第18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。

18.1　退出状态

在书写正确脚本前，大概讲一下退出状态。任何命令进行时都将返回一个退出状态。如果要观察其退出状态，使用最后状态命令：

```
$ echo $?
```

主要有4种退出状态。前面已经讲到了两种，即最后命令退出状态 $?和控制次序命令（$$、||）。其余两种是处理 shell脚本或shell退出及相应退出状态或函数返回码。在第 19章讲到函数时，也将提到其返回码。

要退出当前进程， shell提供命令exit，一般格式为：

```
exit n
```

其中， n为一数字。

[redacted]有在当前状态创建另一个 shell，将退出当前shell。如果在脚本中键入 exit[redacted]返回上一个命令返回值。有许多退出脚本值，但其中相对于脚本和一[redacted]，即：

退出状态0 [redacted]无错误。

退出状态1　退出失败，某处有错误。

可以在shell脚本中加入自己的退出状态（它将退出脚本）。本书鼓励这样做，因为另一个shell脚本或返回函数可能要从 shell脚本中抽取退出脚本。另外，相信加入脚本本身的退出脚本值是一种好的编程习惯。

如果愿意，用户可以在一个用户输入错误后或一个不可覆盖错误后或正常地处理结束后退出脚本。

注意　从现在起，本书所有脚本都将加入注释行。注释行将解释脚本具体含义，帮助用户理解脚本。可以在任何地方加入注释行，因为其本身被解释器忽略。注释行应以#开头。

18.2　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？这里需要从不同的目录中拷贝不同的文件。必须在命令执行前或最后的命令退出前决定处理方法。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。

18.2.1 流控制

if、then、else语句提供条件测试。测试可以基于各种条件。例如文件的权限、长度、数值或字符串的比较。这些测试返回值或者为真（ 0），或者为假（1）。基于此结果，可以进行相关操作。在讲到条件测试时已经涉及了一些测试语法。

case语句允许匹配模式、单词或值。一旦模式或值匹配，就可以基于这个匹配条件作其他声明。

18.2.2 循环

循环或跳转是一系列命令的重复执行过程，本书提到了 3种循环语句：

for 循环　每次处理依次列表内信息，直至循环耗尽。

Until 循环　此循环语句不常使用，until循环直至条件为真。条件部分在循环末尾部分。

While 循环　while循环当条件为真时，循环执行，条件部分在循环头。

流控制语句的任何循环均可嵌套使用，例如可以在一个 for循环中嵌入另一个for循环。

现在开始讲解循环和控制流，并举一些脚本实例。

从现在起，脚本中echo语句使用LINUX或BSD版本，也就是说使用 echo方法echo -e -n，意即从echo结尾中下一行执行命令。应用于 UNIX（系统V和BSD）的统一的echo命令参阅19章shell函数。

18.3 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
   then 命令1
  elif 条件2
then 命令2
else 命令3
```

fi

让我们来具体讲解if语句的各部分功能。

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不成立

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

if语句必须以单词fi终止。在if语句中漏写fi是最一般的错误。我自己有时也是这样。

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。

下面看一些例子。

18.3.1 简单的if语句

最普通的if语句是：

if条件

then 命令

fi

使用if语句时，必须将then部分放在新行，否则会产生错误。如果要不分行，必须使用命令分隔符。本书其余部分将采取这种形式。现在简单 if语句变为：

if 条件；then

命令

fi

注意，语句可以不这样缩排，但建议这样做，因为可以增强脚本的清晰程度。在条件流下采取命令操作更方便。下面的例子测试 10是否小于12，此条件当然为真。因为条件为真， if语句内部继续执行，这里只有一个简单的 echo命令。如果条件为假，脚本退出，因为此语句无else部分。

```
$ 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
```

18.3.2 变量值测试

通过测试设置为接受用户输入的变量可以测知用户是否输入信息。下面的例子中测试用

户键入return键后变量name是否包含任何信息。

```
$ pg iftest2
#!/bin/sh
# if test2
echo -n "Enter your name :"
read NAME
# did the user just hit return ????
if [ "$NAME" = "" ] ; then
  echo "You did not enter any information"
fi
$ iftest2
Enter your name :
You did not enter any information
```

18.3.3 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。对 grep使用if语句找出grep是否成功返回信息。下面的例子中 grep用于查看Dave是否在数据文件 data.file中，注意‘ Dave\> ’用于精确匹配。

```
$ 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
```

上面的例子中，grep输出定向到系统垃圾堆。如果匹配成功， grep返回0，将grep嵌入if语句；如果grep成功返回，if部分为真。

18.3.4 用变量测试grep输出

正像前面看到的，可以用 grep作字符串操作。下面的脚本中，用户输入一个名字列表，grep在变量中查找，要求其包含人名 Peter。

```
$ pg grepstr
#!/bin/sh
# grepstr
echo -n "Enter a list of names:"
read list
if echo $list | grep "Peter" > /dev/null 2>&1
then
  echo "Peter is here"
  # could do some processing here..
else
  echo "Peter's not in the list. No comment!"
fi
```

以下是对应输入名称的输出信息。

```
$ grepstr
Enter a list of names:John Louise Peter James
Peter is here
```

18.3.5 文件拷贝输出检查

下面测试文件拷贝是否正常，如果 cp命令并没有拷贝文件 myfile到myfile.bak，则打印错误信息。注意错误信息中`basename $0`打印脚本名。

如果脚本错误退出，一个好习惯是显示脚本名并将之定向到标准错误中。用户应该知道产生错误的脚本名。

```
$ 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。既然已经显示系统错误信息获知脚本失败，就没必要显示两次。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
```

脚本运行时，所有输出包括错误重定向至系统垃圾堆。

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

18.3.6 当前目录测试

当运行一些管理脚本时，可能要在根目录下运行它，特别是移动某种全局文件或进行权限改变时。一个简单的测试可以获知是否运行在根目录下。下面脚本中变量 DIRECTORY使用当前目录的命令替换操作，然后此变量值与 "/"字符串比较（/为根目录）。如果变量值与字符串不等，则用户退出脚本，退出状态为 1意味错误信息产生。

```
$ pg ifpwd
#!/bin/sh
# ifpwd
DIRECTORY=`pwd`
# grab the current directory
```

