



Linux shell编程

——控制语句函数数组

信息工程学院

❑ 选择结构

- `if` 条件语句
- `case` 选择语句

❑ 循环结构

- `for` 循环语句
- `while` 循环语句
- `until` 循环语句
- `select` 循环与菜单

❑ 循环控制

- ❑ `break` 语句
- ❑ `continue` 语句

条件测试命令

◆ 测试命令

test condition 或者 **[condition]** 命令进行条件测试。

◆ 条件测试的值

- **0** 表示命令**成功**或表达式为**真**
- **非0** 则表示命令**失败**或表达式为**假**

◆ 条件测试的种类

- **命令成功或失败**
- **表达式为真或假**

\$? 中保存了退出状态的值
与一般高级语言相反

```
[wlw@CM00 ~]$ pwd
/home/wlw
[wlw@CM00 ~]$ echo $?
0
[wlw@CM00 ~]$ PWD
bash: PWD: command not found
[wlw@CM00 ~]$ echo $?
127
[wlw@CM00 ~]$
```

```
[root@localhost sh]# ls
count2.sh count3.sh count4.sh count.sh for.sh hello.sh ip.txt
[root@localhost sh]#
[root@localhost sh]#
[root@localhost sh]# test -e ip.txt
[root@localhost sh]#
[root@localhost sh]# echo $?
0
[root@localhost sh]#
[root@localhost sh]# [ -e abc ]
[root@localhost sh]#
[root@localhost sh]# echo $?
1
```

◆语句

- 格式1: **test** <测试表达式>
- 格式2: **[** <测试表达式> **]**
- 格式3: **[[** <测试表达式> **]]**

[]和[[]]的前后必须有空格

◆说明

- 格式1 和 格式2 是等价的，格式3是扩展的 **test** 命令
- 在 **[[]]** 中可以使用通配符进行模式匹配，能够使用符号**&&**, **||**, **<**, 和**>**
- 在 **[]** 中不能使用符号**&&**, **||**, **<**, 和**>**，若要出现需要转义
- 要对整数进行关系运算也可以使用 **(())** 进行测试

例如，**test -f "\$1"** 也可写成: **[-f "\$1"]**

作用：测试位置参数**\$1**是否是已存在的普通文件

◆ 常用条件测试有以下四种情况:

✓ 文件操作

✓ 数值比较

✓ 字符比较

✓ 逻辑操作

文件类型和文件权限测试

[-f fname]	fname 存在且是普通文件时，返回真（即返回 0）
[-L fname]	fname 存在且是链接文件时，返回真
[-d fname]	fname 存在且是一个目录时，返回真
[-s fname]	fname 存在且大小大于 0 时，返回真
[-e fname]	fname（文件或目录）存在时，返回真
[-r fname]	fname（文件或目录）存在且可读时，返回真
[-w fname]	fname（文件或目录）存在且可写时，返回真
[-x fname]	fname（文件或目录）存在且可执行时，返回真

补充

-b 文件	判断该文件是否存在，并且是否为块设备文件（是块设备文件为真）
-c 文件	判断该文件是否存在，并且是否为字符设备文件（是字符设备文件为真）
-L 文件	判断该文件是否存在，并且是否为符号链接文件（是符号链接文件为真）
-p 文件	判断该文件是否存在，并且是否为管道文件（是管道文件为真）
-u 文件	判断该文件是否存在，并且是否该文件拥有 SUID 权限（有 SUID 权限为真）
-g 文件	判断该文件是否存在，并且是否该文件拥有 SGID 权限（有 SGID 权限为真）
文件 1 -nt 文件 2	判断文件 1 的修改时间是否比文件 2 的新（如果新则为真）
文件 1 -ot 文件 2	判断文件 1 的修改时间是否比文件 2 的旧（如果旧则为真）
文件 1 -ef 文件 2	判断文件 1 是否和文件 2 的 Inode 号一致，可以理解为两个文件是否为同一个文件。这个判断用于判断硬链接是很好的方法

例1：测试“/etc/hosts”是否是文件，并通过“\$?”变量查看返回状态值，据此判断测试结果。

```
# [ -f /etc/hosts ]; echo $?
```

0

//返回值为0，表示上一步测试的条件成立。

例2：测试“/media/cdrom/Server”及其父目录是否存在，如果存在则显示“YES”否则显示NO。

```
# [ -e /media/cdrom/Server ] && echo "YES" || echo "NO"
```

NO

数值测试

[int1 -eq int2]	int1 等于 int2 返回真
[int1 -ne int2]	int1 不等于 int2 返回真
[int1 -gt int2]	int1 大于 int2 返回真
[int1 -ge int2]	int1 大于或等于 int2 返回真
[int1 -lt int2]	int1 小于 int2 返回真
[int1 -le int2]	int1 小于或等于 int2 返回真
[[int1 -eq int2]]	int1 等于 int2 返回真
[[int1 -ne int2]]	int1 不等于 int2 返回真
[[int1 -gt int2]]	int1 大于 int2 返回真
[[int1 -ge int2]]	int1 大于或等于 int2 返回真
[[int1 -lt int2]]	int1 小于 int2 返回真
[[int1 -le int2]]	int1 小于或等于 int2 返回真
((int1==int2))	int1 等于 int2 返回真
((int1!=int2))	int1 不等于 int2 返回真
((int1>int2))	int1 大于 int2 返回真
((int1>=int2))	int1 大于或等于 int2 返回真
((int1<int2))	int1 小于 int2 返回真
((int1<=int2))	int1 小于或等于 int2 返回真

[]和[[]]
以及操作符
两边必须留
空格!

(())两边的
空格和操作符
两边的空格都
可省略!

- ◆ **[root@localhost ~]# [-1 -gt -2] && echo yes || echo no**
yes

```
[root@localhost ~]# (( 8 > 3 ));echo $?
```

```
0
```

```
[root@localhost ~]# ((8>3));echo $?
```

```
0
```

```
[root@localhost ~]# [[ 8 > 3 ]];echo $?
```

```
0
```

```
[root@localhost ~]# [[ 8>3 ]];echo $?
```

```
-bash: unexpected token 284 in conditional command
```

```
-bash: syntax error near `8>'
```

```
[root@localhost ~]# [[8 > 3]];echo $?
```

```
bash: [[8: command not found...
```

```
127
```

[-z string]	如果字符串string长度为0，返回真
[-n string]	如果字符串string长度不为0，返回真
[str1 == str2]	两字符串相等（也可使用 = ）返回真
[str1 != str2]	两字符串不等返回真
[[str1 == str2]]	两字符串相同返回真
[[str1 != str2]]	两字符串不相同返回真
[[str1 =~ str2]]	str2是str1的子串返回真
[[str1 > str2]]	str1大于str2返回真
[[str1 < str2]]	str1小于str2返回真

字符串按从左到右对应字符的**ASCII**码进行比较
[[]]中>和<不需要转义

- ◆ **[root@localhost ~]# [-z "\$name"] && echo "yes" || echo "no"**
yes
- ◆ **[root@localhost ~]# name=cos**
[root@localhost ~]# [-z "\$name"] && echo "yes" || echo "no"
[root@localhost ~]# no

```
[root@localhost ~]# [[ str1 =~ str2 ]];echo $?
```

```
1
```

```
[root@localhost ~]# [[ str1=~str ]];echo $?
```

```
0
```

```
[root@localhost ~]# [[str1=~str]];echo $?
```

```
bash: [[str1=~str]]: command not found...
```

```
127
```

```
[root@localhost ~]# [[str1 =~ str]];echo $?
```

```
bash: [[str1: command not found...
```

```
127
```

```
[root@localhost ~]# [ str1 =~ str ];echo $?
```

```
-bash: [: =~: binary operator expected
```

```
2
```

[expr1 -a expr2]	逻辑与，都为真时，结果为真
[expr1 -o expr2]	逻辑或，有一个为真时，结果为真
[! expr]	逻辑非，expr为假时，结果为真

