第 7 章　Shell 编 程

通常情况下，从命令行每输入一次命令就能够得到系统响应，如果需要一个接着一个地输入命令才得到结果的时候，这样的做法效率很低。使用 Shell 程序或者 Shell 脚本可以很好地解决这个问题。

7.1　Shell 基础

在 Linux 系统中，Shell 是最常使用的程序，其主要作用是侦听用户指令、启动命令所指定的进程并将结果返回给用户，本节主要讲述 Shell 的基础知识。

7.1.1　Shell 简介

原来的操作系统都带有命令解释器。命令解释器接收用户的命令，然后解释它们，因而计算机可以使用这些命令。人们想要的不只是这些功能，他们想要比命令解释器具备更优异功能的工具，这驱动的 Shell 的诞生。Shell 接收用户命令，然后调用相应的应用程序，同时它还是一种程序设计语言，是系统管理维护时的重要工具。作为命令语言，它交互式地解释和执行用户输入的命令或者自动地解释和执行预先设定好的一连串的命令。作为程序设计语言，它可以定义各种变量和参数，并提供了许多在高级语言中才具有的选择和循环控制结构。

Shell 命令重新初始化用户的登录会话。当给出该命令时，就会重新设置进程的控制终端的端口特征，并取消对端口的所有访问。然后 Shell 命令为用户把进程凭证和环境重新设置为缺省值，并执行用户的初始程序。根据调用进程的登录用户标识建立所有的凭证和环境。

由于 Linux 系统对 Shell 的处理采用独立、自由、开放的方式，因此 Shell 的种类相当多，目前流行的 Shell 有 sh、csh、ksh、tcsh 和 bash 等。大部分 Linux 系统的默认 Shell 类型为 bash.

7.1.2　bash 简介

bash（Bourne-Again Shell）最早是在 1987 年由布莱恩·福克斯开发的一个为 GNU 计

划编写的 Unix Shell。bash 目前是大多数 Linux 系统默认的 Shell，它还能运行于大多数 Unix 风格的操作系统上，甚至被移植到了 Windows 上的 Cygwin 系统中，以实现 Windows 的 POSIX 虚拟接口。

bash 的命令语法是 Bourne shell 命令语法的超集。数量庞大的 Bourne shell 脚本大多不经过修就可以在 bash 中执行，只有那些引用了 Bourne 特殊变量或使用了 Bourne 内置命令的脚本才需修改。bash 的命令语法很多来自 ksh 和 csh，比如命令行编辑、命令历史、目录栈、$RANDOM 变量、$PPID 变量以及 POSIX 命令置换语法。

对于刚接触 Linux 系统的人而言，bash 就相当于 Windows 系统上的 DOS 命令提示符，可以交互操作，也可以进行批处理操作。然而不同的是，bash 的开发具有较强针对性，因而其功能及易用性远比 DOS 命令提示符大得多。

7.1.3 bash 命令

当登录系统或打开一个终端窗口时，首先看到的是 bash Shell 提示符。Linux 系统的标准提示符包括了用户登录名、登录的主机名、当前所在的工作目录路径和提示符号。

以普通用户 feng 登录名为 localhost 的主机，它的工作目录是/home/feng。

【例 7.1】以普通用户名 feng 登录系统。

```
[feng@localhost ~] $
```

【例 7.2】以 root 用户登录系统。

```
[root@localhost ~]#
```

除了不同的用户名外，提示符号由“$”变成了“#”。根据 bash 的传统，普通用户的提示符以“$”结尾，而超级用户以“#”结尾，提示符的每个部分都可以定制。

要运行命令的话，只需要在提示符后敲进命令，然后再按回车键。Shell 将在其路径中搜索这个命令，找到以后就运行，并在终端里输出相应的结果，命令结束后，再给出新的提示符。

【例 7.3】执行命令 whoami 后，又给出新的提示符。

```
[feng@rhe1 ~] $ whoami
feng
//显示当前登录 Linux 系统的用户是 feng
[feng@localhost ~] $
```

一个 Shell 命令可能含有一些选项和参数,其一般格式为：

[Shell 命令］[选项］[参数］

下面举一个例子来详细描述 Shell 命令格式。

```
[root@localhost ~] #ls -l /root
```

其中“-l”是命令 ls 的一个选项，而/root 则是参数。所有选项在该命令的 man 手册页中都有详细的介绍，而参数则由用户提供。选项决定命令如何工作，而参数则用于确定命令作用的目标。选项有短命令行选项和长命令选项两种。

下面这个示例就使用了短命令行选项。

```
[root@localhost ~] # 1s -1 /root
```

下面两种方法实现了一样的效果。

```
[root@localhost ~] # 1s -1 -a /root
[root@localhost ~] # 1s -la /root
```

下面这个示例就使用了长命令行选项。

```
[root@localhost ~] #1s --size /root
```

在 Linux 系统中，命令可以分为以下两大类：bash 内置的命令和应用程序。

如果是 bash 内置的命令，则由 bash 负责回应；如果是应用程序，那么 Shell 会找出该应用程序，然后将控制权交给内核，由内核执行该应用程序，执行完之后，再将控制权交回给 Shell。使用 which 命令可以查看哪些命令是 bash 内置的命令，哪些是应用程序。

【例 7.4】查看 echo 和 ls 命令。

```
[root@localhost~] # which echo
/usr/bin/echo
[root@localhost ~] # which 1s
alias 1s='1s --color=auto'
/usr/bin/1s
```

7.2　使用 bash

7.2.1　常用控制组合键

在操作 Linux 系统时，经常会使用一些组合键来控制 Shell 的活动。表 7-1 列出了一些常用的控制组合键。

表 7-1　常用控制组合键

控制组合键	功能
Ctrl+1	清屏
Ctrl+o	执行当前命令，并选择上一条命令
Ctrl+s	阻止屏幕输出
Ctrl+q	允许屏幕输出
Ctrl+c	终止命令
Ctrl+z	挂起命令
Ctrl+m	相当于按回车键
Ctrl+d	输入结束，即 EOF 的意思，或者注销 Linux 系统