```
if [ "$DIRECTORY" != "/" ]; then
# is it the root directory ?
# no, the direct output to standard error, which is the screen by default.
  echo "You need to be in the root directory not $DIRECTORY to run this
       script" >&2
# exit with a value of 1, an error
  exit 1
fi
```

18.3.7 文件权限测试

可以用if语句测试文件权限，下面简单测试文件 test.txt是否被设置到变量LOGNAME。

```
$ pg ifwr
#!/bin/sh
# ifwr
LOGFILE=test.txt
echo $LOGFILE
if [ ! -w "$LOGFILE" ]; then
  echo " You cannot write to $LOGFILE " >&2
fi
```

18.3.8 测试传递到脚本中的参数

if语句可用来测试传入脚本中参数的个数。使用特定变量 $#，表示调用参数的个数。可以测试所需参数个数与调用参数个数是否相等。

以下测试确保脚本有三个参数。如果没有，则返回一个可用信息到标准错误，然后代码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
```

18.3.9 决定脚本是否为交互模式

有时需要知道脚本运行是交互模式（终端模式）还是非交互模式（ cron或at）。脚本也许

需要这个信息以决定从哪里取得输入以及输出到哪里，使用 test命令并带有-t选项很容易确认这一点。如果test返回值为1，则为交互模式。

```
$ pg ifinteractive
#!/bin/sh
# ifinteractive
if [ -t ]; then
  echo "We are interactive with a terminal"
else
  echo "We must be running from some background process probably
        cron or at "
fi
```

18.3.10 简单的if else语句

下一个if语句有可能是使用最广泛的：

if 条件

then

　命令1

else

　命令2

fi

使用if语句的else部分可在条件测试为假时采取适当动作。

18.3.11 变量设置测试

下面的例子测试环境变量 EDITOR是否已设置。如果 EDITOR变量为空，将此信息通知用户。如果已设置，在屏幕上显示编辑类型。

```
$ pg ifeditor
#!/bin/sh
# ifeditor
if [ -z $EDITOR ]; then
# the variable has not been set
  echo "Your EDITOR environment is not set"
else
# let's see what it is
  echo "Using $EDITOR as the default editor"
fi
```

18.3.12 检测运行脚本的用户

下面例子中，环境变量用于测试条件，即 LOGNAME是否包含root值。这类语句是加在脚本开头作为一安全性准则的普遍方法。当然 LOGNAME可用于测试任何有效用户。

如果变量不等root，返回信息到标准错误输出即屏幕，也就是通知用户不是 root，脚本然后退出，并带有错误值1。

如果字符串root等于LOGNAME变量，else部分后面语句开始执行。

实际上，脚本会继续进行正常的任务处理，这些语句在 fi后面，因为所有非root用户在脚本的前面测试部分已经被剔出掉了。

```
$ pg ifroot
#!/bin/sh
# ifroot
if [ "$LOGNAME" != "root" ]
# if the user is not root
then
  echo "You need to be root to run this script" >&2
  exit 1
else
# yes it is root
  echo "Yes indeed you are $LOGNAME proceed"
fi
# normal processing statements go here
```

18.3.13 将脚本参数传入系统命令

可以向脚本传递位置参数，然后测试变量。这里，如果用户在脚本名字后键入目录名，脚本将重设$1特殊变量为一更有意义的名字。即DIRECTORY。这里需测试目录是否为空，如果目录为空，ls -A将返回空，然后对此返回一信息。

```
$ pg ifdirec
#!/bin/sh
# ifdirec
# assigning $1 to DIRECTORY variable
DIRECTORY=$1
if [ "`ls -A $DIRECTORY`" = "" ] ; then
# if it's an empty string, then it's empty
  echo "$DIRECTORY is indeed empty"
else
# otherwise it is not
  echo "$DIRECTORY is not empty"
fi
```

也可以使用下面的脚本替代上面的例子并产生同样的结果。

```
$ pg ifdirec2
#!/bin/sh
# ifdirec2
DIRECTORY=$1
if [ -z "`ls -A $DIRECTORY`" ]
then
  echo "$DIRECTORY is indeed empty"
else
  echo "$DIRECTORY is not empty"
fi
```

18.3.14 null：命令用法

到目前为止，条件测试已经讲完了then和else部分，有时也许使用者并不关心条件为真或为假。

不幸的是if语句各部分不能为空——一些语句已经可以这样做。为解决此问题， shell提供了：空命令。空命令永远为真（也正是预想的那样）。回到前面的例子，如果目录为空，可以只在then部分加入命令。

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

18.3.15 测试目录创建结果

现在继续讨论目录，下面的脚本接受一个参数，并用之创建目录，然后参数被传入命令行，重设给变量DIRECTORY，最后测试变量是否为空。

```
if ["$DIRECTORY"=""]
```

也可以用

```
if[$# -lt 1]
```

来进行更普遍的参数测试。

如果字符串为空，返回一可用信息，脚本退出。如果目录已经存在，脚本从头至尾走一遍，什么也没做。

创建前加入提示信息，如果键入Y或y，则创建目录，否则使用空命令表示不采取任何动作。

使用最后命令状态测试创建是否成功执行，如果失败，返回相应信息。

```
$ pg ifmkdir
!/bin/sh
# ifmkdir
# parameter is passed as $1 but reassigned to DIRECTORY
DIRECTORY=$1
# is the string empty ??
if [ "$DIRECTORY" = "" ]
then
  echo "Usage :`basename $0` directory to create" >&2
  exit 1
fi
if [ -d $DIRECTORY ]
then : # do nothing
else
  echo "The directory does exist"
  echo -n "Create it now? [y..n] :"
  read ANS
  if [ "$ANS" = "y" ] || [ "$ANS" = "Y" ]
  then
    echo "creating now"
    # create directory and send all output to /dev/null
    mkdir $DIRECTORY >/dev/null 2>&1
    if [ $? != 0 ]; then
      echo "Errors creating the directory $DIRECTORY" >&2
      exit 1
    fi
  else :    # do nothing
  fi
```

执行上述脚本，显示：

```
$ ifmkdir dt
The directory does exist
Create it now? [y..n]:y
creating now
```

18.3.16 另一个拷贝实例

在另一个拷贝实例中，脚本传入两个参数（应该包含文件名），系统命令cp将$1拷入$2，输出至/dev/null。如果命令成功，则仍使用空命令并且不采取任何动作。

另一方面，如果失败，在脚本退出前要获知此信息。

```
$ 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
```

