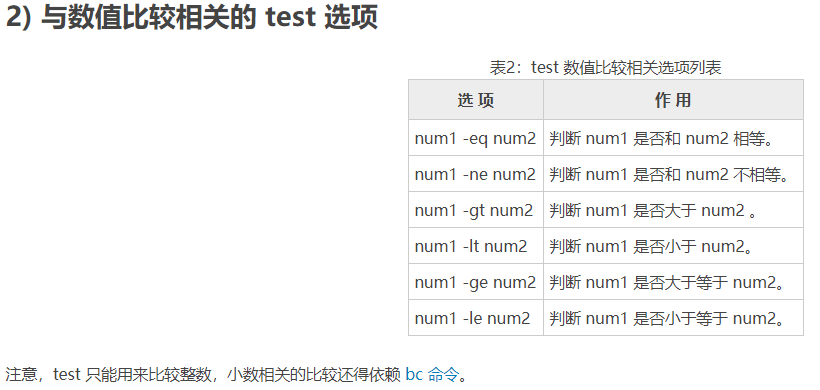


===============================================================================

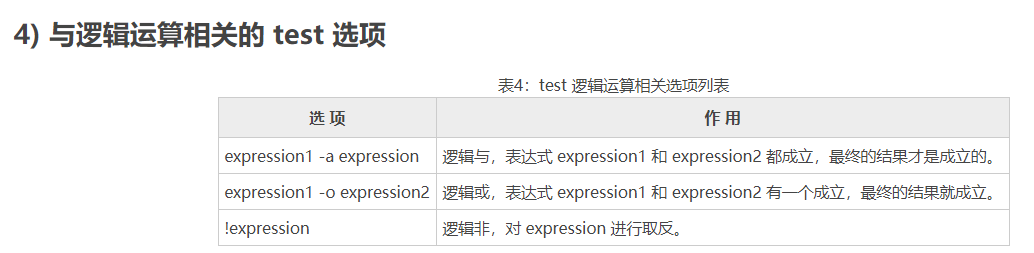
## Test

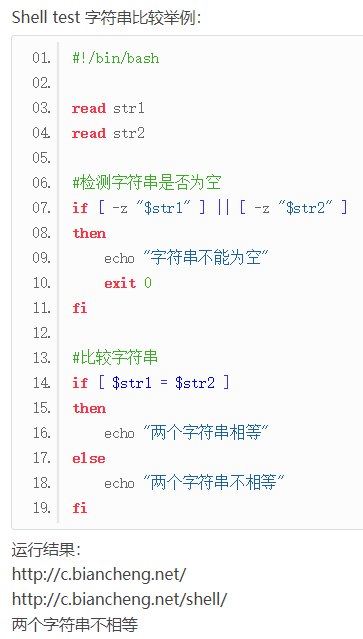
****

****

****

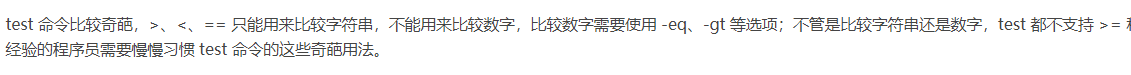
****

****

****

****

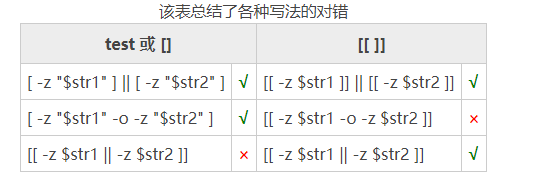
**小Q截图-20190512020444**

****

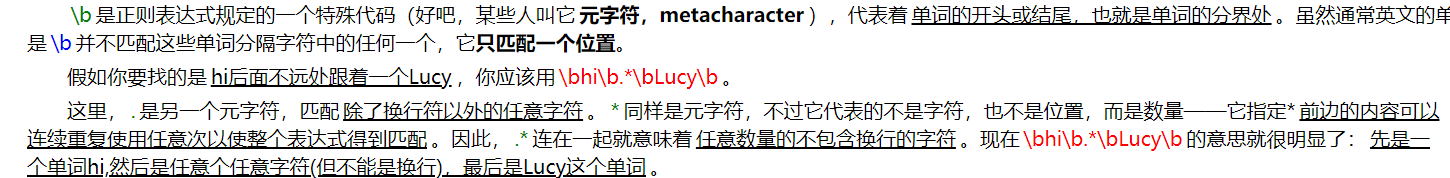
**小Q截图-20190512131907**

****

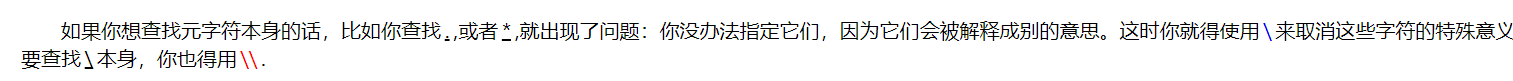
**=====================================================================**

****

**如果要精确地查找hi这个单词的话，我们应该使用\bhi\b。**

****

**小Q截图-20190512132534**

****

## 零宽断言

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，也就是说它们像\b,^,$那样用于指定一个位置，这个位置应该满足一定的条件(即断言)，因此它们也被称为**零宽断言**。最好还是拿例子来说明吧：

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。

下面这个例子同时使用了这两种断言：(?<=\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)。

## 负向零宽断言

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(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]总要匹配一个字符，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*。**负向零宽断言**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。

**零宽度负预测先行断言**(?!exp)，断言此位置的后面不能匹配表达式exp。例如：\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。

同理，我们可以用(?<!exp),**零宽度负回顾后发断言**来断言此位置的前面不能匹配表达式exp：(?<![a-z])\d{7}匹配前面不是小写字母的七位数字。

请详细分析表达式(?<=<(\w+)>).\*(?=<\/\1>)，这个表达式最能表现零宽断言的真正用途。

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)。

## 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

) # 后缀结束

## 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。以这个表达式为例：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（第一到第三个字符）和ab（第四到第五个字符）。

为什么第一个匹配是aab（第一到第三个字符）而不是ab（第二到第三个字符）？简单地说，因为正则表达式有另一条规则，比懒惰／贪婪规则的优先级更高：最先开始的匹配拥有最高的优先权——The match that begins earliest wins。

| **表5.懒惰限定符** | |
| --- | --- |
| **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 |

