第12章　shell 脚本介绍

一个shell脚本可以包含一个或多个命令。当然可以不必只为了两个命令就编写一个shell脚本，一切由用户自己决定。

本章内容有：

- 使用shell脚本的原因。
- shell脚本基本元素。
- shell脚本运行方式。

12.1 使用shell脚本的原因

shell脚本在处理自动循环或大的任务方面可节省时间，且功能强大。如果有一个大的任务的需要完成，你不得不一个个敲进去，然后观察输出结果，如果正确，可继续下一个任务，否则要回到清单一步步观察。

一个任务可能是将文件分类、向文件插入文本、迁移文件、从文件中删除行、清除系统过期文件、以及系统一般的管理维护工作等等。

在使用一系列系统命令的同时，可以使用变量、条件、算术和循环快速创建脚本以完成相应工作。这比在命令行下一个个敲入命令要节省大量的时间。

shell脚本可以在行命令中接收信息，并使用它作为另一个命令的输入。

对于不同的UNIX和LINUX，使用一段shell脚本将需要一些小小的改动才能运行。实际上shell的可迁移性不成问题，但是系统间命令的可迁移性存在差别。

写一段脚本，若其执行结果与预想的不同，不必着急。无论多不可思议的结果，先把它保存起来，这是修改的基础。这里要说的是不要害怕新事物，否则将不能树立信心，学起来会更加困难。

12.2 脚本内容

若能通过一些易理解的脚本就可实现同样功能时，没有必要使脚本复杂化。

脚本不是复杂的程序，它是按行解释的。脚本第一行总是以#!/bin/sh开始，它通知**shell**使用系统上的**Bourne shell**解释器。

任何脚本都可能有注释，加注释需要此行的第一个字符为#，解释器对此行不予解释。在第二行注释中写入脚本名是一个好习惯。

脚本从上到下执行，运行脚本前需要增加其执行权限。确保正确建立脚本路径(**PATH**)，这样只用文件名就可以运行它了。

12.3 运行一段脚本

下面是一个经常讨论的例子，此文件为**cleanup**。

```
$ pg cleanup
#!/bin/sh
# name: cleanup
# this is a general cleanup script
echo "starting cleanup...wait"
rm /usr/lcoal/apps/log/*.log
tail -40 /var/adm/mesages >/tmp/messages
rm /var/adm/messages
mv /tmp/messages /var/adm/messages
echo "finished cleanup"
```

上述脚本清除/usr/adm /下信息，并删除/usr/local/apps/log下所有注册信息。

可以使用chmod命令增加脚本执行权限。

```
$ chmod u+x cleanup
```

现在运行脚本，只敲入文件名即可。

```
$ cleanup
```

如果返回错误信息：

```
$ cleanup
  sh:cleanup:command not found
```

再试：

```
$ ./cleanup
```

如果脚本运行前必须键入路径名，或者shell通知无法找到命令，就需要在.profile PATH下加入用户可执行程序目录。要确保程序在用户的$HOME可执行程序目录下，应键入：

```
$ pwd

$ /home/dave/bin
```

如果pwd命令最后一部分是bin，那么需要在路径中加入此信息。编辑用户.profile文件，加入可执行程序目录$HOME/bin：

```
PATH = $PATH; $HOME/bin
```

如果没有bin目录，就创建它。首先确保在用户根目录下。

```
$ cd $HOME
```

```
$ mkdir bin
```

现在可以在.profile文件中将bin目录加入PATH变量了，然后重新初始化.profile。

```
$./.profile
```

脚本将会正常运行。

第13章　条件测试

写脚本时，有时要判断字符串是否相等、检查文件状态或是数字测试。基于这些测试才能做进一步动作。test命令用于测试字符串，文件状态和数字，也很适合于下一章将提到的if、then、else条件结构。

本章内容有：

- 对文件、字符串和数字使用test命令。
- 对数字和字符串使用expr命令。

expr命令测试和执行数值输出。使用最后退出状态命令$?可测知test和expr，二者均以0表示正确，1表示返回错误。

13.1 测试文件状态

test一般有两种格式，即：

test condition

或

[condition]