[[pattern1 && pattern2]]	逻辑与
[[pattern1 pattern2]]	逻辑或
[[! pattern]]	逻辑非

((expr1 && expr2))	逻辑与
((expr1 expr2))	逻辑或
((! expr))	逻辑非

说明：-a、-o只能出现在[]内，[[]]、(())内要改用&&、||

例：展示测试语句应用。

```
$ cat exam10
```

```
echo -n 'key in a number (1-10) : '
```

```
read a
```

```
if [ "$a" -lt 1 -o "$a" -gt 10 ]
```

```
then echo "Error Number."
```

```
    exit 2
```

```
elif [ ! "$a" -lt 5 ]
```

```
then echo "It's not less 5. "
```

```
else echo "It's less 5. "
```

```
fi
```

```
echo "accept key in value."
```

#提示输入1-10之间的一个数字

#读取输入的数字

#如果该数小于1或者大于10

#显示输入数字有错

#退出，返回码为2

#否则，若该数不小于5

#显示不小于5 的信息

#否则，显示该数小于5

#结束if语句

#显示接受了键入的值

test 命令非常强大，但是很难满足其转义需求以及字符串和算术比较之间的区别。

- **[expr]** 和 **test expr** 是等价的，因为**shell** 也用 **<** 和 **>** 两个操作符进行重定向，所以在 **[expr]** 内必须用 **\<** 或 **\>** 加以转义，并且操作符两边要加空格，否则会认为是一个字符串。
- **[[]]** 可以对文件名和字符串使用更自然的语法。在使用 **=** 或 **!=** 两个操作符时，**[[]]**还能在字符串上进行模式匹配即使用通配符。甚至还可以在**[[]]**内执行算术测试，但是千万要小心**<** 和 **>** 操作符会把操作数当成字符串比较，除非在 **(())**内。
- **(())**用于计算算术表达式，如果表达式求值为 **0**，则设置退出状态为 **1**；如果求值为非 **0** 值，则设置为 **0**。不需要对 **((** 和 **))** 之间的操作符转义。算术只对整数进行。除**0**会产生错误，但不会产生溢出。可以执行 **C** 语言中常见的算术、逻辑和位操作。

总之，**[]**内有些字符要转义；**[[]]**常用于字符串的比较；**(())**常用于整数的比较。



思考题



例：测试当前的用户是否是**teacher**，若不是则提示” **Not teacher** “。

```
[root@localhost ~]# [ $USER == "teacher" ] || echo "Not teacher"
```

例：测试” **/etc/profile** “文件是否有可执行权限，若确实没有可执行权限，则提示” **No x mode.** “的信息。

```
[root@localhost ~]# [ ! -x "/etc/profile" ] && echo "No x mode."
```

条件测试举例

```
$vi contest.sh
#!/bin/bash
User=may
grep ^$User /etc/passwd
echo $?
grep ^$User /etc/passwd > /dev/null \
&& echo "$User is a user in /etc/passwd." \
|| echo "$User isn't a user in /etc/passwd."
```

```
$ bash contest.sh
may:x:1000:1000:may:/home/may:/bin/bash
0
may is a user in /etc/passwd.
```

说明:

- 1./dev/null是空设备，，常用于当不想把标准输出和标准错误输出信息输出到终端时丢垃圾桶中，类型于**window**的回收站，不同的是这个设备有去无回。
- 2.^表示行的开头，\$表示行的结尾，‘^\$’表示空行。

程序控制结构——if语句

if语句一般格式为: (if和fi必须成对出现)

```
if [ 测试条件 ]  
then 命令1  
else 命令2  
fi
```

```
if [ 测试条件 ]; then  
命令1  
else 命令2  
fi
```

- ◆ if 语句可以嵌套使用, else 最多只能有一个
- ◆ 条件测试表达式也可以是多个命令, 以最后一个命令的退出状态为条件值。

程序控制结构——if语句

- ◆ if 语句的**else**部分还可以是**else-if**结构。

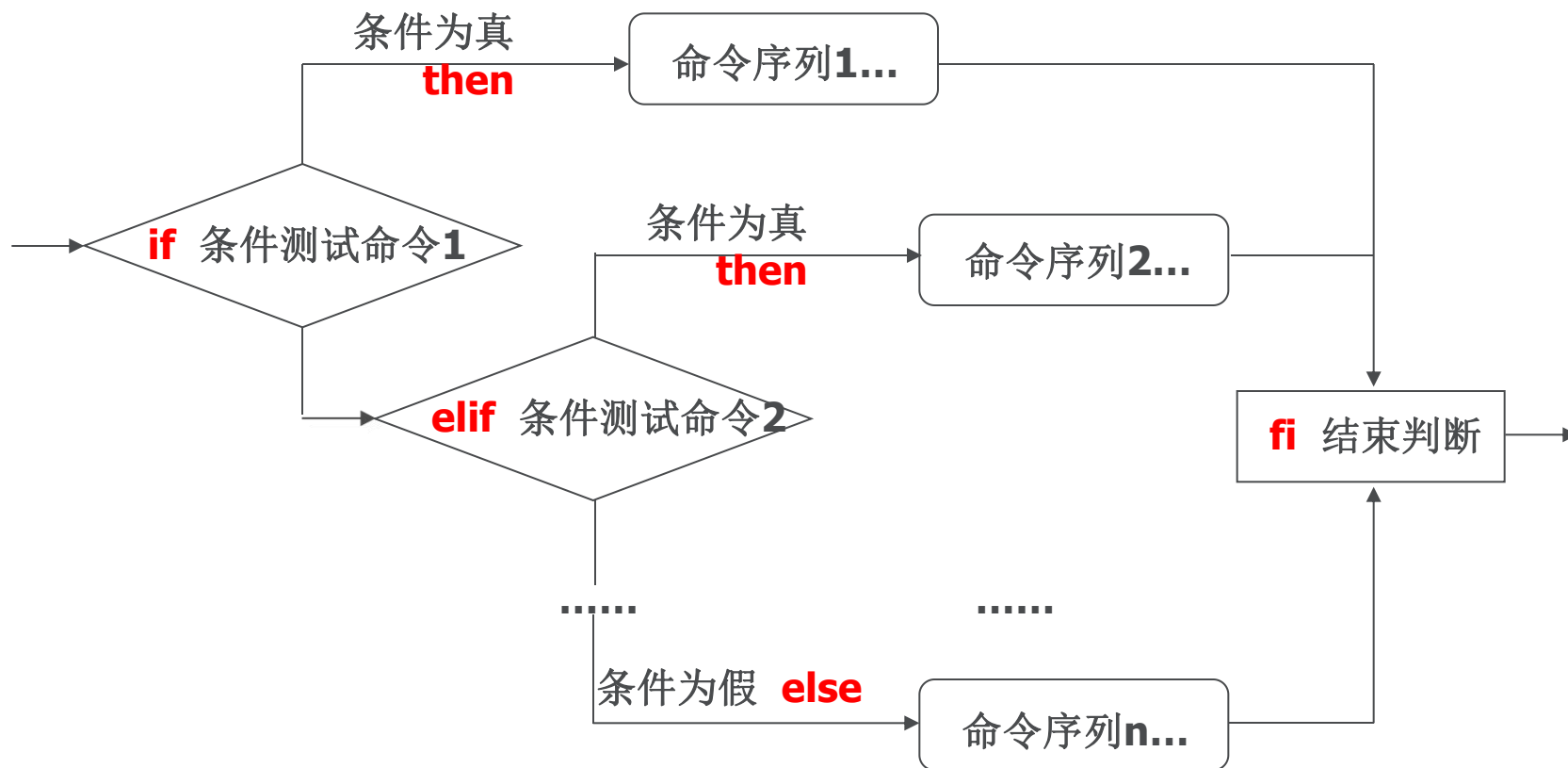
如: **if** [-f "\$1"]
 then echo \$1
 else if [-d "\$1"]
 then (cd \$1 ; ls -l)
 else echo "\$1 is neither a file nor a directory."
 fi
 fi

(注意：比上面的写法少了一个**fi**)

- ◆ 也可用关键字“**elif**”代替“**else if**”。

如: **if** [-f "\$1"]
 then echo \$1
 elif [-d "\$1"]
 then (cd \$1 ; ls -l)
 else echo "\$1 is neither a file nor a directory."
 fi

