句点"."匹配单字符  
1) . : 匹配任意单ASCII 字符,可以为字母，或为数字。  
2) 举例: ..XC..匹配deXC1t、23XCdf 等，.w..w..w.匹配rwxrw-rw-

行首以^匹配字符串或字符序列  
1) ^ : 允许在一行的开始匹配字符或单词。  
2) 举例: ^.01 匹配0011cx4、c01sdf 等，^d 匹配drwxr-xr-x、drw-r--r--等

行尾以$匹配字符串或字符  
1) $ : 在行尾匹配字符串或字符，$符号放在匹配单词后。  
2) 举例: trouble$ 匹配以单词trouble 结尾的所有行  
^$匹配所有空行

使用\*匹配字符串中的单字符或其重复序列(与文件名置换中的"\*"不一样)  
1) \* : 一个单字符后紧跟\*，匹配0 个或多个此单字符。  
2) 举例: compu\*t 将匹配字符u 一次或多次，即匹配computer computing compuuute 等  
1033\* 可以匹配 101333 10133 1013444 等  
3) 在正则表达式中使用"\*"，有时会产生非预期的结果。

使用/屏蔽一个特殊字符的含义  
1) / : 用来屏蔽一个元字符的特殊含义。因为有时在shell 中元字符有特殊含义。/可以使其失去应有意义。  
2) 举例: 在正则表达式中匹配以\*.pas 结尾的所有文件: /\*/.pas$

使用[]匹配属于一个范围或集合单个字符  
1) [ ] : 匹配"[ ]"内的字符。可以是一个单字符，也可以是字符序列。可以使用"-"表示括号"[ ]"内字符序列范围，  
如用[1-5]代替[12345]。可以用逗号","分隔括号"[]"内的字符。

2) 当"^"符号当直接靠着"["，意指否定或不匹配括号"[]"里内容  
3) 举例: [0-9]匹配任意一个数字；[a-z]匹配任意一个小写字母；[0-9A-Za-z]匹配任意字母或数字；  
[C,c]omputer 匹配Computer 和computer；[^a-zA-Z]匹配任一非字母型字符

使用"/{/}"匹配模式结果出现的次数  
1) pattern/{n/}   : 匹配模式pattern 出现n 次的情形。  
2) pattern/{n,/}  : 匹配模式pattern 最少出现n 次的情形。  
3) pattern/{,m/}  : 匹配模式pattern 最多出现m 次的情形。  
4) pattern/{n,m/} : 匹配模式pattern 出现次数在n 与m 之间的情形。

5) 举例: A/{2/}B 匹配的值为AAB  
A/{2,/}B   匹配的值可以是AAB 或AAAAAB，但不能匹配AB  
A/{2,4/}B  匹配的值可以是AAB、AAAB、AAAAB，但不能匹配AB 或AAAAAB 等  
[0-9]/{4/}CX[0-9]/{4/}  匹配数字出现4 次后跟CX，最后是数字出现4 次的情形  
6) 实际上真正的格式是 {n} {n,} {,m} {n,m}，只不过对"{"和"}"应用了Esacpe 字符"/"。

经常使用的正则表达式举例  
[Ss]igna[lL]   匹配单词signal、signaL、Signal、SignaL  
[Ss]igna[lL]/. 同上，但加一句点  
^USER$         只包含USER 的行  
/.             带句点的行  
^d..x..x..x    对用户、用户组及其他用户、组成员有可执行权限的目录  
^[^l]          排除符号链接文件后的文件目录列表(即不是以"l"开始的行)  
[yYnN]         大写或小写y 或n  
^.\*$           匹配行中任意字符串  
^......$       包括6 个字符的行  
[a-zA-Z]       任意单个字母  
[a-z]\*         至少一个小写字母  
[^0-9/$]       非数字或美元符号  
[123]          1 到3 中一个数字  
/^q            以^q 开始行  
^.$            仅有一个字符的行  
^/.[0-9][0-9]  以一个句点和两个数字开始的行  
[0-9]/{2/}-[0-9]/{2/}-[0-9]/{4/}                日期格式dd-mm-yyyy  
[0-9]/{3/}/.[0-9]/{3/}/.[0-9]/{3/}/.[0-9]/{3/}  类IP 地址格式

                                                nnn.nnn.nnn.nnn  
.\*             匹配任意多个字符

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。

/   
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。

$   
匹配输入字符串的结束位置。

\*   
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。 "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VisualBasic 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。

(?!pattern)   
负向预查，在任何不匹配Negative lookahead matches the search string at any point where astring 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'。

/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"。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。

/un   
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 (?)。

**Shell字符串处理**

1. 构造字符串  
   直接构造  
   STR\_ZERO=hello  
   STR\_FIRST="i am a string"  
   STR\_SECOND='success'  
     