使用方括号时，要注意在条件两边加上空格。

文件状态测试

-d 目录	-s 文件长度大于0、非空
-f 普通文件	-w 可写
-l 符号连接	-e 文件存在
-r 可读	-x 可执行

```
$ ls -l scores.txt
-rw-r--r--   1 dave   admin  0 May 15
$ [ -w scores.txt ]
$ echo $?
0


$ test -w scores.txt
$ echo $?
0
$ [ -x scores.txt ]
$ echo $?
1
```

13.2 测试时使用逻辑操作符

有时要比较两个以上文件状态，shell提供三种逻辑操作完成此功能。

-a 逻辑与：操作符两边均为真，结果为真，否则为假。

-o 逻辑或：操作符两边一边为真，结果为真，否则为假。

！逻辑否：条件为假，结果为真。

```
-rw-r--r--    1 root   root      0 May 15 11:29 scores.txt
-rwxr-xr--    1 root   root      0 May 15 11:49 results.txt
```

```
$ [ -w results.txt -a -w scores.txt ]
$ echo $?
0
```

```
$ [ -x results.txt -o -x scores.txt ]
$ echo $?
0
```

```
$ [ -w results.txt -a -x results.txt ]
$ echo $?
0
```

13.3 字符串测试

字符串测试是错误捕获很重要的一部分，特别在测试用户输入或比较变量时尤为重要。

字符串测试有5种格式。

```
test "string"
test string_operator "string"
test "string" string_operator "string"
[ string_operator string ]
[ string string_operator string ]
```

这里，string _ operator可为：

= 两个字符串相等。

! = 两个字符串不等。

-z 空串。

-n 非空串。

要测试环境变量EDITOR是否为空：

```
$ [ -z $EDITOR ]
$ echo $?
1
```

```
$ [ $EDITOR = "vi" ]
$ echo $?
0
```

测试变量tape与变量tape2是否相等：

```
$ TAPE="/dev/rmt0"
$ TAPE2="/dev/rmt1"
$ [ "$TAPE" = "$TAPE2" ]
$ echo $?
1
```

没有规定在设置变量时一定要用双引号，但在进行字符串比较时必须这样做。

13.4 测试数值

测试数值可以使用许多操作符，一般格式如下：

"number" numeric_operator "number "

或者

["number" numeric_operator "number"]

numeric_operator可为：

-eq 数值相等。

-ne 数值不相等。

-gt 第一个数大于第二个数。

-lt 第一个数小于第二个数。

-le 第一个数小于等于第二个数。

-ge 第一个数大于等于第二个数。

```
$ NUMBER=130
$ [ "$NUMBER" -eq "130" ]
$ echo $?
0
$ [ "$NUMBER" -eq "100" ]
$ echo $?
1
```

```
$ [ "$NUMBER" -gt "100" ]
$ echo $?
0
```

也可以测试两个整数变量。

```
$ SOURCE_COUNT=13
$ DEST_COUNT=15
$ [ "$DEST_COUNT" -gt "$SOURCE_COUNT" ]
$ echo $?
0
```

可以不必将整数值放入变量，直接用数字比较，但要加引号。

```
$ [ "990" -le "995" ]
$ echo $?
0
```

可以用逻辑操作符将两个测试表达式结合起来。仅需要用到一对方括号，而不能用两个，否则将返回错误信息“ too many arguments”。

```
$ [ "990" -le "995" ] -a [ "123" -gt "33" ]
sh:[: too many arguments
```

正确使用方式应为：

```
$ [ "990" -le "995"  -a  "123" -gt "33" ]
$ echo $?
0
```

13.5 expr用法

expr命令一般用于整数值，但也可用于字符串。
一般格式为：

expr argument operator argument

expr也是一个手工命令行计数器。

```
$ expr 10 + 10
20
$ expr 900 + 600
1500

$ expr 30 / 3
10

$ expr 30 / 3 / 2
5
```

```
$ expr 30 \* 3
90
```

13.5.1 增量计数

expr在循环中用于增量计算。首先，循环初始化为0，然后循环值加1，反引号的用法意即替代命令。最基本的一种是从（expr）命令接受输出并将之放入循环变量。

```
$ LOOP=0
$ LOOP=`expr $LOOP + 1`
```