◆ 针对多个条件执行不同操作



◆ 多重比较时，elif可以有任意多个，else if只能出现一次

if 语句实例

```
[may@localhost ~]$ cat testif.sh
#!/bin/bash
if [ $1 -le 10 ];then
    echo "a<=10"
    elif [ $1 -le 20 ];then
        echo "10<a<=20"
    else
        echo "a>20"
fi
[may@localhost ~]$ . testif.sh 10
a<=10
[may@localhost ~]$ . testif.sh 20
10<a<=20
[may@localhost ~]$ . testif.sh 30
a>20
[may@localhost ~]$
```

常见错误：当if与[之间无空格时，运行结果如下：

```
[may@localhost ~]$ . testif.sh 56
bash: if[ 56 -le 10 ]: command not found...
-bash: testif.sh: line 3: syntax error near unexpected token `then'
-bash: testif.sh: line 3: `then'
[may@localhost ~]$ _
```

if单分支实例

例：对某系统硬盘占用率超过**80%**时报警。

查看当前分区和目录的挂载情况
命令：**df** [参数] [分区号/装载点]
参数：**-h/-H** 以**KB、MB、GB**等单位输出
(以**1000B**为**1KB**，而非**1024B**)

```
[root@localhost sh]# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda3        19G   2.1G   16G   12% /
tmpfs            495M     0   495M    0% /dev/shm
/dev/sda1        190M    33M   147M   19% /boot
```

```
[root@localhost sh]#
```

```
[root@localhost sh]# df -h | grep /dev/sda3
/dev/sda3        19G   2.1G   16G   12% /
```

```
[root@localhost sh]#
```

```
[root@localhost sh]# df -h | grep /dev/sda3 | awk '{print $5}' #awk无条件输出第5列
12%
```

```
[root@localhost sh]#
```

```
[root@localhost sh]# df -h | grep /dev/sda3 | awk '{print $5}' | cut -d "%" -f 1
12
```

#cut提取以**%**为分隔符的第一列

为什么不先提取列?

```
[root@localhost sh]# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda3        19G   2.1G   16G   12% /
tmpfs            495M    0   495M    0% /dev/shm
/dev/sda1        190M   33M   147M   19% /boot
```

```
[root@localhost sh]# df -h | awk '{print $5}'
Use%
12%
0%
19%
```

如何区分是哪个系统的使用率?

```
[root@localhost sh]# df -h | grep "dev/sda3"
/dev/sda3        19G   2.1G   16G   12% /
```

```
[root@localhost sh]#
```

```
[root@localhost sh]#
```

```
[root@localhost sh]# df -h | grep "dev/sda3" | awk '{print $5}'
12%
```

```
[root@localhost sh]# df -h | grep "dev/sda3" | awk '{print $5}' | cut -d "%" -f 1
12
```

先过滤出系统的行，再提取相应列，得到一个带%的数，判断时只要数字，再%作分隔符提取前面的数字，则可以将运行结果送到变量去判断运行。

- ◆ 例：对某系统硬盘占用率超过**80%**时报警。

```
rate=$(df -h | grep "/dev/sda3" | awk ' {print $5}' | cut -d "%" -f1)
#把根分区使用率作为变量值赋予变量 rate
if [ $rate -ge 80 ]
#判断 rate 的值如果大于等于 80，则执行 then 程序
    then
        echo "Warning! /dev/sda3 is full!!"
        #打印警告信息。在实际工作中，也可以向管理员发送邮件。
    fi
```

#测试时，将程序中的**80**改为**10**调试即可。

补充：awk编程格式

```
[root@localhost ~]# awk '条件 1{动作 1} 条件 2{动作 2} ...' 文件名
```

条件（Pattern）：一般使用关系表达式作为条件。

动作（Action）：格式化输出

流程控制语句

条件的类型	条 件	说 明
awk 保留字	BEGIN	在 awk 程序一开始时，尚未读取任何数据之前执行。BEGIN 后的动作只在程序开始时执行一次
	END	在 awk 程序处理完所有数据，即将结束时执行。END 后的动作只在程序结束时执行一次
关系运算符	>	大于
	<	小于
	>=	大于等于
	<=	小于等于
	==	等于。用于判断两个值是否相等，如果是给变量赋值，请使用“=”号
	!=	不等于
	A~B	判断字符串 A 中是否包含能匹配 B 表达式的子字符串
	A!~B	判断字符串 A 中是否不包含能匹配 B 表达式的子字符串
正则表达式	/正则/	如果在“//”中可以写入字符，也可以支持正则表达式



补充：awk编程——比cut强大，能识别空格分隔符

```
[root@localhost ~]# awk '条件 1{动作 1} 条件 2{动作 2} ...' 文件名
```

条件（Pattern）：一般使用关系表达式作为条件。

动作（Action）： 格式化输出

流程控制语句

格式化输出：一般用printf和print。

因为print在linux中不能直接运行，**一般常用printf**。

printf在管道后无法识别，因此，它一般都用于awk中。

```
[root@localhost ~]# printf '输出类型输出格式' 输出内容
```

输出类型：

%ns: 输出字符串。n 是数字指代输出几个字符

%ni: 输出整数。n 是数字指代输出几个数字

%m.nf: 输出浮点数。m 和 n 是数字，指代输出的整数位数和小数位数。如%8.2f
代表共输出 8 位数，其中 2 位是小数，6 位是整数。

输出格式：

\a: 输出警告声音

\b: 输出退格键，也就是 Backspace 键

\f: 清除屏幕

\n: 换行

\r: 回车，也就是 Enter 键

\t: 水平输出退格键，也就是 Tab 键

\v: 垂直输出退格键，也就是 Tab 键

补充：awk编程——printf格式化输出

```
[root@localhost ~]# awk '条件1{动作1} 条件2{动作2}...' 文件名
```

条件 (Pattern)：一般使用关系表达式作为条件。

动作 (Action)：格式化输出

流程控制语句

例：无条件输出**student.txt**文件中的第二和第六列。

```
[root@localhost sh]# awk '{printf $2 "\t" $6 "\n"}' student.txt
```

```
Name    Average
Lm       87.66
Sc       85.66
Tg       91.66
```

I

```
[root@localhost sh]# cat student.txt
ID      Name    PHP    Linux  MySQL  Average
1       Liming  82     95     86     87.66
2       Sc      74     96     87     85.66
3       Tg      99     83     93     91.66
```

说明：1、在awk的条件动作作用单引号 ‘ ’ 括起来，因此，printf或print中的输出格式要用双引号 “ ” 括起来。

2、print不用加\n会自动换行，printf则需要加\n才行。

```
[root@localhost sh]# awk '{print $2 "\t" $6 }' student.txt
```

```
Name    Average
Lm       87.66
Sc       85.66
Tg       91.66
```

```
[root@localhost sh]# awk '{printf $2 "\t" $6}' student.txt
```

```
Name    AverageLm      87.66Sc 85.66Tg 91.66 [root@localhost sh]# |
```

补充: **cut** 提取列 (**grep** 提取行)

◆ **cut** [选项] 文件名

- **-f** 列号1,列号n //提取第几列
- **-d** 分隔符 //按指定分隔符分隔列（默认为**TAB**键，不识别空格）
- **-c** 字符范围 //按字符范围来进行字段提取。“n-”从第n个字符到行尾；“-m”从第1个字符到第m个字符；“n-m”从第n个字符到第m个字符，行首为0。

```
[root@localhost sh]# cat student.txt
ID      Name      gender  Mark
1       Liming    M       86
2       Sc        M       90
3       Tg        M       83
```

说明：文本文件中的
字段分隔要用
TAB键，**cut**不识别
空格

```
[root@localhost sh]# cut -f 2 student.txt
Name
Liming
Sc
Tg
```

补充：cut 实例

例：提取普通用户名。（**/etc/passwd**中分隔符是“：”）

```
[root@localhost sh]# cut -d ":" -f 1,3 /etc/passwd
root:0
bin:1
daemon:2[
adm:3
lp:4
sync:5
shutdown:6
halt:7
mail:8
```

这提取了第一列，但没法判断是否是普通用户。

```
[root@localhost sh]# grep "/bin/bash" /etc/passwd | grep -v "root"
user1:x:500:500::/home/user1:/bin/bash
user2:x:501:501::/home/user2:/bin/bash
[root@localhost sh]#
[root@localhost sh]# grep "/bin/bash" /etc/passwd | grep -v "root" | cut -d ":" -f 1
user1
user2
```