7.2.2　光标操作

在 Linux 系统中，使用表 7-2 所示组合键通过可以快速地进行光标操作。

表 7-2　光标操作

组合键	功能
Ctrl+a	移动光标到命令行首
Ctrl+e	移动光标到命令行尾
Ctrl+f	按字符前移（向右）
Ctrl+b	按字符后移（向左）
Ctrl+xx	在命令行首和光标之间移动
Ctrl+u	删除从光标到命令行首的部分
Ctrl+k	删除从光标到命令行尾的部分
Ctrl+w	删除从光标到当前单词开头的部分
Ctrl+d	删除光标处的字符
Ctrl+h	删除光标前的一个字符
Ctrl+y	插入最近删除的单词
Ctrl+t	交换光标处字符和光标前面的字符
Alt+f	按单词前移（向右）
Alt+b	按单词后移（向左）
Alt+d	从光标处删除至单词尾
Alt+c	从光标处更改单词为首字母大写
Alt+u	从光标处更改单词为全部大写
Alt+1	从光标处更改单词为全部小写
Alt+t	交换光标处单词和光标前面的单词
Alt+Backspace	与 Ctrl+w 功能类似，分隔符有些差别

7.2.3　特殊字符

在 Linux 系统中，许多字符对于 Shell 来说是具有特殊意义的。表 7-3 列出了一些常用的特殊字符的意义。

表 7-3　特殊字符

符号	功能
~	用户主目录
`	倒引号，用来命令替代（在 Tab 键上面的那个键）
#	注释
$	变量取值
&	后台进程工作

续表

符号	功能
(	子 Shell 开始
)	子 Shell 结束
\	使命令持续到下一行
|	管道
<	输入重定向
>	输出重定向
≫	追加重定向
‘	单引号（不具有变数置换的功能）
“	双引号（具有置换的功能）
/	路径分隔符
;	命令分隔符

7.2.4　通配符

如果命令的参数中含有文件名，那么通配符可以带来十分便利的操作。表 7-4 列出了 bash 中经常使用的通配符。

表 7-4　通配符

符号	功能
?	代表任何单一字符
*	代表任何字符
[字符组合]	在中括号中的字符都符合，比如 [a-z] 代表所有的小写字母
[! 字符组合]	不在中括号中的字符都符合，比如 [! 0-9] 代表非数字的都符合

7.3　Shell 程序的创建

Shell 除了解释和执行用户输入的命令，也可以用来设计程序。它提供了定义变量和参数的手段以及丰富的过程控制结构。使用 Shell 编程类似于用 DOS 中的批处理文件，称为 Shell 程序（又叫 Shell 脚本或 Shell 命令文件）。

7.3.1　基本语法介绍

Shell 程序基本语法较为简单，主要由开头、注释的执行命令组成。

1. 开头

Shell 程序必须以下面的行开始（必须放在文件的第一行）。

#! /bin/bash

符号“#!”用来告诉系统它后面的参数是用来执行该文件的程序，这里使用/bin/bash 执行程序。当编辑好脚本时，如果要执行该脚本，还必须设置权限使其可执行。

要使脚本可执行，需赋予该文件可执行的权限，使用 chmod 命令才能使文件运行。

chmod u+x ［Shell 程序］

2. 注释

在进行 Shell 编程时，以“#”开头的语句直到这一行的结束表示注释，建议在程序中使用注释。如果使用注释，那么即使相当长的时间内没有使用该脚本，也能在很短的时间内明白该脚本作用及工作原理。

3. 执行命令

在 Shell 程序中可以输入多行命令以得到命令的结果信息，这样就提高系统管理的工作效率。

7.3.2 Shell 程序的创建过程

Shell 程序就是放在一个文件中的一系列 Linux 命令和实用程序，在执行的时候，通过 Linux 系统一个接着一个地解释和执行每个命令，这和 Windows 系统下的批处理程序非常相似。

下面通过一个简单的实例来了解 Shell 程序是如何创建和执行的。

1. 创建文件

使用 vi 编辑器创建/root/date 文件，该文件内容如下所示，共有 3 行命令。

```
#! /bin/bash
# filename:date
echo "Mr. $ USER, Today is:"
echo 'date'
echo Wish you a lucky day!
```

2. 设置可执行权限

创建完/root/date 文件之后它还不能执行，需要给它设置可执行权限，使用如下命令给文件设置权限。

```
[root@localhost ~]# chmod u+x /root/date
[root@localhost ~]# ls -l /root/date
-rwxr--r--. 1 root root 94  1 月  2 20:33 /root/date
//可以看到当前/root/date 文件具有可执行权限
```

3. 执行 Shell 程序

输入整个文件的完整路径执行 Shell 程序，使用以下命令执行。

[root@localhost ~]# /root/date

Mr. feng, Today is:

2022 年 01 月 02 日　星期日　20:39:20 PST

Wish you a lucky day!

//可以看到 Shell 程序的输出信息

4. 使用 bash 命令执行程序

在执行 Shell 程序时需要将/root/date 文件设置为可执行权限。如果不设置文件的可执行权限，那么需要使用 bash 命令告诉系统它是一个可执行的脚本。

使用以下命令执行 Shell 程序。

[root@localhost ~]# bash /root/date

Mr. feng, Today is:

2022 年 01 月 02 日 星期日 20:40:12 PST

Wish you a lucky day!