13.5.2 数值测试

可以用**expr**测试一个数。如果试图计算非整数，将返回错误。

```
$ expr rr + 1
expr: non-numeric argument
```

```
$ VALUE=12
$ expr $VALUE + 10 > /dev/null 2>&1
$ echo $?
0
```

下面的例子测试两个字符串是否相等，这里字符串为“hello”和“hello”。

```
$ VALUE=hello
$ expr $VALUE = "hello"
1
$ echo $?
0
```

expr返回1，表明成功。其最后退出状态返回0表示测试成功，两个字符串确实相等。

13.6 小结

本章涉及expr和test基本功能，讲到了怎样进行文件状态测试和字符串赋值，使用其他的条件表达式如if then else和case可以进行更广范围的测试及对测试结果采取一些动作。

第14章　控制流结构

所有功能脚本必须有能力进行判断，也必须有能力基于一定条件处理相关命令。本章讲述这方面的功能，在脚本中创建和应用控制结构。

本章内容有:

- 退出状态。
- while、for和until loops循环。
- if then else语句。
- 脚本中动作。
- 菜单。

14.1 控制结构

几乎所有的脚本里都有某种流控制结构，很少有例外。流控制是什么？假定有一个脚本包含下列几个命令：

```
#!/bin/sh
# make a directory
mkdir /home/dave/mydocs
# copy all doc files
cp *.docs /home/dave/docs
# delete all doc files
rm *.docs
```

上述脚本问题出在哪里？如果目录创建失败或目录创建成功文件拷贝失败，如何处理？

shell会提供一系列命令声明语句等补救措施来帮助你在命令成功或失败时，或需要处理一个命令清单时采取正确的动作。

这些命令语句大概分两类：

循环和流控制。

14.2 if then else语句

if语句测试条件，测试条件返回真（0）或假（1）后，可相应执行一系列语句。if语句结构对错误检查非常有用。其格式为：

```
if 条件1          如果条件1为真
then              那么
   命令1          执行命令1
elif 条件2        如果条件1为假，条件2为真
then              那么
   命令2          执行命令2
else              如果条件1，2均不成立
   命令3          那么执行命令3
fi                完成
```

i f语句必须以单词f i终止。

elif和else为可选项，如果语句中没有否则部分，那么就不需要elif和else部分。If语句可以有许多elif部分。最常用的if语句是if then fi结构。

14.2.1 简单的if语句

```
if  条件
then 命令
fi
```

```
if 条件; then
命令
fi
```

```
$ pg iftest
#!/bin/sh
# iftest
# this is a comment line, all comment lines start with a #
if [ "10" -lt "12" ]
then
  # yes 10 is less than 12
  echo "Yes, 10 is less than 12"
fi
```

14.2.2 grep输出检查

不必拘泥于变量或数值测试，也可以测知系统命令是否成功返回。

```
$ pg grepif
#!/bin/sh
# grepif
if grep 'Dave\>' data.file > /dev/null 2>&1
then
  echo "Great Dave is in the file"
else
  echo "No Dave is not in the file"
fi

$ grepif
No Dave is not in the file
```

14.2.3 文件拷贝输出检查

```
$ pg ifcp
#!/bin/sh
# ifcp
if cp myfile myfile.bak; then
  echo "good copy"
else
  echo "`basename $0`: error could not copy the files" >&2
fi

$ ifcp
cp: myfile: No such file or directory
ifcp: error could not copy the files
```

注意，文件可能没找到，系统也产生本身的错误信息，这类错误信息可能与输出混在一起。要去除系统产生的错误和系统输出，只需简单的将标准错误和输出重定向即可。修改脚本为： **>/dev/null 2>&1**。

```
$ pg ifcp
#!/bin/sh
# ifcp
if cp myfile myfile.bak >/dev/null 2> then
  echo "good copy"
else
  echo "`basename $0`: error could not copy the files" >&2
fi
```

```
$ ifcp
ifcp: error could not copy the files
```

