第 3 章 Linux 脚本编程

CHAPTER 3

在 Linux 系统中，虽然有各种各样的图形化接口工具，但是 shell 仍然是一个非常灵活的工具。shell 不仅是一种命令语言，也是一种程序设计语言。用户可以通过使用 shell 编程实现大量的任务自动化，shell 擅长系统管理任务，尤其适合那些对易用性、可维护性和便携性要求高的任务。脚本应用知识是必需的。一般来说，一个 Linux 机器启动后，它会执行在/etc/rc.d 目录下的 shell 脚本重建系统环境并且启动各种服务，理解这些启动脚本的细节对分析系统的运作行为并修改系统行为具有重大意义。

微课视频

3.1 常用 shell 命令

在 shell 脚本中可以使用任意的 UNIX 命令，有一些相对常用的命令用来进行文件和文字操作。常用 shell 命令如表 3-1 所示。

表 3-1 常用 shell 命令

命 令	说 明
echo "some text"	将文字内容打印在屏幕上
ls	文件列表
wc -l file	计算文件行数
wc -w file	计算文件中的单词数
wc -c file	计算文件中的字符数
cp sourcefile destfile	文件复制
mv oldname newname	重命名文件或移动文件
rm file	删除文件
grep 'pattern' file	在文件内搜索字符串，例如：grep 'searchstring' file.txt
cat file.txt	输出文件内容到标准输出设备（屏幕）上
file somefile	获取文件类型
read var	提示用户输入，并将输入值赋值给变量
sort file.txt	对 file.txt 文件中的行进行排序
uniq	删除文本文件中出现的行列，例如：sort file.txt | uniq
expr	进行数学运算，例如：expr 2 "+" 3

续表

命　令	说　明
find	搜索文件，例如，根据文件名搜索：find . -name filename -print
tee	将数据输出到标准输出设备(屏幕)和文件上，例如：somecommand | tee outfile
basename file	返回不包含路径的文件名，例如：basename /bin/tux 将返回 tux
dirname file	返回文件所在路径，例如：dirname /bin/tux 将返回/bin
head file	打印文本文件开头几行
tail file	打印文本文件末尾几行
sed	sed 是一个基本的查找替换程序。可以从标准输入（比如命令管道）读入文本，并将结果输出到标准输出（屏幕）上。不要和 shell 中的通配符相混淆。例如，将 linuxfocus 替换为 LinuxFocus：cat text.file | sed 's /linuxfocus/LinuxFocus/' > newtext.file
awk	awk 用来从文本文件中提取字段。默认的字段分割符是空格，可以使用-F 指定其他分割符。例如：cat file.txt | awk -F， '{print "，" }'，这里使用“，”作为字段分割符，同时打印第一个和第三个字段

微课视频

3.2 脚本编写基础

shell 脚本的第一行必须是#!/bin/sh 格式，符号#！用来指定该脚本文件的解析程序。当编译好脚本后，如果要执行该脚本，还必须使其具有可执行的属性，例如：chmod +x filename。

3.2.1 特殊字符

脚本文件涉及特殊字符较多，本节重点介绍在脚本中出现频率较高的字符。

1．#

该字符表示注释。以#开头的行（#!是例外）是注释行，注释也可以出现在一个命令语句的后面，注释行前面也可以有空白字符。

在同一行中，命令不能跟在注释语句的后面，因为这种情况下，系统无法分辨注释的结尾。命令只能放在同一行的行首。用另外的一个新行开始下一个注释。

2．；

该字符为命令分割符（分号），分割符允许在同一行里有两个或更多的命令。

3．；；

该字符为 case 语句分支的结束符（双分号）。

4．.