这先过滤掉系统用户，再过滤**root**用户的信息，即留下普通用户，再提取第一列即用户名。

◆ 可执行命令语句块，如果为空，则使用空命令“：”即冒号，该命令不做任何事情，只返回一个退出状态 0。

(1) : 表示不做任何事情，只起占位的作用，其退出值为0。

如：if 条件测试
then
:
fi

(2) true 表示总为真，其退出值总是0。

(3) false 表示总为假，其退出值是255。

如：死循环的写法1：

while true; do commands; done

死循环的写法2：

while :;do commands; done

例：统计当前登录到系统中的用户数量，并判断是否超过三个，若是则显示实际数量并给出警告信息，否则列出登录的用户账号名及所在终端。

```
#!/bin/bash
UserNum=`who | wc -l`
if [ $UserNum -gt 3 ];
then
    echo "Alert , too many login users ( Total: $UserNum )."
else
    echo "Login users:"
    who | awk '{print $1,$2}'
fi
```

```
[may@localhost ~]$ who
may      tty1      2017-03-30 15:23
root     tty2      2017-03-30 15:33
[may@localhost ~]$ who | wc -l
2
[may@localhost ~]$ who | awk '{print $1,$2}'
may tty1
root tty2
[may@localhost ~]$
```

```
[may@localhost ~]$ . whowc.sh
login users.
may tty1
```

补充：if双分支实例

例：备份mysql数据库。

```
#!/bin/bash
#备份 mysql 数据库。

ntpdate asia.pool.ntp.org &>/dev/null
#同步系统时间

date=$(date +%y%m%d)
#把当前系统时间按照“年月日”格式赋予变量 date

size=$(du -sh /var/lib/mysql)
#统计 mysql 数据库的大小，并把大小赋予 size 变量
```

(此方法的问题：**1**、备份在一块硬盘里；**2**、非增量和实时备份，是完全备份；**3**、要求数据库版本相同。由于目前已学知识的限制，此代码可供学习并后继完善。)

```
if [ -d /tmp/dbbak ]
#判断备份目录是否存在，是否为目录
then
    [ #如果判断为真，执行以下脚本
    echo "Date : $date!" > /tmp/dbbak/dbinfo.txt
    #把当前日期写入临时文件
    echo "Data size : $size" >> /tmp/dbbak/dbinfo.txt
    #把数据库大小写入临时文件
    cd /tmp/dbbak

    #进入备份目录
    tar -zcf mysql-lib-$date.tar.gz /var/lib/mysql dbinfo.txt &>/dev/null
    #打包压缩数据库与临时文件，把所有输出丢入垃圾箱（不想看到任何输出）
    rm -rf /tmp/dbbak/dbinfo.txt
    #删除临时文件
else
    mkdir /tmp/dbbak
    #如果判断为假，则建立备份目录
    echo "Date : $date!" > /tmp/dbbak/dbinfo.txt
    echo "Data size : $size" >> /tmp/dbbak/dbinfo.txt
    #把日期和数据库大小保存如临时文件
    cd /tmp/dbbak
    tar -zcf mysql-lib-$date.tar.gz dbinfo.txt /var/lib/mysql &>/dev/null
    #压缩备份数据库与临时文件
    rm -rf /tmp/dbbak/dbinfo.txt
    #删除临时文件
fi
```

补充：if双分支实例

在工作中，服务器上的服务经常宕机而管理员却不知道，会造成服务中断。因此写一脚本监听本机的服务，如果服务停止或宕机，可以自动重启服务。以**apache**服务举例，判断**apache**服务是否启动，如果没有启动则自动启动。

```
[root@localhost sh]# netstat -tuln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:47365           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 :::48651                :::*                    LISTEN
tcp        0      0 :::111                  :::*                    LISTEN
tcp        0      0 :::80                   :::*                    LISTEN
tcp        0      0 :::22                   :::*                    LISTEN
tcp        0      0 :::1:631                :::*                    LISTEN
tcp        0      0 :::1:25                 :::*                    LISTEN
udp        0      0 0.0.0.0:111            0.0.0.0:*
```

```
[root@localhost sh]# netstat -tuln | awk '{print $4}'
(only
Local
0.0.0.0:47365
0.0.0.0:111
0.0.0.0:22
127.0.0.1:631
127.0.0.1:25
:::48651
:::111
:::80
:::22
:::1:631
:::1:25
0.0.0.0:111
```

```
[root@localhost sh]# netstat -tuln | awk '{print $4}' | grep ":80$"
```

提取出**80**端口作判断即可

补充：if双分支实例

例：判断apache服务是否启动，如果没有启动则自动启动。

方法一：监听80端口来判断。

```
#!/bin/bash

aa=$( netstat -tuln | awk ' {print $4}' | grep ":80$" )

if [ "$aa" == "" ]
then
    echo "httpd is down ,must restart"
    /etc/rc.d/init.d/httpd start &>/dev/null
else
    echo "httpd is ok"
fi
```

```
[root@localhost sh]# ./if2.sh
httpd is ok
```

补充：if双分支实例

```
[root@localhost sh]# service httpd stop
停止 httpd: [确定]
[root@localhost sh]# netstat -tuln
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:47365           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 :::48651                :::*                     LISTEN
tcp        0      0 :::111                  :::*                     LISTEN
tcp        0      0 :::22                   :::*                     LISTEN
tcp        0      0 :::1:631                :::*                     LISTEN
tcp        0      0 :::1:25                  :::*                     LISTEN
udp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
udp        0      0 0.0.0.0:34933           0.0.0.0:*               LISTEN
udp        0      0 0.0.0.0:631             0.0.0.0:*               LISTEN
udp        0      0 0.0.0.0:1018            0.0.0.0:*               LISTEN
udp        0      0 127.0.0.1:659           0.0.0.0:*               LISTEN
udp        0      0 :::111                  :::*                     LISTEN
udp        0      0 :::1018                  :::*                     LISTEN
udp        0      0 :::55616                 :::*                     LISTEN
[root@localhost sh]# ./if2.sh
httpd is down ,must restart
```

关闭之后用netstat查看无80，运行之后再用netstat查看即可。



补充：if双分支实例



例：判断apache服务是否启动，如果没有启动则自动启动。

方法二：扫描服务器状态来判断。即当端口还在，但访问拥挤时用nmap端口扫描命令，80有应答更准确，提取出http的状态是open即可。

```
[root@localhost sh]# nmap -sT 192.168.44.8

Starting Nmap 5.51 ( http://nmap.org ) at 2017-02-19 07:43 CST
Nmap scan report for 192.168.44.8
Host is up (0.0010s latency).
Not shown: 997 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
80/tcp    open  http
111/tcp   open  rpcbind

Nmap done: 1 IP address (1 host up) scanned in 0.29 seconds
[root@localhost sh]# nmap -sT 192.168.44.8 | grep tcp
22/tcp    open  ssh
80/tcp    open  http
111/tcp   open  rpcbind
[root@localhost sh]# nmap -sT 192.168.44.8 | grep tcp | grep http
80/tcp    open  http
[root@localhost sh]# nmap -sT 192.168.44.8 | grep tcp | grep http | awk '{print $3}'
http
[root@localhost sh]# nmap -sT 192.168.44.8 | grep tcp | grep http | awk '{print $2}'
open
```

补充：if双分支实例

例：判断apache服务是否启动，如果没有启动则自动启动。

方法二：扫描服务器状态来判断。即当端口还在，但访问拥挤时用nmap端口扫描命令，80有应答更准确，提取出http的状态是open即可。

```
#!/bin/bash
port=$(nmap -sT 192.168.4.210 | grep tcp | grep http | awk '{print $2}')
#使用 nmap 命令扫描服务器，并截取 apache 服务的状态，赋予变量 port
if [ "$port" == "open" ]
#如果变量 port 的值是 "open"
then
    echo "$(date) httpd is ok!" >> /tmp/autostart-acc.log
    #则证明 apache 正常启动，在正常日志中写入一句话即可
else
    /etc/rc.d/init.d/httpd start &>/dev/null
    #否则证明 apache 没有启动，自动启动 apache
    echo "$(date) restart httpd !!" >> /tmp/autostart-err.log
    #并在错误日志中记录自动启动 apache 的时间
fi
```