7.3.3　Shell 里的特殊字符

和其他编程语言一样，Shell 里也有特殊字符，常见的有美元符号（$）、反斜线（\）和引号。美元符号“$”表示变量替换，即用其后指定的变量的值来代替变量。反斜线“\”为转义字符，用于告诉 Shell 不要对其后面的那个字符进行特殊处理，只是当做普通字符。而 Shell 下的引号情况比较复杂，分为 3 种：双引号（”）、单引号（’）和倒引号（‘）。它们的作用不尽相同，下面逐一说明。

1. 双引号（”）

由双引号括起来的字符，除美元符号（$）、反斜线（\）和倒引号（‘）仍保留其特殊功能外，其余字符均作为普通字符对待。

2. 单引号（’）

由单引号括起来的字符都作为普通字符出现。

3. 倒引号（‘）

倒引号一般是键盘上 Tab 键上方的按键符号。由倒引号括起来的字符串被 Shell 解释为命令行，在执行时，Shell 会先执行该命令行，并以它的标准输出结果取代整个倒引号部分。

【例 7.5】通过以下示例理解 3 种引号的用法。

[root @ localhost 桌面] # echo "My current directory is ' pwd ' and logname is $ LOGNAME"

My current directory is /home/feng/桌面 and logname is feng

[root@ localhost 桌面] # echo " My current directory is ' pwd ' and logname is \ $ LOGNAME"

My current directory is /home/feng/桌面 and logname is $ LOGNAME

```
[root@localhost 桌面]# echo 'My current directory is 'pwd'and logname is  $ LOGNAME'
My current directory is 'pwd'and logname is  $ LOGNAME
```

7.4 Shell 变量

像高级程序设计语言一样，Shell 也提供说明和使用变量的功能。对 Shell 来讲，所有变量的取值都是一个字符，Shell 程序采用“$var”的形式来引用名为 var 的变量的值。

7.4.1 Shell 定义的环境变量

Shell 在开始执行时就已经定义了一些与系统的工作环境有关的变量，用户还可以重新定义这些变量。

常用的 Shell 环境变量如表 7-5 所示。

表 7-5 常用的 Shell 环境变量

Shell 环境变量	描述
HOME	用于保存用户主目录的完全路径名
PATH	用于保存用冒号分隔的目录路径名，Shell 将按 PATH 变量中给出的顺序搜索这些目录，找到的第一个与命令名称一致的可执行文件将被执行
TERM	终端的类型
UID	当前用户的 UID，由数字构成
PWD	当前工作目录的绝对路径名，该变量的取值随 cd 命令的使用而变化
PS1	主提示符，在 root 用户下，默认的主提示符是“#”，在普通用户下，默认的主提示符是“$”
PS2	在 Shell 接收用户输入命令的过程中，如果用户在输入行的末尾输入“\”然后按回车键，或者当用户按回车键时 Shell 判断出用户输入的命令没有结束时，就显示这个辅助提示符，提示用户继续输入命令的其余部分，默认的辅助提示符是“>”

【例 7.6】查看当前用户 Shell 定义的环境变量的值。

```
[root@localhost ~]# echo  $ HOME
/root
[root@localhost ~]# echo  $ PWD
/root
[root@localhost ~]# echo  $ PS1
[\u@ \h  \W]\$
[root@localhost ~]# echo  $ PS2
>
[root@localhost ~]# echo  $ PATH
```

/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/feng/bin:/home/feng/bin

[root@localhost ~]# echo $UID

7.4.2　用户定义的变量

用户可以按照下面的语法规则定义自己的变量。

变量名=变量值

在定义变量时，变量名前不应该加符号“$”，在引用变量的内容时则应在变量名前加符号“$”。在给变量赋值时，等号两边一定不能留空格，若变量中本身就包含了空格，则整个字符串都要用双引号括起来。

[root@localhost ~]# var1=100

[root@localhost ~]# echo $var1

100

//在引用变量名时,在变量名前加符号“ $ ”

[root@localhost ~]# var2="red hat linux"

[root@localhost ~]# echo $var2

red hat linux

//变量值中包含了空格,需将整个字符串用双引号括起来

在说明一个变量并将它设置为一个特定值后就不再改变它的值时，可以用下面的命令来保证一个变量的只读性。

readonly 变量名

【例 7.7】设置变量 var1 为只读。

[root@localhost ~]# readonly var1

[root@localhost ~]# echo $var1

100

[root@localhost ~]# var1=200

bash: var1: readonly variable

//无法更改只读变量的值

在任何时候创建的变量都只是当前 Shell 的局部变量，所以不能被 Shell 运行的其他命令或 Shell 程序所利用，而 export 命令可以将一个局部变量提供给 Shell 命令使用，其格式如下所示。

export 变量名

【例 7.8】将局部变量 var3 提供给 Shell 程序使用。

[root@localhost ~]# var3=300

[root@localhost ~]# export var3

[root@localhost ~]# echo $var3

300

```
[root@localhost ~]# env |grep var3
var3=300
```

也可以在给变量赋值的同时使用 export 命令。

```
export 变量名=变量值
```

【例 7.9】将局部变量 var4 提供给 Shell 程序使用。

```
[[root@localhost ~]# export var4=400
[root@localhost ~]# echo $var4
400
[root@localhost ~]# env |grep var4
var4=400
```

使用 export 设置的变量在 Shell 以后运行的所有命令或程序中都可以访问到。

7.4.3 位置参数

位置参数是一种在调用 Shell 程序的命令行中按照各自的位置决定的变量，是在程序名之后输入的参数。位置参数之间用空格分隔，Shell 取第一个位置参数替换程序文件中的 $1，第二个替换 $2，依次类推。$0 是一个特殊的变量，它的内容是当前这个 Shell 程序的文件名，所以 $0 不是一个位置参数，在显示当前所有的位置参数时是不包括 $0 的。

7.4.4 预定义变量

预定义变量和环境变量相类似，也是在 Shell 一开始时就定义了的变量。所不同的是，用户只能根据 Shell 的定义来使用这些变量，所有预定义变量都是由符号“$”和另一个符号组成的。

常用的 Shell 预定义变量如表 7-6 所示。

表 7-6 Shell 预定义变量

预定义变量	描述
$#	位置参数的数量
$*	所有位置参数的内容
$?	命令执行后返回的状态，0 表示没有错误，非 0 表示有错误
$$	当前进程的进程号
$!	后台运行的最后一个进程号
$0	当前执行的进程名