“点”（.）作为一个文件名的组成部分，当点（.）以一个文件名为前缀时，使该文件变成了隐藏文件，在使用 ls 命令时，一般不会显示这种隐藏文件。作为目录名时，单个点（.）表示当前目录，两个点（..）表示上一级目录。

5．"

该字符为部分引用（双引号）。"STRING"的引用会使 STRING 里的特殊字符能够被解释。

6．'

该字符为完全引用（单引号）。'STRING'能引用 STRING 里的所有字符（包括特殊字符

也会被原样引用)。这是一个比使用双引号（"）更强的引用。

7．，

该字符为逗号操作符，用于连接多个数学表达式，每个数学表达式都被求值，但只有最后一个表达式的值被返回。例如：

```
let "t2=((a=9，15/3))" #设置"a=9"且"t2=15/3"
```

8．\

该字符为转义符（反斜杠)。用于单个字符的引用机制。

\X“转义”字符为 X，它有“引用”X 的作用，也等同于直接在单引号里的'X'。\也可以用于引用双引号（"）和单引号（'），这时双引号和单引号就表示普通的字符，而不表示引用。

9．/

该字符为文件路径的分隔符（斜杠)。用于分隔一个文件路径的各个部分。例如：/home/bozo/projects/Makefile。同时，它也是算术操作符中的除法运算符。

10．`

该字符表示命令替换。`command`结构使字符（`）引住的命令（command）的执行结果能赋值给一个变量。它也被称为后引号或斜引号。

11．:

该字符表示空命令（冒号)。该命令的意思是空操作。它一般被认为与 shell 的内建命令 true 是一样的。

12．!

该字符为取反操作符。取反一个测试结果或退出状态（感叹号)。取反操作符（!）取反一个命令的退出状态，它也取反一个测试操作。例如，它能改相等符(=)为不等符(!=)。取反操作符（!）是 bash 的关键字。

13．*

该字符可表示通配符（*)，是用于匹配文件名扩展的通配符。它自动匹配给定的目录下的每个文件。

该字符也可表示算术操作符。在计算时，星号（*）表示乘法运算符。两个星号（**）表示求幂运算符。

14．?

该字符为测试操作符。在一些表达式中，问号（?）表示一个条件测试；在双括号结构里，问号（?）表示 C 语言风格的三元操作符；在参数替换表达式里，问号（?）测试一个变量是否被设置了值。

15．$

该字符表示变量替换（引用一个变量的内容)。一个变量名前面加一个$字符前缀表示引用该变量的内容。

16．( )

该字符表示命令组。一组由圆括号括起来的命令是新开一个子 shell 来执行的。因为是在子 shell 里执行，所以在圆括号里的变量不能被脚本的其他部分访问。因此父进程（即脚

本进程）不能存取子进程（即子 shell）创建的变量。

17．{}

该字符表示代码块（花括号）。这个结构也是一组命令代码块，它是匿名的函数。与函数不同的是，在代码块里的变量仍然能被脚本后面的代码访问。由花括号括起的代码块可以引起输入/输出的 I/O 重定向。

18．>，&>，>&，>>

该字符表示重定向。例如：

scriptname>filename，把命令 scriptname 的输出重定向到文件 filename 中。如果文件 filename 存在则将会被覆盖。

command&>filename，把命令 command 的标准输出（stdout）和标准错误（stderr）重定向到文件 filename 中。

command>&2，把命令 command 的标准输出（stdout）重定向到标准错误（stderr）。

scriptname>>filename appends，把脚本 scriptname 的输出追加到文件 filename。如果 filename 不存在，则它会被创建。

19．|

该字符表示管道。把上一个命令的输出传给下一个命令，这是连接命令的一种方法。

3.2.2 变量和参数

1．变量替换

变量名表示变量的值保存的地方，引用变量的值称为变量替换。如果 variable1 是一个变量名，那么$variable1 就是引用该变量的值，即这个变量包含的数据。

2．变量赋值

用“=”对变量进行赋值，“=”的左右两边不能有空白符。

3．bash 变量无类型

不同于许多其他编程语言，bash 不以“类型”区分变量。本质上说，bash 变量是字符串，但是根据环境的不同，bash 允许变量有整数计算和比较操作，其中决定因素是变量的值是否只含有数字。

示例 3.2.2-1 对变量操作的脚本如下：

```
#!/bin/sh
a="hello world"          #对变量赋值
echo "A is:"             #打印变量 a 的内容
echo $a
```