脚本运行，没有拷贝错误：

```
$ cp2 myfile.lex myfile.lex.bak
```

脚本运行带有拷贝错误：

```
$ ifcp2 myfile.lexx myfile.lex.bak
ifcp2: ERROR failed to copy myfile.lexx myfile.lex.bak
```

下面的脚本用sort命令将文件accounts.qtr分类，并输出至系统垃圾堆。没人愿意观察屏幕上300行的分类页。成功之后不采取任何动作。如果失败，通知用户。

```
$ 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
```

18.3.17 多个if语句

可能有时要嵌入if语句。为此需注意if和fi的相应匹配使用。

18.3.18 测试和设置环境变量

前面已经举例说明了如何测试环境变量 EDITOR是否被设置。现在如果未设置，则进一步为其赋值，脚本如下：

```
$ pg ifseted
#!/bin/sh
# ifseted
# is the EDITOR set ?

if [ -z $EDITOR ] ; then
  echo "Your EDITOR environment is not set"
  echo "I will assume you want to use vi..OK"
  echo -n "Do you wish to change it now? [y..n] :"
  read ANS

  # check for an upper or lower case 'y'
  if [ "$ANS" = "y" ] || [ "$ANS" = "Y" ]; then
    echo "enter your editor type :"
    read EDITOR
    if [ -z $EDITOR ] || [ "$EDITOR" = "" ]; then
      # if EDITOR not set and no value in variable EDITOR,
      # then set it to vi
      echo "No, editor entered, using vi as default"
      EDITOR=vi
      export EDITOR
    fi

    # got a value use it for EDITOR
    EDITOR=$EDITOR
    export EDITOR
    echo "setting $EDITOR"
  fi
else
# user
  echo "Using vi as the default editor"
  EDITOR=vi
  export vi
fi
```

脚本工作方式如下：首先检查是否设置了该变量，如果已经赋值，输出信息提示使用 vi作为缺省编辑器。vi被设置为编辑器，然后脚本退出。

如果未赋值，则提示用户，询问其是否要设置该值。检验用户输入是否为大写或小写 y，输入为其他值时，脚本退出。

如果输入Y或y，再提示输入编辑类型。使用 $EDITOR=“ ”测试用户是否未赋值和未点击return键。一种更有效的方法是使用 -z $EDITOR方法，本文应用了这两种方法。如果测试失败，返回信息到屏幕，即使用 vi做缺省编辑器，因而EDITOR赋值为vi。

如果用户输入了一个名字到变量 EDITOR，则使用它作为编辑器并马上让其起作用，即导出变量EDITOR。

18.3.19 检测最后命令状态

前面将目录名传入脚本创建了一个目录，脚本然后提示用户是否应创建目录。下面的例子创建一个目录，并从当前目录将所有 *.txt文件拷入新目录。但是这段脚本中用最后状态命令检测了每一个脚本是否成功执行。如果命令失败则通知用户。

```
$ pg ifmkdir2
#!/bin/sh
# ifmkdir2
DIR_NAME=testdirec
# where are we ?
THERE=`pwd`
# send all output to the system dustbin
mkdir $DIR_NAME > /dev/null 2>&1
# is it a directory ?
if [ -d $DIR_NAME ]; then
# can we cd to the directory
  cd $DIR_NAME
  if [ $? = 0 ]; then
  # yes we can
    HERE=`pwd`
    cp $THERE/*.txt $HERE
  else
    echo "Cannot cd to $DIR_NAME"  >&2
    exit 1
  fi
else
  echo "$cannot create directory $DIR_NAME" >&2
  exit 1
fi
```

18.3.20 增加和检测整数值

下面的例子进行数值测试。脚本包含了一个计数集，用户将其赋予一个新值就可改变它。脚本然后将当前值100加入一个新值。工作流程如下：

用户输入一个新值改变其值，如果键入回车键，则不改变它，打印当前值，脚本退出。

如果用户用y或Y响应新值，将提示用户输入增量。如果键入回车键，原值仍未变。键入一个增量，首先测试是否为数字，如果是，加入计数 COUNTOR中，最后显示新值。

```
$ pg ifcounter
#!/bin/sh
# ifcounter
COUNTER=100
echo "Do you wish to change the counter value currently set at $COUNTER ?
      [y..n] :"
read ANS
if [ "$ANS" = "y" ] || [ "$ANS" = "Y" ]; then
# yes user wants to change the value
  echo "Enter a sensible value "
  read VALUE
  # simple test to see if it's numeric, add any number to VALUE,
  # then check out return
  # code
  expr $VALUE + 10 > /dev/null 2>&1
    STATUS=$?
  # check return code of the expr
  if [ "$VALUE" = "" ] || [ "$STATUS" != "0" ]; then
  # send errors to standard error
    echo " You either entered nothing or a non-numeric " >&2
    echo " Sorry now exiting..counter stays at $COUNTER" >&2
```

```
    exit 1
  fi
# if we are here, then it's a number, so add it to COUNTER
  COUNTER=`expr $COUNTER + $VALUE`
  echo " Counter now set to $COUNTER"
else
# if we are here then user just hit return instead of entering a number
# or answered n to the change a value prompt
  echo " Counter stays at $COUNTER"
fi
```

运行结果如下：

```
$ ifcount
Do you wish to change the counter value currently set at 100 ? [y..n]:n
Counter stays at 100


$ ifcount
Do you wish to change the counter value currently set at 100 ? [y..n]:y
Enter a sensible value
fdg
 You either entered nothing or a non-numeric
 Sorry now exiting..counter stays at 100


$ ifcount
Do you wish to change the counter value currently set at 100 ? [y..n]:y
Enter a sensible value  250
Counter now set to 350
```

18.3.21 简单的安全登录脚本

以下是用户登录时启动应用前加入相应安全限制功能的基本框架。首先提示输入用户名和密码，如果用户名和密码均匹配脚本中相应字符串，用户登录成功，否则用户退出。

脚本首先设置变量为假 —— 总是假定用户输入错误， stty当前设置被保存，以便隐藏passwd域中字符，然后重新保存stty设置。

如果用户ID和密码正确（密码是mayday），明亮INVALID_USER和INVALID_PASSWD设置为no表示有效用户或密码，然后执行测试，如果两个变量其中之一为 yes，缺省情况下，脚本退出用户。