   重复多次  
   #repeat the first parm($1) by $2 times  
   strRepeat()  
   {  
   local x=$2  
   if [ "$x" == "" ]; then  
   x=0  
   fi  
     
   local STR\_TEMP=""  
   while [ $x -ge 1 ];  
   do  
   STR\_TEMP=`printf "%s%s" "$STR\_TEMP" "$1"`  
   x=`expr $x - 1`  
   done  
   echo $STR\_TEMP  
   }  
     
   举例：  
   STR\_REPEAT=`strRepeat "$USER\_NAME" 3`  
   echo "repeat = $STR\_REPEAT"
2. 赋值与拷贝  
   直接赋值  
   与构造字符串一样  
   USER\_NAME=terry  
     
   从变量赋值  
   ALIASE\_NAME=$USER\_NAME
3. 联接  
   直接联接两个字符串  
   STR\_TEMP=`printf "%s%s" "$STR\_ZERO" "$USER\_NAME"`  
   使用printf可以进行更复杂的联接
4. 求长  
   求字符数(char)  
   COUNT\_CHAR=`echo "$STR\_FIRST" | wc -m`  
   echo $COUNT\_CHAR  
     
   求字节数(byte)  
   COUNT\_BYTE=`echo "$STR\_FIRST" | wc -c`  
   echo $COUNT\_BYTE  
     
   求字数(word)  
   COUNT\_WORD=`echo "$STR\_FIRST" | wc -w`  
   echo $COUNT\_WORD
5. 比较  
   相等比较  
   str1 = str2  
     
   不等比较  
   str1 != str2  
     
   举例：  
   if [ "$USER\_NAME" = "terry" ]; then  
   echo "I am terry"  
   fi  
     
   小于比较  
   #return 0 if the two string is equal, return 1 if $1 < $2, else 2strCompare() { local x=0 if [ "$1" != "$2" ]; then x=2 localTEMP=`printf "%s/n%s" "$1" "$2"` local TEMP2=`(echo "$1"; echo "$2") |sort` if [ "$TEMP" = "$TEMP2" ]; then x=1 fi fi echo $x }
6. 测试  
   判空  
   -z str  
     
   判非空  
   -n str  
     
   是否为数字  
   # return 0 if the string is num, otherwise 1  
   strIsNum()  
   {  
   local RET=1  
   if [ -n "$1" ]; then  
   local STR\_TEMP=`echo "$1" | sed 's/[0-9]//g'`  
   if [ -z "$STR\_TEMP" ]; then  
   RET=0   
   fi  
   fi  
   echo $RET  
   }  
     
   举例：  
   if [ -n "$USER\_NAME" ]; then  
   echo "my name is NOT empty"  
   fi  
     
   echo `strIsNum "9980"`
7. 分割  
   以符号＋为准，将字符分割为左右两部分  
   使用sed  
   举例：  
   命令 date --rfc-3339 seconds 的输出为  
   2007-04-14 15:09:47+08:00  
   取其＋左边的部分  
   date --rfc-3339 seconds | sed 's/+[0-9][0-9]:[0-9][0-9]//g'  
   输出为  
   2007-04-14 15:09:47  
   取+右边的部分  
   date --rfc-3339 seconds | sed 's/.\*+//g'  
   输出为  
   08:00  
     
   以空格为分割符的字符串分割  
   使用awk  
   举例：  
   STR\_FRUIT="Banana 0.89 100"  
   取第3字段  
   echo $STR\_FRUIT | awk '{ print $3; }'
8. 子字符串  
   字符串1是否为字符串2的子字符串  
   # return 0 is $1 is substring of $2, otherwise 1  
   strIsSubstring()  
   {  
   local x=1  
   case "$2" in  
   \*$1\*) x=0;;  
   esac  
   echo $x  
   }

模仿秀

您可以将 UNIX 命令行看作是一句话：

* 可执行命令，如 *cat* 或 *ls*，是**动词**——操作。
* 命令的输出是**名词**——要查阅或使用的数据。
* Shell 操作符，如 |（管道）或 >（重定向标准输出），是**连词**——用于连接句子。

例如，命令行：ls -A | wc -l 用于计算当前目录下的条目数（忽略特殊条目 . 和 ..），它包含两个句子。第一个句子 ls -A 是动词结构，列举当前目录下的内容，第二个句子 wc -l 是另一个动词结构，用于计算行数。第一个句子输出的结果作为第二个句子的输入，并由连接词（管道）连接这两个句子。