有时变量名很容易与其他文字混淆，例如：

```
num=2
echo "this is the $numnd"
```

上面代码并不会打印出“this is the 2nd”，而仅打印“this is the”，因为 shell 会去搜索变量 numnd 的值，但是这个变量是没有值的。可以使用花括号来告诉 shell 所要打印的是 num 变量，修改如下：

```
num=2
echo "this is the ${num}nd"
```

这将打印“this is the 2nd”。

4．局部变量

局部变量只在代码块或一个函数里有效。如果变量用 local 来声明，那么它只能在该变量声明的代码块中可见。这个代码块就是局部“范围”。在一个函数内，局部变量意味着该变量只有在函数代码块内才有意义。

示例 3.2.2-2　局部变量使用方法的脚本如下:

```
#!/bin/bash
hello="var1"
echo $hello
function func1 {
     local hello="var2"
     echo $hello
     }
func1
echo $hello
```

打开超级终端，建立该脚本文件，运行结果如下：

```
[root@bogon chapter2]# ./Example2.1.2-2
var1
var2
var1
```

从结果中能看出局部变量的使用方法。

5．位置参数

命令行传递给脚本的参数是$0，$1，$2，$3，…

$0 是脚本的名字，$1 是第一个参数，$2 是第二个参数，$3 是第三个参数，以此类推。在位置参数$9 之后的参数必须用括号括起来，例如：${10}，${11}，${12}。

特殊变量$*和$@表示所有的位置参数。

示例 3.2.2-3　位置参数代码如下：

```
#!/bin/sh
echo "number of vars:"$#
echo "values of vars:"$*
echo "value of var1:"$1
echo "value of var2:"$2
echo "value of var3:"$3
echo "value of var4:"$4
```

打开超级终端，建立该脚本文件，运行结果为：

```
root@ubuntu ://home/linux/chapter3# ./Example3.2.2-3  1 2 3 4 5
number of vars:5
values of vars:1 2 3 4 5
```

```
value of var1:1
value of var2:2
value of var3:3
value of var4:4
```

3.2.3 退出和退出状态

exit 命令一般用于结束一个脚本，就像 C 语言的 exit 一样。它也能返回一个值给父进程。每个命令都能返回一个退出状态（有时也看作返回状态）。如果一个命令执行成功，则返回 0；如果执行不成功，则返回一个非零值，此值通常可以被解释成一个对应的错误值。同样地，脚本里的函数和脚本自身都会返回一个退出状态码。在脚本或函数里被执行的最后一个命令将决定退出状态码。如果一个脚本以不带参数的 exit 命令结束，则脚本的退出状态码将会是执行 exit 命令前的最后一个命令的退出状态码。脚本结束时，没有 exit 命令，有不带参数的 exit 命令和 exit $?命令三者是等价的。以下三段代码是等价的。

```
#!/bin/bash
COMMAND_1
…
COMMAND_LAST
# 脚本将会以最后命令 COMMAND_LAST 的状态码退出
exit  n
```

```
#!/bin/bash
COMMAND_1
…
COMMAND_LAST
exit $?
```

```
#!/bin/bash
COMMAND1
…
COMMAND_LAST
```

在一个脚本里，exit n 命令将会返回 shell 一个退出状态码 n（n 必须是 0～255 的十进制整数）。

$?变量保存了最后一个命令执行后的退出状态。当一个函数返回时，$?变量保存了函数中最后一个命令的退出状态码，这就是 bash 里函数返回值的处理办法。当一个脚本运行结束，$?变量保存脚本的退出状态，而脚本的退出状态就是脚本中最后一个已执行命令的退出状态。并且依照惯例，0 表示执行成功，1～255 的整数表示错误。

示例 3.2.3-1 退出和退出状态代码如下：

```
#!/bin/bash
echo hello
echo $?
lskdf
echo $?
```

```
echo
exit 113                    #返回 113 状态码给 shell
```