## 处理选项

在C#中，你可以使用[Regex(String, RegexOptions)构造函数](http://msdn2.microsoft.com/zh-cn/library/h5845fdz.aspx" \o "MSDN 相关文档)来设置正则表达式的处理选项。如：Regex regex = new Regex(@"\ba\w{6}\b", RegexOptions.IgnoreCase);

上面介绍了几个选项如忽略大小写，处理多行等，这些选项能用来改变处理正则表达式的方式。下面是.Net中常用的正则表达式选项：

| **表6.常用的处理选项** | |
| --- | --- |
| **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。 |
| ExplicitCapture(显式捕获) | 仅捕获已被显式命名的组。 |

一个经常被问到的问题是：是不是只能同时使用多行模式和单行模式中的一种？答案是：不是。这两个选项之间没有任何关系，除了它们的名字比较相似（以至于让人感到疑惑）以外。

## 平衡组/递归匹配

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 <bbb> <bbb> 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

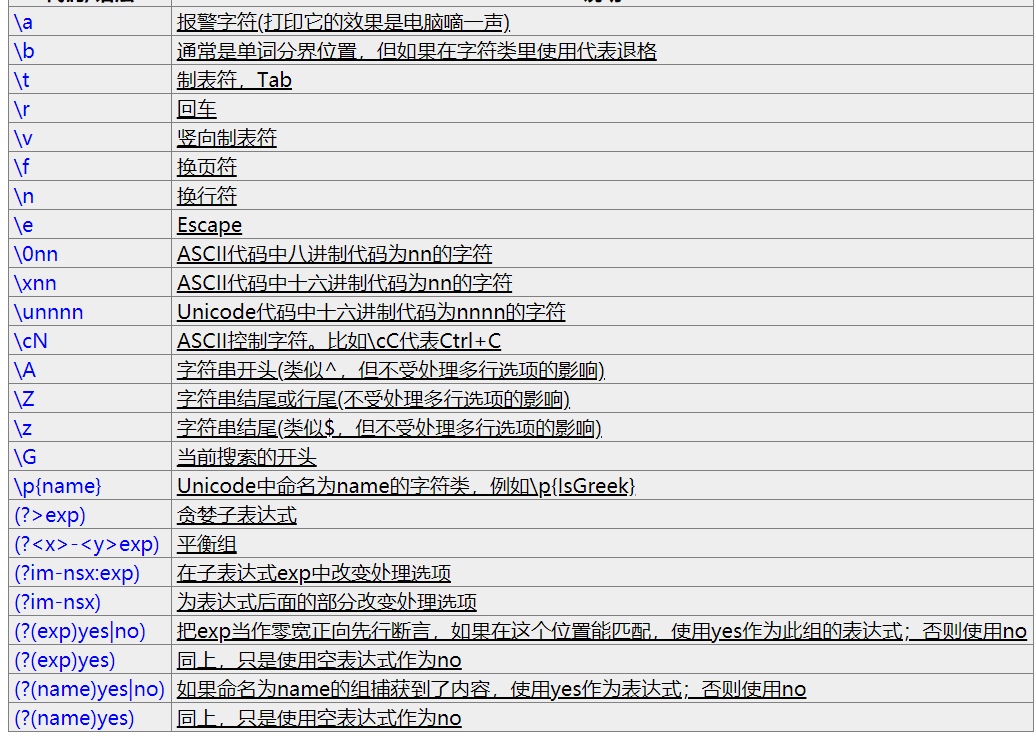
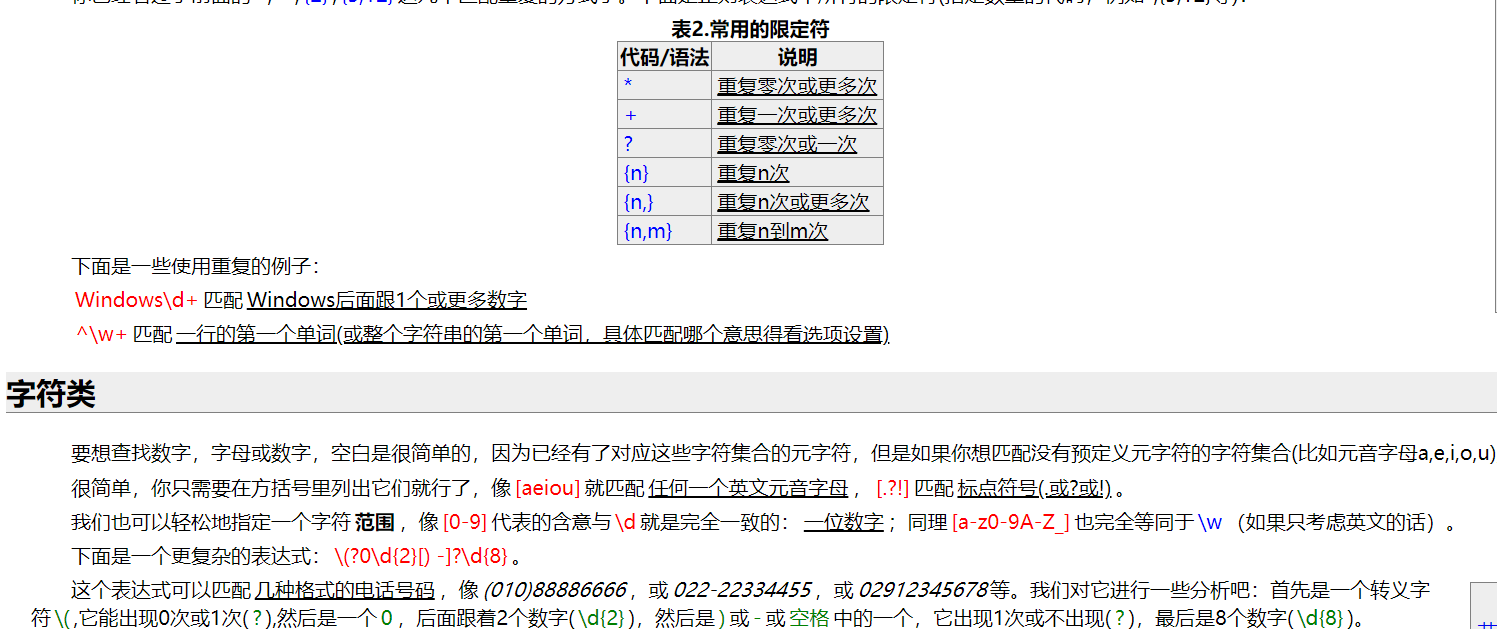
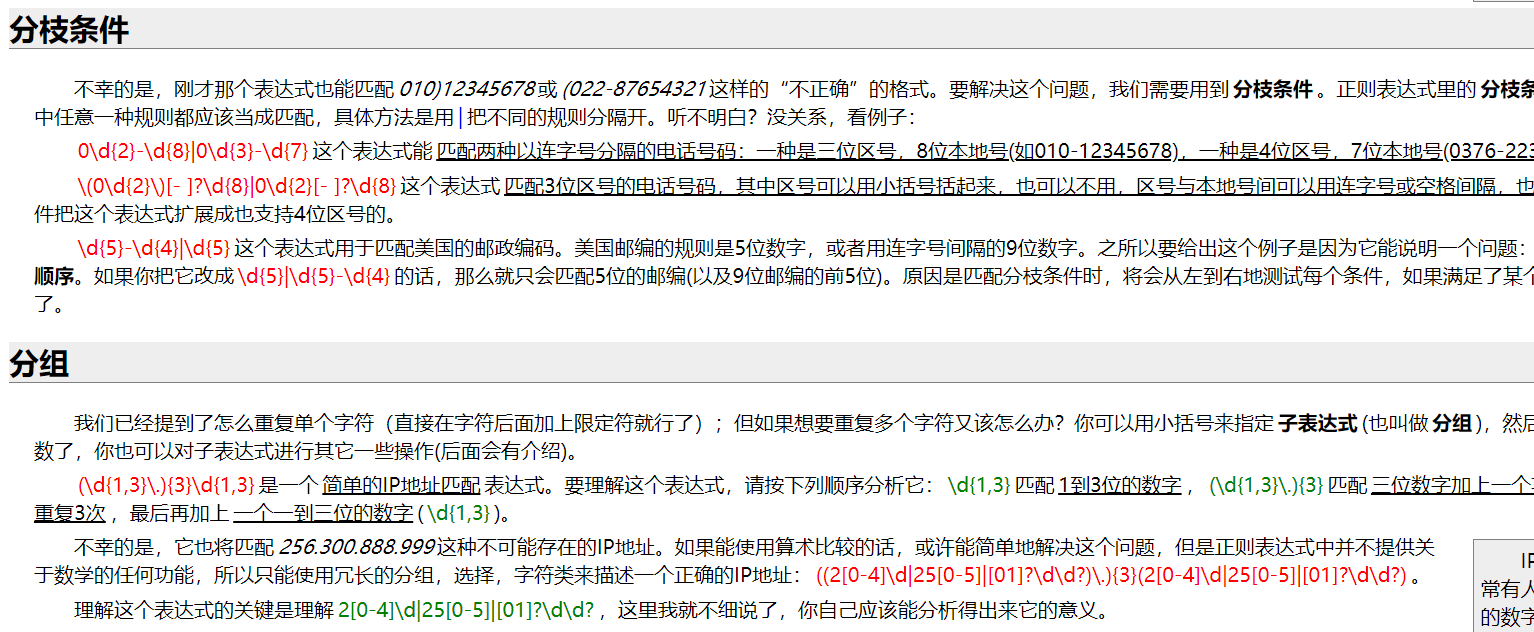
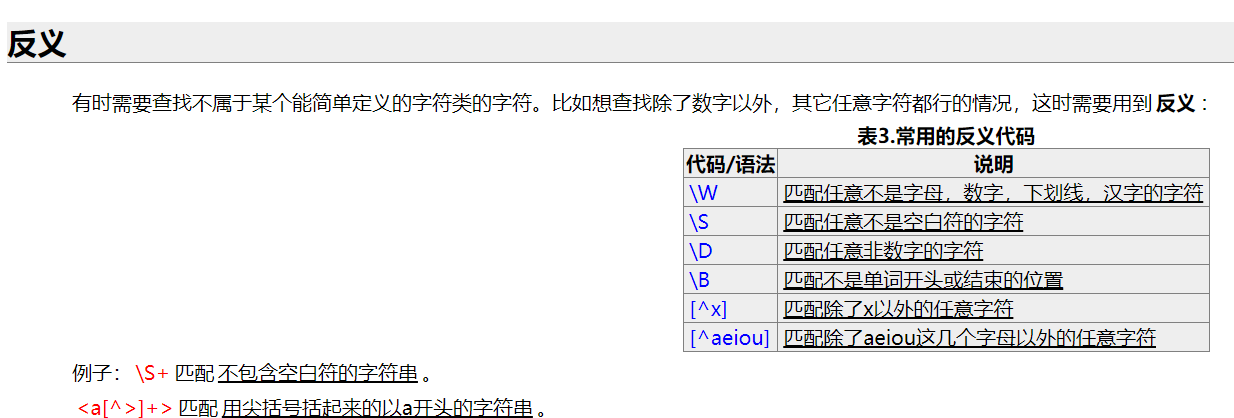
)+

