本教程针对 Linux 后端服务器方面的知识

Linux 常用命令教程

Linux 处理目录的常用命令

接下来我们就来看几个常见的处理目录的命令吧：
ls：列出目录
cd：切换目录
pwd：显示目前的目录
mkdir：创建一个新的目录
rmdir：删除一个空的目录
cp：复制文件或目录
rm：移除文件或目录
mv：移动文件与目录，或修改文件与目录的名称
你可以使用 *man [命令]* 来查看各个命令的使用文档，如 ：man cp。

ls（列出目录）

在 Linux 系统当中， ls 命令可能是最常被运行的。语法：

```
1. 0voice@ubuntu:~$ ls [-aAdfFhilnrRSt] 目录名称
2. 0voice@ubuntu:~$ ls [--color={never,auto,always}] 目录名称
3. 0voice@ubuntu:~$ ls [--full-time] 目录名称
```

选项与参数：
-a ：全部的文件，连同隐藏档(开头为 . 的文件) 一起列出来(常用)
-d ：仅列出目录本身，而不是列出目录内的文件数据(常用)
-l ：长数据串列出，包含文件的属性与权限等等数据；(常用)
将家目录下的所有文件列出来(含属性与隐藏档)

```
1. 0voice@ubuntu:~$ ls -al
```

cd（切换目录）

cd 是 Change Directory 的缩写，这是用来变换工作目录的命令。语法：

```
1. cd [相对路径或绝对路径]
```

案例

```
# 表示回到自己的目录，亦即是 /0voice 这个目录
0voice@ubuntu:~$ cd ~

# 表示去到目前的上一级目录，亦即是 /0voice 的上一级目录的意思；
0voice@ubuntu:~$ cd ..
```

接下来大家多操作几次应该就可以很好的理解 cd 命令的。

pwd（显示目前所在的目录）

pwd 是 Print Working Directory 的缩写，也就是显示目前所在目录的命令。
选项与参数：
-P ：显示出确实的路径，而非使用连结 (link) 路径。
实例：单纯显示出目前的工作目录：

```
0voice@ubuntu:~$ pwd
/home/0voice   <== 显示出目录啦～
```

实例显示出实际的工作目录，而非连结档本身的目录名而已。

```
0voice@ubuntu:~$ cd /var/mail   <==注意，/var/mail 是一个连结档
0voice@ubuntu:/var/mail$ pwd
/var/mail         <==列出目前的工作目录
0voice@ubuntu:/var/mail$ pwd -P
/var/spool/mail   <==怎么回事？有没有加 -P 差很多～
0voice@ubuntu:/var/mail$ ls -ld /var/mail
lrwxrwxrwx 1 0voice 0voice 10 AUG 4 17:54 /var/mail -> spool/mail
# 看到这里应该知道为啥了吧？因为 /var/mail 是连结档，连结到 /var/spool/mail
# 所以，加上 pwd -P 的选项后，会不以连结档的数据显示，而是显示正确的完整路径啊！
```

mkdir（创建新目录）

如果想要创建新的目录的话，那么就使用 mkdir (make directory)吧。语法：

```
mkdir [-mp] 目录名称
```

选项与参数：
-m ：配置文件的权限喔！直接配置，不需要看默认权限 (umask) 的脸色～
-p ：帮助你直接将所需要的目录(包含上一级目录)递归创建起来！
实例：请到/tmp 底下尝试创建数个新目录看看：

```
0voice@ubuntu:~$ cd /tmp
0voice@ubuntu:tmp$ mkdir test    <==创建一名为 test 的新目录
0voice@ubuntu:tmp$ mkdir test1/test2/test3/test4
mkdir: cannot create directory `test1/test2/test3/test4':
```

```
5.  No such file or directory       <== 没办法直接创建此目录啊！
6.  0voice@ubuntu:tmp$ mkdir -p test1/test2/test3/test4
```

加了这个 -p 的选项，可以自行帮你创建多层目录！实例：创建权限为 rwx--x--x 的目录。

```
1.  0voice@ubuntu:/tmp$ mkdir -m 711 test2
2.  0voice@ubuntu:/tmp$ ls -l
3.  drwxr-xr-x  3 0voice  0voice  4096 Jul 18 12:50 test
4.  drwxr-xr-x  3 0voice  0voice  4096 Jul 18 12:53 test1
5.  drwx--x--x  2 0voice  0voice  4096 Jul 18 12:54 test2
```

上面的权限部分，如果没有加上 -m 来强制配置属性，系统会使用默认属性。
如果我们使用 -m ，如上例我们给予 -m 711 来给予新的目录 drwx--x--x 的权限。

rmdir（删除空的目录）

语法：

```
1.  rmdir [-p] 目录名称
```

选项与参数：
-p ：连同上一级『空的』目录也一起删除
删除 0voice 目录

```
1.  0voice@ubuntu:/tmp$ rmdir 0voice/
```

将 mkdir 实例中创建的目录(/tmp 底下)删除掉！

```
1.  0voice@ubuntu:/tmp$ ls -l   <==看看有多少目录存在？
2.  drwxr-xr-x  3 0voice  0voice 4096 Jul 18 12:50 test
3.  drwxr-xr-x  3 0voice  0voice 4096 Jul 18 12:53 test1
4.  drwx--x--x  2 0voice  0voice 4096 Jul 18 12:54 test2
5.  0voice@ubuntu:/tmp$ rmdir test   <==可直接删除掉，没问题
6.  0voice@ubuntu:/tmp$ rmdir test1  <==因为尚有内容，所以无法删除！
7.  rmdir: `test1': Directory not empty
8.  0voice@ubuntu:/tmp$ rmdir -p test1/test2/test3/test4
9.  0voice@ubuntu:/tmp$ ls -l        <==您看看，底下的输出中 test 与 test1 不见了！
10. drwx--x--x  2 0voice  0voice 4096 Jul 18 12:54 test2
```

利用 -p 这个选项，立刻就可以将 test1/test2/test3/test4 一次删除。不过要注意的是，这个 rmdir 仅能删除空的目录，你可以使用 rm 命令来删除非空目录。

cp（复制文件或目录）

cp 即拷贝文件和目录。语法:

```
1.  0voice@ubuntu:~$ cp [-adfilprsu] 来源档(source) 目标档(destination)
2.  0voice@ubuntu:~$ cp [options] source1 source2 source3 .... directory
```

选项与参数：
-a：相当於 -pdr 的意思，至於 pdr 请参考下列说明；(常用)
-d：若来源档为连结档的属性(link file)，则复制连结档属性而非文件本身；
-f：为强制(force)的意思，若目标文件已经存在且无法开启，则移除后再尝试一次；
-i：若目标档(destination)已经存在时，在覆盖时会先询问动作的进行(常用)
-l：进行硬式连结(hard link)的连结档创建，而非复制文件本身；
-p：连同文件的属性一起复制过去，而非使用默认属性(备份常用)；
-r：递归持续复制，用於目录的复制行为；(常用)
-s：复制成为符号连结档 (symbolic link)，亦即『捷径』文件；
-u：若 destination 比 source 旧才升级 destination ！
用 0voice 身份，将 0voice 目录下的 .bashrc 复制到 /tmp 下，并命名为 bashrc

```
1.  0voice@ubuntu:~$ cp ~/.bashrc /tmp/bashrc
2.  0voice@ubuntu:~$ cp -i ~/.bashrc /tmp/bashrc
3.  cp: overwrite `/tmp/bashrc'? n  <==n 不覆盖，y 为覆盖
```

rm（移除文件或目录）

语法：

```
1.  rm [-fir] 文件或目录
```

选项与参数：
-f ：就是 force 的意思，忽略不存在的文件，不会出现警告信息；
-i ：互动模式，在删除前会询问使用者是否动作
-r ：递归删除啊！最常用在目录的删除了！这是非常危险的选项!!!
将刚刚在 cp 的实例中创建的 bashrc 删除掉！

```
1.  0voice@ubuntu:~$ rm -i bashrc
2.  rm: remove regular file `bashrc'? y
```