7.4.5 参数置换的变量

Shell 提供了参数置换功能以便用户可以根据不同的条件来给变量赋不同的值。参数置换的变量有四种，这些变量通常与某一个位置参数相联系，根据指定的位置参数是否已

经设置决定变量的取值，它们的语法和功能分别如下。

1. 变量=$｛参数-word｝

如果设置了参数，则用参数的值置换变量的值，否则用 word 置换，即这种变量的值等于某一个参数的值。如果该参数没有设置，则变量就等于 word 的值。

2. 变量=$｛参数=word｝

如果设置了参数，则用参数的值置换变量的值，否则把变量设置成 word，然后再用 word 替换参数的值。位置参数不能用于这种方式，因为在 Shell 程序中不能为位置参数赋值。

3. 变量=$｛参数? word｝

如果设置了参数，则用参数的值置换变量的值，否则就显示 word 并从 Shell 中退出，如果省略了 word，则显示标准信息。这种变量要求一定等于某一个参数的值。如果该参数没有设置，就显示一个信息，然后退出，这种方式常用于出错指示。

4. 变量=$｛参数+word｝

如果设置了参数，则用 word 置换变量，否则不进行置换。

注意：所有这四种形式中的参数既可以是位置参数，也可以是另一个变量，只是用位置参数的情况比较多。

7.5　变量表达式

test 是 Shell 程序中的一个表达式，通过和 Shell 提供的 if 等条件语句相结合可以方便地测试字符串、文件状态和数字。其语法如下所示。

test［表达式］

表达式所代表的操作符有字符串操作符、数字操作符、逻辑操作符和文件操作符。其中文件作符是一种 Shell 特有的操作符，因为 Shell 里的变量都是字符串，为了对文件进行操作，Shell 提供了这样的一种操作符。

7.5.1　字符串比较

字符串比较用来测试字符串是否相同、长度是否为 0、字符串是否为 null。常用的字符串比较符号如表 7-7 所示。

表 7-7　字符串比较符号

字符串比较符号	描述
=	比较两个字符串是否相同，相同则为“是”
! =	比较两个字符串是否相同，不同则为“是”
-n	比较字符串的长度是否大于 0，如果大于 0 则为“是”
-z	比较字符串的长度是否等于 0，如果等于 0 则为“是”

【例 7.10】字符串比较的使用。

```
[root@localhost ~]# str1=linux
[root@localhost ~]# test $str1=linux
[root@localhost ~]# echo $?
0
//结果显示 0 表示字符串 str1 等于 linux
```

在 test 处理含有空格的变量时最好用引号将变量括起来，否则会出现错误的结果。因为 Shell 在处理命令行时将会去掉多余的空格，而用引号括起来则可以防止 Shell 去掉这些空格。

7.5.2 数字比较

数字比较用来测试数字的大小。

常用的数字比较符号如表 7-8 所示。

表 7-8 数字比较符号

数字比较符号	描述
-eq	相等
-ge	大于等于
-le	小于等于
-ne	不等于
-gt	大于
-lt	小于

【例 7.11】数字相等比较。

```
[root@localhost ~]# int1=325
[root@localhost ~]# int2=335
[root@localhost ~]# test $int1 -eq $int2
[root@localhost ~]# echo $?
1
//结果显示 1 表示字符 int1 和 int2 比较,二者值不相等。
```

【例 7.12】数字大于比较。

```
[root@localhost ~]# int2=15
[root@localhost ~]# test $int2 -gt 15
[root@localhost ~]# echo $?
1
[root@localhost ~]# test $int2 -lt 16
[root@localhost ~]# echo $?
```

0
//结果显示 0 表示字符 int1 的值确实大于 2。

7.5.3　逻辑测试

逻辑测试用来测试文件是否存在。
常用的逻辑测试符号如表 7-9 所示。

表 7-9　逻辑测试符号

逻辑测试符号	描述
!	与一个逻辑值相反的逻辑值
-a	两个逻辑值为“是”返回值才为“是”，反之为“否”
-0	两个逻辑值有一个为“是”，返回值就为“是”

【例 7.13】逻辑测试。

```
[root@localhost ~]# test - r empty - a - s empty
[root@localhost ~]# echo $?
1
```

//结果显示 1 表示文件 empty 存在且只读以及长度为 0

7.5.4　文件操作测试

文件操作测试表达式通常是为了测试文件的文件操作逻辑。常用的文件操作测试符号如表 7-10 所示。

表 7-10　文件操作测试符号

文件操作测试符号	描述
-d	对象存在且为目录则返回值为“是”
-f	对象存在且为文件则返回值为“是”
-L	对象存在且为符号链接则返回值为“是”
-r	对象存在且可读则返回值为“是”
-s	对象存在且长度非 0 则返回值为“是”
-w	对象存在且可写则返回值为“是”
-X	对象存在且可执行则返回值为“是”
!	测试条件的否定

【例 7.14】文件操作测试

```
[root@localhost ~]# cat /dev/null>empty
[root@localhost ~]# cat empty
[root@localhost ~]# test -r empty
[root@localhost ~]# echo $?
0
```

```
//结果显示 0 表示文件 empty 存在且只读
[root@localhost ~]# test ! -s empty
[root@localhost ~]# echo $?
0
//结果显示 0 表示文件 empty 存在且文件长度为 0
```

7.6 Shell 条件判断语句

Shell 提供了用来控制程序和执行流程的命令，包括条件分支和循环结构，用户可以用这些的命令创建非常复杂的程序。在 Shell 程序中使用条件判断语句可以使用 if 条件语句和 case 条件语句，两者的区别在于使用 case 语句的选项比较多而已。