14.2.4 测试传递到脚本中的参数

以下测试确保脚本有三个参数。如果没有，则输出一个提示信息到标准错误，然后退出并显示退出状态。如果参数数目等于3，则显示所有参数。

```
$ pg ifparam
#!/bin/sh
# ifparam
if [ $# -lt 3 ]; then
# less than 3 parameters called, echo a usage message and exit
  echo "Usage: `basename $0`arg1 arg2 arg3" >&2
  exit 1
fi
# good, received 3 params, let's echo them
echo "arg1: $1"
echo "arg2: $2"
echo "arg3: $3"
```

如果只传入两个参数，则显示一可用信息，然后脚本退出。

```
$ ifparam cup medal
Usage:ifparam arg1 arg2 arg3
```

这次传入三个参数。

```
$ ifparam cup medal trophy
arg1: cup
arg2: medal
arg3: trophy
```

14.2.5 null: 命令用法

```
$ pg ifdirectory
#!/bin/sh
# ifdirectory
DIRECTORY=$1
if [ "`ls -A $DIRECTORY`" = "" ]
then
   echo "$DIRECTORY is indeed empty"
   else :     # do nothing
fi
```

```
$ pg ifcp2
#!/bin/sh
# ifcp2
if cp $1 $2 > /dev/null 2>&1
# successful, great do nothing
then :
else
# oh dear, show the user what files they were.
  echo "`basename $0`: ERROR failed to copy $1 to $2"
  exit 1
fi
```

```
$ pg ifsort
#!/bin/sh
# ifsort
if sort accounts.qtr > /dev/null
# sorted. Great
then :
else
# better let the user know
  echo "`basename $0`: Oops..errors could not sort accounts.qtr"
fi
```

14.3 case语句

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，执行相应的命令。case语句格式如下:

```
case 值 in
    模式1)
        命令1
        ...
        ;;
    模式2)
        命令2
        ...
        ;;
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，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，再接受其他输入。

模式部分可能包括元字符，与在命令行文件名例子中使用过的匹配模式类型相同，

*　　任意字符。

？　　任意单字符。

[..] 类或范围中任意字符。

14.3.1 简单的case语句

```
$ pg caseselect
#!/bin/sh
# caseselect
echo -n "enter a number from 1 to 5  :"
read ANS
  case $ANS in
  1) echo "you select 1"
    ;;
  2) echo "you select 2"
    ;;
  3) echo "you select 3"
    ;;
  4) echo "you select 4"
    ;;
  5) echo "you select 5"
    ;;
  *) echo "`basename $0`: This is not between 1 and 5" >&2
    exit 1
    ;;
  esac
```

给出不同输入，运行此脚本。

```
$ caseselect
enter a number from 1 to 5 : 4
you select 4
```

使用模式*捕获范围之外的取值情况。

```
$ caseselect
enter a number from 1 to 5  :pen
caseselect: This is not between 1 and 5
```

14.3.2 对匹配模式使用|（或）

```
$ pg caseterm
#!/bin/sh
# caseterm
echo " choices are.. vt100, vt102, vt220"
echo -n "enter your terminal type :"
read TERMINAL
  case $TERMINAL in
  vt100|vt102) TERM=vt100
     ;;
  vt220) TERM=vt220
     ;;
  *) echo "`basename $0` : Unknown response" >&2
     echo "setting it to vt100 anyway, so there"
     TERM=vt100
     ;;
  esac
  export TERM
echo "Your terminal is set to $TERM"
```

14.3.3 case与命令参数传递

```
$ pg caseparam
#!/bin/sh
# caseparam
if [ $# != 1 ]; then
  echo "Usage:`basename $0`[start|stop|help]"  >&2
  exit 1
fi
# assign the parameter to the variable OPT
OPT=$1
  case $OPT in
  start) echo "starting..`basename $0`"
    # code here to start a process
    ;;
  stop) echo "stopping..`basename $0`"
    # code here to stop a process
    ;;
  help)
    # code here to display a help page
    ;;
  *) echo "Usage:`basename $0`[start|stop|help]"
    ;;
  esac
```

14.4 for 循环

For 循环一般格式为：