如果加上 -i 的选项就会主动询问喔，避免你删除到错误的档名！

mv（移动文件与目录，或修改名称）

移动文件与目录，或者修改名称，语法：

```
1.  0voice@ubuntu:~$ mv [-fiu] source destination
2.  0voice@ubuntu:~$ mv [options] source1 source2 source3 .... directory
```

选项与参数：
-f ：force 强制的意思，如果目标文件已经存在，不会询问而直接覆盖；

-i ：若目标文件（destination）已经存在时，就会询问是否覆盖！
-u ：若目标文件已经存在，且 source 比较新，才会升级（update）

复制一文件，创建一目录，将文件移动到目录中

```
0voice@ubuntu:~$ cd /tmp
0voice@ubuntu:/tmp$ cp ~/.bashrc bashrc
0voice@ubuntu:/tmp$ mkdir mvtest
0voice@ubuntu:/tmp$ mv bashrc mvtest
```

将某个文件移动到某个目录去，就是这样做！
将刚刚的目录名称更名为 mvtest2

```
0voice@ubuntu:/tmp$ mv mvtest mvtest2
```

Linux 文件内容查看

Linux 系统中使用以下命令来查看文件的内容：
cat：由第一行开始显示文件内容
tac：从最后一行开始显示，可以看出 tac 是 cat 的倒著写！
nl：显示的时候，顺道输出行号！
more：一页一页的显示文件内容
less：与 more 类似，但是比 more 更好的是，他可以往前翻页！
head：只看头几行
tail：只看尾巴几行
你可以使用 man [命令]来查看各个命令的使用文档，如 ：man cp。

cat(文本输出)

由第一行开始显示文件内容
语法：

```
cat [-AbEnTv]
```

选项与参数：
-A ：相当於 -vET 的整合选项，可列出一些特殊字符而不是空白而已；
-b ：列出行号，仅针对非空白行做行号显示，空白行不标行号！
-E ：将结尾的断行字节 $ 显示出来；
-n ：列印出行号，连同空白行也会有行号，与 -b 的选项不同；
-T ：将 [tab] 按键以 ^I 显示出来；
-v ：列出一些看不出来的特殊字符

检看 /etc/issue 这个文件的内容：

```
0voice@ubuntu:~$ cat /etc/magic
# Magic local data for file(1) command.
# Insert here your local magic data. Format is described in magic(5).
```

tac（与 cat 相反）

tac 与 cat 命令刚好相反，文件内容从最后一行开始显示，可以看出 tac 是 cat 的倒着写！如：

```
0voice@ubuntu:~$ tac /etc/magic
# Insert here your local magic data. Format is described in magic(5).
# Magic local data for file(1) command.
```

nl(显示行号)

显示行号，语法：

```
nl [-bnw] 文件
```

选项与参数：
-b ：指定行号指定的方式，主要有两种：
-b a ：表示不论是否为空行，也同样列出行号(类似 cat -n)；
-b t ：如果有空行，空的那一行不要列出行号(默认值)；
-n ：列出行号表示的方法，主要有三种：
-n ln ：行号在荧幕的最左方显示；
-n rn ：行号在自己栏位的最右方显示，且不加 0 ；
-n rz ：行号在自己栏位的最右方显示，且加 0 ；
-w ：行号栏位的占用的位数。

实例一：用 nl 列出 /etc/issue 的内容

```
0voice@ubuntu:~$ nl /etc/magic
     1  # Magic local data for file(1) command.
     2  # Insert here your local magic data. Format is described in magic(5)
```

more（一页一页翻动）

一页一页翻动

```
0voice@ubuntu:~$ more /etc/profile
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).
```

```
4.
5.  if [ "$PS1" ]; then
6.    if [ "$BASH" ] && [ "$BASH" != "/bin/sh" ]; then
7.      # The file bash.bashrc already sets the default PS1.
8.      # PS1='\h:\w\$ '
9.      if [ -f /etc/bash.bashrc ]; then
10.       . /etc/bash.bashrc
11.     fi
12.   else
13.     if [ "`id -u`" -eq 0 ]; then
14.       PS1='# '
15.     else
16.       PS1='$ '
17.     fi
18.   fi
19. fi
20. --More--(70%)
```

在 more 这个程序的运行过程中，你有几个按键可以按的：
空白键 (space)：代表向下翻一页；
Enter ：代表向下翻『一行』；
/字串：代表在这个显示的内容当中，向下搜寻『字串』这个关键字；
:f ：立刻显示出档名以及目前显示的行数；
q ：代表立刻离开 more ，不再显示该文件内容。
b 或 [ctrl]-b ：代表往回翻页，不过这动作只对文件有用，对管线无用。

less

一页一页翻动，以下实例输出/etc/man.config 文件的内容：

```
1.  0voice@ubuntu:~$ less /etc/profile
2.  # /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
3.  # and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).
4.
5.  if [ "$PS1" ]; then
6.    if [ "$BASH" ] && [ "$BASH" != "/bin/sh" ]; then
7.      # The file bash.bashrc already sets the default PS1.
8.      # PS1='\h:\w\$ '
9.      if [ -f /etc/bash.bashrc ]; then
10.       . /etc/bash.bashrc
11.     fi
12.   else
13.     if [ "`id -u`" -eq 0 ]; then
14.       PS1='# '
```

```
    else
      PS1='$ '
    fi
  fi
fi
```

less 运行时可以输入的命令有：
空白键 ：向下翻动一页；
[pagedown]：向下翻动一页；
[pageup]：向上翻动一页；
/字串 ：向下搜寻『字串』的功能；
?字串 ：向上搜寻『字串』的功能；
n ：重复前一个搜寻（与 / 或 ? 有关！）
N ：反向的重复前一个搜寻（与 / 或 ? 有关！）
q ：离开 less 这个程序；

head

取出文件前面几行，语法：

```
head [-n number] 文件
```

选项与参数：
-n ：后面接数字，代表显示几行的意思

```
0voice@ubuntu:~$ head -n 10  /etc/profile
```

默认的情况中，显示前面 10 行！若要显示前 20 行，就得要这样：

```
0voice@ubuntu:~$ head -n 20  /etc/profile
```

tail（取出文件后面几行）

取出文件后面几行，语法：

```
tail [-n number] 文件
```

选项与参数：
-n ：后面接数字，代表显示几行的意思
-f ：表示持续侦测后面所接的档名，要等到按下[ctrl]-c 才会结束 tail 的侦测

```
0voice@ubuntu:~$ tail -n 10 /etc/profile
```

```
2. 0voice@ubuntu:~$ tail -n 20 /etc/profile
```