键入有效的ID和密码，用户将允许进入。这是一种登录脚本的基本框架。下面的例子中有效用户ID为dave或pauline。

```
$ pg ifpass
#!/bin/sh
# ifpass
# set the variables to false
INVALID_USER=yes
INVALID_PASSWD=yes
# save the current stty settings
SAVEDSTTY=`stty -g`
echo "You are logging into a sensitive area"
echo -n "Enter your ID name :"
read NAME
# hide the characters typed in
stty -echo
```

```
echo "Enter your password :"
read PASSWORD
# back on again
stty $SAVEDSTTY
if [ "$NAME" = "dave" ] || [ "$NAME" = "pauline" ]; then
  # if a valid then set variable
  INVALID_USER=no
fi

if [ "$PASSWORD" = "mayday" ]; then
# if valid password then set variable
INVALID_PASSWD=no
fi
if [ "$INVALID_USER" = "yes" -o "$INVALID_PASSWD" = "yes" ]; then
  echo "`basename $0 :` Sorry wrong password or userid"
  exit 1
fi
# if we get here then their ID and password are OK.
echo "correct user id and password given"
```

如果运行上述脚本并给一个无效用户：

```
$ ifpass
You are logging into a sensitive area
Enter your ID name  : dave
Enter your password :
ifpass :Sorry wrong password or userid
```

现在给出正确的用户和密码：

```
$ ifpass
You are logging into a sensitive area
Enter your ID name  : dave
Enter your password :
correct user id and password given
```

18.3.22 elif用法

if then else语句的elif部分用于测试两个以上的条件。

18.3.23 使用elif进行多条件检测

使用一个简单的例子，测试输入脚本的用户名。脚本首先测试是否输入一个名字，如果没有，则什么也不做。如果输入了，则用 elif测试是否匹配root、louise或dave，如果不匹配其中任何一个，则打印该名字，通知用户不是 root、louise或dave。

```
$ pg ifelif
#!/bin/sh
# ifelif
echo -n "enter your login name :"
read NAME
# no name entered do not carry on
if [ -z $NAME ] || [ "$NAME" = "" ]; then
  echo "You did not enter a name"
elif
  # is the name root
  [ "$NAME" = "root" ]; then
```

```
  echo "Hello root"
elif
  # or is it louise
  [ $NAME = "louise" ]; then
  echo "Hello louise"
elif
  # or is it dave
  [ "$NAME" = "dave" ]; then
  echo "Hello dave"
else
  # no it's somebody else
  echo "You are not root or louise or dave but hi $NAME"
fi
```

运行上述脚本，给出不同信息，得结果如下：

```
$ ifelif
enter your login name : dave
Hello dave

$ ifelif
enter your login name :
You did not enter a name

$ ifelif2
enter your login name : peter
You are not root or louise or dave but hi peter
```

18.3.24 多文件位置检测

假定要定位一个用户登录文件，已知此文件在 /usr/opts/audit/logs或/usr/local/audit/logs中，具体由其安装人决定。在定位此文件前，首先确保文件可读，此即脚本测试部分。如果未找到文件或文件不可读，则返回错误信息。脚本如下：

```
$ pg ifcataudit
#!/bin/sh
# ifcataudit
# locations of the log file
LOCAT_1=/usr/opts/audit/logs/audit.log
LOCAT_2=/usr/local/audit/audit.logs

if [ -r $LOCAT_1 ]; then
  # if it is in this directory and is readable then cat it
  echo "Using LOCAT_1"
  cat $LOCAT_1
elif
  # else it then must be in this directory, and is it readable
  [ -r $LOCAT_2 ]
then
  echo "Using LOCAT_2"
  cat $LOCAT_2
else
  # not in any of the directories...
  echo "`basename $0`: Sorry the audit file is not readable or cannot be
       located." >&2
  exit 1
```

```
fi
```

运行上面脚本，如果文件在上述两个目录之一中并且可读，将可以找到它。如果不是，返回错误并退出，下面结果失败，因为假想的文件并不存在。

```
$ ifcataudit
ifcataudit: Sorry the audit file is not readable or cannot be located.
```

18.4 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，执行相匹配的命令。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，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，再接受其他输入。

模式部分可能包括元字符，与在命令行文件扩展名例子中使用过的匹配模式类型相同，即：

* 任意字符。

？ 任意单字符。

[..] 类或范围中任意字符。

下面举例说明。

18.4.1 简单的case语句

下面的脚本提示输入 1到5，输入数字传入case语句，变量ANS设置为case取值测试变量ANS，ANS将与每一模式进行比较。

如果匹配成功，则执行模式里面的命令直至；;，这里只反馈非用户数字选择的信息，然后case退出，因为匹配已找到。

进程在case语句后仍可继续执行。

如果匹配未找到，则使用 *模式捕获此情况，这里执行错误信息输出。

```
$ 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
```

18.4.2 对匹配模式使用|

使用case时，也可以指定“|”符号作为或命令，例如vt100|vt102匹配模式vt100或vt102。下面的例子中，要求用户输入终端类型。如果输入为vt100或vt102，将匹配模式‘vt100|vt102’，执行命令是设置TERM变量为vt100。如果用户输入与模式不匹配，*用来捕获输入，其中命令为将TERM设置为vt100。最后在case语句外，导出TERM变量。由于使用*模式匹配，无论用户输入什么，TERM都将有一个有效的终端类型值。

```
$ 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"
```

运行脚本，输入一无效终端类型，

```
$ caseterm
 choices are.. vt100, vt102, vt220
enter your terminal type :vt900
caseterm : Unknown response
setting it to vt100 anyway, so there
Your terminal is set to vt100
```

如果输入一正确的终端类型，

```
$ case2
 choices are.. vt100, vt102, vt220
enter your terminal type :vt220
Your terminal is set to vt220
```

无论怎样，一个有效的终端类型被赋予用户。

18.4.3 提示键入y或n

case的一个有效用法是提示用户响应以决定是否继续进程。这里提示输入 y以继续处理，n退出。如果用户输入Y、y或yes，处理继续执行case语句后面部分。如果用户输入N、n或no或其他响应，用户退出脚本。

```
$ pg caseans
#!/bin/sh
# caseans
echo -n "Do you wish to proceed [y..n] :"
read ANS
  case $ANS in
  y|Y|yes|Yes) echo "yes is selected"
    ;;
  n|N) echo "no is selected"
    exit 0    # no error so only use exit 0 to terminate
    ;;
  *) echo "`basename $0` : Unknown response" >&2
    exit 1
    ;;
  esac
# if we are here then a y|Y|yes|Yes was selected only.
```