```
for 变量名 in 列表
do
    命令1
    命令2
done
```

当变量值在列表里， for循环即执行一次所有命令。命令可为任何有效的shell命令和语句。变量名为任何单词。

in列表是可选的，如果不用，for循环使用命令行的位置参数。

in列表可以包含替换、字符串和文件名

14.4.1 简单的 for 循环

```
$ pg for_i
#!/bin/sh
# for_i
for loop in 1 2 3 4 5
do
   echo $loop
done
```

```
$ for_i
1
2
3
4
5
```

14.4.2 对 for 循环使用 ls 命令

```
$ pg forls

#!/bin/sh
# forls
for loop in `ls `
do
   echo $loop
done
```

```
$ forls
array
arrows
center
center1
center2
centerb
```

14.4.3 对 for 循环使用命令行参数

在for循环中省去in列表选项时，它将接受命令行位置参数作为选项内容。相当于：

```
for  params  in  "$@"
```

或

```
for  params  in  "$*"
```

下面的例子不使用in列表选项，for循环查看特定参数$@或$*，以从命令行中取得参数。

```
$ pg forparam2
#!/bin/sh
# forparam2
for params
do
  echo "You supplied $params as a command line option"
done
```

```
$ forparam2 myfile1 myfile2 myfile3
 You supplied myfile1 as a command line option
 You supplied myfile2 as a command line option
 You supplied myfile3 as a command line option
```

可在for循环里使用find命令，利用命令行参数，传递所有要查找的文件。

```
$ pg forfind
#!/bin/sh
# forfind
for loop
do
   find / -name $loop -print
done
```

执行结果：

```
$ forfind passwd LPSO.AKSOP
/etc/passwd
/etc/pam.d/passwd
/etc/uucp/passwd
/usr/bin/passwd
/usr/local/accounts/LPSO.AKSOP
```

14.4.4 多文件大小写转换

查找所有以LPSO开头的文件并将其内容转换为大写。这里使用了ls和cat命令，ls用于找到相关文件，cat后将之管道输出至tr命令。目标文件扩展名为.UC，注意在for循环中使用ls命令时反引号的用法。

```
$ pg forUC
#!/bin/sh
# forUC
for files in `ls LPSO*`
do
  cat $files |tr "[a-z]" "[A-Z]" >$files.UC
done
```

14.4.5 for循环嵌入

嵌入循环可以将一个for循环嵌在另一个for循环内：

```
for 变量名1 in 列表1
    do
        for 变量名2 in 列表2
            do
                命令1
                . . .
            done
    done
```

下面脚本即为嵌入for循环，这里有两个列表APPS和SCRIPTS。第一个包含服务器上应用的路径，第二个为运行在每个应用上的管理脚本。对列表APPS上的每一个应用，列表SCRIPTS里的脚本将被运行，脚本实际上为后台运行。脚本使用tee命令在登录文件上放一条目，因此输出到屏幕的同时也输出到一个文件。查看输出结果就可以看出嵌入for循环怎样使用列表SCRIPTS以执行列表APPS上的处理。

```
$ pg audit_run
#!/bin/sh
# audit_run
APPS="/apps/accts /apps/claims /apps/stock /apps/serv"
SCRIPTS="audit.check report.run cleanup"
LOGFILE=audit.log
MY_DATE=`date +%H:%M" on "%d/%m%Y`


# outer loop
for loop in $APPS
do
  # inner loop
  for loop2 in $SCRIPTS
  do
    echo "system $loop now running $loop2 at $MY_DATE" | tee -a $LOGFILE
    $loop $loop2 &


  done
done
```

执行结果：

```
$ audit_run
system /apps/accts now running audit.check at 20:33 on 23/051999
system /apps/accts now running report.run at 20:33 on 23/051999
system /apps/accts now running cleanup at 20:33 on 23/051999
system /apps/claims now running audit.check at 20:33 on 23/051999
system /apps/claims now running report.run at 20:33 on 23/051999
system /apps/claims now running cleanup at 20:34 on 23/051999
system /apps/stock now running audit.check at 20:34 on 23/051999
system /apps/stock now running report.run at 20:34 on 23/051999
system /apps/stock now running cleanup at 20:34 on 23/051999
system /apps/serv now running audit.check at 20:34 on 23/051999
system /apps/serv now running report.run at 20:34 on 23/051999
system /apps/serv now running cleanup at 20:34 on 23/051999
```

14.5 until循环

until循环执行一系列命令直至条件为真时停止。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循环格式为：