在本系列文章以及其他文章中展示的许多您可能已经学习过的命令行句式都具有这种句子结构。

但是，如果缺少了文法上的修饰语，命令行将显得不专业。当然，基本句子也能完成工作，但是这样显得不*优美*。（在此对高中英语演唱二人组 Rad 女士和 Perlstein 女士表示歉意。）解决更有趣的问题需要用到*形容词*。

几乎所有重要问题都需要从无用数据中过滤出有用数据。虽然属性的数量和种类会有所不同，但是每种方案都通过某种方式（形式或格式），隐式或显式地描述了它要查找并处理的信息，从而生成另外一种形式的其他信息。

在命令行中，*正则表达式* 的作用相当于形容词——一种描述或限定词。在应用到输出时，正则表达式可辨别相关数据和无关数据。

标点概述

让我们看一个示例问题。

grep 实用工具逐行过滤输入并寻找匹配。grep 的最简单应用是打印那些包含与某个模式匹配的文本的行。grep 可以查找具有固定顺序的字符组合，甚至可以通过使用 -i 选项来忽略大小写。

因此，假定文件 heroes.txt 包含以下行：

|  |
| --- |
| Catwoman Batman The Tick Spider Man Black Cat Batgirl Danger Girl Wonder Woman Luke Cage The Punisher Ant Man Dead Girl Aquaman SCUD Spider Woman Blackbolt Martian Manhunter |

命令行：

|  |
| --- |
| grep -i man heroes.txt |

将生成：

|  |
| --- |
| Catwoman Batman Spider Man Wonder Woman Ant Man Aquaman Martian Manhunter |

其中 grep 扫描 heroes.txt 文件中的每一行并查找字母 *m*，后面紧跟 *a*，然后紧跟 *n*。除了必须保证相邻，这些字母可以出现在行的任何位置，甚至可以位于较大的单词中间。在不考虑大小写的情况下（-i 选项），Catwoman、Batman、Spider Man、Wonder Woman、Ant Man、Aquaman 和 Martian Manhunter 都包含字符串 man。

grep 实用工具包含其他可优化搜索的内置选项。例如，-w 选项限制于匹配整个单词，因此 grep -i -w man 将*排除*Catwoman 和 Batman（举例来说）。

该工具还有一个优秀的功能，可以排除而不是包括所有匹配的搜索结果。使用 -v 选项来*排除* 匹配的行。例如：

|  |
| --- |
| grep -v -i 'spider' heroes.txt |

将打印除了包含字符串 spider 之外的所有行。

|  |
| --- |
| Catwoman Batman The Tick Black Cat Batgirl Danger Girl Wonder Woman Luke Cage The Punisher Ant Man Dead Girl Aquaman SCUD Blackbolt Martian Manhunter |

但是，对于以下这些情况，您该如何处理？只希望得到那些*开头为*“Bat”的单词；或者以“bat”、“Bat”、“cat”或“Cat”开头的单词？或者希望知道有多少漫画复仇者的名字以“man”*结束*。在这些实例中，类似于上述三个示例的简单字符串搜索将无法满足要求，因为这些搜索不区分位置。

位置、位置、位置和备选项

正则表达式*可以* 过滤特定的位置，例如行的开始或结束，以及单词的开始和结束。正则表达式（通常简写为 *regex*）还可以描述：备选项（您可将其称为“this”或“that”）；固定长度、可变长度或不定长度的重复；范围（例如，“a-m 之间的任意字母”）；还有字符的类别或种类（“可打印字符”或“标点符号”），以及其他技术。

[表 1](http://www.ibm.com/developerworks/cn/aix/library/au-speakingunix9/#table1) 显示了一些常用的正则表达式操作符。您可以连接[表 1](http://www.ibm.com/developerworks/cn/aix/library/au-speakingunix9/#table1) 中显示的元素（以及其他操作符）并加以组合使用，从而构建（非常）复杂的正则表达式。

**表 1. 常用的正则表达式操作符**

|  |  |
| --- | --- |
| **操作符** | **用途** |
| .（句号） | 匹配任意单个字符。 |
| ^（脱字号） | 匹配出现在行首或字符串开始位置的空字符串。 |
| $（美元符号） | 匹配出现在行末的空字符串。 |
| A | 匹配大写字母 *A*。 |
| a | 匹配小写字母 *a*。 |
| /d | 匹配任意一位数字。 |
| /D | 匹配任意单个非数字字符。 |
| /w | 匹配任意单个字母数字字符，同义词是 [:alnum:]。 |
| [A-E] | 匹配任意大写的 *A、B、C、D* 或 *E*。 |
| [^A-E] | 匹配*除* *A、B、C、D* 和 *E* 之外的任意字符。 |
| X? | 匹配出现零次或一次的大写字母 *X*。 |
| X\* | 匹配零个或任意个大写 *X*。 |
| X+ | 匹配一个或多个字母 *X*。 |
| X{*n*} | 精确匹配 *n* 个字母 *X*。 |
| X{*n,m*} | 匹配最少 *n* 个并且不超过 *m* 个字母 *X*。如果省略 *m*，表达式将尝试匹配最少 *n 个 X*。 |
| (abc|def)+ | 匹配一连串的（最少一个） abc 或 def；abc 和 def 将匹配。 |