补充：if多分支实例

例：判断用户的输入是否是文件？

```
[root@localhost sh]# ls
abc          count3.sh  count.sh  hello.sh  if2.sh  ip.txt  para2.sh  student.txt
count2.sh  count4.sh  for.sh    if1.sh    if3.sh  mail.txt para.sh    test.txt
[root@localhost sh]#
[root@localhost sh]# ./if3.sh
please input a filename: abc
abc is putong
[root@localhost sh]#
[root@localhost sh]#
[root@localhost sh]# ./if3.sh
please input a filename: /root
/root is dire
[root@localhost sh]# ./if3.sh
please input a filename: /dev/sda3
/dev/sda3 is other file
[root@localhost sh]#
```

```
[root@localhost sh]# ./if3.sh
please input a filename:
11111111111111111111
[root@localhost sh]# echo $?
101
[root@localhost sh]#
[root@localhost sh]#
[root@localhost sh]# ./if3.sh
please input a filename: dgasgasdgsa
22222222222222222222
[root@localhost sh]# echo $?
102
[root@localhost sh]#
```

```
#!/bin/bash

read -t 30 -p "please input a filename: " filename

if [ -z $filename ]
then
    echo "11111111111111111111"
    exit 101
elif [ ! -e $filename ]
then
    echo "22222222222222222222"
    exit 102
elif [ -f $filename ]
then
    echo "$filename is putong"
elif [ -d $filename ]
then
    echo "$filename is dire"
else
    echo "$filename is other file"
fi
```

程序控制结构——case语句

case语句允许进行多重选择。

其一般语法形式是：

```
case    表达式    in
    模式字符串1)    命令
                        ...
                        命令;;
    模式字符串2)    命令
                        ...
                        命令;;
        ...
                *)    命令
                        ...
                        命令;;

esac
```

◆ 表达式按顺序匹配每个模式，一旦有一个模式匹配成功，则执行该模式后面的所有命令，然后退出 **case**。

◆ 表达式没有找到匹配的模式，则执行缺省值 “***)**” 后面的命令块； “***)**” 可以不出现。

◆ 每个模式字符串后面可有一条或多条命令，其最后一条命令必须以**两个分号(即;;)**结束。

◆ 各模式字符串应是唯一的，不应重复出现。并且要合理安排它们的出现顺序。

case语句允许进行多重选择。

其一般语法形式是：

```
case    表达式    in
    模式字符串1)    命令
                    ...
                    命令;;
    模式字符串2)    命令
                    ...
                    命令;;
    ...
                    *)    命令
                    ...
                    命令;;

esac
```

◆ 模式字符串中通配符可以含有通配符和“|”。

如果一个模式字符串中包含多个模式以竖线（|）隔开，表示“或”的关系，即只要给定表达式与其中一个模式相配，就会执行其后的命令表。

```
case $yn in
    [Yy] | [Yy][Ee][Ss] ) echo "agree." ;;
    [Nn] | [Nn][Oo]      ) echo "not agree." ;;
    * ) echo "invalid" ;;
esac
```

◆ case的退出（返回）值是整个结构中最后执行的那个命令的退出值。若没有执行任何命令，则退出值为0。

分支结构—case 语句举例

```
#!/bin/bash
echo -n "please input: "
read yn
case $yn in
    [Yy] | [Yy][Ee][Ss] ) echo "agree." ;;
    [Nn] | [Nn][Oo]      ) echo "not agree." ;;
    *                    ) echo "invalid" ;;
esac
```

```
[may@localhost ~]$ . case.sh
please input:yes
agree
[may@localhost ~]$ . case.sh
please input:no
not agree
[may@localhost ~]$ . case.sh
please input:Yes
agree
[may@localhost ~]$ . case.sh
please input:AAFG
invalid
```

程序控制结构——for语句

使用方式主要有两种：值表方式、算术表达式方式。

算术表达式方式一般格式是：

```
for ((expr1;expr2;expr3))  
do  
    commands  
done
```

```
#!/bin/bash  
for ((i=1;i<=5;i++))  
do  
    echo $i  
done
```

- ◆ 首先执行 **expr1**，一般是**算数表达式**，仅在循环开始之初执行一次
- ◆ 执行 **expr2**，一般是**逻辑表达式**，在每次执行循环体之前执行一次
 - 其值为假时，终止循环
 - 其值为真时，执行**do**和**done**之间的 **commands**
- ◆ 执行**expr3**，一般是**算数表达式**，在每次执行循环体之后执行一次
- ◆ 进入下一次循环

程序控制结构——for语句

使用方式主要有两种：值表方式、算术表达式方式。

值表方式一般格式是：

```
for 变量 in 值表
do
    命令表
done
```

```
#!/bin/bash
for i in 1 2 3 4 5
do
    echo $i
done
```

- ◆ 将值表的每一项依次赋给变量，执行do和done之间的命令
- ◆ 如此循环，直到值表中的所有项值都已经用完，循环次数取决于值表中单词的个数。
- ◆ 值表可以是命令替换、变量名替换、字符串和文件名列表(可包含通配符)，每个列表项以空格间隔。
- ◆ 可以省略in 值表，省略相当于in "\$@"，所有位置参数

for循环举例

用**for**循环实现**1~100**累加，并显示结果。

```
#!/bin/bash

s=0
for ((i=1;i<=100;i++))
do
    s=$((s+i))
done
echo "The sum of 1..100 is $S."
```

```
#!/bin/bash

s=0
for i in {1..100}
do
    s=$((s+i))
done
echo "The sum of 1..100 is $S."
```

- 数值型更适合于数值运算类；
- 值表型更适于运维管理，除了循环还要将循环内容赋给变量。
- 值表型比数值型更常用。

for循环举例-用数值型for(())

用for循环实现批量解压缩。（设在~/sh/tar目录下已存若干压缩包）

```
[root@localhost tar]# ls
apr-1.4.6.tar.gz      httpd-2.4.7.tar.gz  php-5.6.15.tar.gz
apr-util-1.4.1.tar.gz mysql-5.5.23.tar.gz phpMyAdmin-4.1.4-all-languages.tar.gz
```

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

```
cd /root/sh/tar
```

```
ls *.tar.gz > tar.log
```

```
ls *.tgz >> tar.log &> /dev/null
```

```
aa=$(cat tar.log|wc -l)
```

```
for (( i=1;i<="$aa";i=i+1 ))
```

```
do
```

```
bb=$(cat tar.log | awk 'NR== '$i' {print $1}' )
```

```
tar -zxvf $bb -C /root/sh/tar
```

```
done
```

将ls列出的目录下的压缩文件名重定向写入tar.log文件中，一行一个文件名

用wc -l命令统计tar.log文件中的行数，即压缩文件个数，即得到循环次数给aa

用awk的NR行数一行一行处理，提取出每一行的第一列字符串给bb，即文件名

解压到指定目录

for循环举例-用值表型for in

用**for**循环实现批量解压缩。（设在~/sh/tar目录下已存若干压缩包）

```
[root@localhost tar]# ls  
apr-1.4.6.tar.gz      httpd-2.4.7.tar.gz  php-5.6.15.tar.gz  
apr-util-1.4.1.tar.gz mysql-5.5.23.tar.gz phpMyAdmin-4.1.4-all-languages.tar.gz
```

```
#!/bin/bash  
  
cd /root/sh/tar/  
  
ls *.tar.gz > tar.log  
ls *.tgz >> tar.log &>/dev/null  
  
for i in $(cat tar.log)  
do  
    tar -zxvf $i  
done
```

不需要统计文件个数，提取文件名。此值表形式比数值形式在shell运维管理中更常用

实验：for循环举例-用数值型for(())

用for循环实现批量添加用户。

```
#!/bin/bash
# 成批添加50个用户

for (( n=1; n<=50; n++ ))
do
    useradd user${n}
    echo "123456"|passwd --stdin user${n}
    chage -d 0 user${n}
done
```

说明：

1. **echo <newword> | passwd --stdin <user> #设置用户的新密码**
2. **chage -d 0 <user> #强制用户登陆时修改口令**