```
until 条件(条件可为任意测试条件)
    do
        命令1
        . . .
    done
```

until循环实例：

这段脚本不断的搜寻who命令中用户root，变量IS-ROOT保存grep命令的结果。

若找到了root，循环结束，并向用户simon发送邮件，通知他root已经登录，注意sleep命令用法，它常用于until循环中，因为必须让循环体内命令睡眠几秒钟再执行，否则会消耗大量系统资源。亦可转入后台运行。

```
$ pg until_who
#!/bin/sh
# until_who
IS_ROOT=`who | grep root`
until [ "$IS_ROOT" ]
do
  sleep 5
done
echo "Watch it. roots in " | mail simon
```

14.6 while循环

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，其格式为：

```
while 命令
      do
          命令1
          命令2
          . . .
      done
```

虽然通常只使用一个命令，但在**while**和**do**之间可以放*几个命令*。命令通常用作测试条件。

只有当命令的退出状态为**0**时，**do**和**done**之间命令才被执行，如果退出状态不是**0**，则循环终止。

命令执行完毕，控制返回循环顶部，从头开始直至测试条件为假。

14.6.1 简单的while循环

以下是一个基本的while循环，测试条件是：如果COUNTER小于5，那么条件返回真。COUNTER从0开始，每次循环处理时，COUNTER加1。

```
$ pg whilecount
#!/bin/sh
# whilecount
COUNTER=0
# does the counter = 5 ?
while [ $COUNTER -lt 5 ]
do
  # add one to the counter
  COUNTER=`expr $COUNTER + 1`
  echo $COUNTER
done
```

```
$ whilecount
1
2
3
4
5
```

14.6.2 用while循环从文件中读取数据

while循环最常用于从一个文件中读取数据，因此编写脚本可以处理这样的信息。

假定要从下面包含雇员名字、从属部门及其ID号的一个文件中读取信息。

```
$ pg names.txt
Louise Conrad:Accounts:ACC8987
Peter James:Payroll:PR489
Fred Terms:Customer:CUS012
James Lenod:Accounts:ACC887
Frank Pavely:Payroll:PR489
```

可以用一个变量保存每行数据，当不再有数据可读时条件为假。**while**循环使用输入重定向以保证从文件中读取数据。注意整行数据被赋予变量$LINE。

```
$ pg whileread
#!/bin/sh
# whileread
while read LINE
do
   echo $LINE
done < names.txt

$ whileread
Louise Conrad:Accounts:ACC8987
Peter James:Payroll:PR489
Fred Terms:Customer:CUS012
James Lenod:Accounts:ACC887
Frank Pavely:Payroll:PR489
```

14.6.3 使用IFS读文件

输出时要去除冒号域分隔符，可使用变量IFS。在改变它之前保存IFS的当前设置。然后在脚本执行完后恢复此设置。使用IFS可以将域分隔符改为冒号而不是空格或tab键。这里有3个域需要分隔，即NAME、DEPT和ID。

为使输出看起来更清晰，对echo命令使用tab键将各个域分隔得更开一些，脚本如下：

```
$ pg   whilereadifs
#!/bin/sh
# whilereadifs
# save the setting of IFS
SAVEDIFS=$IFS
# assign new separator to IFS
IFS=:
while read NAME DEPT ID
do
   echo -e "$NAME\t $DEPT\t $ID"
done < names.txt
# restore the settings of IFS
IFS=$SAVEDIFS
```

脚本运行，输出果然清晰多了。

```
$ whilereadifs
Louise Conrad          Accounts    ACC8987
Peter James            Payroll     PR489
Fred Terms             Customer    CUS012
James Lenod            Accounts    ACC887
Frank Pavely           Payroll     PR489
```

还可以采取进一步动作，统计各部门雇员数。