7.6.1 if 条件语句

Shell 程序中的条件分支是通过 if 条件语句来实现的，其语法格式有 if-then-fi 语句和 if-then-else-fi 语句两种。

1. if-then-fi 语句

if-then-fi 语句的语法格式如下所示。

```
if    命令行 1
then
      命令行 2
fi
```

【例 7.15】使用 if-then-fi 语句创建简单的 Shell 程序。

使用 vi 编辑器创建 Shell 程序，文件名为/root/continue，文件内容如下所示。

```
#! /bin/bash
#filename:continue
echo -n "Do you want to continue: Y or N"
read ANSWER
if [ $ANSWER=N -0 $ANSWER=n ]
then
exit
fi
```

运行 Shell 程序/root/continue，输出内容如下所示。

```
[root@localhost ~] # bash /root/continue
Do you want to continue: Y or N
```

2. if-then-else-fi 语句

if-then-else-fi 语句的语法格式如下所示。

```
if
命令行 1
then
命令行 2
else
命令行 3
fi
```

【例 7.16】使用 if-then-else-fi 语句创建一个根据输入的分数判断分数是否及格的 Shell 程序。使用 vi 编辑器创建 Shell 程序，文件名为/root/score，文件内容如下所示。

```
#! /bin/bash
#filename:score
echo -n "please input a score: "
read SCORE
Echo" You input Score is  $ SCORE"
if[ $ SCORE -ge 60 ];
then
echo- n "Congratulation! You Pass the examination. "
else
echo -n "Sorry ! You Fail the examination! "
fi
echo -n "press any key to continue! "
read  $ GOOUT
```

运行 Shell 程序/root/score,输出内容如下所示。

```
[root@rhe1l~] # bash /root/score
please input a score: 80                    //输入数值 80
You input Score is 80
Congratulation! You Pass the examination. press any key to continue!
[root@localhost~] #bash /root/score
please input a score: 30                    //输入数值 30
You input Score is 30
Sorry ! You Fail the examination! press any key to continue!
```

7.6.2　case 条件语句

if 条件语句用于在两个选项中选定一项，而 case 条件选择为用户提供了根据字符串或变量的值从多个选项中选择一项的方法。

case 条件语句的语法格式如下所示。

```
case string in
```

```
exp-1)
若干个命令行 1
; ;
exp-2)
若干个命令行 2
; ;
……
* )
其他命令行
esac
```

Shell 通过计算字符串 string 的值，将其结果依次与运算式 exp-1 和 exp-2 等进行比较，直到找到一个匹配的运算式为止。如果找到了匹配项，则执行它下面的命令直到遇到一对分号（;;）为止。

在 case 运算式中也可以使用 Shell 通配符（*、?、[ ]）。通常用“*”作为 case 命令的最后运算式，以便在前面找不到任何相应的匹配项时执行“其他命令行”的命令。

【例 7.17】使用 case 语句创建一个菜单选择的 Shell 脚本。

使用 vi 编辑器创建 Shell 程序，文件名为/root/selection，文件内容如下所示。

```
#! /bin/bash
#filename:selection
#Display a menu
echo-
echo "1 Restore"
echo "2 Backup"
echo "3 Unload"
echo
#Read and excute the user's selection
echo -n "Enter Choice: "
read CHOICE
case "$ CHOICE" in
1)echo "Restore"; ;
2)echo "Backup"; ;
3)echo "Unload"; ;
*) echo "Sorry  $ CHOICE is not a valid choice
exit 1
Esac
```

运行 Shell 程序/root/selection，输出内容如下所示。

```
[root@localhost ~] # bash /root/selection
```

```
-
1 Restore
2 Backup
3 Unload
Enter Choice:
```

7.7　Shell 循环控制语句

在 Shell 程序中循环控制语句可以使用 for 循环语句、while 循环语句以及 until 循环语句，下面分别进行介绍。

7.7.1　for 循环语句

for 循环语句对一个变量的可能的值都执行一个命令序列。赋给变量的几个数值既可以在程序中以数值列表的形式提供，也可以在程序以外以位置参数的形式提供。

for 循环语句的语法格式如下所示。

```
for　　变量名　［in 数值列表］
  do
      若干个命令行
  done
```

变量名可以是用户选择的任何字符串，如果变量名是 var，则在 in 之后给出的数值将按顺序替换循环命令列表中的“$var”。如果省略了 in，则变量 var 的取值将是位置参数。对变量的每一个可能的赋值都将执行 do 和 done 之间的命令列表。

【例 7.18】使用 for 语句创建简单的 Shell 程序。

使用 vi 编辑器创建 Shell 程序，文件名为/root/for，文件内容如下所示。

```
#! /bin/bash
# filename:for
for ab in 1 2 3 4
do
echo  $ ab
done
```

运行 Shell 程序/root/for,输出内容如下所示。

```
[root@localhost ~] # bash/root/for
  1
  2
  3
  4
```

【例 7.19】使用 for 语句创建求命令行上所有整数之和的 Shell 程序。

使用 vi 编辑器创建 Shell 程序，文件名为/root/sum，文件内容如下所示。

```
#! /bin/bash
# filename:sum
sum=0
for INT in $*
do
sum='expr $sum + $INT
done
echo $sum
```

运行 Shell 程序/root/sum,输出内容如下所示。

```
[root@localhost ~] bash/root/sum 1 2 3 4 5
15
```

7.7.2 while 循环语句

While 语句是用命令的返回状态值来控制循环的。

while 循环语句的语法格式如下所示。

```
while
    若干个命令行 1
do
    若干个命令行 2
done
```

只要 while 的“若干个命令行 1”中最后一个命令的返回状态为真，while 循环就继续执行“若干个命令行 2”。

【例 7.20】使用 while 语句创建一个计算 1~5 的平方的 Shell 程序。

使用 vi 编辑器创建 Shell 程序，文件名为/root/zx，文件内容如下所示。

```
#! /bin/bash
# f1lename:zx
int=1
while [ $Sint -le 5 ]
do
sq='expr $int\* $int'
echo $sq
int='expr $int +1'
done
echo "Job completed"
```

运行 Shell 程序/root/zx,输出内容如下所示。

```
[root@localhost ~] # bash/root/zx
```

```
1
4
9
16
25
Job completed
```