Shell 教程

Shell 是一个用 c 语言编写的程序，它是用户使用 Linux 的桥梁。Shell 既是命令语言，又是一种程序设计语言。
Shell 脚本（shell script），是一种为 shell 编写的脚本程序。业界所说的 shell 通常指的是 shell 脚本。Shell 与 Shell 脚本是两个不同的概念。
Linux 中的 shell 有很多种类，常用的几种：
1> Bourne Shell（/usr/bin/sh 或/bin/sh）
2> Bourne Again Shell（/bin/bash）
3> C Shell（/usr/bin/csh）
4> K Shell（/usr/bin/ksh）
5> Shell for Root（/sbin/sh）
本教程使用的是 Bash，也就是 Bourne Again Shell，由于易用和免费，Bash 在日常工作中被广泛使用。同时，Bash 也是大多数 Linux 系统默认的 Shell。

第一个 Shell 脚本

打开文本编辑器（vi/vim），新建一个文件 first.sh，扩展名为 sh（sh 代表 shell）。扩展名并不影响脚本执行。
实例：

```
1. #!/bin/bash
2. echo “Hello World!”
```

#! 是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种 shell。
echo 命令用于向窗口输出文本。

运行 Shell 脚本方式
1. 作为可执行程序
   将上面的代码保存为 first.sh，并 cd 到相应目录：
   ```
   $ chmod +x first.sh
   $ ./first.sh
   ```
2. 作为解释器参数
   ```
   /bin/bash ./first.sh
   ```

Shell 变量

定义变量时，变量名不加美元符号（$），如：

```
1. domain="www.0voice.com"
```

注意，变量名与等号之间不能有空格，变量名的命名需遵循如下规则：
1> 命名只能使用英文字母，数字和下划线，首个字符不能以数字开头。
2> 中间不能有空格，可以使用下划线(_)。
3> 不能使用标点符号。
4> 不能使用 bash 里的关键字（可用 help 命令查看保留关键字）

例如，有效的 shell 变量名

```
1. zerovoice
2. ZERO_VOICE
3. _ZERO_VOICE
4. Zerovoice0
```

例如，无效的 shell 变量名

```
1. 0voice
2. ?voice
3. zero*voice
```

使用变量

使用一个定义过的变量，只要在变量名前面加美元符号即可，如：

```
1. domain="www.0voice.com"
2. echo $domain
3. echo ${domain}
```

变量外的大括号，$domain 与${domain}效果一样。也是为了帮助解释器识别变量边界。

```
1. for skill in C CPP Linux Shell; do
2.     echo "I am good at ${skill}Code"
3. done
```

如果不给 skill 变量加上大括号，写成了 echo "I am good at ${skill}Code"，解释器就会把$skillCode 当成一个变量。

只读变量

使用 readonly 命令可以将变量定义为只读变量，只读变量的值不能被改变。下面的例子尝试更改只读变量，结果报错：

```
1.  #!/bin/bash
2.  url="http://www.google.com"
3.  readonly url
4.  url="http://www.0voice.com"
```

运行脚本，结果如下：

```
./var.sh: line 4: url: readonly variable
```

删除变量

使用 unset 命令可以删除变量。语法：

```
1.  unset variable_name
```

变量被删除后不能再次使用。unset 命令不能删除只读变量。

实例

```
1.  #!/bin/bash
2.  url="http://www.0voice.com"
3.  unset url
4.  echo $url
```

以上实例执行将没有任何输出。

变量类型

运行 shell 时，会同时存在三种变量：
1）**局部变量** 局部变量在脚本或命令中定义，仅在当前 shell 实例中有效，其他 shell 启动的程序不能访问局部变量。
2）**环境变量** 所有的程序，包括 shell 启动的程序，都能访问环境变量，有些程序需要环境变量来保证其正常运行。必要的时候 shell 脚本也可以定义环境变量。
3）**shell 变量** shell 变量是由 shell 程序设置的特殊变量。shell 变量中有一部分是环境变量，有一部分是局部变量，这些变量保证了 shell 的正常运行

Shell 字符串

字符串是 shell 编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。

单引号

单引号字符串的限制：

```
1. str='this is a string'
```

单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；
单引号字串中不能出现单独一个的单引号(对单引号使用转义符后也不行)，但可成对出现，作为字符串拼接使用。

双引号

```
1. your_name='0voice'
2. str="Hello, I know you are \"$your_name\"! \n"
3. echo -e $str
```

输出结果为：

```
1. Hello, I know you are "0voice"!
```

双引号的优点：
双引号里可以有变量
2. 双引号里可以出现转义字符

拼接字符串

```
1. your_name="0voice"
2. # 使用双引号拼接
3. greeting="hello, "$your_name" !"
4. greeting_1="hello, ${your_name} !"
5. echo $greeting  $greeting_1
6. # 使用单引号拼接
7. greeting_2='hello, '$your_name' !'
8. greeting_3='hello, ${your_name} !'
9. echo $greeting_2  $greeting_3
```

输出结果为：

```
1. hello, 0voice ! hello, 0voice !
```

获取字符串长度

```
1. string="abcd"
2. echo ${#string} #输出 4
```

提取子字符串

以下实例从字符串第 2 个字符开始截取 4 个字符：

```
1. string="0voice is a great college"
2. echo ${string:1:4} # 输出 voic
```

查找子字符串

查找字符 i 或 o 的位置(哪个字母先出现就计算哪个)：

```
1. string="0voice is a great college "
2. echo `expr index "$string" io`  # 输出 3
```

注意： 以上脚本中 ` 是反引号，而不是单引号 '，不要看错了哦。

Shell 数组

bash 支持一维数组（不支持多维数组），并且没有限定数组的大小。
类似于 C 语言，数组元素的下标由 0 开始编号。获取数组中的元素要利用下标，下标可以是整数或算术表达式，其值应大于或等于 0。

定义数组

在 Shell 中，用括号来表示数组，数组元素用"空格"符号分割开。定义数组的一般形式为：

```
1. 数组名=(值 1 值 2 ... 值 n)
```

例如：

```
1. array_name=(value0 value1 value2 value3)
```

还可以单独定义数组的各个分量

```
array_name[0]=value0
array_name[1]=value1
array_name[n]=valuen
```

可以不使用连续的下标，而且下标的范围没有限制。

读取数组

读取数组元素值的一般格式是：

```
${数组名[下标]}
```

例如：

```
valuen=${array_name[n]}
```

使用@符号可以获取数组中的所有元素，例如：

```
echo ${array_name[@]}
```

获取数组的长度

获取数组长度的方法与获取字符串长度的方法相同，例如：

```
# 取得数组元素的个数
length=${#array_name[@]}
# 或者
length=${#array_name[*]}
# 取得数组单个元素的长度
lengthn=${#array_name[n]}
```

Shell 注释

以 # 开头的行就是注释，会被解释器忽略。通过每一行加一个 # 号设置多行注释，像这样：

```
#--------------------------------------------
# slogan：一切只为渴望更优秀的你
#--------------------------------------------
##### 用户配置区 开始 #####
```

```
#
#
# 这里可以添加脚本描述信息
#
#
##### 用户配置区 结束  #####
```

如果在开发过程中，遇到大段的代码需要临时注释起来，过一会儿又取消注释，怎么办呢？每一行加个#符号太费力了，可以把这一段要注释的代码用一对花括号括起来，定义成一个函数，没有地方调用这个函数，这块代码就不会执行，达到了和注释一样的效果。