以下是一些使用 grep 作为搜索工具的正则表达式示例。许多其他 UNIX 工具，包括交互式编辑器 vi 和 Emacs、流编辑器 sed 和 awk，以及所有现代编程语言都支持正则表达式。在您学会正则表达式的语法（也许相当晦涩）之后，就可以将您的专业知识灵活运用到不同的工具、编程语言和操作系统。

**查找以“Bat”开头的名称**

要查找以“Bat”开头的名称，请使用：

|  |
| --- |
| grep -E '^Bat' |

可以使用 -E 选项来指定正则表达式。^（脱字号）字符匹配行首或字符串的开头，这是一个出现在每行或每个字符串开头字符之前的假想字符。字母 B、a 和 t 只具有字面含义并且仅匹配那些特定的字符。因此，命令 grep -E '^Bat'将生成：

|  |
| --- |
| Batman Batgirl |

由于许多 regex 操作符也为 Shell 所使用（其中一些具有不同的用途，另外一些则有类似的用途），因此一个好的习惯是使用单引号将命令行中的每个 regex 括起来，以保护 regex 操作符免遭 Shell 的误解。例如，\*（星号）和 $（美元符号）都是 regex 操作符，并且对于您的 Shell 具有特殊的含义。

**查找以“man”结尾的名称**

要查找以“man”结尾的名称，可以使用 regex man$ 来匹配序列 m、a 和 n，并且后面紧接与 regex 操作符 $ 匹配的行（字符串）。

**查找空行**

基于 ^ 和 $ 的作用，您可以使用 regex ^$ 来查找空行（相当于在开始之后立即结束的行）。

**备选项或集合操作符**

要查找以“bat”、“Bat”、“cat”或“Cat”开头的单词，可以使用以下两个技巧。首先是*备选项*，如果备选项中的*任意* 模式匹配，都会产生匹配的结果。例如，命令：

|  |
| --- |
| grep -E '^(bat|Bat|cat|Cat)' heroes.txt |

可实现这一技巧。regex 操作符 |（竖线）表示备选项，因此 this|that 匹配字符串 this 或字符串 that。因此，^(bat|Bat|cat|Cat) 表示“行首紧跟 bat、Bat、cat 或 Cat之一。”当然，可以使用 grep -i 来简化该 regex，这样可以忽略大小写，从而将命令简化为：

|  |
| --- |
| grep -i -E '^(bat|cat)' heroes.txt |

匹配“bat”、“Bat”、“cat”或“Cat”的另一个方法是使用 [ ]（方括号）*集合* 操作符。如果将一组字符放在一个集合中，则可以匹配那些字符中的任意一个。（您可以将*集合* 看作是字符备选项的简写法。）

例如，命令行：

|  |
| --- |
| grep -E '^[bcBC]at' heroes.txt |

与以下命令生成的结果相同：

|  |
| --- |
| grep -E '^(bat|Bat|cat|Cat)' heroes.txt |

您可以再次使用 -i 将 regex 简化为 ^[bc]at。

而且，还可以使用 -（连字符）操作符在集合中指定包含的字符范围。例如，用户名通常以字母开头。假定要在提交给您的服务器的 Web 表格中验证这样的用户名，可以使用类似于 ^[A-Za-z] 的 regex。此 regex 表示“字符串的开头后紧跟任意大写字母 (*A-Z*) 或任意小写字母 (*a-z*)。”顺便说明一下，[A-z] 与 [A-Za-z] 作用相同。

还可以在集合中混合使用范围和单个字符。regex [A-MXYZ] 将匹配任意大写的 *A-M、X、Y* 和 *Z*。

并且，如果希望反转集合（即排除集合中的任意字符），可以使用特殊集合 [^ ] 并包含要排除的范围或字符。以下是反转集合的示例。要查找所有名称中包含 *at* 的超级英雄，并排除 Dark Knight 和 Batman，请键入：

|  |
| --- |
| grep -i -E '[^b]at' heroes.txt |

此命令生成：

|  |
| --- |
| Catwoman Black Cat |

由于某些集合需要经常使用，所以设计出简化符号以代替大量字符。例如，集合 [A-z0-9\_] 十分常用，因此可以简写为 /w。与此类似，操作符 /W 是集合 [^A-z0-9\_] 的简写。还可以使用符号 [:alnum:] 代替 /w，使用 [^[:alnum:]]代替 /W。