【例 7.21】使用 while 语句创建一个根据输入的数值求累加和（1+2+3+4+・・・・・+n）的 Shell 程序。

使用 vi 编辑器创建 Shell 程序，文件名为/root/number，文件内容如下所示。

```
#! /bin/bash
# filename:number
echo - n "Plese Input Number: "
read NUM
number = 0
sum = 0
while [ $ number - le  $ NUM ]
do
echo number
echo "$ number"
number = 'expr  $ number +1'
echo sum
echo "$ sum"
sum = 'expr  $ sum  + $ number'
done
echo
```

运行 Shell 程序/root/number,输出内容如下所示。

```
[root@ rhe1 ~] #bash /root/number
Please Input Number: 4    //在这里输入了数字 4
number
0
Sum
0
Number
1
Sum
1
number
```

```
2
Sum
3
number
3
Sum
6
number
4
Sum
10
```

7.7.3 until 循环语句

until 循环语句是另外一种循环结构，它和 while 语句相类似。

until 循环语句的语句格式如下所示。

```
until
        若干个命令行 1
do
        若干个命令行 2
done
```

until 循环语句和 while 循环语句的区别在于：while 循环语句在条件为真时继续执行循环，而 until 循环语句则是在条件为假时继续执行循环。

Shell 还提供了 true 和 false 两条命令用于创建无限循环结构，它们的返回状态分别是总为 0 或总为非 0。

【例 7.22】使用 until 语句创建一个输入 exit 退出的 Shell 程序。

使用 vi 编辑器创建 Shell 程序，文件名为/root/hk，文件内容如下所示。

```
#! /bin/bash
#filename:hk
echo "This example is for test until. . . . do"
echo "If you input [exit] then quit the system"
echo -n "please input: "
read EXIT
until [ $ EXIT = "exit"]
do
read EXIT
done
echo "OK! "
```

运行 Shell 程序/root/hk,输出内容如下所示。

```
[root@localhost ~] # bash/root/hk
This example is for test until. . . . do
If you input [exit] then quit the system
please input:exit                                    //输入 exit 退出
OK!
```

7.7.4　break、continue 和 exit 语句

break 语句的作用是在正常结束之前退出当前循环。例如，下面求和的这个例子中，为了避免 while 的循环条件永远为真而导致程序永远执行，就在循环体内部用了一个 if 语句跳出循环。

```
#! /bin/bash
i=1
sum=0
while true; do
if [ $ i -gt 100 ]; then
break:
fi
sum=- $  [ $ sum+ $ i]
i= $  [ $ i+1]
done
echo  $ sum
```

Continue 语句的作用是不执行本次循环，而直接跳到下一次循环，例如下面这个计算 100 以内奇数和的例子。

```
#! /bin/bash
i=1
sum=0
until [  $ i -gt 100]; do
    if [  $ [ $ 1%2) -eq 0 ]; then
       i= $  [ $ i+1]
       continue;
    fi
    sum= $  [ $ sum+ $ i]
    i= $  [ $ i+1]
done
echo  $ sum
```

exit 语句用于终止脚本程序并返回值。该值可用“ $?”在下一命令中获取。通常情况

下，正执行的程序将返回 0（TRUE），而未正常结束的程序则返回 1-255 间的错误代码。

```
#! /bin/bash
for age in 58 14 -25 26
do
    if [ $ age -1t 0] ; then
        echo "Sage is not a valid age. Exit. . . "
        exit 100
    else
        echo "Sage is a valid age"
    fi
done
```

执行该脚本会发现到-25 处程序就停止并跳出运行。此时执行“echo $?”命令就会显示“100”。

7.8 Shell 函数

Shell 里也可以使用函数。Shell 函数的名字必须是唯一的，且符合变量命名规则。所有用来织函数的命令就像普通命令一样执行。当以一个简单的命令名来调用函数的时候，和该函数名相同的命令就被执行。

7.8.1 声明 shell 函数

函数必须声明，然后才能在 Shell 里执行。自定义函数可以采用如下所示两种方法声明。

```
//方法一
function FUNCTION_NAME {
[EXPRESSIONS]
}
//方法二
FUNCTION_NAME ( ){
[EXPRESSIONS]
}
```

如果采用第二种方法,则括号是不能省略的。

例如,如下代码可定义一个简单的函数。

```
function hello {
    echo "Hello World"
}
```

Shell 函数的参数不通过 C 语言中形式参数的方法指定，而是通过系统变量指定。例

如“ $n”，“ $ * ”等．下面的 sayhello 函数就会获取“ $ @ ”变量。

```
sayhello ( ){
    for name in  $ @; do
        echo "Hello  $  (name) ! "
    done
}
```

Shell 函数还可以带上返回值，方法为 retum ［返回值］。retum 也可以不带返回值，将直接跳出函数体。

7.8.2　调用 Shell 函数

Shell 函数的调用也和 C 语言中调用函数的方法有所区别，其参数是直接跟在函数名后，而无需通过括号括起来，如下所示。

```
FUNCTION_ NAME PARAMI PARAM2. . .
```

例如调用前文中的 sayhello 命令，方法如下所示。

```
#! /bin/bash
sayhello ( ) {
    for name in  $ @; do
        echo "Hello  $  (name) ! "
    done
}
sayhello Derek Jeff Carol
```

执行该脚本则会输出如下内容。

```
Hello Derek!
Hello Jeff!
Hello Carol!
```

7.9　编写交互脚本

前面介绍的都是非交互脚本，而实际上 Linux 中有许多脚本，需要来自用户的输入，或者在运行的时候向用户输出信息。交互脚本有如下优势。

（1）可以建立更加灵活的脚本。

（2）用户可自定义脚本使得其在运行时产生不同的行为。

（3）脚本可以在运行过程中报告状态。

7.9.1　提示用户

提示用户最常用的命令是 echo，其基本用法前面已经使用过很多次了。这里仅列出其常用的些选项。

（1）-e：解释反斜杠转义字符。

（2）-n：禁止换行。

echo 中常用的转义字符序列见表 7-11。

表 7-11　echo 转义字符序列

选项	含义
\ a	响铃
\ b	退格
\ c\ e	强制换行
\ e	退出
\ f	清除屏幕
\ n	换行
\ r	回车
\ t	水平制表符
\ v	垂直制表符
\ \	反斜杠
\ ONNN	值为八进制值 NNN（0 到 3 个八进制数字）的 8 比特字符
\ NNN	值为八进制值 NNN（1 到 3 个八进制数字）的 8 比特字符
\ xHH	值为 1 六进制值（1 或者 2 个十六进制数字）的 8 比特字符