多行注释

多行注释还可以使用以下格式：

```
:<<EOF
注释内容...
注释内容...
注释内容...
EOF
```

EOF 也可以使用其他符号:

```
:<<'
注释内容...
注释内容...
注释内容...
'

:<<!
注释内容...
注释内容...
注释内容...
!
```

shell 传递参数

我们可以在执行 Shell 脚本时，向脚本传递参数，脚本内获取参数的格式为：**$n**。**n** 代表一个数字，1 为执行脚本的第一个参数，2 为执行脚本的第二个参数，以此类推……

以下实例我们向脚本传递三个参数，并分别输出，其中 **$0** 为执行的文件名：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

echo "Shell 传递参数实例！";
echo "执行的文件名：$0";
echo "第一个参数为：$1";
echo "第二个参数为：$2";
echo "第三个参数为：$3";
```

为脚本设置可执行权限，并执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
$ ./test.sh 1 2 3
Shell 传递参数实例！
执行的文件名：./test.sh
第一个参数为：1
第二个参数为：2
第三个参数为：3
```

特殊字符

几个特殊字符用来处理参数：

参数处理	说明
$#	传递到脚本的参数个数
$*	以一个单字符串显示所有向脚本传递的参数。 如"$*"用「"」括起来的情况、以"$1 $2 … $n"的形式输出所有参数。
$$	脚本运行的当前进程 ID 号
$!	后台运行的最后一个进程的 ID 号
$@	与$*相同，但是使用时加引号，并在引号中返回每个参数。 如"$@"用「"」括起来的情况、以"$1" "$2" … "$n" 的形式输出所有参数。
$-	显示 Shell 使用的当前选项，与 set 命令功能相同。
$?	显示最后命令的退出状态。0 表示没有错误，其他任何值表明有错误。

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

echo "Shell 传递参数实例！";
echo "第一个参数为：$1";

echo "参数个数为：$#";
echo "传递的参数作为一个字符串显示：$*";
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
$ ./test.sh 1 2 3
Shell 传递参数实例！
第一个参数为：1
参数个数为：3
传递的参数作为一个字符串显示：1 2 3
```

$* 与 $@ 区别：

相同点：都是引用所有参数。
不同点：只有在双引号中体现出来。假设在脚本运行时写了三个参数 1、2、3,，则 "*" 等价于 "1 2 3"（传递了一个参数），而 "@" 等价于 "1" "2" "3"（传递了三个参数）。

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

echo "-- \$* 演示 ---"
for i in "$*"; do
    echo $i
done

echo "-- \$@ 演示 ---"
for i in "$@"; do
    echo $i
done
```

执行脚本，输出结果如下所示：

```
$ chmod +x test.sh
$ ./test.sh 1 2 3
```

```
-- $* 演示 ---
1 2 3
-- $@ 演示 ---
1
2
3
```

Shell 基本运算符

Shell 和其他编程语言一样，支持多种运算符，包括：

1. 算数运算符
2. 关系运算符
3. 布尔运算符
4. 字符串运算符
5. 文件测试运算符

原生 bash 不支持简单的数学运算，但是可以通过其他命令来实现，例如 awk 和 expr，expr 最常用。expr 是一款表达式计算工具，使用它能完成表达式的求值操作。例如，两个数相加(**注意使用的是反引号 ` 而不是单引号 '**)：

```
#!/bin/bash
val=`expr 2 + 2`
echo "两数之和为 : $val"
```

执行脚本，输出结果如下所示：

```
两数之和为 : 4
```

两点注意：

表达式和运算符之间要有空格，例如 2+2 是不对的，必须写成 2 + 2，这与我们熟悉的大多数编程语言不一样。

完整的表达式要被``包含，注意这个字符不是常用的单引号，在 Esc 键下边。

算术运算符

下表列出了常用的算术运算符，假定变量 a 为 10，变量 b 为 20：

运算符	说明	举例
+	加法	`expr $a + $b` 结果为 30。

-	减法	`expr $a - $b` 结果为 -10。
*	乘法	`expr $a \* $b` 结果为 200。
/	除法	`expr $b / $a` 结果为 2。
%	取余	`expr $b % $a` 结果为 0。
=	赋值	a=$b 将把变量 b 的值赋给 a。
==	相等。用于比较两个数字，相同则返回 true。	[$a == $b] 返回 false。
!=	不相等。用于比较两个数字，不相同则返回 true。	[$a != $b] 返回 true。

注意：条件表达式要放在方括号之间，并且要有空格，例如: **[$a==$b]** 是错误的，必须写成 **[$a == $b]**。

算术运算符实例如下：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

a=10
b=20

val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
```

```
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a 等于 b"
fi
if [ $a != $b ]
then
   echo "a 不等于 b"
fi
```

执行脚本，输出结果如下所示：

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a 不等于 b
```

注意：

- 乘号(*)前边必须加反斜杠(\)才能实现乘法运算；
- if...then...fi 是条件语句，后续将会讲解。
- 在 MAC 中 shell 的 expr 语法是：**$((表达式))**，此处表达式中的"*"不需要转义符号 "\" 。

关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。下表列出了常用的关系运算符，假定变量 a 为 10，变量 b 为 20：

运算符	说明	举例
-eq	检测两个数是否相等，相等返回 true。	[$a -eq $b] 返回 false。
-ne	检测两个数是否不相等，不相等返回 true。	[$a -ne $b] 返回 true。

-gt	检测左边的数是否大于右边的，如果是，则返回 true。	[$a -gt $b] 返回 false。
-lt	检测左边的数是否小于右边的，如果是，则返回 true。	[$a -lt $b] 返回 true。
-ge	检测左边的数是否大于等于右边的，如果是，则返回 true。	[$a -ge $b] 返回 false。
-le	检测左边的数是否小于等于右边的，如果是，则返回 true。	[$a -le $b] 返回 true。

关系运算符实例如下：

```
#!/bin/bash
# author: 零声学院
# url:www.0voice.com

a=10
b=20

if [ $a -eq $b ]
then
   echo "$a -eq $b : a 等于 b"
else
   echo "$a -eq $b: a 不等于 b"
fi
if [ $a -ne $b ]
then
   echo "$a -ne $b: a 不等于 b"
else
   echo "$a -ne $b : a 等于 b"
fi
if [ $a -gt $b ]
then
   echo "$a -gt $b: a 大于 b"
else
   echo "$a -gt $b: a 不大于 b"
fi
if [ $a -lt $b ]
then
```

```
    echo "$a -lt $b: a 小于 b"
else
    echo "$a -lt $b: a 不小于 b"
fi
if [ $a -ge $b ]
then
    echo "$a -ge $b: a 大于或等于 b"
else
    echo "$a -ge $b: a 小于 b"
fi
if [ $a -le $b ]
then
    echo "$a -le $b: a 小于或等于 b"
else
    echo "$a -le $b: a 大于 b"
fi
```

执行脚本，输出结果如下所示：

```
10 -eq 20: a 不等于 b
10 -ne 20: a 不等于 b
10 -gt 20: a 不大于 b
10 -lt 20: a 小于 b
10 -ge 20: a 小于 b
10 -le 20: a 小于或等于 b
```

布尔运算符

下表列出了常用的布尔运算符，假定变量 a 为 10，变量 b 为 20：

运算符	说明	举例
!	非运算，表达式为 true 则返回 false，否则返回 true。	[! false] 返回 true。
-o	或运算，有一个表达式为 true 则返回 true。	[$a -lt 20 -o $b -gt 100] 返回 true。
-a	与运算，两个表达式都为 true 才返回 true。	[$a -lt 20 -a $b -gt 100] 返回 false。