运行脚本，输入无效响应，得结果：

```
$ caseans
Do you wish to proceed [y..n] :df
caseans : Unknown response
```

给出有效响应：

```
$ caseans
Do you wish to proceed [y..n] :y
yes is selected
```

18.4.4 case与命令参数传递

可以使用case控制到脚本的参数传递。

下面脚本中，测试特定变量 $#，它包含传递的参数个数，如果不等于 1，退出并显示可用信息。

然后case语句捕获下列参数：passwd、start、stop或help，相对于每一种匹配模式执行进一步处理脚本。如果均不匹配，显示可用信息到标准错误输出。

```
$ 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
```

运行脚本，输入无效参数。

```
$ caseparam what
Usage:caseparam [start|stop|help]
```

输入有效参数，结果为：

```
$ caseparam stop
stopping..caseparam
```

18.4.5 捕获输入并执行空命令

不一定要在匹配模式后加入命令，如果你原本不想做什么，只是在进一步处理前过滤出意外响应，这样做是一种好办法。

如果要运行对应于一个会计部门的帐目报表，必须首先在决定运行报表的类型前确认用户输入一个有效的部门号，匹配所有可能值，其他值无效。用 case可以很容易实现上述功能。

下面的脚本中如果用户输入部门号不是 234、453、655或454，用户退出并返回可用信息。一旦响应了用户的有效部门号，脚本应用同样的技术取得报表类型，在 case语句末尾显示有效的部门号和报表类型。脚本如下：

```
$ pg casevalid
!/bin/sh
# casevalid
TYPE=""
echo -n "enter the account dept No: :"
read ACC
  case $ACC in
    234);;
    453);;
```

```
    655);;
    454);;
  *) echo "`basename $0`: Unknown dept No:"  >&2
    echo "try..234,453,655,454"
    exit 1
    ;;
  esac

# if we are here, then we have validated the dept no
echo " 1 . post"
echo " 2 . prior"
echo -n "enter the type of report: "
read ACC_TYPE
  case $ACC_TYPE in
  1)TYPE=post;;
  2)TYPE=prior;;
  *)  echo "`basename $0`: Unknown account type." >&2
    exit 1
    ;;
  esac

# if we get here then we are validated!
echo "now running report for dept $ACC for the type $TYPE"
# run the command report..
```

输入有效部门号：

```
$ casevalid
enter the account dept No: :234
 1 . post
 2 . prior
enter the type of report:2
now running report for dept 234 for the type  prior
```

输入无效部门号：

```
$ casevalid
enter the account dept No: :432
casevalid: Unknown dept No:
try..234,453,655,454
```

输入无效的报表类型：

```
$ casevalid
enter the account dept No: :655
 1 . post
 2 . prior
enter the type of report:4
casevalid: Unknown account type.
```

18.4.6 缺省变量值

如果在读变量时输入回车键，不一定总是退出脚本。可以先测试是否已设置了变量，如果未设置，可以设置该值。

下面的脚本中，要求用户输入运行报表日期。如果用户输入回车键，则使用缺省日期星期六，并设置为变量when的取值。

如果用户输入另外一天，这一天对于case语句是运行的有效日期，即星期六、星期四、星

期一。注意，这里结合使用了日期缩写作为捕获的可能有效日期。

脚本如下：

```
$ pg caserep
#!/bin/sh
# caserep
echo "      Weekly Report"
echo -n "What day do you want to run report [Saturday] :"


# if just a return is hit then except default which is Saturday
read WHEN
echo "validating..${WHEN:="Saturday"}"
  case $WHEN in
  Monday|MONDAY|mon)
    ;;
  Sunday|SUNDAY|sun)
    ;;
  Saturday|SATURDAY|sat)
    ;;
  *) echo " Are you nuts!, this report can only be run on " >&2
    echo " on a Saturday, Sunday or Monday" >&2
    exit 1
    ;;
  esac
echo "Report to run on $WHEN"
# command here to submitted actual report run
```

对于正确输入：

```
$ caserep
      Weekly Report
What day do you want to run report [Saturday] :
validating..Saturday
Report to run on Saturday
```

对于错误输入：

```
$ caserep
      Weekly Report
What day do you want to run report [Saturday] :Tuesday
validating..Tuesday
 Are you nuts!, this report can only be run on
 on a Saturday, Sunday or Monday
```

可以推断出case语句有时与if then else语句功能相同，在某些条件下，这种假定是正确的。

18.5 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，使用变量名访问列表中取值。命令可为任何有效的shell命令和语句。变量名为任何单词。 In列表用法是可选的，如果不用它， 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，下面看一些例子。

18.5.1 简单的for循环

此例仅显示列表1 2 3 4 5，用变量名访问列表。

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

运行上述脚本，输出：

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

18.5.2 打印字符串列表

下面for循环中，列表包含字符串“ orange red blue grey ”，命令为echo，变量名为loop，echo命令使用$loop反馈出列表中所有取值，直至列表为空。

```
$ pg forlist
#!/bin/sh
# forlist
for loop in "orange red blue grey"
do
  echo $loop
done

$ forlist
orange red blue grey
```

也可以在循环体中结合使用变量名和字符串。

```
echo " this is the fruit $loop"
```

结果为：

```
This is the fruit orange red blue grey
```

18.5.3 对for循环使用ls命令

这个循环执行ls命令，打印当前目录下所有文件。

```
$ pg forls
#!/bin/sh
```

```
# forls
for loop in `ls `
do
  echo $loop
done

$ forls
array
arrows
center
center1
center2
centerb
```

18.5.4 对for循环使用参数

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参数。实际上即指明：

```
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
  echo $params
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
```

下面的脚本包含in"$@"，结果与上面的脚本相同。

```
$ pg forparam3
#!/bin/sh
# forparam3
for params in "$@"
do
  echo "You supplied $params as a command line option"
done
  echo $params
done
```

对上述脚本采取进一步动作。如果要查看一系列文件，可在 for循环里使用find命令，利用命令行参数，传递所有要查阅的文件。

```
$ pg forfind
#!/bin/sh
# forfind
```

```
for loop
do
  find / -name $loop -print
done
```