```
$ pg whileread_cond
!/bin/sh
# whileread_cond
# initialise variables
ACC_LOOP=0; CUS_LOOP=0; PAY_LOOP=0;
```

```
SAVEDIFS=$IFS
IFS=:
while read NAME DEPT ID
do
  # increment counter for each matched dept.
  case $DEPT in
  Accounts) ACC_LOOP=`expr $ACC_LOOP + 1`
    ACC="Accounts"
    ;;
  Customer) CUS_LOOP=`expr $CUS_LOOP + 1`
    CUS="Customer"
    ;;
  Payroll) PAY_LOOP=`expr $PAY_LOOP + 1`
    PAY="Payroll"
    ;;
  *) echo "`basename $0`: Unknown department $DEPT" >&2
    ;;
  esac
done < names.txt
```

```
IFS=$SAVEDIFS
echo "there are $ACC_LOOP employees assigned to $ACC dept"
echo "there are $CUS_LOOP employees assigned to $CUS dept"
echo "there are $PAY_LOOP employees assigned to $PAY dept"
```

运行脚本，输出：

```
$ whileread_cond
there are 2 employees assigned to Accounts dept
there are 1 employees assigned to Customer dept
there are 2 employees assigned to Payroll dept
```

14.6.4 忽略#字符

读文本文件时，可能要忽略或丢弃遇到的注释行，下面是一个读配置文件的典型的例子。

```
$ pg config
# THIS IS THE SUB SYSTEM AUDIT CONFIG FILE
# DO NOT EDIT!!!!.IT WORKS
#
# type of admin access
AUDITSCM=full
# launch place of sub-systems
AUDITSUB=/usr/opt/audit/sub
# serial hash number of product
HASHSER=12890AB3
# END OF CONFIG FILE!!!
```

```
$ pg ignore_hash
#!/bin/sh
# ignore_hash
INPUT_FILE=config
if [-s $INPUT_FILE ]; then
  while read LINE
  do
    case $LINE in
    \#*) ;;    # ignore any hash signs
    *) echo $LINE
       ;;
    esac
  done <$INPUT_FILE
else
  echo "`basename $0` : Sorry $INPUT_FILE does not
  exit 1
fi
```

```
$ ignore_hash
AUDITSCM=full
AUDITSUB=/usr/opt/audit/sub
HASHSER=12890AB3
```

14.6.5 每次读一对记录

有时可能希望每次处理两个记录，也许可从记录中进行不同域的比较。每次读两个记录很容易，就是要在第一个**while**语句之后将第二个读语句放在其后。

```
$ pg record.txt
record 1
record 2
record 3
record 4
record 5
record 6
```

```
$ pg readpair
#!/bin/sh
# readpair
# first record
while read rec1
do
  # second record
  read rec2
  # further processing/testing goes here to test or compare both records
  echo "This is record one of a pair :$rec1"
  echo "This is record two of a pair :$rec2"
  echo "----------------------------"
done < record.txt
```

首先来检查确实读了很多记录，可以使用 wc命令：

```
$ cat record.txt | wc -l
       6
```

共有6个记录，观察其输出：

```
$ readpair
This is record one of a pair :record 1
This is record two of a pair :record 2
-----------------------------
This is record one of a pair :record 3
This is record two of a pair :record 4
-----------------------------
This is record one of a pair :record 5
This is record two of a pair :record 6
-----------------------------
```

14.6.6 使用while循环读键盘输入

```
$ pg whileread
#!/bin/sh
# whileread
echo " type <CTRL-D> to terminate"
echo -n "enter your most liked film :"
while read FILM
do
  echo "Yeah, great film the $FILM"
done
```

```
$ whileread
enter your most liked film: Sound of Music
Yeah, great film the Sound of Music
<CTRL-D>
```

14.7 使用break和continue控制循环

有时需要基于某些准则退出循环或跳过循环步。shell提供两个命令实现此功能。

- break
- continue

14.7.1 break

break命令允许跳出循环。break通常在进行一些处理后退出循环或case语句。如果是在一个嵌入循环里，可以指定跳出的循环个数。例如：在两层循环内，用break 2刚好跳出整个循环。

14.7.2 跳出case语句

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，break使用脚本如下：