实验：for循环举例-用值表型for in

用for循环实现批量添加用户。

```
#!/bin/bash
# 成批添加50个用户

for n in {1..50}    #或`seq 1 50`或$(seq 1 50)
do
    useradd user${n}
    echo "123456"|passwd --stdin user${n}
    chage -d 0  user${n}
done
```

说明：

1. **seq** 命令用于产生从某个数到另外一个数之间的所有整数



实验补充：for循环举例-用数值型for(())



用for循环实现批量添加指定数量的用户。

```
#!/bin/bash
#批量添加指定数量的用户
```

```
read -p "Please input user name: " -t 30 name
#让用户输入用户名，把输入保存入变量name
read -p "Please input the number of users: " -t 30 num
#让用户输入添加用户的数量，把输入保存入变量num
read -p "Please input the password of users: " -t 30 pass
#让用户输入初始密码，把输入保存如变量pass
```

```
if [ ! -z "$name" -a ! -z "$num" -a ! -z "$pass" ]
#判断三个变量不为空
```

```
then
y=$(echo $num | sed 's/[0-9]//g')
#定义变量的值为后续命令的结果
#后续命令作用是，把变量num的值替换为空。如果能替换为空，证明num的值为数字
#如果不能替换为空，证明num的值为非数字。我们使用这种方法判断变量num的值为数字
```

```
if [ -z "$y" ]
#如果变量y的值为空，证明num变量是数字
```

```
then
for (( i=1;i<=$num;i=i+1 ))
#循环num变量指定的次数
```

```
do
/usr/sbin/useradd $name$i &>/dev/null
#添加用户，用户名为变量name的值加变量i的数字
echo $pass | /usr/bin/passwd --stdin $name$i &>/dev/null
#给用户设定初始密码为变量pass的值
chage -d 0 $name$i &>/dev/null
#强制用户登录之后修改密码
```

```
done
```

```
fi
```

```
fi
```

```
[root@localhost sh]# ./useradd.sh
Please input user name: test
Please input the number of users: 5
Please input the password of users: 123
[root@localhost sh]#
test1:x:502:502::/home/test1:/bin/bash
test2:x:503:503::/home/test2:/bin/bash
test3:x:504:504::/home/test3:/bin/bash
test4:x:505:505::/home/test4:/bin/bash
test5:x:506:506::/home/test5:/bin/bash
```

实验补充: for循环举例-用值表型for in

用for循环实现批量删除普通用户。

```
#!/bin/bash

name=$( cat /etc/passwd | grep "/bin/bash" | grep -v root | cut -d ":" -f 1)

for i in $name
do
    userdel -r $i &>/dev/null
done
```

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash" | grep -v root | cut -d ":" -f 1
```

```
user1
user2
test1
test2
test3
test4
test5
```

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash"
```

```
root:x:0:0:root:/root:/bin/bash
user1:x:500:500::/home/user1:/bin/bash
user2:x:501:501::/home/user2:/bin/bash
test1:x:502:502::/home/test1:/bin/bash
test2:x:503:503::/home/test2:/bin/bash
test3:x:504:504::/home/test3:/bin/bash
test4:x:505:505::/home/test4:/bin/bash
test5:x:506:506::/home/test5:/bin/bash
```

```
[root@localhost sh]# cat /etc/passwd | grep "/bin/bash" | grep -v root
```

```
user1:x:500:500::/home/user1:/bin/bash
user2:x:501:501::/home/user2:/bin/bash
test1:x:502:502::/home/test1:/bin/bash
test2:x:503:503::/home/test2:/bin/bash
test3:x:504:504::/home/test3:/bin/bash
test4:x:505:505::/home/test4:/bin/bash
test5:x:506:506::/home/test5:/bin/bash
```

for循环说明-值表中含空格

```
#!/bin/bash
#使用字符串列表作为值表

for x in centos ubuntu gentoo
do    echo "$x" ; done
```

比较

```
for x in "centos" "ubuntu" "gentoo"
```

作为三个变量
运行结果：
centos
ubuntu
gentoo

```
for x in "centos ubuntu gentoo"
```

作为一个变量，运行结果：
centos ubuntu gentoo

值表项中包含空格必需使用双引号

for循环说明-值表用变量

```
#!/bin/bash
#使用变量作值表

i=1; weekdays="Mon Tue Wed Thu Fri"
for day in $weekdays ; do
    echo "Weekday $((i++)) : $day"
done
```

```
for day in $weekdays
```

分为多个值匹配
Weekday 1: Mon
Weekday 2: Tue
Weekday 3: Wed
Weekday 4: Thu
Weekday 5: Fri

比较

```
for day in "$weekdays"
```

作为一个值匹配
Weekday 1: Mon Tue Wed Thu Fri

值表项中变量作为一个值处理需要添加双引号

for循环说明-值表省时

```
#!/bin/bash

i=1
for day          # 使用位置参数变量$@作为值表, in $@可以省略
do
    echo -n "Positional parameter $((i++)): $day "
    case $day in
        [Mm]on|[Tt]ue|[Ww]ed|[Tt]hu|[Ff]ri) echo "(weekday)";;
        [Ss]at|[Ss]un) echo "(WEEKEND)";;
        *) echo "(Invalid)";;
    esac
done
```

```
$ ./for3.sh Mon Sun lundi
Positional parameter 1:Mon(weekday)
Positional parameter 2:Sun(WEEKEND)
Positional parameter 3:lundi(Invalid)
```

若无值表项则是将位置参数传递给变量

while 循环语句

```
while expr    # 执行 expr
do            # 若expr的退出状态为0，进入循环，否则退出while
    commands  # 循环体
done          # 循环结束标志，返回循环顶部
```

- ◆ 表达式expr部分除使用test或[]测试命令外，还可以是一组命令；
- ◆ 根据其最后一个命令的退出值决定是否进入循环体执行。

```
#!/bin/bash

i=1
s=0
while [ $i -le 100 ] // (($i<=100))
do
    s=$((s+$i))
    i=$((i+1))
done
echo "The sum is $s."
```



while 循环语句举例



```
#!/bin/bash
# $RANDOM是一个系统随机数的环境变量，它的范围是0--32767，模100
# 运算用于生成1-100的随机整数
# 使用永真循环、条件退出 (break) 的方式接收用户的猜测并进行判断

num=$((RANDOM%100))
while :
do
    read -p "Please guess my number [0-99]: " answer
    if [ $answer -lt $num ]
    then echo "The number < my NUMBER."
    elif [[ $answer -gt $num ]]
    then echo "The number > my NUMBER."
    elif ((answer == num))
    then echo "Congratulate: my NUMBER is $num." ; break
    fi
done
```

```
[root@localhost mayl]# . random.sh
please guess my number[0-99]:45
the number > my number.
please guess my number[0-99]:34
the number > my number.
please guess my number[0-99]:20
the number < my number.
please guess my number[0-99]:25
the number < my number.
please guess my number[0-99]:28
the number > my number.
please guess my number[0-99]:27
Congratulat: my number is 27
[root@localhost mayl]# _
```

until 循环语句

```
until expr    # 执行 expr
do            # 若expr的退出状态非0，进入循环，否则退出until
    commands  # 循环体
done          # 循环结束标志，返回循环顶部
```

- ◆ 它与while语句很相似，只是表达式expr不同；
- ◆ 当表达式expr为假时，才进入循环体，直至表达式expr为真时终止循环。

```
#!/bin/bash

i=1
s=0
until [ $i -gt 100 ] // (($i>100))
do
    s=$((s+i))
    i=$((i+1))
done
echo "The sum is $s."
```

程序控制结构——break和continue

break [n]

- 用于强行退出当前循环。
- 如果是嵌套循环，则 **break** 命令后面可以跟一数字 **n**，表示退出第 **n** 重循环（最里面的为第一重循环）。

continue [n]

- 用于忽略本次循环的剩余部分，回到循环的顶部，继续下一次循环。
- 如果是嵌套循环，**continue** 命令后面也可跟一数字 **n**，表示回到第 **n** 重循环的顶部。

break和continue举例

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

```
for ((i=1;i<=10;i=i+1))  
do  
    if [ "$i" -eq 4 ]  
    then  
        break  
    fi  
    echo $i  
done
```

```
[root@localhost sh]# ./break.sh  
1  
2  
3
```

