s/foo\(([^,]\*),([^,]\*),([^,]\*)\)/foo\(\2,\1,\3\)/g*

让我们把它打散来加以分析。写出这个表达式的基本思路是找出foo()和它的括号中的三个参数的位置。第一个参数是用这个表达式来识别的：：([^,]\*)，我们可以从里向外来分析它：

[^,] 除了逗号之外的任何字符

[^,]\* 0或者多个非逗号字符

([^,]\*) 将这些非逗号字符标记为\1，这样可以在之后的替换模式表达式中引用它

([^,]\*), 我们必须找到0或者多个非逗号[字符](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6)后面跟着一个逗号，并且非逗号字符那部分要标记出来以备后用。

正是指出一个[使用正则表达式](https://baike.baidu.com/item/%E4%BD%BF%E7%94%A8%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F)常见错误的最佳时机。为什么我们要使用[^,]\*这样的一个表达式，而不是更加简单直接的写法，例如：.\*，来匹配第一个参数呢？设想我们使用模式.\*来匹配字符串"10,7,2"，它应该匹配"10,"还是"10,7,"？为了解决这个两义性（ambiguity），正则表达式规定一律按照最长的串来，在上面的例子中就是"10,7,"，显然这样就找出了两个参数而不是我们期望的一个。所以，我们要使用[^,]\*来强制取出第一个逗号之前的部分。

这个表达式我们已经分析到了：foo\(([^,]\*)，这一段可以简单的翻译为“当你找到foo(就把其后直到第一个逗号之前的部分标记为\1”。然后我们使用同样的办法标记第二个参数为\2。对第三个参数的标记方法也是一样，只是我们要搜索所有的字符直到右括号。我们并没有必要去搜索第三个参数，因为我们不需要调整它的位置，但是这样的模式能够保证我们只去替换那些有三个参数的foo()方法调用，在foo()是一个[重载](https://baike.baidu.com/item/%E9%87%8D%E8%BD%BD)（overloading）方法时这种明确的模式往往是比较保险的。然后，在替换部分，我们找到foo()的对应实例，然后利用标记好的部分进行替换，是把第一和第二个参数交换位置。

**正则表达式语法支持情况**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **命令或环境** | **.** | **[ ]** | **^** | **$** | **\( \)** | **\{ \}** | **?** | **+** | **|** | **( )** |
| **vi** | √ | √ | √ | √ | √ |  |  |  |  |  |
| **Visual C++** | √ | √ | √ | √ | √ |  |  |  |  |  |
| **awk** | √ | √ | √ | √ |  | awk是支持该语法的，只是要在命令  行加入 --posix or --re-interval参数即可，可见  man awk中的interval expression | √ | √ | √ | √ |
| **sed** | √ | √ | √ | √ | √ | √ |  |  |  |  |
| **delphi** | √ | √ | √ | √ | √ |  | √ | √ | √ | √ |
| **python** | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| **java** | √ | √ | √ | √ | √ | √ | √ | √ | √ | √ |
| **javascript** | √ | √ | √ | √ | √ |  | √ | √ | √ | √ |
| **php** | √ | √ | √ | √ | √ |  |  |  |  |  |
| **perl** | √ | √ | √ | √ | √ |  | √ | √ | √ | √ |
| **C#** | √ | √ | √ | √ |  |  | √ | √ | √ | √ |