```
$ pg breakout
#!/bin/sh
# breakout
# while : means loop forever
while :
do
  echo -n "Enter any number [1..5] :"
  read ANS
  case $ANS in
    1|2|3|4|5) echo "great you entered
    ;;
  *) echo "Wrong number..bye"
    break
    ;;
  esac
done
```

14.7.3 continue

continue命令有别于break命令，它不会跳出循环，只是跳过这个循环步。其语法格式是：

continue [n]

循环层数是由内向外编号

- exit命令

利用exit命令可以立即退出正在执行的shell脚本。

其语法格式是：

exit [n]

若未显式给出n值，则为最后一个命令的执行状态。

```
for i in 1 2 3 4 5
  do
      if [ "$i" –eq 3 ]
          then continue
      else echo "$i"
      fi
  done
```

14.8 菜单

创建菜单时，在while循环里null空命令很合适。while加空命令null意即无限循环，这正是一个菜单所具有的特性。

菜单界面应是友好的，不应该让用户去猜做什么，主屏幕也应该带有主机名和日期，并伴随有运行此菜单的用户名。

下面是即将显示的菜单。

```
User: dave                Host:Bumper                Date:31/05/1999

                1 :  List files in current directory
                2 :  Use the vi editor
                3 :  See who is on the system
                H :  Help screen
                Q :  Exit Menu

              Your Choice [1,2,3,H,Q] >
```

可以给变量一个更有意义的名字：

```
MYDATE = `date +%d/%m/%Y`
THIS_HOST = `hostname -s`
USER = `whoami`
```

要注意实际屏幕显示，不要浪费时间使用大量的echo语句或不断地调整它们。这里使用本地文档，在分界符后面接受输入，直至分界符被再次定位。格式为：

```
command << WORD
any input
WORD
```

脚本如下：

```
$ pg menu
#!/bin/sh
# menu
# set the date, user and hostname up
MYDATE=`date +%d/%m/%Y`
THIS_HOST=`hostname -s`
USER=`whoami`
```

```
# loop forever !
while :
do
  # clear the screen
  tput clear
  # here documents starts here
  cat <<MAYDAY

  ---------------------------------------------------------------------
  User: $USER                Host:$THIS_HOST                Date:$MYDATE
  ---------------------------------------------------------------------

                  1 :  List files in current directory
                  2 :  Use the vi editor
                  3 :  See who is on the system
                  H :  Help screen
                  Q :  Exit Menu

  ---------------------------------------------------------------------
  MAYDAY
```

```
# here document finished
echo -e -n "\tYour Choice [1,2,3,H,Q] >"
read CHOICE
  case $CHOICE in
  1) ls
     ;;
  2) vi
     ;;
  3) who
     ;;
  H|h)
```

```
    # use a here document for the help screen
    cat <<MAYDAY
    This is the help screen, nothing here yet to help you!
    MAYDAY
      ;;
    Q|q) exit 0
      ;;
    *) echo -e "\t\007unknown user response"
      ;;
    esac
  echo -e -n "\tHit the return key to continue"
  read DUMMY
done
```

附：向脚本传递参数

shell提供shift命令以帮助偏移选项，即可以去除只使用$1到$9传递参数的限制。

shift命令

向脚本传递参数时，有时需要将每一个参数偏移以处理选项，这就是shift命令的功能。它每次将参数位置向左偏移一位，下面用一段简单脚本详述其功能。脚本使用while循环输出所有传递到脚本的参数。

```
$ pg opt2
#!/bin/sh
# opt2
loop=0
while [ $# -ne 0 ]
do
   echo $1
done
```

```
$ opt2 file1 file2 file3
file1
file1
file1
...
```

shift命令简单用法

使用shift命令来处理传递到脚本的每一个参数。改动后脚本如下：

```
$ pg opt2
#!/bin/sh
# opt2
loop=0
while [ $# -ne 0 ]
do
   echo $1
   shift
done
```

```
$ opt2 file1 file2 file3
file1
file2
file3
```