布尔运算符实例如下：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a 不等于 b"
else
   echo "$a == $b: a 等于 b"
fi
if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a 小于 100 且 $b 大于 15 : 返回 true"
else
   echo "$a 小于 100 且 $b 大于 15 : 返回 false"
fi
if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a 小于 100 或 $b 大于 100 : 返回 true"
else
   echo "$a 小于 100 或 $b 大于 100 : 返回 false"
fi
if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a 小于 5 或 $b 大于 100 : 返回 true"
else
   echo "$a 小于 5 或 $b 大于 100 : 返回 false"
fi
```

执行脚本，输出结果如下所示：

```
10 != 20 : a 不等于 b
10 小于 100 且 20 大于 15 : 返回 true
10 小于 100 或 20 大于 100 : 返回 true
10 小于 5 或 20 大于 100 : 返回 false
```

逻辑运算符

以下介绍 Shell 的逻辑运算符，假定变量 a 为 10，变量 b 为 20:

运算符	说明	举例
&&	逻辑的 AND	[[$a -lt 100 && $b -gt 100]] 返回 false
||	逻辑的 OR	[[$a -lt 100 || $b -gt 100]] 返回 true

逻辑运算符实例如下：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

a=10
b=20

if [[ $a -lt 100 && $b -gt 100 ]]
then
   echo "返回 true"
else
   echo "返回 false"
fi

if [[ $a -lt 100 || $b -gt 100 ]]
then
   echo "返回 true"
else
   echo "返回 false"
fi
```

执行脚本，输出结果如下所示：

```
返回 false
返回 true
```

字符串运算符

下表列出了常用的字符串运算符，假定变量 a 为 "abc"，变量 b 为 "efg"：

运算符	说明	举例
=	检测两个字符串是否相等，相等返回 true。	[$a = $b] 返回 false。

!=	检测两个字符串是否相等，不相等返回 true。	[$a != $b] 返回 true。
-z	检测字符串长度是否为 0，为 0 返回 true。	[-z $a] 返回 false。
-n	检测字符串长度是否为 0，不为 0 返回 true。	[-n "$a"] 返回 true。
$	检测字符串是否为空，不为空返回 true。	[$a] 返回 true。

字符串运算符实例如下：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

a="abc"
b="efg"

if [ $a = $b ]
then
   echo "$a = $b : a 等于 b"
else
   echo "$a = $b: a 不等于 b"
fi
if [ $a != $b ]
then
   echo "$a != $b : a 不等于 b"
else
   echo "$a != $b: a 等于 b"
fi
if [ -z $a ]
then
   echo "-z $a : 字符串长度为 0"
else
   echo "-z $a : 字符串长度不为 0"
fi
if [ -n "$a" ]
then
   echo "-n $a : 字符串长度不为 0"
else
   echo "-n $a : 字符串长度为 0"
fi
if [ $a ]
then
   echo "$a : 字符串不为空"
```

```
else
   echo "$a : 字符串为空"
fi
```

执行脚本，输出结果如下所示：

```
abc = efg: a 不等于 b
abc != efg : a 不等于 b
-z abc : 字符串长度不为 0
-n abc : 字符串长度不为 0
abc : 字符串不为空
```

文件测试运算符

文件测试运算符用于检测 Unix 文件的各种属性。属性检测描述如下：

操作符	说明	举例
-b file	检测文件是否是块设备文件，如果是，则返回 true。	[-b $file] 返回 false。
-c file	检测文件是否是字符设备文件，如果是，则返回 true。	[-c $file]返回 false。
-d file	检测文件是否是目录，如果是，则返回 true。	[-d $file]返回 false。
-f file	检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。	[-f $file]返回 true。
-g file	检测文件是否设置了 SGID 位，如果是，则返回 true。	[-g $file]返回 false。
-k file	检测文件是否设置了粘着位(Sticky Bit)，如果是，则返回 true。	[-k $file]返回 false。
-p file	检测文件是否是有名管道，如果是，则返回 true。	[-p $file]返回 false。

-u file	检测文件是否设置了 SUID 位，如果是，则返回 true。	[-u $file]返回 false。
-r file	检测文件是否可读，如果是，则返回 true。	[-r $file] 返回 true。
-w file	检测文件是否可写，如果是，则返回 true。	[-w $file] 返回 true。
-x file	检测文件是否可执行，如果是，则返回 true。	[-x $file] 返回 true。
-s file	检测文件是否为空（文件大小是否大于 0），不为空返回 true。	[-s $file] 返回 true。
-e file	检测文件（包括目录）是否存在，如果是，则返回 true。	[-e $file] 返回 true。

其他检查符：
-S: 判断某文件是否 socket。
-L: 检测文件是否存在并且是一个符号链接。
变量 file 表示文件 /home/king/test.sh，它的大小为 100 字节，具有 rwx 权限。下面的代码，将检测该文件的各种属性：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

file="/home/king/test.sh"
if [ -r $file ]
then
   echo "文件可读"
else
   echo "文件不可读"
fi
if [ -w $file ]
then
   echo "文件可写"
else
   echo "文件不可写"
```

```
fi
if [ -x $file ]
then
   echo "文件可执行"
else
   echo "文件不可执行"
fi
if [ -f $file ]
then
   echo "文件为普通文件"
else
   echo "文件为特殊文件"
fi
if [ -d $file ]
then
   echo "文件是个目录"
else
   echo "文件不是个目录"
fi
if [ -s $file ]
then
   echo "文件不为空"
else
   echo "文件为空"
fi
if [ -e $file ]
then
   echo "文件存在"
else
   echo "文件不存在"
fi
```

执行脚本，输出结果如下所示：

```
文件可读
文件可写
文件可执行
文件为普通文件
文件不是个目录
文件不为空
文件存在
```

Shell echo 命令

Shell 的 echo 指令与 PHP 的 echo 指令类似，都是用于字符串的输出。命令格式：

```
echo string
```

您可以使用 echo 实现更复杂的输出格式控制。

显示普通字符串：

```
echo "It is a test"
```

这里的双引号完全可以省略，以下命令与上面实例效果一致：

```
echo It is a test
```

显示转义字符

```
echo "\"It is a test\""
```

结果将是:

```
"It is a test"
```

同样，双引号也可以省略

显示变量

read 命令从标准输入中读取一行,并把输入行的每个字段的值指定给 shell 变量

```
#!/bin/sh
read name
echo "$name It is a test"
```

以上代码保存为 test.sh，name 接收标准输入的变量，结果将是:

```
[root@www ~]# sh test.sh
OK                     #标准输入
OK It is a test        #输出
```

显示换行

```
echo -e "OK! \n" # -e 开启转义
echo "It is a test"
```

输出结果：

```
OK!

It is a test
```

显示不换行

```
#!/bin/sh
echo -e "OK! \c" # -e 开启转义 \c 不换行
echo "It is a test"
```

输出结果：

```
OK! It is a test
```

显示结果定向至文件

```
echo "It is a test" > myfile
```

原样输出字符串，不进行转义或取变量(用单引号)

```
echo '$name\"'
```

输出结果：

```
$name\"
```

显示命令执行结果

```
echo `date`
```

注意： 这里使用的是反引号 `，而不是单引号 '。

结果将显示当前日期

```
Thu Jul 24 10:08:46 CST 2014
```

Shell printf 命令

printf 命令模仿 C 程序库（library）里的 printf() 程序。printf 由 POSIX 标准所定义，因此使用 printf 的脚本比使用 echo 移植性好。printf 使用引用文本或空格分隔的参数，外面可以在 printf 中使用格式化字符串，还可以制定字符串的宽度、左右对齐方式等。默认 printf 不会像 echo 自动添加换行符，我们可以手动添加 \n。

printf 命令的语法：

```
printf  format-string  [arguments...]
```

参数说明：
format-string: 为格式控制字符串
arguments: 为参数列表。
实例如下：

```
$ echo "Hello, Shell"
Hello, Shell
$ printf "Hello, Shell\n"
Hello, Shell
$
```