使用转义字符的示例如下所示。

```
[feng@localhost 桌面] $ echo -e "Red hat\nLinux"
Red hat
Linux
//解释转义字符
[feng@localhost 桌面] $ echo "Red hat\nLinux"
Red hat\nLinux
//不解释转义字符
```

7.9.2　用户输入

接受用户输入的命令为 read 命令。read 命令的语法如下所示。

read［选项］名称 1 名称 2 … …

相关选项如表 7-12 所示。

表 7-12　read 选项列表

选项	含义
-a ANAME	将输入读入 ANAME 的数组

续表

选项	含义
-d DELIM	用于截断输入的字符，默认是换行符“n”
-n NCHARS	读入 n 个字符显示一个提示
-p PROMPT	取消转义，例如启用时“n”将可能不会被解释为换行符
-S	安静模式，输入的字符将不会显示
-t　TIMEOUT	超时，超过指定时间，read 自动停止

如下命令将读入一个字符串，并显示出来。

```
// 将用户的输入读入 str
[root@localhost 桌面]# read -p "Please input some words:" str
Please input some words:linux
[root@localhost 桌面]# echo  $ str
linux
```

将前文中的 actorial. sh 改为互动模式,其内容如下。

```
#!  /bin/bash
factorial ( )
{
local 1- $ 1
if [ $ 1-eq 0]
Then
echo 1else
localj = 'expr  $ 1-1'
local k = 'factoria $ j'
echo 'expr si  \* sk'
                fi
}
while true
do
read-p "Please input an integer (input 'q'to exit): " num
if [Snum-"q"]; then
Break
fi
rtn = 'factorial  $ num'
echo "The factorial of Snum is  $ rtn
done
```

执行 sh factorial. sh 即可与之交互。

```
[root@localhost root] # sh
    factorial. shPlease input an integer (input 'q'to exit):3
The factorial of 3 is 6
Please input an integer (input 'q'to exit):6
The factorial of 6 is 720
Please input an integer (input 'q'to exit):9
The factorial of 9 is 362880
Please input an integer (input 'q'to exit):q
root@localhost root]#
```

7.10 数组

几乎所有程序语言都支持数组，目前的 bash 版本也提供了创建一维数组的能力。本节将介绍 Shell 编程中数组的使用方法。

7.10.1 为什么使用数组

目前我们使用的变量都是由变量名和变量值组成的，并且一个变量只能包含一个值，这种数据结构称为标量变量。如果在编程中需要使用多个数据，使用标量变量的方法将会产生多个变量，这样会影响计算机处理的效率，并且使用多个变量名也会给编程造成极大的麻烦。为了解决这些问题，在许多计算机语言中使用数组来存放数据类型相同的数据。在 Shell 中，因为没有数据类型的约束，数组的使用将更加灵活。

数组是可以在内存中连续存储多个元素的结构，也可以理解为是一次存放多个值的变量。数组由数组名、数组元素和元素下标组成。

在计算机内存中，同一数组的所有元素按顺序依次存放在相邻的存储单元中，使用下标访问每一个元素，这样大大提高了内存的访问效率，并且在同一数组中存放逻辑功能相同的数据，为编程提供了便利。

7.10.2 数组的创建、赋值和删除

与 bash 中的其他变量一样，数组的创建和赋值可以同时进行，但是需要注明数组元素的下标。数组创建与赋值的命令格式如下。

```
array［subscript］=value
```

其中 array 为数组名，是数组的唯一标识；subscript 为从 0 开始的数组元素下标；value 为对应元素的值。下面是数组创建和赋值的例子。

```
[user@localhost ~] $ arraY[2]=10
[user@localhost ~] $ echo $ array[2][2]
[user@localhost ~] $ echo $ (array[2])
10
```

```
[user@localhost ~] $
```

在这个例子中，第一条命令将 10 赋值给数组 array 中下标为 2 的元素。在后面的两条命令中演示了访问数组元素的方法，在读取数组元素的值时，需要用花括号指明数组元素的完整名称，以免 Shell 将元素名扩展成路径名。

作为能够存储多个数值的数据结构，数组还支持一次赋值多个数据，命令格式如下。

```
array = (valuet1 value2…)
```

其中 value1、value2 等值依次赋予从下标为 0 开始的数组元素。例如:

```
[user@localhost ~] $ array=(one two three four)
[user@localhost ~] $ echo $ (array[0]) $ (array[1]) $ (array[2]) $ (array[3]))
one two three four
[user@localhost ~] $
```

另外，也可以指定一个下标来给特定的元素赋值。例如：

```
[user@loclhost ~] $ array=([0]=1 [3]- 4)
[user@localhost ~] $ echo $ {array[0]) $ {array[1]) $ {array[2]) $ {array[3])
1 two three 4
[userr@localhost ~] $
```

在许多编程语言中出现的+=运算符也可以使用在 Shell 的数组中，作用是在数组的尾部追加新元素。例如，在数组 array 尾部追加了两个元素并分别赋值为 5 和 6。

```
[user@ localhost ~] $ array+=(56)
[user@localhost ~] $ echo $ {array[0]) $ {array[1]) $ {array[2]) $ {array[3])
 $ {array[4]) $ {array[5])
1 two three 4 5 6
[user@localhost ~] $
```