脚本执行时，从命令行参数中取值并使用 find命令，这些取值形成-name选项的参数值。

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

18.5.5 使用for循环连接服务器

因为for循环可以处理列表中的取值，现设变量为网络服务器名称，并使用 for循环连接每一服务器。

```
$ pg forping
#!/bin/sh
# forping
HOSTS="itserv dnssevr acctsmain ladpd ladware"
for loop in $HOSTS
do
  ping -c 2 $loop
done
```

18.5.6 使用for循环备份文件

可以用for循环备份所有文件，只需将变量作为 cp命令的目标参数。这里有一变量 .bak，当在循环中使用cp命令时，它作为此命令目标文件名。列表命令为 ls。

18.5.7 多文件转换

匹配所有以LPSO开头文件并将其转换为大写。这里使用了 ls和cat命令。ls用于查询出相关文件，cat用于将之管道输出至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
```

18.5.8 多sed删除操作

下面的例子中，sed用于删除所有空文件，并将输出导至以 .HOLD.mv为扩展名的新文件中，mv将这些文件移至初始文件中。

```
#!/bin/sh
# forsed
for files in `ls LPSO*`
do
  sed -e "/^$/d" $files >$files.HOLD
  mv $files.HOLD $files
done
```

18.5.9 循环计数

前面讨论expr时指出，循环时如果要加入计数，使用此命令。下面使用 ls在for循环中列出文件及其数目。

```
$ pg forcount
#!/bin/sh
# forcount
counter=0
for files in *
do
  # increment
  counter=`expr $counter + 1`
done
  echo "There are $counter files in `pwd` we need to process"

$ forcount
There are 45 files in /apps/local we need to process
```

使用wc命令可得相同结果。

```
$ ls |wc -l
     45
```

18.5.10 for循环和本地文档

在for循环体中可使用任意命令。下面的例子中，一个变量包含所有当前登录用户。使用who命令并结合awk语言可实现此功能。然后 for循环循环每一用户，给其发送一个邮件，邮件信息部分用一个本地文档完成。

```
$ pg formailit
#!/bin/sh
```

```
# formailit
WHOS_ON=`who -u | awk '{print $1}'`
for user in $WHOS_ON
do
mail $user << MAYDAY
Dear Colleagues,
It's my birthday today, see you down the
club at 17:30 for a drink.

See ya.
$LOGNAME
MAYDAY
Done
```

上述脚本的邮件信息输出为：

```
$ pg mbox
Dear Colleagues,
It's my birthday today, see you down the
club at 17:30 for a drink.

See ya.
dave
```

18.5.11 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
```

```
$ 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
```

18.6 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
          命令1
          ...
done
```

条件可为任意测试条件，测试发生在循环末尾，因此循环至少执行一次 —— 请注意这一点。

下面是一些实例。

18.6.1 简单的until循环

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

如果找到了root，循环结束，并向用户 simon发送邮件，通知他用户 root已经登录，注意这里sleep命令用法，它经常用于 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。

```
$ 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
```

18.6.2 监视文件

下面例子中，until循环不断挂起做睡眠，直至文件 /tmp/monitor.lck被删除。文件删除后，脚本进入正常处理过程。

```
$ pg until_lck
!/bin/sh
# until_lck
LOCK_FILE=/tmp/process.LCK
until  [ !  -f $LOCK_FILE ]

do
  sleep 1
done
echo "file deleted "
# normal processing now, file is present
```

上述例子是使脚本与其他处理过程协调工作的一种方法。还有另外一种方法使脚本间互相通信。假定有另一段脚本 process.main用于搜集本地网络所有机器的信息并将之放入一个报表文件。

当脚本 process.main运行时，创建了一个 LCK文件（锁文件），上面脚本必须接收process.main搜集的信息，但是如果 process仍然在修改报表文件时试图处理该文件就不太好了。

为克服这些问题，脚本process.main创建了一个LCK文件，当它完成时，就删除此文件。

上述脚本将挂起，等待 LCK文件被删除，一旦 LCK文件删除，上述脚本即可处理报表文件。

18.6.3 监视磁盘空间

until循环做监视条件也很有用。假定要监视文件系统容量，当它达到一定水平时通知超级用户。

下面的脚本监视文件系统 /logs，不断从变量 $LOOK_OUT中抽取信息， $LOOK_OUT包含使用awk和grep得到的/logs容量。

如果容量达到 90%，触发命令部分，向超级用户发送邮件，脚本退出。必须退出，如果不退出，条件保持为真（例如，容量总是保持在 90%以上），将会不断的向超级用户发送邮件。

```
$ pg until_mon
#!/bin/sh
# until_mon
# get percent column and strip off header row from df
LOOK_OUT=`df |grep /logs | awk '{print $5}'| sed 's/%//g'`
echo $LOOK_OUT
until [ "$LOOK_OUT" -gt "90" ]
do
```

```
  echo "Filesystem..logs is nearly full" | mail root
  exit 0
done
```

18.7 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。

18.7.1 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
```

运行上述脚本，返回数字1到5，然后终止。

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

18.7.2 使用while循环读键盘输入

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。

```
$ 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
```

脚本运行时，输入可能是：

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

18.7.3 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，当不再有读取数据时条件为真。 while循环使用输入重定向以保证从文件中读取数据。注意整行数据被设置为单变量 $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
```

18.7.4 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个域需要加域分隔，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
```

18.7.5 带有测试条件的文件处理

大部分while循环里都带有一些测试语句，以决定下一步的动作。

这里从人员文件中读取数据，打印所有细节到一个保留文件中，直至发现 James Lenod，脚本退出。测试前反馈的信息要确保“ James Lenod ”加入保留文件中。

注意，所有变量在脚本顶端被设置完毕。这样当不得不对变量进行改动时可以节省时间和输入。所有编辑都放在脚本顶端，而不是混于整个脚本间。

```
$ pg whileread_file
#!/bin/sh
# whileread_file
# initialise variables
SAVEDIFS=$IFS
IFS=:
HOLD_FILE=hold_file
NAME_MATCH="James Lenod"
INPUT_FILE=names.txt

# create a new HOLD_FILE each time, in case script is continuously run
>$HOLD_FILE
while read NAME DEPT ID
do
  # echo all information into holdfile with redirection
  echo $NAME $DEPT $ID >>$HOLD_FILE
  # is it a match ???
  if [ "$NAME" = "$NAME_MATCH" ]; then
    # yes then nice exit
    echo "all entries up to and including $NAME_MATCH are in $HOLD_FILE"
    exit 0
  fi
done < $INPUT_FILE
# restore IFS
IFS=$SAVEDIFS
```