接下来,我来用一个脚本来体现 printf 的强大功能：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

printf "%-10s %-8s %-4s\n" 姓名 性别 体重kg
printf "%-10s %-8s %-4.2f\n" 郭靖 男 66.1234
printf "%-10s %-8s %-4.2f\n" 杨过 男 48.6543
printf "%-10s %-8s %-4.2f\n" 郭芙 女 47.9876
```

执行脚本，输出结果如下所示：

```
姓名     性别   体重kg
郭靖     男      66.12
杨过     男      48.65
郭芙     女      47.99
```

%s %c %d %f 都是格式替代符
%-10s 指一个宽度为 10 个字符（-表示左对齐，没有则表示右对齐），任何字符都会被显示在 10 个字符宽的字符内，如果不足则自动以空格填充，超过也会将内容全部显示出来。
%-4.2f 指格式化为小数，其中.2 指保留 2 位小数。
更多实例：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

# format-string 为双引号
printf "%d %s\n" 1 "abc"

# 单引号与双引号效果一样
printf '%d %s\n' 1 "abc"

# 没有引号也可以输出
printf %s abcdef

# 格式只指定了一个参数，但多出的参数仍然会按照该格式输出，format-string 被重用
printf %s abc def

printf "%s\n" abc def

printf "%s %s %s\n" a b c d e f g h i j

# 如果没有 arguments，那么 %s 用 NULL 代替，%d 用 0 代替
printf "%s and %d \n"
```

执行脚本，输出结果如下所示：

```
1 abc
1 abc
abcdefabcdefabc
def
a b c
d e f
g h i
j
 and 0
```

printf 的转义序列

序列	说明
\a	警告字符，通常为 ASCII 的 BEL 字符
\b	后退
\c	抑制（不显示）输出结果中任何结尾的换行字符（只在%b 格式指示符控制下的参数字符串中有效），而且，任何留在参数里的字符、任何接下来的参数以及任何留在格式字符串中的字符，都被忽略
\f	换页（formfeed）
\n	换行
\r	回车（Carriage return）
\t	水平制表符
\v	垂直制表符
\\	一个字面上的反斜杠字符
\ddd	表示 1 到 3 位数八进制值的字符。仅在格式字符串中有效
\0ddd	表示 1 到 3 位的八进制值字符

以下为演示案例

```
$ printf "a string, no processing:<%s>\n" "A\nB"
a string, no processing:<A\nB>

$ printf "a string, no processing:<%b>\n" "A\nB"
a string, no processing:<A
B>

$ printf "www.0voice.com \a"
www.0voice.com $                #不换行
```

Shell test 命令

Shell 中的 test 命令用于检查某个条件是否成立，它可以进行数值、字符和文件三个方面的测试。

数值测试

参数	说明
-eq	等于则为真
-ne	不等于则为真
-gt	大于则为真
-ge	大于等于则为真
-lt	小于则为真
-le	小于等于则为真

实例演示：

```
num1=100
num2=100
if test $[num1] -eq $[num2]
then
    echo '两个数相等! '
else
    echo '两个数不相等! '
fi
```

输出结果：

```
两个数相等!
```

代码中的 [] 执行基本的算数运算，如：

```
#!/bin/bash
```

```
a=5
b=6

result=$[a+b] # 注意等号两边不能有空格
echo "result 为： $result"
```

结果为:

```
result 为： 11
```

字符串测试

参数	说明
=	等于则为真
!=	不相等则为真
-z 字符串	字符串的长度为零则为真
-n 字符串	字符串的长度不为零则为真

实例演示：

```
num1="0voice"
num2="0vo1ice"
if test $num1 = $num2
then
    echo '两个字符串相等!'
else
    echo '两个字符串不相等!'
fi
```

输出结果：

```
两个字符串不相等!
```

文件测试

参数	说明
-e 文件名	如果文件存在则为真
-r 文件名	如果文件存在且可读则为真
-w 文件名	如果文件存在且可写则为真
-x 文件名	如果文件存在且可执行则为真
-s 文件名	如果文件存在且至少有一个字符则为真
-d 文件名	如果文件存在且为目录则为真
-f 文件名	如果文件存在且为普通文件则为真
-c 文件名	如果文件存在且为字符型特殊文件则为真
-b 文件名	如果文件存在且为块特殊文件则为真

实例演示：

```
cd /bin
if test -e ./bash
then
    echo '文件已存在!'
else
    echo '文件不存在!'
fi
```

输出结果：

```
文件已存在!
```

另外，Shell 还提供了与(-a)、或(-o)、非(!)三个逻辑操作符用于将测试条件连接起来，其优先级为："!"最高，"-a"次之，"-o"最低。例如：

```
cd /bin
if test -e ./notFile -o -e ./bash
then
```

```
    echo '至少有一个文件存在!'
else
    echo '两个文件都不存在'
fi
```

输出结果：

```
至少有一个文件存在!
```

Shell 流程控制

if

if 语句语法格式：

```
if condition
then
    command1
    command2
    ...
    commandN
fi
```

写成一行（适用于终端命令提示符）：

```
if [ $(ps -ef | grep -c "ssh") -gt 1 ]; then echo "true"; fi
```

末尾的 fi 就是 if 倒过来拼写，后面还会遇到类似的。

if else

if else 语法格式：

```
if condition
then
    command1
    command2
    ...
    commandN
else
    command
```

```
fi
```

if else-if else

if else-if else 语法格式：

```
if condition1
then
    command1
elif condition2
then
    command2
else
    commandN
fi
```

以下实例判断两个变量是否相等：

```
a=10
b=20
if [ $a == $b ]
then
   echo "a 等于 b"
elif [ $a -gt $b ]
then
   echo "a 大于 b"
elif [ $a -lt $b ]
then
   echo "a 小于 b"
else
   echo "没有符合的条件"
fi
```

输出结果：

```
a 小于 b
```

if else 语句经常与 test 命令结合使用，如下所示：

```
num1=$[2*3]
num2=$[1+5]
if test $[num1] -eq $[num2]
then
    echo '两个数字相等!'
else
    echo '两个数字不相等!'
```

```
fi
```

输出结果：

```
两个数字相等!
```

---

for 循环

与其他编程语言类似，Shell 支持 for 循环。
for 循环一般格式为：

```
for var in item1 item2 ... itemN
do
    command1
    command2
    ...
    commandN
done
```

写成一行：

```
for var in item1 item2 ... itemN; do command1; command2… done;
```

当变量值在列表里，for 循环即执行一次所有命令，使用变量名获取列表中的当前取值。命令可为任何有效的 shell 命令和语句。in 列表可以包含替换、字符串和文件名。
in 列表是可选的，如果不用它，for 循环使用命令行的位置参数。
例如，顺序输出当前列表中的数字：

```
for loop in 1 2 3 4 5
do
    echo "The value is: $loop"
done
```

输出结果：

```
The value is: 1
The value is: 2
The value is: 3
The value is: 4
The value is: 5
```