顺便说明一下，/w（以及同义词 [:alnum:]）是特定于区域的，而 [A-z0-9\_] 即表示字母 A-z、数字 *0-9* 和下划线。如果要开发国际化应用程序，请使用区域特定的格式以使代码可以在许多区域之间移植。

跟我一起重复：重复，重复，重复

到目前为止，已经介绍了字面值、位置和两种备选项操作符。仅使用这些内容，就可以匹配大多数具有*可预测* 长度的模式。现在回到用户名，通过以下 regex 命令可以确保每个用户名以字母开头并紧跟恰好七个字母或数字：

|  |
| --- |
| [a-z][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9] |

但是这样有点笨拙。而且，它只匹配恰好八个字符的用户名。它不会匹配三到八个字符之间的名称，这通常也是有效的用户名。

正则表达式还可以包括*重复修饰符*。重复修饰符可以指定数量，如没有、一个、多个、一个或多个，零或一个、五到十个，以及恰好三个。重复修饰符必须与其他模式组合，修饰符本身没有含义。

例如，regex：

|  |
| --- |
| ^[A-z][A-z0-9]{2,7}$ |

可以实现前面描述的用户名过滤功能。*用户名* 是以字母开头，后面紧跟至少两个，但不超过七个字母或数字的字符串，并且紧跟字符串结尾。

此处的位置定位点非常重要。如果没有两个位置操作符，则会错误地接受任意长度的用户名。为什么呢？请考虑 regex：

|  |
| --- |
| ^[A-z][A-z0-9]{2,7} |

此命令辨别：字符串是否以字母开头并紧跟二到七个字母？但是它未提到终止条件。因此，字符串 samuelclemens 满足条件，但是它的长度显然超出了有效用户名的范围。与此类似，省略开始定位点 ^，或同时省略两个定位点将分别匹配以类似 munster1313 结束或包含该字符串的字符串。如果必须匹配特定的长度，请记得在要求的模式的开头和结尾分别加上分隔符。

以下是其他一些示例：

* 可以使用 {2,} 查找两次或多次重复。regex ^G[o]{2,}gle 匹配 Google、Gooogle、Goooogle 等等。
* 重复修饰符 ?、+ 和 \* 分别查找零次或一次、一次或多次，以及零次或多次重复。（例如，您可以将 ? 看作是{0,1} 的简写法。）

regex boys? 匹配 boy 或 boys；regex Goo?gle 匹配 Gogle 或 Google。

regex Goo+gle 匹配 Google、Gooogle、Goooogle 等等。

construct Goo\*gle 匹配 Gogle、Google、Gooogle 等等。

* 可以将重复修饰符应用到单个字符（如上所示），还可以应用到更复杂的组合。使用 ( 和 ) 圆括号（就像数学中的用法）将修饰符应用到子表达式。下面是一个示例：给定文本文件 test.txt：

|  |
| --- |
| The rain in Spain falls mainly  on the the plain.  It was the best of of times; it was the worst of times. |

* 命令 grep -i -E '(/b(of|the)/W+){2,}' test.txt 将生成：

|  |
| --- |
| on the the plain. It was the best of of times; |

* regex 操作符 /b 匹配*单词边界* 或 (/W/w|/w/W)。该 regex 表示“一连串完整单词‘the’或‘of’后面紧跟非文字字符。”您可能会提出疑问，为什么 /W+ 是必需的：/b 是位于单词开头或结尾的空字符串。在单词之间必须包括这一（或这些）字符，否则该 regex 将无法找到匹配。

捕获需要注意的内容

查找文本是常见的问题，但是更常见的问题则是希望在找到文本之后将其提取出来。换句话说，您希望去粗取精。

正则表达式通过*捕获* 来提取信息。如果希望将需要的文本与其他内容分开，请使用圆括号将模式括起来。实际上，您已经使用圆括号收集术语；在默认情况下，圆括号自动进行捕获。

要查看捕获，请切换到 Perl。（grep 实用工具不支持捕获，因为其目标是打印包含模式的行。）

以下命令：

|  |
| --- |
| perl -n -e '/^The/s+(.\*)$/ && print "$1/n"' heroes.txt |

将打印：

|  |
| --- |
| Tick Punisher |

使用命令 perl -e 可以直接从命令行运行 Perl 程序。perl -n 命令针对输入文件的每一行运行一次程序。命令的 regex 部分，即位于斜杠之间的文本（/）表示“匹配字符串的开头，然后字母‘T’、‘h’、‘e’后紧跟一个或多个空格字符/s+，然后捕获直到字符串结尾的所有字符。