还可以采取进一步动作，列出多少个雇员属于同一部门。这里保持同样的读方式。假定每个域都有一个变量名，然后在case语句里用expr增加每行匹配脚本。任何发现的未知部门知

识反馈到标准错误中，如果一个无效部门出现，没有必要退出。

```
$ 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
```

18.7.6 扫描文件行来进行数目统计

一个常用的任务是读一个文件，统计包含某些数值列的数值总和。下面的文件包含有部门STAT和GIFT所卖的商品数量。

```
$ pg total.txt
STAT    3444
GIFT     233
GIFT     252
GIFT     932
STAT     212
STAT     923
GIFT     129
```

现在的任务是要统计部门 GIFT所卖的各种商品数量。使用 expr保存统计和，看下面的expr语句。变量LOOP和TOTAL首先在循环外初始化为0，循环开始后， ITEMS加入TOTAL，第一次循环只包含第一种商品，但随着过程继续， ITEMS逐渐加入TOTAL。

下面的expr语句不断增加计数。

```
LOOP=0
TOTAL=0
...
while...
TOTAL=`expr $TOTAL + $ITEMS`
ITEMS=`expr $ITEMS + 1`
done
```

使用expr语句时容易犯的一个错误是开始忘记初始化变量。

```
LOOP=0
TOTAL=0
```

如果真的忘了初始化，屏幕上将布满 expr错误。

如果愿意，可以在循环内初始化循环变量。

```
TOTAL=`expr ${TOTAL:=0} + ${ITEMS}`
```

上面一行如果变量TOTAL未赋值，将其初始化为0。这是在expr里初始化变量的第一个例子。另外在循环外要打印出最后总数。

```
$ pg $total
#!/bin/sh
# total
# init variables
LOOP=0
TOTAL=0
COUNT=0

echo "Items  Dept"
echo "____________"
while read DEPT ITEMS
do
  # keep a count on total records read
  COUNT=`expr $COUNT + 1`
  if [ "$DEPT" = "GIFT" ]; then
    # keep a running total
    TOTAL=`expr $TOTAL + $ITEMS`
    ITEMS=`expr $ITEMS + 1`
    echo -e "$ITEMS\t$DEPT"
  fi
  #echo $DEPT $ITEMS
done < total.txt
echo "============"
echo $TOTAL
echo "There were $COUNT entries altogether in the file"
```

运行脚本，得到：

```
$ total
Items  Dept

____________
234     GIFT
253     GIFT
933     GIFT
130     GIFT
============
```

```
1546
There were 7 entries altogether in the file
```

18.7.7 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。使用这项技术时，不要忘了不断进行检查，因为它实际上读了大量的记录。

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

每次读两个记录，下面的例子对记录并不做实际测试。

脚本如下：

```
$ 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
-----------------------------
```

18.7.8 忽略#字符

读文本文件时，可能要忽略或丢弃遇到的注释行，下面是一个典型的例子。

假定要使用一般的while循环读一个配置文件，可拣选每一行，大部分都是实际操作语句。有时必须忽略以一定字符开头的行，这时需要用 case语句，因为#是一个特殊字符，最好首先

用反斜线屏蔽其特殊意义，在#符号后放一个星号*，指定*后可包含任意字符。

配置文件如下：

```
$ 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!!!
```

忽略#符号的实现脚本如下：

```
$ 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 exist or is empty"
  exit 1
fi
```

运行得：

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

18.7.9 处理格式化报表

读报表文件时，一个常用任务是将不想要的行剔除。以下是库存商品水平列表，我们感兴趣的是那些包含商品记录当前水平的列

```
$ pg order
#########  RE-ORDER REPORT #########
 ITEM              ORDERLEVEL     LEVEL
########################################
Pens               14             12
Pencils            15             15
Pads                7              3
Disks               3              2
Sharpeners          5              1
########################################
```

我们的任务是读取其中取值，决定哪些商品应重排。如果重排，重排水平应为现在商品

的两倍。输出应打印需要重排的每种商品数量及重排总数。

我们已经知道可以忽略以某些字符开始的行，因此这里没有问题。首先读文件，忽略所有注释行和以‘ ITEM ’开始的标注行。读取文件至一临时工作文件中，为确保不存在空行，用sed删除空行，需要真正做的是过滤文本文件。脚本如下：

```
$ pg whileorder
#!/bin/sh
# whileorder
INPUT_FILE=order
HOLD=order.tmp

if [ -s $INPUT_FILE ]; then
  # zero the output file, we do not want to append!
  >$HOLD
  while read LINE
  do
    case $LINE in
    \#*|ITEM*) ;;    # ignore any # or the line with ITEM

    *)
      # redirect the output to a temp file
      echo $LINE >>$HOLD
      ;;
    esac
  done <$INPUT_FILE
  # use to sed to delete any empty lines, if any
  sed -e '/^$/d' order.tmp >order.$$
  mv order.$$ order.tmp
else
  echo "`basename $0` : Sorry $INPUT_FILE does not exist or is empty"
fi
```

输出为：

```
$ pg order.tmp
Pens        14 12
Pencils     15 15
Pads         7  3
Disks        3  2
Sharpeners   5  1
```

现在要在另一个while循环中读取临时工作文件，使用 expr对数字进行数值运算。

```
$ pg whileorder2
#!/bin/sh
# whileorder2
# init the variables
HOLD=order.tmp
RE_ORDER=0
ORDERS=0
STATIONERY_TOT=0
if [ -s $HOLD ]; then
  echo "============ STOCK RE_ORDER REPORT ================"
  while read ITEM REORD LEVEL
  do
    # are we below the reorder level for this item??
```

```
    if [ "$LEVEL" -lt "$REORD" ]; then
      # yes, do the new order amount
      NEW_ORDER=`expr $REORD + $REORD`
      # running total of orders
      ORDERS=`expr $ORDERS + 1`
      # running total of stock levels
      STATIONERY_TOT=`expr $STATIONERY_TOT + $LEVEL`
      echo "$ITEM need reordering to the amount $NEW_ORDER"
    fi
  done <$HOLD
  echo "$ORDERS new items need to be ordered"
  echo "Our reorder total is $STATIONERY_TOT"
else
else
  echo "`basename $0` : Sorry $HOLD does not exist or is empty"
fi
```