顺序输出字符串中的字符：

```
for str in 'This is a string'
do
    echo $str
```

```
done
```

输出结果：

```
This is a string
```

---

while 语句

while 循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。其格式为：

```
while condition
do
    command
done
```

以下是一个基本的 while 循环，测试条件是：如果 int 小于等于 5，那么条件返回真。int 从 0 开始，每次循环处理时，int 加 1。运行上述脚本，返回数字 1 到 5，然后终止。

```
#!/bin/bash
int=1
while(( $int<=5 ))
do
    echo $int
    let "int++"
done
```

运行脚本，输出：

```
1
2
3
4
5
```

以上实例使用了 Bash let 命令，它用于执行一个或多个表达式，变量计算中不需要加上 $ 来表示变量

while 循环可用于读取键盘信息。下面的例子中，输入信息被设置为变量 FILM，按<Ctrl-D>结束循环。

```
echo '按下 <CTRL-D> 退出'
echo -n '输入你最喜欢的学院名: '
while read FILM
do
    echo "是的！$FILM 是一个好学院"
```

```
done
```

运行脚本，输出类似下面：

```
按下 <CTRL-D> 退出
输入你最喜欢的学院名:零声学院
是的！零声学院 是一个好学院
```

无限循环

无限循环语法格式：

```
while :
do
    command
done
```

或者

```
while true
do
    command
done
```

或者

```
for (( ; ; ))
```

until 循环

until 循环执行一系列命令直至条件为 true 时停止。
until 循环与 while 循环在处理方式上刚好相反。
一般 while 循环优于 until 循环，但在某些时候—也只是极少数情况下，until 循环更加有用。
until 语法格式:

```
until condition
do
    command
done
```

condition 一般为条件表达式，如果返回值为 false，则继续执行循环体内的语句，否则跳出循环。
以下实例我们使用 until 命令来输出 0 ~ 9 的数字：

```
#!/bin/bash

a=0

until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

运行结果：
输出结果为：

```
0
1
2
3
4
5
6
7
8
9
```

case

Shell case 语句为多选择语句。可以用 case 语句匹配一个值与一个模式，如果匹配成功，执行相匹配的命令。case 语句格式如下：

```
case 值 in
模式 1)
    command1
    command2
    ...
    commandN
    ;;
模式 2）
    command1
    command2
    ...
    commandN
    ;;
esac
```

case 工作方式如上所示。取值后面必须为单词 in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至 ;;。取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 * 捕获该值，再执行后面的命令。
下面的脚本提示输入 1 到 4，与每一种模式进行匹配：

```
echo '输入 1 到 4 之间的数字:'
echo '你输入的数字为:'
read aNum
case $aNum in
    1)  echo '你选择了 1'
    ;;
    2)  echo '你选择了 2'
    ;;
    3)  echo '你选择了 3'
    ;;
    4)  echo '你选择了 4'
    ;;
    *)  echo '你没有输入 1 到 4 之间的数字'
    ;;
esac
```

输入不同的内容，会有不同的结果，例如：

```
输入 1 到 4 之间的数字:
你输入的数字为:
3
你选择了 3
```

跳出循环

在循环过程中，有时候需要在未达到循环结束条件时强制跳出循环，Shell 使用两个命令来实现该功能：break 和 continue。

break 命令

break 命令允许跳出所有循环（终止执行后面的所有循环）。
下面的例子中，脚本进入死循环直至用户输入数字大于 5。要跳出这个循环，返回到 shell 提示符下，需要使用 break 命令。

```
#!/bin/bash
while :
do
```

```
    echo -n "输入 1 到 5 之间的数字:"
    read aNum
    case $aNum in
        1|2|3|4|5) echo "你输入的数字为 $aNum!"
        ;;
        *) echo "你输入的数字不是 1 到 5 之间的! 游戏结束"
            break
        ;;
    esac
done
```

执行以上代码，输出结果为：

```
输入 1 到 5 之间的数字:3
你输入的数字为 3!
输入 1 到 5 之间的数字:7
你输入的数字不是 1 到 5 之间的! 游戏结束
```

continue

continue 命令与 break 命令类似，只有一点差别，它不会跳出所有循环，仅仅跳出当前循环。

对上面的例子进行修改：

```
#!/bin/bash
while :
do
    echo -n "输入 1 到 5 之间的数字: "
    read aNum
    case $aNum in
        1|2|3|4|5) echo "你输入的数字为 $aNum!"
        ;;
        *) echo "你输入的数字不是 1 到 5 之间的!"
            continue
            echo "游戏结束"
        ;;
    esac
done
```

运行代码发现，当输入大于 5 的数字时，该例中的循环不会结束，语句 **echo "游戏结束"** 永远不会被执行。

esac

case 的语法和 C family 语言差别很大，它需要一个 esac（就是 case 反过来）作为结束标记，每个 case 分支用右圆括号，用两个分号表示 break。

Shell 函数

linux shell 可以用户定义函数，然后在 shell 脚本中可以随便调用。
shell 中函数的定义格式如下：

```
[ function ] funname [()]

{

    action;

    [return int;]

}
```

说明：
1、可以带 function fun() 定义，也可以直接 fun() 定义,不带任何参数。
2、参数返回，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值。 return 后跟数值 n(0-255
下面的例子定义了一个函数并进行调用：

```
#!/bin/bash
# author:King 老师
# url:www.0voice.com

demoFun(){
    echo "这是我的第一个 shell 函数!"
}
echo "-----函数开始执行-----"
demoFun
echo "-----函数执行完毕-----"
```

输出结果：

```
-----函数开始执行-----
这是我的第一个 shell 函数!
```

```
-----函数执行完毕-----
```

下面定义一个带有 return 语句的函数：

```
#!/bin/bash
# author:零声学院 King 老师
# url:www.0voice.com

funWithReturn(){
    echo "这个函数会对输入的两个数字进行相加运算..."
    echo "输入第一个数字: "
    read aNum
    echo "输入第二个数字: "
    read anotherNum
    echo "两个数字分别为 $aNum 和 $anotherNum !"
    return $(($aNum+$anotherNum))
}
funWithReturn
echo "输入的两个数字之和为 $? !"
```

输出类似下面：

```
这个函数会对输入的两个数字进行相加运算...
输入第一个数字:
1
输入第二个数字:
2
两个数字分别为 1 和 2 !
输入的两个数字之和为 3 !
```

函数返回值在调用该函数后通过 $? 来获得。
注意：所有函数在使用前必须定义。这意味着必须将函数放在脚本开始部分，直至 shell 解释器首次发现它时，才可以使用。调用函数仅使用其函数名即可。

函数参数

在 Shell 中，调用函数时可以向其传递参数。在函数体内部，通过 $n 的形式来获取参数的值，例如，$1 表示第一个参数，$2 表示第二个参数...
带参数的函数示例：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

funWithParam(){
    echo "第一个参数为 $1 !"
```

```
    echo "第二个参数为 $2 !"
    echo "第十个参数为 $10 !"
    echo "第十个参数为 ${10} !"
    echo "第十一个参数为 ${11} !"
    echo "参数总数有 $# 个!"
    echo "作为一个字符串输出所有参数 $* !"
}
funWithParam 1 2 3 4 5 6 7 8 9 34 73
```

输出结果：

```
第一个参数为 1 !
第二个参数为 2 !
第十个参数为 10 !
第十个参数为 34 !
第十一个参数为 73 !
参数总数有 11 个!
作为一个字符串输出所有参数 1 2 3 4 5 6 7 8 9 34 73 !
```