Perl 捕获内容被放在以 $1 开头的特殊 Perl 变量中。Perl 程序的其余部分打印捕获的内容。

每个嵌套的括号对，从左开始算起，每个左圆括号加一，放在下一个特殊的数字变量中。例如：

|  |
| --- |
| perl -n -e '/^(/w)+-(/w+)$/ && print "$1 $2"' |

将生成：

|  |
| --- |
| Spider Man Ant Man Spider Woman |

捕获感兴趣的文本仅仅是隔靴搔痒。如果能够准确确定材料，就可以使用其他材料改变其外观。类似于 vi 和 Emacs 的编辑器将模式匹配与替换组合，从而将查找和替换文本组合成一步操作。还可以使用模式、替换和 sed 从命令行更改文本。

丰富的主题

正则表达式非常强大；可供使用的操作符的数量庞大，种类繁多。它包含如此丰富的信息和实践知识，我们在这里所能列举的实属凤毛麟角。

幸运的是，有以下三种优秀的正则表达式理论来源可供使用：

* 如果在您的系统上有 Perl，可以参阅 Perl Regular Expression man 页面（键入 perldoc perlre）。它会提供 regex 的精彩介绍，并包含许多有用的示例。许多编程语言都已采用 Perl 兼容的正则表达式 (PCRE)，因此您在此 man 页面读到的内容已被直接转换到 PHP、[**Python**](http://lib.csdn.net/base/11)、[**Java**](http://lib.csdn.net/base/17)™ 和 Ruby 编程语言，以及许多其他最新工具。
* Jeffrey Friedl 编著的《正则表达式》（第三版）被认为是 regex 用法方面的圣经。该书细致、准确、清晰、务实地说明了匹配的工作方式、所有的 regex 操作符、多数优先性（限制 + 和 \* 匹配字符的数量），以及更多内容。此外，Friedl 的书还包括一些令人惊叹的正则表达式，可以准确地匹配完全限定的电子邮件地址和其他 Request for Comments (RFC) 特定的字符串。
* Nathan Good 编著的 *Regular Expression Recipes* 一书提供了针对许多常见数据处理和过滤问题的有用的解决方案。如果需要提取邮政编码、电话号码或引用的字符串，请尝试 Nathan 的解决方案。

在命令行中，可以采用许多方法使用正则表达式。几乎每个处理文本的命令都支持某种形式的正则表达式。大多数 Shell 命令语法还或多或少地扩展正则表达式以匹配文件名（尽管操作符的功能可能有所不同）。

例如，键入 ls [a-c] 以查找名为 *a、b* 或 *c* 的文件。键入 ls [a-c]\* 以查找以 *a、b* 或 *c* 开头的所有文件名。此处的\* 在 Shell 中不像 grep 的解释器那样修饰 [a-c]，\* 被解释为 .\*。? 操作符在 Shell 中也可以工作，但是被解释为 .，即匹配任意单个字符。

查看您最喜欢的实用工具或 Shell 的文档以确定哪些 regex 操作符受支持，以及操作符可能具有的独特性。

下面我们就以具体的实例来看一下如何使用正则表达式。其中用黑体着重标出的是匹配到的字符串。

一个最简单的例子便是 **/all/**，比如下面一段文字：

John’s b**all** fell into the hole

John cried because it is **all** his life.

这个正则表达式不含任何的原字符，它查找的是字符串all，这个字符串all可以是独成一个单词，也可以是其它单词的一部分，因此正则表达式**/all/**既匹配**ball**里的**all**，也匹配完整的单词all。

下面我们着重讨论正则表达式里原字符的用法。

**3.1            行首、行尾定位符**

行首定位符^

Here is a tongue twister:

**Bobby** Bippy bought a bat.

**Bobby** Bippy bought a ball.  
With his bat Bob banged the ball  
Banged it bump against the wall  
But so boldly Bobby banged it  
That he burst his rubber ball, "Boo!" cried Bobby  
Bad luck ball, Bad luck Bobby, bad luck ball  
Now to drown his many troubles  
**Bobby** Bippy's blowing bubbles.

/^Bobby/

匹配位于行首的Bobby。

Here is a tongue twister:

Bobby Bippy bought a bat.

Bobby Bippy bought a ball.  
With his bat Bob banged the ball  
Banged it bump against the wall  
But so boldly Bobby banged it  
That he burst his rubber ball, "Boo!" cried **Bobby**  
Bad luck ball, Bad luck Bobby, bad luck ball  
Now to drown his many troubles  
Bobby Bippy's blowing bubbles.