)\*

(?(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<div>标签：<div[^>]\*>[^<>]\*(((?'Open'<div[^>]\*>)[^<>]\*)+((?'-Open'</div>)[^<>]\*)+)\*(?(Open)(?!))</div>.

****

**===============================================================================**

# **Shell case in语句详解**

当分支较多，并且判断条件比较简单时，使用 case in 语句就比较方便了。  
  
如下所示。

1. #!/bin/bash
2. **printf** "Input integer number: "
3. **read** num
4. **case** $num **in**
5. 1)
6. echo "Monday"
7. ;;
8. 2)
9. echo "Tuesday"
10. ;;
11. 3)
12. echo "Wednesday"
13. ;;
14. 4)
15. echo "Thursday"
16. ;;
17. 5)
18. echo "Friday"
19. ;;
20. 6)
21. echo "Saturday"
22. ;;
23. 7)
24. echo "Sunday"
25. ;;
26. \*)
27. echo "error"
28. **esac**

运行结果：  
Input integer number:3↙  
Wednesday  
  
看了这个例子，相信大家对 case in 语句有了一个大体上的认识，那么，接下来我们就正式开始讲解 case in 的用法，它的基本格式如下：

case expression in  
    pattern1) 匹配模式  
        statement1  
        ;;  
    pattern2)  
        statement2  
        ;;  
    pattern3)  
        statement3  
        ;;  
    ……  
    \*)  
        statementn  
esac

case、int 和 esac 都是 Shell 关键字，expression 表示表达式，pattern 表示匹配模式。

* expression 既可以是一个变量、一个数字、一个字符串，还可以是一个数学计算表达式，或者是命令的执行结果，只要能够得到 expression 的值就可以。
* pattern 可以是一个数字、一个字符串，甚至是一个简单的正则表达式。  
  case 会将 expression  的值与 pattern1、pattern2、pattern3 逐个进行匹配：
* 如果 expression 和某个模式（比如 pattern2）匹配成功，就会执行这模式（比如 pattern2）后面对应的所有语句（该语句可以有一条，也可以有多条），直到遇见双分号;;才停止；然后整个 case 语句就执行完了，程序会跳出整个 case 语句，执行 esac 后面的其它语句。
* 如果 expression 没有匹配到任何一个模式，那么就执行\*)后面的语句（\*表示其它所有值），直到遇见双分号;;或者esac才结束。\*)相当于多个 if 分支语句中最后的 else 部分。

如果你有[C语言](http://c.biancheng.net/c/" \t "http://c.biancheng.net/view/_blank)、[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank)、[Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 等编程经验，这里的;;和\*)就相当于其它编程语言中的 break 和 default。

对\*)的几点说明：

* Shell case in 语句中的\*)用来“托底”，万一 expression 没有匹配到任何一个模式，\*)部分可以做一些“善后”工作，或者给用户一些提示。
* 可以没有\*)部分。如果 expression 没有匹配到任何一个模式，那么就不执行任何操作。

除最后一个分支外（这个分支可以是普通分支，也可以是\*)分支），其它的每个分支都必须以;;结尾，;;代表一个分支的结束，不写的话会有语法错误。最后一个分支可以写;;，也可以不写，因为无论如何，执行到 esac 都会结束整个 case in 语句。  
  
上面的代码是 case in 最常见的用法，即 expression 部分是一个变量，pattern 部分是一个数字或者表达式。

## case in 和正则表达式

case in 的 pattern 部分支持简单的正则表达式，具体来说，可以使用以下几种格式：

|  |  |
| --- | --- |
| **格式** | **说明** |
| \* | 表示任意字符串。 |
| [abc] | 表示 a、b、c 三个字符中的任意一个。比如，[15ZH] 表示 1、5、Z、H 四个字符中的任意一个。 |
| [m-n] | 表示从 m 到 n 的任意一个字符。比如，[0-9] 表示任意一个数字，[0-9a-zA-Z] 表示字母或数字。 |
| | | 表示多重选择，类似逻辑运算中的或运算。比如，abc | xyz 表示匹配字符串 "abc" 或者 "xyz"。 |

如果不加以说明，Shell 的值都是字符串，expression 和 pattern 也是按照字符串的方式来匹配的；本节第一段代码看起来是判断数字是否相等，其实是判断字符串是否相等。  
  
最后一个分支\*)并不是什么语法规定，它只是一个正则表达式，\*表示任意字符串，所以不管 expression 的值是什么，\*)总能匹配成功。  
  
下面的例子演示了如何在 case in 中使用正则表达式：

[纯文本复制](http://c.biancheng.net/view/2767.html)

1. #!/bin/bash
2. **printf** "Input a character: "
3. **read** -n 1 char
4. **case** $char **in**
5. [a-zA-Z])
6. **printf** "\nletter\n"
7. ;;
8. [0-9])
9. **printf** "\nDigit\n"
10. ;;
11. [0-9])
12. **printf** "\nDigit\n"
13. ;;
14. [,.?!])
15. **printf** "\nPunctuation\n"
16. ;;
17. \*)
18. **printf** "\nerror\n"
19. **esac**

运行结果1：  
Input integer number: S  
letter  
  
运行结果2：  
Input integer number: ,  
Punctuation

================================================================================================================

# **Shell while循环详解**

while 循环是 [Shell 脚本](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank)中最简单的一种循环，当条件满足时，while 重复地执行一组语句，当条件不满足时，就退出 while 循环。  
  
[Shell](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank) while 循环的用法如下：

while condition  
do  
    statements  
done

condition表示判断条件，statements表示要执行的语句（可以只有一条，也可以有多条），do和done都是 Shell 中的关键字。  
  
while 循环的执行流程为：

* 先对 condition 进行判断，如果该条件成立，就进入循环，执行 while 循环体中的语句，也就是 do 和 done 之间的语句。这样就完成了一次循环。
* 每一次执行到 done 的时候都会重新判断 condition 是否成立，如果成立，就进入下一次循环，继续执行 do 和 done 之间的语句，如果不成立，就结束整个 while 循环，执行 done 后面的其它 Shell 代码。
* 如果一开始 condition 就不成立，那么程序就不会进入循环体，do 和 done 之间的语句就没有执行的机会。  
  注意，在 while 循环体中必须有相应的语句使得 condition 越来越趋近于“不成立”，只有这样才能最终退出循环，否则 while 就成了死循环，会一直执行下去，永无休止。  
    
  while 语句和 if else 语句中的 condition 用法都是一样的，你可以使用 test 或 [] 命令，也可以使用 (()) 或 [[]]，

## while 循环举例

【实例1】计算从 1 加到 100 的和。

1. #!/bin/bash
2. i=1
3. sum=0
4. **while** ((i <= 100))
5. **do**
6. ((sum += i))
7. ((i++))
8. **done**
9. echo "The sum is: $sum"

运行结果：  
The sum is: 5050  
  
在 while 循环中，只要判断条件成立，循环就会执行。对于这段代码，只要变量 i 的值小于等于 100，循环就会继续。每次循环给变量 sum 加上变量 i 的值，然后再给变量 i 加 1，直到变量 i 的值大于 100，循环才会停止。  
  
i++语句使得 i 的值逐步增大，让判断条件越来越趋近于“不成立”，最终退出循环。  
  
对上面的例子进行改进，计算从 m 加到 n 的值。

1. #!/bin/bash
2. **read** m
3. **read** n
4. sum=0
5. **while** ((m <= n))
6. **do**
7. ((sum += m))
8. ((m++))
9. **done**
10. echo "The sum is: $sum"