注意，$10 不能获取第十个参数，获取第十个参数需要${10}。当 n>=10 时，需要使用${n}来获取参数。

另外，还有几个特殊字符用来处理参数：

参数处理	说明
$#	传递到脚本的参数个数
$*	以一个单字符串显示所有向脚本传递的参数
$$	脚本运行的当前进程 ID 号
$!	后台运行的最后一个进程的 ID 号
$@	与$*相同，但是使用时加引号，并在引号中返回每个参数。
$-	显示 Shell 使用的当前选项，与 set 命令功能相同。
$?	显示最后命令的退出状态。0 表示没有错误，其他任何值表明有错误。

Shell 输入/输出重定向

大多数 UNIX 系统命令从你的终端接受输入并将所产生的输出发送回到您的终端。一个命

令通常从一个叫标准输入的地方读取输入，默认情况下，这恰好是你的终端。同样，一个命令通常将其输出写入到标准输出，默认情况下，这也是你的终端。
重定向命令列表如下：

命令	说明
command > file	将输出重定向到 file。
command < file	将输入重定向到 file。
command >> file	将输出以追加的方式重定向到 file。
n > file	将文件描述符为 n 的文件重定向到 file。
n >> file	将文件描述符为 n 的文件以追加的方式重定向到 file。
n >& m	将输出文件 m 和 n 合并。
n <& m	将输入文件 m 和 n 合并。
<< tag	将开始标记 tag 和结束标记 tag 之间的内容作为输入。

需要注意的是文件描述符 0 通常是标准输入（STDIN），1 是标准输出（STDOUT），2 是标准错误输出（STDERR）。

输出重定向

重定向一般通过在命令间插入特定的符号来实现。特别的，这些符号的语法如下所示:

```
command1 > file1
```

上面这个命令执行 command1 然后将输出的内容存入 file1。
注意任何 file1 内的已经存在的内容将被新内容替代。如果要将新内容添加在文件末尾，请使用>>操作符。
执行下面的 who 命令，它将命令的完整的输出重定向在用户文件中(users):

```
$ who > users
```

执行后，并没有在终端输出信息，这是因为输出已被从默认的标准输出设备（终端）重定向到指定的文件。
你可以使用 cat 命令查看文件内容：

```
$ cat users
_mbsetupuser console  Oct 31 17:35
```

```
tianqixin    console  Oct 31 17:35
tianqixin    ttys000  Dec  1 11:33
```

输出重定向会覆盖文件内容，请看下面的例子：

```
$ echo "零声学院 www.0voice.com" > users
$ cat users
零声学院 www.0voice.com
$
```

如果不希望文件内容被覆盖，可以使用 >> 追加到文件末尾，例如：

```
$ echo "零声学院 www.0voice.com " >> users
$ cat users
零声学院 www.0voice.com
零声学院 www.0voice.com
$
```

输入重定向

和输出重定向一样，Unix 命令也可以从文件获取输入，语法为：

```
command1 < file1
```

这样，本来需要从键盘获取输入的命令会转移到文件读取内容。
注意：输出重定向是大于号(>)，输入重定向是小于号(<)。
接着以上实例，我们需要统计 users 文件的行数,执行以下命令：

```
$ wc -l users
       2 users
```

也可以将输入重定向到 users 文件：

```
$  wc -l < users
       2
```

注意：上面两个例子的结果不同：第一个例子，会输出文件名；第二个不会，因为它仅仅知道从标准输入读取内容。

```
command1 < infile > outfile
```

同时替换输入和输出，执行 command1，从文件 infile 读取内容，然后将输出写入到 outfile 中。

重定向深入讲解

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

1. 标准输入文件(stdin)：stdin 的文件描述符为 0，Unix 程序默认从 stdin 读取数据。
2. 标准输出文件(stdout)：stdout 的文件描述符为 1，Unix 程序默认向 stdout 输出数据。
3. 标准错误文件(stderr)：stderr 的文件描述符为 2，Unix 程序会向 stderr 流中写入错误信息。

默认情况下，command > file 将 stdout 重定向到 file，command < file 将 stdin 重定向到 file。
如果希望 stderr 重定向到 file，可以这样写：

```
$ command 2 > file
```

如果希望 stderr 追加到 file 文件末尾，可以这样写：

```
$ command 2 >> file
```

2 表示标准错误文件(stderr)。
如果希望将 stdout 和 stderr 合并后重定向到 file，可以这样写：

```
$ command > file 2>&1
或者
$ command >> file 2>&1
```

如果希望对 stdin 和 stdout 都重定向，可以这样写：

```
$ command < file1 >file2
```

command 命令将 stdin 重定向到 file1，将 stdout 重定向到 file2。

Here Document

Here Document 是 Shell 中的一种特殊的重定向方式，用来将输入重定向到一个交互式 Shell 脚本或程序。
它的基本的形式如下：

```
command << delimiter
    document
delimiter
```

它的作用是将两个 delimiter 之间的内容(document) 作为输入传递给 command。
注意：
结尾的 delimiter 一定要顶格写，前面不能有任何字符，后面也不能有任何字符，包括空格和 tab 缩进。
开始的 delimiter 前后的空格会被忽略掉。
在命令行中通过 wc -l 命令计算 Here Document 的行数：

```
$ wc -l << EOF
```

```
    欢迎来到
    零声学院
    www.0voice.com
EOF
3          # 输出结果为 3 行
$
```

我们也可以将 Here Document 用在脚本中，例如：

```
#!/bin/bash
# author: 零声学院
# url:www.0voice.com
cat << EOF
欢迎来到
零声学院
www.0voice.com
EOF
```

执行以上脚本，输出结果：

```
欢迎来到
零声学院
www.0voice.com
```

/dev/null 文件

如果希望执行某个命令，但又不希望在屏幕上显示输出结果，那么可以将输出重定向到 /dev/null：

```
$ command > /dev/null
```

/dev/null 是一个特殊的文件，写入到它的内容都会被丢弃；如果尝试从该文件读取内容，那么什么也读不到。但是 /dev/null 文件非常有用，将命令的输出重定向到它，会起到"禁止输出"的效果。
如果希望屏蔽 stdout 和 stderr，可以这样写：

```
$ command > /dev/null 2>&1
```

注意：0 是标准输入（STDIN），1 是标准输出（STDOUT），2 是标准错误输出（STDERR）。

Shell 文件包含

和其他语言一样，Shell 也可以包含外部脚本。这样可以很方便的封装一些公用的代码作为

一个独立的文件。

Shell 文件包含的语法格式如下：

```
. filename   # 注意点号(.)和文件名中间有一空格
或
source filename
```

创建两个 shell 脚本文件。

test1.sh 代码如下：

```
#!/bin/bash
# author:零声学院
# url:www.0voice.com

url="http://www.0voice.com"
```

test2.sh 代码如下：

```
#!/bin/bash
# author: 零声学院
# url: www.0voice.com

#使用 . 号来引用 test1.sh 文件
. ./test1.sh

# 或者使用以下包含文件代码
# source ./test1.sh

echo "零声学院官网地址：$url"
```

接下来，我们为 test2.sh 添加可执行权限并执行：

```
$ chmod +x test2.sh
$ ./test2.sh
零声学院官网地址：http://www.0voice.com
```

注：被包含的文件 test1.sh 不需要可执行权限。