打开超级终端，建立该脚本文件。运行结果如下：

```
1.  root@ubuntu:/home/linux/chapter3# ./Example3.2.3-1
2.  hello
3.  0
4.  ./Example3.2.3-1: line 4: lskdf: command not found
5.  127
```

在上面的结果中第 2 行显示 hello，因为第 2 行执行成功，所以第 3 行打印 0，第 4 行为无效命令，第 4 行因为上面的无效命令，执行失败，打印一个非 0 的值。

微课视频

3.3　流程控制

对代码块的操作是构造和组织 shell 脚本的关键，循环和分支结构为脚本编程提供了操作代码块的工具，流程控制主要包括条件测试、操作符相关主题、循环控制语句、测试与分支控制语句及与其相关的操作控制符。

3.3.1　条件测试

每个完善的编程语言都应该能测试一个条件，然后依据测试的结果执行进一步的动作。bash 由 test 命令、各种括号和内嵌的操作符，以及 if/then 语句来完成条件测试的功能。

大多数情况下，可以使用测试命令对条件进行测试。例如：可以使用文件测试操作符判断文件是否存在以及文件是否可读等。通常用“[ ]”表示条件测试。注意，这里的空格很重要，要确保方括号两侧的空格。

1．比较操作符

比较操作符包括整数比较操作符、字符串比较操作符和混合比较操作符。其中，常用整数比较操作符如表 3-2 所示，常用字符串比较操作符如表 3-3 所示，常用混合比较操作符如表 3-4 所示。

表 3-2　常用整数比较操作符

整数比较操作符	说　　明
-eq	等于，例如：if ["$a" -eq "$b"]
-ne	不等于，例如：if ["$a" -ne "$b"]
-gt	大于，例如：if ["$a" -gt "$b"]
-ge	大于或等于，例如：if ["$a" -ge "$b"]
-lt	小于，例如：if ["$a" -lt "$b"]
-le	小于或等于，例如：if ["$a" -le "$b"]
<	小于（在双括号里使用），例如：(("$a" < "$b"))
<=	小于或等于（在双括号里使用），例如：(("$a" <= "$b"))
>	大于（在双括号里使用），例如：(("$a" > "$b"))
>=	大于或等于（在双括号里使用），例如：(("$a" >= "$b"))

表 3-3　常用字符串比较操作符

字符串比较操作符	说　　明
=	等于，例如：if ["$a" = "$b"]
==	等于，例如：if ["$a" == "$b"] 它和=是同义词
!=	不相等，例如：if ["$a" != "$b"]操作符在[[...]]结构里使用模式匹配
<	小于，依照 ASCII 字符排列顺序，例如：if [["$a" < "$b"]]，if ["$a" \< "$b"]。注意，"<"字符在[ ]结构里需要转义
>	大于，依照 ASCII 字符排列顺序，例如：if [["$a" > "$b"]]，if ["$a" \> "$b"]。注意，">"字符在[ ]结构里需要转义
-z	字符串为"null"，即指字符串长度为零
-n	字符串不为"null"，即指字符串长度不为零

表 3-4　常用混合比较操作符

混合比较操作符	说　　明
-a	逻辑与，如果 exp1 和 exp2 都为真，则 exp1 -a exp2 返回真
-o	逻辑或，如果 exp1 和 exp2 任何一个为真，则 exp1 -o exp2 返回真

2．文件测试操作符

常用文件测试操作符如表 3-5 所示，如果条件成立，则返回真。

表 3-5　常用文件测试操作符

文件测试操作符	说　　明
-e	文件存在
-f	文件是一个普通文件（不是一个目录或一个设备文件）
-s	文件大小不为零
-d	文件是一个目录
-b	文件是一个块设备（例如软盘、光驱等）
-c	文件是一个字符设备（例如键盘、调制解调器、声卡等）
-p	文件是一个管道
-h	文件是一个符号链接
-l	文件存在且为链接文件
-s	文件是一个 socket
-t	文件（描述符）与一个终端设备相关
-r	文件是否可读（指运行这个测试命令的用户的读权限）
-w	文件是否可写（指运行这个测试命令的用户的写权限）
-x	文件是否可执行（指运行这个测试命令的用户的可执行权限）
-g	文件或目录的 sgid 标志被设置。如果一个目录的 sgid 标志被设置，在该目录下创建的文件都属于拥有此目录的用户组，而不必是创建文件时用户所属的组。这个特性对在一个工作组中共享目录很有用处
-u	文件的 suid 标志被设置