运行结果：  
1↙  
100↙  
The sum is: 5050  
  
【实例2】实现一个简单的加法计算器，用户每行输入一个数字，计算所有数字的和。

1. #!/bin/bash
2. sum=0
3. echo "请输入您要计算的数字，按 Ctrl+D 组合键结束读取"
4. **while** **read** n
5. **do**
6. ((sum += n))
7. **done**
8. echo "The sum is: $sum"

运行结果：  
12↙  
33↙  
454↙  
6767↙  
1↙  
2↙  
The sum is: 7269  
  
在终端中读取数据，可以等价为在文件中读取数据，按下 Ctrl+D 组合键表示读取到文件流的末尾，此时 read 就会读取失败，得到一个非 0 值的退出状态，从而导致判断条件不成立，结束循环。

**===============================================================================**

# **Shell until循环用法详解**

unti 循环和 while 循环恰好相反，当判断条件不成立时才进行循环，一旦判断条件成立，就终止循环。  
  
until 的使用场景很少，一般使用 while 即可。  
  
[Shell](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank) until 循环的用法如下：

until condition  
do  
    statements  
done

condition表示判断条件，statements表示要执行的语句（可以只有一条，也可以有多条），do和done都是 Shell 中的关键字。  
  
until 循环的执行流程为：

* 先对 condition 进行判断，如果该条件不成立，就进入循环，执行 until 循环体中的语句（do 和 done 之间的语句），这样就完成了一次循环。
* 每一次执行到 done 的时候都会重新判断 condition 是否成立，如果不成立，就进入下一次循环，继续执行循环体中的语句，如果成立，就结束整个 until 循环，执行 done 后面的其它 Shell 代码。
* 如果一开始 condition 就成立，那么程序就不会进入循环体，do 和 done 之间的语句就没有执行的机会。

注意，在 until 循环体中必须有相应的语句使得 condition 越来越趋近于“成立”，只有这样才能最终退出循环，否则 until 就成了死循环，会一直执行下去，永无休止。  
  
上节《[Shell while循环](http://c.biancheng.net/view/1006.html" \t "http://c.biancheng.net/view/_blank)》演示了如何求从 1 加到 100 的值，这节我们改用 until 循环，请看下面的代码：

1. #!/bin/bash
2. i=1
3. sum=0
4. **until** ((i > 100))
5. **do**
6. ((sum += i))
7. ((i++))
8. **done**
9. echo "The sum is: $sum"

运行结果：  
The sum is: 5050  
  
在 while 循环中，判断条件为((i<=100))，这里将判断条件改为((i>100))，两者恰好相反，请读者注意区分。

# **Shell for循环和for int循环详解**

## [C语言](http://c.biancheng.net/c/" \t "http://c.biancheng.net/view/_blank)风格的 for 循环

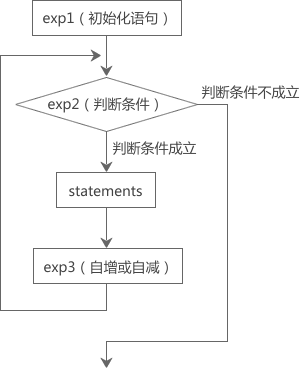
C语言风格的 for 循环的用法如下：

for((exp1; exp2; exp3))  
do  
    statements  
done

几点说明：

* exp1、exp2、exp3 是三个表达式，其中 exp2 是判断条件，for 循环根据 exp2 的结果来决定是否继续下一次循环；
* statements 是循环体语句，可以有一条，也可以有多条；
* do 和 done 是 Shell 中的关键字。

它的运行过程为：  
1) 先执行 exp1。  
  
2) 再执行 exp2，如果它的判断结果是成立的，则执行循环体中的语句，否则结束整个 for 循环。  
  
3) 执行完循环体后再执行 exp3。  
  
4) 重复执行步骤 2) 和 3)，直到 exp2 的判断结果不成立，就结束循环。  
  
上面的步骤中，2) 和 3) 合并在一起算作一次循环，会重复执行，for 语句的主要作用就是不断执行步骤 2) 和 3)。  
  
exp1 仅在第一次循环时执行，以后都不会再执行，可以认为这是一个初始化语句。exp2 一般是一个关系表达式，决定了是否还要继续下次循环，称为“循环条件”。exp3 很多情况下是一个带有自增或自减运算的表达式，以使循环条件逐渐变得“不成立”。  
  
for 循环的执行过程可用下图表示：



下面我们给出一个实际的例子，计算从 1 加到 100 的和。

1. #!/bin/bash
2. sum=0
3. **for** ((i=1; i<=100; i++))
4. **do**
5. ((sum += i))
6. **done**
7. echo "The sum is: $sum"

运行结果：  
The sum is: 5050  
  
代码分析：  
1) 执行到 for 语句时，先给变量 i 赋值为 1，然后判断 i<=100 是否成立；因为此时 i=1，所以 i<=100 成立。接下来会执行循环体中的语句，等循环体执行结束后（sum 的值为1），再计算 i++。  
  
2) 第二次循环时，i 的值为2，i<=100 成立，继续执行循环体。循环体执行结束后（sum的值为3），再计算 i++。  
  
3) 重复执行步骤 2)，直到第 101 次循环，此时 i 的值为 101，i<=100 不再成立，所以结束循环。  
  
由此我们可以总结出 for 循环的一般形式为：

for(( 初始化语句; 判断条件; 自增或自减 ))  
do  
    statements  
done

#### for 循环中的三个表达式

for 循环中的 exp1（初始化语句）、exp2（判断条件）和 exp3（自增或自减）都是可选项，都可以省略（但分号;必须保留）。  
  
1) 修改“从 1 加到 100 的和”的代码，省略 exp1：

1. #!/bin/bash
2. sum=0
3. i=1
4. **for** ((; i<=100; i++))
5. **do**
6. ((sum += i))
7. **done**
8. echo "The sum is: $sum"

可以看到，将i=1移到了 for 循环的外面。  
  
2) 省略 exp2，就没有了判断条件，如果不作其他处理就会成为死循环，我们可以在循环体内部使用 break 关键字强制结束循环：

1. #!/bin/bash
2. sum=0
3. **for** ((i=1; ; i++))
4. **do**
5. **if**(( i>100 )); **then**
6. **break**
7. **fi**
8. ((sum += i))
9. **done**
10. echo "The sum is: $sum"

break 是 Shell 中的关键字，专门用来结束循环，后续章节还会深入讲解。  
  
3) 省略了 exp3，就不会修改 exp2 中的变量，这时可在循环体中加入修改变量的语句。例如：

1. #!/bin/bash
2. sum=0
3. **for** ((i=1; i<=100; ))
4. **do**
5. ((sum += i))
6. ((i++))
7. **done**
8. echo "The sum is: $sum"