/Bobby$/

匹配位于行尾的Bobby。

**3.2            词首、词尾定位符**

词首定位符 /<

Here is a tongue twister:

**Bo**bby Bippy bought a bat.

**Bo**bby Bippy bought a ball.  
With his bat **Bo**b banged the ball  
Banged it bump against the wall  
But so boldly **Bo**bby banged it  
That he burst his rubber ball, "**Bo**o!" cried **Bo**bby  
Bad luck ball, Bad luck **Bo**bby, bad luck ball  
Now to drown his many troubles  
**Bo**bby Bippy's blowing bubbles.

//<Bo/

匹配位于词首的字符串Bo。

词尾定位符 />

Here is a tongue twister:

Bobby Bippy bought a bat.

Bobby Bippy bought a **ball**.  
With his bat Bob banged the **ball**  
Banged it bump against the wall  
But so boldly Bobby banged it  
That he burst his rubber ball, "Boo!" cried Bobby  
Bad luck ball, Bad luck Bobby

Bad luck **ball**  
Now to drown his many troubles  
Bobby Bippy's blowing bubbles.

/ball/>/

匹配位于词尾的字符串ball。

在一个表达式中搭配使用词首定位符与词尾定位符

John’s ball fell into the **hole**

John cried because it is his whole life

//<hole/>/

匹配以h作为单词开头并且以e作为单词结尾的模式hole。也就是说，字母h的前面是一个分隔单词的字符（比如空格或换行符），字母l的后面也是一个分隔单词的字符。这样，在这个例子中只有完整的单词hole会被匹配，而单词whole就不会被匹配。

**3.3            匹配单个字符**

匹配任意的一个字符 **.**

Here is a tongue twister:

**Bobby Bippy** bought a bat.

**Bobby Bippy**bought a ball.  
With his bat Bob banged the ball  
Banged it bump against the wall  
But so boldly **Bobby**banged it  
That he burst his rubber ball, "Boo!" cried **Bobby**  
Bad luck ball, Bad luck **Bobby**, bad luck ball  
Now to drown his many troubles  
**Bobby Bippy**'s blowing bubbles.

/B**…**y/

匹配B开头后面紧跟三个任意字符，最后紧接着一个y的字符串。在这个例子中，Bobby和Bippy都会被匹配。

匹配0个或多个前一字符 \*

Here is **a** tongue twister:

Bobby Bippy bought **a** b**a**t.

Bobby Bippy bought **a** b**all**.  
With his b**a**t Bob b**a**nged the b**all**  
B**a**nged it bump **a**g**a**inst the w**all**  
But so boldly Bobby b**a**nged it  
Th**a**t he burst his rubber b**all**, "Boo!" cried Bobby  
B**a**d luck b**all**, B**a**d luck Bobby, b**a**d luck b**alll**  
Now to drown his m**a**ny troubles  
Bobby Bippy's blowing bubbles.

/ al\*/

这里的星号（\*）匹配0个或多个在它前面的那个字符。前面曾提到过，正则表达式里的\*和shell里的\*作用是截然不同的。在shell里\*表示任意个数的任意字符，而在正则表达式里，\*只代表任意个数（包括0个）的前一字符，\*可以看作和它前面那个字符是粘连在一起的，\*只限制它前面那一个字符。这个正则表达式中的\*匹配单独一个或多个连续的l，甚至也匹配一个l也没有的模式，所以，单个字符a也会被匹配。

**3.4            匹配多个字符**

**匹配一组字符里的任意字符 [ ]**

Here is a tongue twister:

Bobby Bippy bought a bat.

Bobby Bippy bought a **ball**.  
With his bat Bob banged the **ball**  
Banged it bump against the **wall**  
But so boldly Bobby banged it  
That he burst his rubber **ball**, "Boo!" cried Bobby  
Bad luck **ball**, Bad luck Bobby, bad luck **ball**l  
Now to drown his many troubles  
Bobby Bippy's blowing bubbles.

/[bw]all/

方括号匹配一组字符中的一个，这个正则表达式查找的是第一个字母是b或w，后面紧跟着all的字符串，因此在这个例子中，wall和ball都会被匹配。

**匹配指定范围内的字符 [x-y]**

Here is a tongue twister:

Bobby **Bip**py bought a bat.

Bobby **Bip**py bought a ball.  
With his bat Bob banged the ball  
Banged it bump against the wall  
But so boldly Bobby banged it  
That he burst his rubber ball, "Boo!" cried Bobby  
Bad luck ball, Bad luck Bobby, bad luck ball  
Now to drown his many troubles  
Bobby **Bip**py's blowing bubbles.

/B[a-z]p/

方括号里的短线（-）匹配某一范围内的一个字符，这个正则表达式将查找第一个字母是B，第二个字母是ASCII码介于a到z的字符（小写字母），第三个字母是p的字符串。

**匹配不在指定范围内的字符 [^ ]**

Here is a tongue twister:

Bobby Bippy bought a bat.