示例 3.3.1-1　分析下列测试命令的含义。

```
[ -f "somefile" ]   #判断是否为一个文件
 [ -x "/bin/ls" ]   #判断/bin/ls 是否存在并有可执行权限
 [ -n "$var" ]      #判断$var 变量是否有值
 [ "$a" = "$b" ]    #判断$a 和$b 是否相等
```

3．嵌套的 if/then 语句

```
if [ condition1 ]
  then
    if [ condition2 ]
    then
     do-something   #仅当 condition1 和 condition2 同时满足才能执行 do-something 语句
    fi
 fi
```

3.3.2　操作符相关主题

常用操作符主要包括赋值操作符、计算操作符、位操作符和逻辑操作符，具体如表 3-6 所示。

表 3-6　常用操作符

操 作 符	符　　号	说　　明
赋值操作符	=	通用的变量赋值操作符，可以用于数值和字符串的赋值
计算操作符	+	加
	–	减
	*	乘
	/	除
	**	求幂
	%	求模
位操作符	<<	位左移（每移一位相当于乘以 2）
	<<=	位左移赋值
	>>	位右移（每移一位相当于除以 2）
	>>=	位右移赋值（和<<=相反）
	&	位与
	&=	位与赋值
	|	位或
	|=	位或赋值
	~	位反
	!	位非
	^	位异或
	^=	位异或赋值
逻辑操作符	&&	逻辑与
	||	逻辑或

示例 3.3.2-1 下面是求最大公约数的实例。代码如下：

```
#!/bin/bash
#最大公约数，使用 Euclid 算法
#参数检测
ARGS=2
E_BADARGS=85
   if [ $# -ne "$ARGS" ]
   then
      echo "Usage: `basename $0` first-number second-number"
      exit $E_BADARGS
   fi
gcd()
{
   dividend=$1      #赋任意值
   divisor=$2       #这里两个参数的赋值大小没有关系，为什么
   remainder=1
  #如果在循环中使用未初始化变量，那么在循环中第一个传递的值会返回一个错误信息
  until [ "$remainder" -eq 0 ]
  do
        let "remainder = $dividend % $divisor"
        dividend=$divisor
        divisor=$remainder
  done
}
   gcd $1 $2
   echo
   echo "GCD of $1 and $2 = $dividend"
   echo
exit 0
```

打开超级终端，建立该脚本文件，任意输入两个数 235 和 200，得到最大公约数为 5，运行结果如下：

```
root@ubuntu:/home /chapter3# ./Example3.3.2-1 235 200
GCD of 235 and 200 = 5
```

3.3.3 循环控制

循环控制是软件编程中非常重要的内容，脚本循环控制主要包括 for、while、until 等语句。对代码块的操作是构造组织 shell 脚本的关键，循环和分支结构为脚本编程提供了操作代码块的工具。

1. for 语句

for 语句是最简单的循环控制语句，它的格式为：

```
for arg in [list]
```

list 中的参数允许包含通配符。如果 do 和 for 想在同一行出现，那么在它们之间需要

添加一个“;”。

下面是一个基本的循环结构，它与 C 语言的 for 结构有很大不同。

```
for arg in [list]
do
   command(s)…
done
```

示例 3.3.3-1　分配行星的名字和它距太阳的距离，代码如下：

```
#!/bin/bash
for planet in "Mercury 36" "Venus 67" "Earth 93"  "Mars 142" "Jupiter 483"
do
    set -- $planet  # Parses variable "planet" and sets positional parameters.
    #"--" 将防止$planet 为空或者是以一个破折号开头
    #可能需要保存原始的位置参数,因为它们被覆盖了
    echo "$1     $2,000,000 miles from the sun"
    #-------two  tabs---把后边的 0 和$2 连接起来
done
exit 0
```