4) 最后给大家看一个更加极端的例子，同时省略三个表达式：

1. #!/bin/bash
2. sum=0
3. i=0
4. **for** (( ; ; ))
5. **do**
6. **if**(( i>100 )); **then**
7. **break**
8. **fi**
9. ((sum += i))
10. ((i++))
11. **done**
12. echo "The sum is: $sum"

这种写法并没有什么实际意义，仅仅是为了给大家做演示。

## [Python](http://c.biancheng.net/python/" \t "http://c.biancheng.net/view/_blank) 风格的 for in 循环

Python 风格的 for in 循环的用法如下：

for variable in value\_list  
do  
    statements  
done

variable 表示变量，value\_list 表示取值列表，in 是 Shell 中的关键字。

in value\_list 部分可以省略，省略后的效果相当于 in $@，本文末尾的「[value\_list 使用特殊变量](http://c.biancheng.net/view/2804.html" \l "use_special_var)」将会详细讲解。

每次循环都会从 value\_list 中取出一个值赋给变量 variable，然后进入循环体（do 和 done 之间的部分），执行循环体中的 statements。直到取完 value\_list 中的所有值，循环就结束了。  
  
Shell for in 循环举例：

1. #!/bin/bash
2. sum=0
3. **for** n **in** 1 2 3 4 5 6
4. **do**
5. echo $n
6. ((sum+=n))
7. **done**
8. echo "The sum is "$sum

运行结果：  
1  
2  
3  
4  
5  
6  
The sum is 21

### 对 value\_list 的说明

取值列表 value\_list 的形式有多种，你可以直接给出具体的值，也可以给出一个范围，还可以使用命令产生的结果，甚至使用通配符，下面我们一一讲解。

#### 1) 直接给出具体的值

可以在 in 关键字后面直接给出具体的值，多个值之间以空格分隔，比如1 2 3 4 5、"abc" "390" "tom"等。  
  
上面的代码中用一组数字作为取值列表，下面我们再演示一下用一组字符串作为取值列表：

1. #!/bin/bash
2. **for** str **in** "C语言中文网" "http://c.biancheng.net/" "成立7年了" "日IP数万"
3. **do**
4. echo $str
5. **done**

运行结果：  
C语言中文网  
http://c.biancheng.net/  
成立7年了  
日IP数万

#### 2) 给出一个取值范围

给出一个取值范围的具体格式为：

{start..end}

start 表示起始值，end 表示终止值；注意中间用两个点号相连，而不是三个点号。根据笔者的实测，这种形式只支持数字和字母。  
  
例如，计算从 1 加到 100 的和：

1. #!/bin/bash
2. sum=0
3. **for** n **in** {1..100}
4. **do**
5. ((sum+=n))
6. **done**
7. echo $sum

运行结果：  
5050  
  
再如，输出从 A 到 z 之间的所有字符：

1. #!/bin/bash
2. **for** c **in** {A..z}
3. **do**
4. **printf** "%c" $c
5. **done**

输出结果：  
ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^\_`abcdefghijklmnopqrstuvwxyz  
  
可以发现，Shell 是根据 [ASCII](http://c.biancheng.net/c/ascii/" \t "http://c.biancheng.net/view/_blank) 码表来输出的。

#### 3) 使用命令的执行结果

使用反引号``或者$()都可以取得命令的执行结果，我们在《[Shell变量](http://c.biancheng.net/view/743.html" \t "http://c.biancheng.net/view/_blank)》一节中已经进行了详细讲解，并对比了两者的优缺点。本节我们使用$()这种形式，因为它不容易产生混淆。  
  
例如，计算从 1 到 100 之间所有偶数的和：

1. #!/bin/bash
2. sum=0
3. **for** n **in** $(seq 2 2 100)
4. **do**
5. ((sum+=n))
6. **done**
7. echo $sum

运行结果：  
2550  
  
seq 是一个 [Linux 命令](http://c.biancheng.net/linux/" \t "http://c.biancheng.net/view/_blank)，用来产生某个范围内的整数，并且可以设置步长，不了解的读者请自行百度。seq 2 2 100表示从 2 开始，每次增加 2，到 100 结束。  
  
再如，列出当前目录下的所有 Shell 脚本文件：

1. #!/bin/bash
2. **for** filename **in** $(ls \*.sh)
3. **do**
4. echo $filename
5. **done**

运行结果：  
demo.sh  
test.sh  
abc.sh  
  
ls 是一个 [Linux](http://c.biancheng.net/linux_tutorial/" \t "http://c.biancheng.net/view/_blank) 命令，用来列出当前目录下的所有文件，\*.sh表示匹配后缀为.sh的文件，也就是 Shell 脚本文件。

#### 4) 使用 Shell 通配符

Shell 通配符可以认为是一种精简化的正则表达式，通常用来匹配目录或者文件，而不是文本，不了解的读者请猛击《[Linux Shell 通配符（glob 模式）](https://www.linuxidc.com/Linux/2016-08/134192.htm" \t "http://c.biancheng.net/view/_blank)》。  
  
有了 Shell 通配符，不使用 ls 命令也能显示当前目录下的所有脚本文件，请看下面的代码：

1. #!/bin/bash
2. **for** filename **in** \*.sh
3. **do**
4. echo $filename
5. **done**

运行结果：  
demo.sh  
test.sh  
abc.sh

#### 5) 使用特殊变量

Shell 中有多个特殊的变量，例如 $#、$\*、$@、$?、$$ 等（不了解的读者请猛击《[Shell特殊变量](http://c.biancheng.net/view/806.html" \t "http://c.biancheng.net/view/_blank)》），在 value\_list 中就可以使用它们。

1. #!/bin/bash
2. function func(){
3. **for** str **in** $@
4. **do**
5. echo $str
6. **done**
7. }
8. func [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) [Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) Python [C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank)

运行结果：  
C++  
Java  
Python  
C#  
  
其实，我们也可以省略 value\_list，省略后的效果和使用$@一样。请看下面的演示：

[纯文本复制](http://c.biancheng.net/view/2804.html)