Bobby Bippy bought a b**all.**

With his bat Bob banged the ball  
Banged it bump against the wall  
But so boldly Bobby banged it  
That he burst his rubber b**all,** "Boo!" cried Bobby  
Bad luck b**all,** Bad luck Bobby, bad luck ball  
Now to drown his many troubles  
Bobby Bippy's blowing bubbles.

/all[^A-Z0-9]/

方括号内的脱字符^是一个否定字符，这个正则表达式查找的是后面带一个特殊字符的all，这个特殊字符既不是小写字母又不是大写字母，也不是0到9的数字，比如它可以是一个标点符号或空格。

**根据字符x出现的次数匹配 x/{m/}  x/{m,/}  x/{m, n/}**

比如这个正则表达式：/Go/{2,5/}gle/将匹配G后面至少出现2个，最多有不超过5个o的模式。Google，Goooogle会被匹配，而Gogle和Goooooogle则不会被匹配。

**3.5            转义字符**

如果要匹配的字符串中含有正则表达式的原字符，需要用斜线将其转义，就像c语言里打印单引号 ’ 要写成 /’ 一样。这里有个例子：我们想要查找字符串google.com，要查找的字符串里含有正则表达式的原字符“**.**”，因此这个正则表达式要写成 /google**/.**com/，如果不用 / 转义，找到的将是google后面跟一个任意的字符，然后跟一个com的字符串。这显然不一定是我们要找的。

**3.6            字符标签**

例如在下面一段文字里：

Occurence and happening are the most general. I mean, the words occurence and happening are most generally used.

在这段文字里有两个拼错的单词，Occurence和occurence，（其实应该是occurrence），我们可以在vi中用下面的表达式将其修改：

:1,$s/**/(**[Oo]ccur**/)**ence/**/1**rence/

我们且不管这个vi命令的用法（其实它是一个替换命令，我们在后面介绍sed时还将提到）我们先拿出这个语句中的两个表达式：

/**/(**[Oo]ccur**/)**ence/

**/1**rence

其中前一个是一个正则表达式。这个命令用后面的表达式内容替换前面的正则表达式匹配到的内容。vi编辑器将查找单词Occurence和occurence，如果找到，就把圆括号中的内容加上标签（Occur或occur被加上标签），因为这是第一个被标记的模式，所以被称为标签1。这个模式被保存在称为寄存器1的内存寄存器中。在第二个正则表达式中用/1引用寄存器1中的内容，/1被替换为寄存器中的内容，后面紧跟一个rence，于是，拼错的Occurence和occurence被改正为正确的Occurrence和occurrence。

**3.7            原字符组合使用的例子**

例1：//<Bob.\*all/>/

Here is a tongue twister:

Bobby Bippy bought a bat.

**Bobby Bippy bought a ball**.  
With his bat **Bob** **banged the ball**  
Banged it bump against the wall  
But so boldly Bobby banged it  
That he burst his rubber ball, "Boo!" cried Bobby  
Bad luck ball, Bad luck **Bobby, bad luck ball**  
Now to drown his many troubles  
Bobby Bippy's blowing bubbles.

//<Bob.\*all/>/

匹配以Bob开头，后面跟任意个数的任意字符，然后以all结尾的字符串。这里再次重复， \* 在Shell里表示任意个数的任意字符，而在正则表达式里表示任意个数的**前一**字符。与 **.**配合使用表示任意个数（包括零个）的任意字符。实际上，\* 也可以表示重复零次或任意次它前面的一组字符，我们称这一组（有时也可能是一个）字符为“原子”。当原子包括多个字符时，这多个字符要用圆括号括起来，并且需要将圆括号转义；当原子只含一个字符时，可以不用圆括号。在这个例子里，**.**表示一个任意字符，紧跟着一个\*表示重复0次或任意次前面的那个任意字符。而下面的例子

//(sup/)\*info/

则表示匹配在字符串info前有0个或多个sup的字符串，因此 supinfo, info, supsupinfo都会被匹配。

例2：/B[a-z][bp]\*y$/

Here is a tongue twister:

Bobby Bippy bought a bat.

Bobby Bippy bought a ball.  
With his bat Bob banged the ball  
Banged it bump against the wall  
But so boldly Bobby banged it  
That he burst his rubber ball, "Boo!" cried **Bobby**  
Bad luck ball, Bad luck Bobby, bad luck ball  
Now to drown his many troubles  
Bobby Bippy's blowing bubbles.

/B[a-z][bp]\*y$/

这个正则表达式匹配这样的字符串：开头字符是B，第二个字母是一个小写字母，后面紧跟0个或多个重复的b或p，最后跟一个y，并且这个字符串位于行的末尾。