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

```
for ((i=1;i<=10;i=i+1))  
do  
    if [ "$i" -eq 4 ]  
    then  
        continue  
    fi  
    echo $i  
done
```

```
[root@localhost sh]# ./continue.sh  
1  
2  
3  
5  
6  
7  
8  
9  
10
```

break和continue举例

```
#!/bin/bash

i=1
for day in Mon Tue Wed Thu Fri Sat Sun
do
    echo "Weekday $((i++)) : $day"
    if [ $i -eq 6 ]; then
        break
    fi
done
```

```
[may@localhost ~]$ . break.sh
weekday 1 :MON
weekday 2 :TUE
weekday 3 :WED
weekday 4 :THU
weekday 5 :FRI
[may@localhost ~]$ _
```

```
#!/bin/bash

i=1
for day in Mon Tue Wed Thu Fri Sat Sun
do
    echo "Weekday $((i++)) : $day"
    if [ $i -eq 6 ]; then
        continue
    fi
done
```

```
[may@localhost ~]$ . breaktest.sh
weekday 1 :MON
weekday 2 :TUE
weekday 3 :WED
weekday 4 :THU
weekday 5 :FRI
weekday 6 :SAT
weekday 7 :SUN
```

- ◆ exit命令是用于退出当前用户的登录状态，exit语句用来退出当前脚本，后续脚本程序不执行。
 - 如果给出了返回值，则在执行脚本之后，可以通过\$? 查询；
 - 如果没有给出返回值，则在执行脚本之后的返回值是执行exit语句前的最后一条命令的返回值。

```
#!/bin/bash

read -t 30 -p "please input num: " num

y=$( echo $num | sed 's/[0-9]//g' )

if [ -n $y ]
then
    echo "please input number,error!!!!"
    exit 19
else
    echo $num
fi
```

```
[root@localhost sh]# ./exit.sh
please input num: 123
123
```

如果变量num的值不是数字不替换；
是数字则把num替换为空并赋给变量y。

判断y变量的值如果不为空输出报错
信息，并退出脚本，返回值为19。

```
[root@localhost sh]# ./exit.sh
please input num: abc
please input number,error!!!!
[root@localhost sh]# echo $?
19
```

select 循环实现菜单

```
select variable in list
do
    # 循环开始的标志
    commands # 循环变量每取一次值，循环体就执行一遍
done        # 循环结束的标志
```

```
#!/bin/bash
echo "What is your favourite fruit?"
select fruit in APPLE ORANGE BANANA
do
    echo "Your favourite fruit is $fruit"
done
```

```
[may@localhost ~]$ . select.sh
what is your favourite fruit?
1) APPLE
2) ORANGE
3) BANANA
#? 1
your favourite is APPLE
#? 2
your favourite is ORANGE
#? 3
your favourite is BANANA
#? 4
your favourite is
#? _
```

select 是**bash**的扩展应用：

- 自动用**1,2,3,4**列出菜单
(没有**echo**指令，自动显示菜单)
- 自动**read**输入选择
(没有 **read**指令，自动输入)
- 赋值给变量
(没有赋值指令，输入数字后自动将字符串赋值给变量)

此处的循环没有结束语句

例：

```
#!/bin/bash
echo "What is your favourite fruit?"
select fruit in APPLE ORANGE BANANA
do
    break          #select本身就是一个循环，break是当选择后，就跳出循环
done
echo "Your favourite fruit is $fruit"
```

此处就解决了上例中不能退出循环的问题

特别说明：虽然**select**本身就是循环，但不建议用它的循环，因为**select**虽然循环却不再显示菜单，只循环输入，所以**select** 语句干脆直接用**break**，只执行一次，在其上另配**while**循环。

```
[may@localhost ~]$ . selectbreak.sh
what is your favourite fruit?
1) APPLE
2) ORANGE
3) BANANA
#? 1
your favourite is APPLE
[may@localhost ~]$ . selectbreak.sh
what is your favourite fruit?
1) APPLE
2) ORANGE
3) BANANA
#? 2
your favourite is ORANGE
[may@localhost ~]$
```

select 循环实现菜单

```
select variable in list
do                # 循环开始的标志
    commands      # 循环变量每取一次值，循环体就执行一遍
done             # 循环结束的标志
```

说明：如果 **in list...** 这一部分被省略，那么参数 **variable** 就以位置参数 (**\$1, \$2, ...**) 作为给定的值。

- ◆ 按数值顺序排列的菜单项 (**list item**) 会显示到标准输出
- ◆ 菜单项的间隔符由环境变量 **IFS** 决定
- ◆ 用于引导用户输入的提示信息存放在环境变量 **PS3** 中
- ◆ 用户输入的值会被存储在内置变量 **REPLY** 中
- ◆ 用户直接输入回车将重新显示菜单
- ◆ 与 **for** 循环类似，省略 **in list** 时等价于 **in "\$*"**

- ◆ 将大型脚本代码分割成小块，将这些被命名的代码块称为函数
 - 一个函数就是一个子程序，用于完成特定的任务
 - 如：添加一个用户、判断用户是否为管理员 等
- ◆ 函数定义之后可以被使用它的主程序调用
 - 调用函数的方法与执行**Shell**命令无异
 - 可以在**Shell**脚本中调用（函数需先定义而后调用）
 - 在命令行上直接调用（定义函数的文件需先加载）

◆ 函数定义

```
function 函数名 {  
    commands  
}
```

```
函数名 () {  
    commands  
}
```

◆ 函数调用

- 函数必须在调用之前定义
- 只需输入函数名即可调用函数

函数名

函数名 参数1 参数2 ...

Shell函数 (选讲)

- ◆ 调用函数时，直接利用函数名，如showfile，不必带圆括号。
- ◆ shell脚本与函数间的参数传递可利用位置参数和变量直接传递。
- ◆ 通常，函数中的最后一个命令执行之后，就退出被调函数。也可利用return命令立即退出函数，其格式是：return [n]

函数定义示例：

showfile()

```
{  
    if [ -d "$1" ]  
    then cd "$1"  
        cat m*.c | pr  
    else echo "$1 is not a directory ."  
    fi  
        echo "End of the function ."  
}
```

函数调用示例： **showfile** /home/mengqc

Shell函数（选讲）

◆ 函数的存储

- 函数和调用它的主程序保存在同一个文件中
 - 函数的定义必须出现在调用之前
- 函数和调用它的主程序保存在不同的文件中
 - 保存函数的文件必须先使用 `source` 命令执行，之后才能调用其中的函数

◆ 函数的显示

- 显示当前Shell可见的所有函数名

```
$ declare -F
```
- 显示当前Shell可见的所有（指定）的函数定义

```
$ declare -f
$ declare -f <functionName>
```

选讲：函数举例-函数和调用程序在同一个文件中

```
#!/bin/bash
# User define Function (UDF) #
sql_bak    () { echo "Running mysqldump tool..."; }
sync_bak   () { echo "Running rsync tool..."; }
git_bak     () { echo "Running gstore tool..."; }
tar_bak     () { echo "Running tar tool..."; }

# Main script starts here #
PS3="Please choose a backup tools : "
select s in  mysqldump rsync gstore tar quit
do
    case $REPLY in
        1) sql_bak ;;
        2) sync_bak ;;
        3) git_bak ;;
        4) tar_bak ;;
        5) exit    ;;
    esac
done
```

选讲：函数举例-函数和调用程序在不同文件中

```
#!/bin/bash
# filename: /root/bin/myfunc.sh #
# User define Function (UDF) #
sql_bak  () { echo "Running mysqldump tool..."; }
sync_bak () { echo "Running rsync tool..."; }
git_bak  () { echo "Running gstore tool..."; }
tar_bak  () { echo "Running tar tool..."; }
```

```
#!/bin/bash
# filename: funcsourc.sh #
source /root/bin/myfunc.sh
PS3="Please choose a backup tools : "
select s in  mysqldump rsync gstore tar quit
do
    case $REPLY in
        1|[mM]ysqldump) sql_bak ;;
        2|[rR]sync)     sync_bak ;;
        3|[gG]istore)   git_bak  ;;
        4|[tT]ar)       tar_bak  ;;
        5) exit         ;;
    esac
done
```