1. #!/bin/bash
2. function func(){
3. **for** str
4. **do**
5. echo $str
6. **done**
7. }
8. func C++ Java Python C#

运行结果：  
C++  
Java  
Python  
C#

# **Shell select in循环详解**

select in 循环用来增强交互性，它可以显示出带编号的菜单，用户输入不同的编号就可以选择不同的菜单，并执行不同的功能。  
  
select in 是 [Shell](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank) 独有的一种循环，非常适合终端（Terminal）这样的交互场景，[C语言](http://c.biancheng.net/c/" \t "http://c.biancheng.net/view/_blank)、[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank)、[Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank)、[Python](http://c.biancheng.net/python/" \t "http://c.biancheng.net/view/_blank)、[C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank) 等其它编程语言中是没有的。  
  
Shell select in 循环的用法如下：

select variable in value\_list  
do  
    statements  
done

variable 表示变量，value\_list 表示取值列表，in 是 Shell 中的关键字。你看，select in 和 [for in](http://c.biancheng.net/view/2804.html" \t "http://c.biancheng.net/view/_blank) 的语法是多么地相似。  
  
我们先来看一个 select in 循环的例子：

1. #!/bin/bash
2. echo "What is your favourite OS?"
3. **select** name **in** "[Linux](http://c.biancheng.net/linux_tutorial/" \t "http://c.biancheng.net/view/_blank)" "Windows" "Mac OS" "UNIX" "[Android](http://c.biancheng.net/android/" \t "http://c.biancheng.net/view/_blank)"
4. **do**
5. echo $name
6. **done**
7. echo "You have selected $name"

运行结果：

What is your favourite OS?  
1) Linux  
2) Windows  
3) Mac OS  
4) UNIX  
5) Android  
#? 4↙  
You have selected UNIX  
#? 1↙  
You have selected Linux  
#? 9↙  
You have selected  
#? 2↙  
You have selected Windows  
#?^D

#?用来提示用户输入菜单编号；^D表示按下 Ctrl+D 组合键，它的作用是结束 select in 循环。  
  
运行到 select 语句后，取值列表 value\_list 中的内容会以菜单的形式显示出来，用户输入菜单编号，就表示选中了某个值，这个值就会赋给变量 variable，然后再执行循环体中的 statements（do 和 done 之间的部分）。  
  
每次循环时 select 都会要求用户输入菜单编号，并使用环境变量 PS3 的值作为提示符，PS3 的默认值为#?，修改 PS3 的值就可以修改提示符。  
  
如果用户输入的菜单编号不在范围之内，例如上面我们输入的 9，那么就会给 variable 赋一个空值；如果用户输入一个空值（什么也不输入，直接回车），会重新显示一遍菜单。  
  
注意，select 是无限循环（死循环），输入空值，或者输入的值无效，都不会结束循环，只有遇到 break 语句，或者按下 Ctrl+D 组合键才能结束循环。

#### 完整实例

select in 通常和 [case in](http://c.biancheng.net/view/2767.html" \t "http://c.biancheng.net/view/_blank) 一起使用，在用户输入不同的编号时可以做出不同的反应。  
  
修改上面的代码，加入 case in 语句：

1. #!/bin/bash
2. echo "What is your favourite OS?"
3. **select** name **in** "Linux" "Windows" "Mac OS" "UNIX" "Android"
4. **do**
5. **case** $name **in**
6. "Linux")
7. echo "Linux是一个类UNIX操作系统，它开源免费，运行在各种服务器设备和嵌入式设备。"
8. **break**
9. ;;
10. "Windows")
11. echo "Windows是微软开发的个人电脑操作系统，它是闭源收费的。"
12. **break**
13. ;;
14. "Mac OS")
15. echo "Mac OS是苹果公司基于UNIX开发的一款图形界面操作系统，只能运行与苹果提供的硬件之上。"
16. **break**
17. ;;
18. "UNIX")
19. echo "UNIX是操作系统的开山鼻祖，现在已经逐渐退出历史舞台，只应用在特殊场合。"
20. **break**
21. ;;
22. "Android")
23. echo "Android是由Google开发的手机操作系统，目前已经占据了70%的市场份额。"
24. **break**
25. ;;
26. \*)
27. echo "输入错误，请重新输入"
28. **esac**
29. **done**

用户只有输入正确的编号才会结束循环，如果输入错误，会要求重新输入。  
  
运行结果1，输入正确选项：

What is your favourite OS?  
1) Linux  
2) Windows  
3) Mac OS  
4) UNIX  
5) Android  
#? 2  
Windows是微软开发的个人电脑操作系统，它是闭源收费的。

运行结果2，输入错误选项：

What is your favourite OS?  
1) Linux  
2) Windows  
3) Mac OS  
4) UNIX  
5) Android  
#? 7  
输入错误，请重新输入  
#? 4  
UNIX是操作系统的开山鼻祖，现在已经逐渐退出历史舞台，只应用在特殊场合。

运行结果3，输入空值：

What is your favourite OS?  
1) Linux  
2) Windows  
3) Mac OS  
4) UNIX  
5) Android  
#?  
1) Linux  
2) Windows  
3) Mac OS  
4) UNIX  
5) Android  
#? 3  
Mac OS是苹果公司基于UNIX开发的一款图形界面操作系统，只能运行与苹果提供的硬件之上。

# **Shell break和continue跳出循环详解**

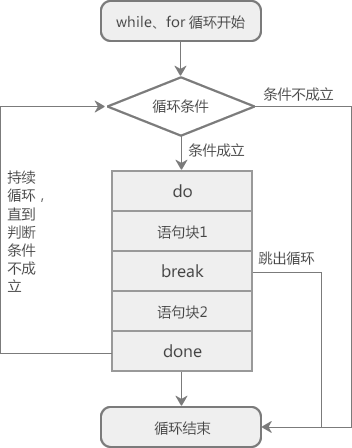
使用 while、until、for、select 循环时，如果想提前结束循环（在不满足结束条件的情况下结束循环），可以使用 break 或者 continue 关键字。  
  
在[C语言](http://c.biancheng.net/c/" \t "http://c.biancheng.net/view/_blank)、[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank)、[C#](http://c.biancheng.net/csharp/" \t "http://c.biancheng.net/view/_blank)、[Python](http://c.biancheng.net/python/" \t "http://c.biancheng.net/view/_blank)、[Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 等大部分编程语言中，break 和 continue 只能跳出当前层次的循环，内层循环中的 break 和 continue 对外层循环不起作用；但是 [Shell](http://c.biancheng.net/shell/" \t "http://c.biancheng.net/view/_blank) 中的 break 和 continue 却能够跳出多层循环，也就是说，内层循环中的 break 和 continue 能够跳出外层循环。  
  