打开超级终端，建立该脚本文件，运行结果如下：

```
[root@bogon chapter2]# ./Example3.3.3-1
Mercury          36,000,000 miles from the sun
Venus            67,000,000 miles from the sun
Earth            93,000,000 miles from the sun
Mars            142,000,000 miles from the sun
Jupiter         483,000,000 miles from the sun
```

2．while 语句

while 语句在循环的开始判断条件是否满足，如果条件一直满足，那就一直循环下去（0 为退出码[exit status]），与 for 循环的区别：while 语句适合用在循环次数未知的情况下。while 语句的循环结构如下：

```
while [condition]
do
    command…
done
```

和 for 循环一样，如果想把 do 和条件放到同一行，还需要添加一个“;”，代码如下：

```
while [condition] ; do
```

示例 3.3.3-2　简单的 while 循环，代码如下：

```
#!/bin/bash
var0=0
LIMIT=10
while [ "$var0" -lt "$LIMIT" ]
do
```

```
    echo -n "$var0 "          #-n 将会阻止产生新行
    var0=`expr $var0 + 1`     #var0=$(($var0+1)) 也可以
                              #var0=$((var0 + 1))也可以
                              #let "var0 += 1"也可以
done                          #使用其他方法也行
echo
exit 0
```

打开超级终端，建立该脚本文件，命名为 while，运行结果如下：

```
root@ubuntu:/home/linux/chapter3# ./Example3.3.3-2
0 1 2 3 4 5 6 7 8 9
```

3．until 语句

until 语句的结构在循环的顶部判断条件，如果条件一直为 false，那就一直循环下去（与 while 循环相反）。until 循环的结构如下：

```
until [condition-is-true]
do
     command…
done
```

until 循环的判断在循环的顶部，这与某些编程语言是不同的。与 for 循环一样，如果想把 do 和条件放在同一行，需要使用“;”。

```
until [condition-is-true] ;  do
```

示例 3.3.3-3 until 循环，代码如下：

```
#!/bin/bash
     END_CONDITION=end
     until [ "$var1" = "$END_CONDITION" ]   #在循环的顶部判断条件
do
     echo "Input variable #1 "
     echo " ($END_CONDITION to exit) "
     read var1
     echo "variable #1 = $var1"
     echo
done
exit 0
```

打开超级终端，建立该脚本文件，运行文件，直到出现结束标志 end，程序运行结束，运行结果如下：

```
root@ubuntu:/home/linux/chapter3# ./Example3.3.3-3
Input variable #1
(end to exit)
3
variable #1 = 3

Input variable #1
```

```
(end to exit)
end
variable #1 = end
```

4．break 和 continue

break 和 continue 这两个循环控制命令与其他语言的类似命令的行为是相同的，break 命令将会跳出循环，continue 命令将会跳过本次循环后面的语句，直接进入下次循环。

break 命令可以带一个参数，不带参数的 break 循环只能退出最内层的循环，而 break N 可以退出 N 层循环。continue 命令也可以带一个参数，不带参数的 continue 命令只跳过本次循环的剩余代码。而 continue N 将会把 N 层循环剩余的代码都跳过，但是循环的次数不变。

3.3.4　测试与分支

case 语句和 select 语句结构从技术上说不是循环，因为它们并不对可执行的代码块进行迭代。但是和循环有相似之处：它们也依靠在代码块的顶部或底部的条件判断决定程序的分支。shell 中的 case 与 C/C++中的 switch 结构是相同的，它允许通过条件判断选择执行代码块中多条路径中的一条。它的作用与多个 if/then/else 语句相同，是它们的简化结构，特别适用于创建目录。

1. case 语句

case 语句结构如下：

```
case "$variable" in
?"$condition1" )
?command…
?;;
?"$condition2" )
?command…
?;;
esac
```