选讲：函数的结束与返回值举例

```
#!/bin/bash
max2() {
    [[ -z $1 || -z $2 ]] && { echo "Need 2 parameters.";exit; }
    (($1==$2)) && { echo "The two nums are equal.";exit; }
    (($1>$2)) && return $1 || return $2
}

read -p "Please input two inegerer nums:" a b
echo "a=$a,b=$b"
max2 $a $b
re_val=$?
echo "The larger is $re_val."
```

函数的返回值

- 在系统变量`$?`中。当函数的最后一条命令执行结束，其退出码即结束函数的返回值。
- **return [N]**结束函数执行，**N**即为指定返回值
- **exit [N]**结束函数及当前shell执行，**N**即为指定返回值。

```
[wlw@CM00 myshfile]$ sh ./max.sh
Please input two inegerer nums:
a=,b=
Need 2 parameters.
[wlw@CM00 myshfile]$ sh ./max.sh
Please input two inegerer nums:3 3
a=3,b=3
The two nums are equal.
[wlw@CM00 myshfile]$ sh ./max.sh
Please input two inegerer nums:34 65
a=34,b=65
The larger is 65.
```

选讲：函数的结束与返回值举例

- ◆ 使用 **return** 或 **exit** 只能返回整数值
- ◆ 使用标准输出实现函数的返回值
 - 是一种通用的方法，既能返回整数又能返回字符串
 - 函数结束前使用 **echo** 命令将结果显示到标准输出
 - 调用函数时使用 **res=\$(functionName)** 将函数输出结果存到变量 **res** 中，之后便使用变量 **\$res** 的值（输出或执行测试或进一步处理等）

```
#!/bin/bash
to_upper(){
    st="$@"
    put=$(tr ' [a-z]' ' [A-Z]' <<<"${st}")
    echo "$put"
}
```

```
res=$(to_upper "$@")
echo "Output:$res"
```

```
[wllw@CM00 myshfile]$ sh ./relocal.sh This Is a Test!
Output:THIS IS A TEST!
[wllw@CM00 myshfile]$ sh ./relocal.sh this is a test
Output:THIS IS A TEST
```

选讲：函数参数与变量

◆ 参数(Arguments)

- 调用函数时，使用位置参数的形式为函数传递参数
- 函数内的 $\$1$ - $\{n\}$ 、 $\$*$ 和 $\$@$ 表示其接收的参数
- 函数调用结束后位置参数 $\$1$ - $\{n\}$ 、 $\$*$ 和 $\$@$ 将被重置为调用函数之前的值
- 在主程序和函数中， $\$0$ 始终代表脚本名

◆ 变量(Variables)

- 函数内使用 **local** 声明的变量是局部（**Local**）变量
 - 局部变量的作用域是当前函数以及其调用的所有函数
- 函数内未使用 **local** 声明的变量是全局（**Global**）变量
 - 即主程序和函数中的同名变量是一个变量（地址一致）
- 使用全局变量引用函数的值不利于结构化编程

数组 (选讲)

- ◆ 数组：
 - **Bash**只提供一维数组，下标从**0**开始，大小无限定
- ◆ 声明数组：**declare -a** 数组名
 - (可以省略声明，按数组方式直接赋值，**BASH**会识别是数组)
- ◆ 数组赋值：
 - (1) **array=(var1 var2 var3 ... varN)**
 - (2) **array[0]=var1**
 - ...
 - array[n]=varN**
- 数组赋值时，其成员不一定要连续，允许空缺元素。

数组 (选讲)

◆ 引用数组

➤ 引用数组元素:

- 如: `$ echo ${array[0]}`

`${数组名[下标]}`

➤ 引用所有数组元素:

- 如: `$ echo ${array[*]}`

`${数组名[*/@]}`

➤ 引用数组元素个数:

- 如: `$ echo ${#array[*]}`

`${#数组名[*/@]}`

◆ 删除数组元素:

➤ `unset 数组[下标]` //带下标, 清除相应的元素;

➤ `unset 数组` //不带下标, 清除整个数据。

数组 (选讲)

```
[w1w@CM00 ~]$ a=(1 2 3 4 5 a b c)
[w1w@CM00 ~]$ echo ${a[2]}
3
[w1w@CM00 ~]$ echo ${a[*]}
1 2 3 4 5 a b c
[w1w@CM00 ~]$ echo ${a[@]}
1 2 3 4 5 a b c
[w1w@CM00 ~]$ echo ${#a[*]}
8
```

```
[w1w@CM00 ~]$ a[20]=hello
[w1w@CM00 ~]$ a[21]=world
[w1w@CM00 ~]$ echo ${a[*]}
1 2 3 4 5 a b c hello world
[w1w@CM00 ~]$ echo ${a[10]}

[w1w@CM00 ~]$ unset a[21]
[w1w@CM00 ~]$ echo ${a[*]}
1 2 3 4 5 a b c hello
[w1w@CM00 ~]$
```

```
[w1w@CM00 ~]$ x=${a[0]}+${a[1]}
[w1w@CM00 ~]$ echo $x
1+2
[w1w@CM00 ~]$ x=$(( ${a[0]}+${a[1]} ))
[w1w@CM00 ~]$ echo $x
3
[w1w@CM00 ~]$ x=$(( ${a[0]}+${a[7]} ))
[w1w@CM00 ~]$ echo $x
1
```

数组（选讲）

数组的特殊使用：分片和替换

- ◆ **分片：** **`${数组名[@或*]:起始位置:长度}`** 切片原先数组，返回是字符串，中间用“空格”分开。

➤ 例：
`$ a=(1 2 3 4 5)`
`$ echo ${a[@]:0:3}`
1 2 3
`$ echo ${a[@]:1:4}`
2 3 4 5

如果加上” ()”，将得到新的切片数组。

```
$ c=(${a[@]:1:4}) #此例中c 就是一个新数据。  
$ echo ${#c[@]}  
4  
$ echo ${c[*]}  
2 3 4 5
```

数组（选讲）

数组的特殊使用：分片和替换

◆ **替换：** `${数组名[@或*]/查找字符/替换字符}` 不会改变原先数组内容。

➤ 例：

```
$ a=(1 2 3 4 5)
$ echo ${a[@]/3/100}
1 2 100 4 5
$ echo ${a[@]}
1 2 3 4 5
```

如果需要修改，可以看下面例子，重新定义数据。

```
$ a=(${a[@]/3/100})
$ echo ${a[@]}
1 2 100 4 5
```

比较说明下列变量引用表达式

- ◆ **\$name**
 - 表达式**\$name**表示变量**name**的值，若变量未定义，则用空值替换。
- ◆ **\${name}**
 - 表达式**\${name}**将被变量**name**的值替换。用花括号括起**name**，目的在于把变量名与后面的字符分隔开，避免出现混淆。替换后花括号被取消。
- ◆ **\${name[n]}**
 - **\${name[n]}**表示数组变量**name**中第**n**个元素的值。
- ◆ **\${#name[*]}** **\${#name[@]}**
 - 计算数组的长度

选讲：比较说明下列变量引用表达式

- ◆ **`${name[*]}` `${name[@]}`**
 - **`${name[*]}`和`${name[@]}`都表示数组`name`中所有非空元素的值，每个元素的值用空格分开。如果数组`name`中没有元素，则`${name[@]}`被扩展为空串。**

说明：用双引号把它们都括起来，那么二者的含义就有区别：

- ◆ **`"${name[*]}"` `"${name[@]}"`**
 - **`"${name[*]}"`,被扩展成一个词(即字符串)，这个词由以空格分开的各个数组元素组成；**
 - **`"${name[@]}"`，被扩展成多个词，每个数组元素是一个词。**

选讲:有效的变量引用表达式

- ◆ **`${name#pattern}`** **`${name##pattern}`**
 - **`${name#pattern}`** 中, 从变量**name**的开头部分去掉与**pattern**匹配的最短的部分;
 - **`${name##pattern}`**中, 从变量**name**的开头部分去掉与**pattern**匹配的最长的部分。
- ◆ **`${name % pattern}`** **`${name %% pattern}`**
 - **`${name % pattern}`**中, 从变量**name**的最尾部去掉与**pattern**匹配的最短部分;
 - **`${name %% pattern}`**中, 从变量**name**的最尾部去掉与**pattern**匹配的最长部分。