在实际开发中，break 和 continue 一般只用来跳出当前层次的循环，很少有需要跳出多层循环的情况。

## break 关键字

Shell break 关键字的用法为：

break n

n 表示跳出循环的层数，如果省略 n，则表示跳出当前的整个循环。break 关键字通常和 if 语句一起使用，即满足条件时便跳出循环。

  
图1：Shell break关键字原理示意图

【实例1】不断从终端读取用户输入的正数，求它们相加的和：

1. #!/bin/bash
2. sum=0
3. **while** **read** n; **do**
4. **if**((n>0)); **then**
5. ((sum+=n))
6. **else**
7. **break**
8. **fi**
9. **done**
10. echo "sum=$sum"

运行结果：  
10↙  
20↙  
30↙  
0↙  
sum=60  
  
while 循环通过 read 命令的退出状态来判断循环条件是否成立，只有当按下 Ctrl+D 组合键（表示输入结束）时，read n才会判断失败，此时 while 循环终止。  
  
除了按下 Ctrl+D 组合键，你还可以输入一个小于等于零的整数，这样会执行 break 语句来终止循环（跳出循环）。  
  
【实例2】使用 break 跳出双层循环。  
  
如果 break 后面不跟数字的话，表示跳出当前循环，对于有两层嵌套的循环，就得使用两个 break 关键字。例如，输出一个 4\*4 的矩阵：

1. #!/bin/bash
2. i=0
3. **while** ((++i)); **do** #外层循环
4. **if**((i>4)); **then**
5. **break** #跳出外层循环
6. **fi**
7. j=0;
8. **while** ((++j)); **do** #内层循环
9. **if**((j>4)); **then**
10. **break** #跳出内层循环
11. **fi**
12. **printf** "%-4d" $((i\*j))
13. **done**
14. **printf** "\n"
15. **done**

运行结果：

1 2 3 4

2 4 6 8

3 6 9 12

4 8 12 16

当 j>4 成立时，执行第二个 break，跳出内层循环；外层循环依然执行，直到 i>4 成立，跳出外层循环。内层循环共执行了 4 次，外层循环共执行了 1 次。  
  
我们也可以在 break 后面跟一个数字，让它一次性地跳出两层循环，请看下面的代码：

1. #!/bin/bash
2. i=0
3. **while** ((++i)); **do** #外层循环
4. j=0;
5. **while** ((++j)); **do** #内层循环
6. **if**((i>4)); **then**
7. **break** 2 #跳出内外两层循环
8. **fi**
9. **if**((j>4)); **then**
10. **break** #跳出内层循环
11. **fi**
12. **printf** "%-4d" $((i\*j))
13. **done**
14. **printf** "\n"
15. **done**

修改后的代码将所有 break 都移到了内层循环里面。读者需要重点关注break 2这条语句，它使得程序可以一次性跳出两层循环，也就是先跳出内层循环，再跳出外层循环。

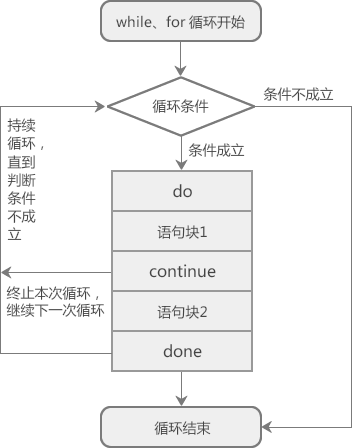
## continue 关键字

Shell continue 关键字的用法为：

continue n

n 表示循环的层数：

* 如果省略 n，则表示 continue 只对当前层次的循环语句有效，遇到 continue 会跳过本次循环，忽略本次循环的剩余代码，直接进入下一次循环。
* 如果带上 n，比如 n 的值为 2，那么 continue 对内层和外层循环语句都有效，不但内层会跳过本次循环，外层也会跳过本次循环，其效果相当于内层循环和外层循环**同时**执行了不带 n 的 continue。这么说可能有点难以理解，稍后我们通过代码来演示。  
  continue 关键字也通常和 if 语句一起使用，即满足条件时便跳出循环。

  
图2：Shell continue关键字原理示意图

【实例1】不断从终端读取用户输入的 100 以内的正数，求它们的和：

1. #!/bin/bash
2. sum=0
3. **while** **read** n; **do**
4. **if**((n<1 || n>100)); **then**
5. **continue**
6. **fi**
7. ((sum+=n))
8. **done**
9. echo "sum=$sum"

运行结果：  
10↙  
20↙  
-1000↙  
5↙  
9999↙  
25↙  
sum=60  
  
变量 sum 最终的值为 60，-1000 和 9999 并没有计算在内，这是因为 -1000 和 9999 不在 1~100 的范围内，if 判断条件成立，所以执行了 continue 语句，跳过了当次循环，也就是跳过了((sum+=n))这条语句。  
  
注意，只有按下 Ctrl+D 组合键输入才会结束，read n才会判断失败，while 循环才会终止。  
  
【实例2】使用 continue 跳出多层循环，请看下面的代码：

1. #!/bin/bash
2. **for**((i=1; i<=5; i++)); **do**
3. **for**((j=1; j<=5; j++)); **do**
4. **if**((i\*j==12)); **then**
5. **continue** 2
6. **fi**
7. **printf** "%d\*%d=%-4d" $i $j $((i\*j))
8. **done**
9. **printf** "\n"
10. **done**

运行结果：

1\*1=1 1\*2=2 1\*3=3 1\*4=4 1\*5=5

2\*1=2 2\*2=4 2\*3=6 2\*4=8 2\*5=10

3\*1=3 3\*2=6 3\*3=9 4\*1=4 4\*2=8 5\*1=5 5\*2=10 5\*3=15 5\*4=20 5\*5=25

从运行结果可以看出，遇到continue 2时，不但跳过了内层 for 循环，也跳过了外层 for 循环。

## break 和 continue 的区别

break 用来结束所有循环，循环语句不再有执行的机会；continue 用来结束本次循环，直接跳到下一次循环，如果循环条件成立，还会继续循环。