对变量使用""并不是强制的，因为不会发生单词分离。每句测试行，都以右小括号“)”结尾。每个条件块都以两个分号结尾。case 块的结束以 esac（case 的反向拼写）结尾。

示例 3.3.4-1　用 case 语句查看计算机的架构，代码如下：

```
#!/bin/bash
#case-cmd.sh: 使用命令替换产生"case"变量
case $( arch ) in       #arch"返回计算机的类型,等价于'uname -m'…
    i386 ) echo "80386-based machine";;
    i486 ) echo "80486-based machine";;
    i586 ) echo "Pentium-based machine";;
    i686 ) echo "Pentium2+-based machine";;
    x86 _64) echo "intel10'cpu";;
*   ) echo "Other type of machine";;
esac
exit 0
```

打开超级终端，建立该脚本文件，运行结果如下：

```
root@ubuntu:/home/linux/chapter3# ./Example3.3.4-1
intel10'cpu
```

从结果可以看到当前运行的计算机的架构。

2．select 语句

select 语句结构是建立菜单的另一种工具，从 ksh 中引入的结构如下：

```
select variable [in list]
do
   ?command…
   ?break
done
```

示例 3.3.4-2 用 select 语句创建菜单，代码如下：

```
#!/bin/bash
PS3='Choose your favorite vegetable: '  #设置提示符字串
echo
select vegetable in "beans" "carrots" "potatoes" "onions" "rutabagas"
do
   echo
   echo "Your favorite veggie is $vegetable."
   echo "Yuck!"
   echo
   break                                  #如果这里没有'break'会发生什么
done
exit 0
```

打开超级终端，建立该脚本文件，运行结果如下：

```
root@ubuntu:/home/linux/chapter3# ./Example3.3.4-2

1) beans
2) carrots
3) potatoes
4) onions
5) rutabagas
Choose your favorite vegetable: 3

Your favorite veggie is potatoes.
Yuck!
```

如果忽略了 in list 列表，那么 select 语句将使用传递到脚本的命令行参数（$@）或者函数参数（当 select 语句在函数中时）。

示例 3.3.4-3 编写脚本，在脚本中对输入的两个参数进行大小比较。代码如下：

```
#!/bin/bash
a=$1
```

```
b=$2
#判断 a 或者 b 变量是否为空,只要有一个为空就打印提示语句并退出
if [ -z $a ] || [ -z $b ]
then
    echo "please enter 2 no"
    exit 1
#判断 a 和 b 的大小,并根据判断结果打印语句
fi
if [ $a -eq $b ]
then
    echo "number a = number b"
else if [ $a -gt $b ]
    then
        echo "number a>number b"
    elif [ $a -lt $b ]
        then
            echo "number a<number b"
    fi
fi
```

打开超级终端，建立该脚本文件，运行结果如下：

```
root@ubuntu:/home/linux/chapter3# ./Example3.3.4-3 7 8
number a<number b
```

示例 3.3.4-4　当前目录下的文件数目统计，代码如下：

```
#!/bin/bash
#变量 counter 用于统计文件的数目
counter=0
#变量 files 遍历当前文件夹
for files in *
do
      #判断 files 是否为文件,如果是,将 counter 变量的值加 1,再赋给自己
      if [ -f "$files" ]
      then
      counter=`expr $counter + 1`
      fi
done
echo "There are $counter file in `pwd` "
```

打开超级终端，建立该脚本文件，命名为 count，运行结果如下：

```
root@ubuntu:/home/linux/chapter3# ./Example3.3.4-4
There are 10 file in /home/linux/chapter3
root@ubuntu:/home/linux/chapter3# ls
Example3.2.2-1  Example3.2.2-3  Example3.3.2-1  Example3.3.3-2  Example3.3.4-1
Example3.2.2-2  Example3.2.3-1  Example3.3.3-1  Example3.3.3-3  Example3.3.4-4
```

3.4 习题

1. 什么是 shell？
2. 什么是局部变量？
3. 编程实现列出系统上的所有用户。
4. 编程实现在一个指定的目录的所有文件中查找字符串。
5. 编程实现检测是否有字母输入。