以下为依据报表文件运行所得输出结果。

```
$ whileorder
============ STOCK REORDER REPORT ===============
Pens need reordering to the amount 28
Pads need reordering to the amount 14
Disks need reordering to the amount 6
Sharpeners need reordering to the amount 10
4 new items need to be ordered
Our reorder total is 18
```

将两段脚本结合在一起很容易。实际上这本来是一个脚本，为讲解方便，才将其分成两个。

18.7.10 while循环和文件描述符

第5章查看文件描述符时，提到有必要用while循环将数据读入一个文件。使用文件描述符3和4，下面的脚本进行文件myfile.txt到myfile.bak的备份。注意，脚本开始测试文件是否存在，如果不存在或没有数据，脚本立即终止。还有while循环用到了空命令（:），这是一个死循环，因为null永远返回真。尝试读至文件结尾将返回错误，那时脚本也终止执行。

```
$ pg copyfile
#!/bin/sh
# copyfile
FILENAME=myfile.txt
FILENAME_BAK=myfile.bak
if [ -s $FILENAME ]; then
  # open FILENAME for writing
  # open FILENAME for reading
  exec 4>$FILENAME_BAK
  exec 3<$FILENAME

# loop forever until no more data and thus an error so we are at end of
file
  while :
  do
    read LINE <&3
    if [ "$?" -ne 0 ]; then
      # errors then close up
```

```
        exec 3<&-
        exec 4<&-
        exit
      fi
    # write to FILENAME_BAK
    echo $LINE>&4
    done
else
    echo "`basename $0` : Sorry, $FILENAME is not present or is empty" >&2
fi
```

18.8 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。

18.8.1 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。例如如果在两层循环内，用 break 2刚好跳出整个循环。

18.8.2 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 a number between 1 and 5"
    ;;
  *) echo "Wrong number..bye"
    break
    ;;
  esac
done
```

18.8.3 continue

continue命令类似于break命令，只有一点重要差别，它不会跳出循环，只是跳过这个循环步。

18.8.4 浏览文件行

下面是一个前面用过的人人文件列表，但是现在加入了一些头信息。

```
$ pg names2.txt
------LISTING OF PERSONNEL FILE-----
--- TAKEN AS AT 06/1999 -----
Louise Conrad:Accounts:ACC8987
Peter James:Payroll:PR489
Fred Terms:Customer:CUS012
James Lenod:Accounts:ACC887
Frank Pavely:Payroll:PR489
```

假定现在需要处理此文件，看过文件之后知道头两行并不包含个人信息，因此需要跳过这两行。

也不需要处理雇员Peter James，这个人已经离开公司，但没有从人员文件中删除。

对于头信息。只需简单计算所读行数，当行数大于 2时开始处理，如果处理人员名字为Peter James，也将跳过此记录。脚本如下：

```
$ pg whilecontinue
#!/bin/sh
# whilecontinue
SAVEDIFS=$IFS
IFS=:
INPUT_FILE=names2.txt
NAME_HOLD="Peter James"
LINE_NO=0

if [ -s $INPUT_FILE ]; then
  while read NAME DEPT ID
  do
    LINE_NO=`expr $LINE_NO + 1`
    if [ "$LINE_NO" -le 2 ]; then
      # skip if the count is less than 2
      continue
    fi
    if [ "$NAME" = "$NAME_HOLD" ]; then
      # skip if the name in NAME_HOLD is Peter James
      continue
    else
      echo " Now processing...$NAME $DEPT $ID"
      # all the processing goes here
    fi
  done < $INPUT_FILE
IFS=$SAVEDIFS
else
  echo "`basename $0 ` : Sorry file not found or there is no data in the
file" >&2
  exit 1
fi
```

运行上面脚本，得出：

```
$ whilecontinue
Louise Conrad Accounts ACC8987
Fred Terms Customer CUS012
```

```
James Lenod Accounts ACC887
Frank Pavely Payroll PR489
```

18.9 菜单

创建菜单时，在while循环里null空命令很合适。hile加空命令null意即无限循环，这正是一个菜单所具有的特性。当然除非用户选择退出或是一个有效选项。

创建菜单只需用while循环和case语句捕获用户输入的所有模式。如果输入无效，则报警，反馈错误信息，然后继续执行循环直到用户完成处理过程，选择退出选项。

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。由于测试原因，所有选项使用的是系统命令。

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] >
```

首先，使用命令替换设置日期，主机名和用户。日期格式为 /DD/MM/YYYY，参数格式为：

```
$ date +%d/%m/%y
32/05/1999
```

对于主机名，使用hostname-s选项只抽取主机名部分。主机名有时也包括了完全确认的域名。当然如果要在屏幕上显示这些，那就更好了。

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`
```

对于while循环，只需将空命令直接放在while后，即为无限循环，格式为：

```
while :
do
    命令
done
```

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
```

此技术用于菜单屏幕，也将用于帮助屏幕。帮助屏幕不像这样复杂。

用case语句控制用户选择。菜单选择有：

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

case语句应控制所有这些模式，不要忘了将大写与小写模式并列在一起，因为有时用户会关闭或打开CAPS LOCK键。因为菜单脚本不断循环，所以应该允许用户退出，如果用户选择Q或q键，脚本应退出，此时脚本带有0值。

如果用户选择无效，应发出警报并带有警告信息。虽然本章开始说过从现在开始一直使用LINUX或BSD echo语句版本，这里必须使用系统V版本发出警报：

```
echo "\007 the bell ring"
```

用一个简单的echo和读语句锁屏直到用户点击回车键，这样任何信息或命令输出将可视。

也需要清屏，为此可使用tput命令（后面讨论tput），如果不这样做，使用clear命令也可以。到此所有功能已经具备了，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
```

18.10 小结

在任何合理脚本的核心部分都有某种流控制，如果要求脚本具有智能性，必须能够进行判断和选择。

本章讲述了怎样使用控制流写一段优美的脚本，而不只是完成基本功能。这里也学到了怎样处理列表和循环直至条件为真或为假。