对于数组的删除操作，可分为删除数组和删除数组中的指定元素。删除操作需要使用 unset 命令，删除数组的命令格式如下。

```
unset array
```

例如：

```
[user@localhost ~] $ array=( 1 2 3)
[user@localhost ~] $ echo $ {array[0]) $ {array[1]) $ {array[2])
[user@localhost ~] $ unset array
[user@localhost ~] $ echo $ {array[0]) $ {array[1]) $ {array[2])
[user@localhost ~] $
```

删除数组中指定元素的命令格式如下。

```
unset array[subscript]
```

例如：

```
[user@localhost ~] $ array=(1 2 3)
[user@localhost ~] $ echo $ {array[0]) $ {array[1]) $ {array[2])
```

```
1  2  3
[user@localhost ~] $ unset array[1]
[user@localhost ~] $   echo $ {array[0]}   $ {array[1]}   $ {array[2]}
1 3
[user@localhost ~] $
```

7.10.3 遍历访问数组元素

在前面的例子中使用了下标的方式对数组元素进行创建和赋值等操作。在处理多个值时，这么做显然没有发挥数组的作用。特别是在一些数组长度不确定的情况下，就需要遍历访问数组中的每一个元素。循环是一种非常适合遍历数组的方法。如下面的例子，使用循环对数组进行赋值和访问。

编写脚本 array1. sh，使用 for 循环对数组进行赋值，例如：

```
#! /bin/bash

array1[0]=1
for((i=1; i<10; i=i+1)); do
array1( $ i]= $ [ $ larray1[ $ 1- 1])* 2]
done
```

上面的脚本用一个等比数列的前 10 项为数组 array1 的前 10 个元素赋值。现在增加脚本内容，使用直到循环访问数组 array 1 的所有元素，代码如下。

```
#! /bin/bash
array[0]=1
for (( i=1; i<10; i=i+1 ));do
array[ $ i]= $ [ $ {array[ $ i- 1)*2]
done
i=0
until [ -Z $ {array[ $ i]1 ];do
echo $ {array[ $ i] }
i= $ [ $ i+1]
do
```

脚本 array1. sh 执行结果如下。

```
  [ user@localhost ~] $ . /array1. sh
1
2
4
8
```

```
16
32
64
128
256
512
[user@localhost ~] $
```

上面的例子演示了数组的循环赋值和遍历方法。一般情况下，使用 until 循环数组下标直到元素不为空的方法就可以遍历数组了。然而有一种更简便、更强大的数组遍历方法。通过学习位置参数，我们知道使用双引号引用的 $@ 可以列出所有参数并保持它们的完整性。在数组中，符号@ 也有类似的作用，使用@ 作为数组的下标将会得到数组的所有元素，如果再使用双引号引用，则可以得到完整的元素序列。修改脚本 array1. sh 修改数组的遍历方式如下。

```
#! /bin/bash
array[0]=1
for (( i=1; 1<10; i=i+1 )) ; do
arrayI $ i]- $ [ $ (array[ $ i- 1])*2]
done
for i in "Slarray[e]}"; do
echo $ i
done
```

这样修改脚本以后和修改前的执行效果完全一样。另外，使用#符号能取得元素序列的长度，添加如下相应代码到脚本 array1. sh 中。

```
#! /bin/bash
array[0]=1
for (( i=1; i<10; 1=i+1 )); do
array[ $ i]= $ [ $ {array[ $ i-1]}*2]
done
echo "S{#array[@]} elements of array show behind:"
for i in " $ {array[@]}"; do
echo $ i
done
```

脚本 array1. sh 执行结果如下。

```
[userelocalhost ~] $ . /test. sh
10 elements of array show behind:
1
2
```

```
4
8
16
32
64
128
256
512
[userelocalhost ~] $
```

这样便能获取到数组元素的数目，这会为编程带来很大的帮助。

7.11 Shell 编程综合案例

7.11.1 判断两个数字的大小

```
#! /bin/bash
#文件名:/root/bin/numcmp.sh
#定义变量
read -p "Please input the first num:" num1
read -p "Please input the second num:" num2
#判断数字是否符合标准
if [[ $num1 =~ ^[0-9]+$ && $num2 =~ ^[0-9]+$ ]]; then
#判断两个数字的大小并输出判断结果
    if [ $num1 -lt $num2 ]; then
        echo "The num2 is biger than the num1"
    elif [ $num1 -eq $num2 ]; then
        echo "Two numbers equal"
    else
        echo "The num1 is biger than the num2"
    fi
else
    echo "Please enter the correct number"
fi
```

7.11.2 创建用户

使用一个用户名做为参数，如果指定参数的用户存在，就显示其存在，否则添加之；显示添加的用户的 id 号等信息 。

```
#! /bin/bash
#文件名:/root/bin/createuser. sh
#定义变量
read -p "请输入一个用户名:" name
#判断用户名是否存在
if  'id  $ name &> /dev/null'; then
#      若存在,则输出 ID 等信息
echo "用户存在,用户的 ID 信息为:'id  $ name'"
else
# 若不存在,则添加用户,设置密码为随机 8 位,下次登录时提示修改密码,同时显示 ID 等信息
passwd='cat /dev/urandom |tr -cd [:alpha:] |head -c8'
'useradd  $ name &> /dev/null'
'echo " $ passwd" | passwd --stdin  $ name &> /dev/null'
echo "用户名: $ name 密码:  $ passwd" >> user. txt
'chage -d 0  $ name'
echo "用户已添加,用户的 ID 信息为:'id  $ name'密码为: $ passwd"
fi
```

7.11.3　判断用户输入的信息

编写一个脚本，提示用户输入信息，判断其输入的是 yes 或 no 或其他信息。

```
#! /bin/bash
#文件名:/root/bin/yesorno. sh
#定义变量
read -p "Yue ma?(yes or no):"ANS
#把变量中的大写转换为小写
ans='echo " $ ANS" |tr [[:upper:]] [[:lower:]] '
#判断输入的信息是什么并输出结果
case  $ ans in
yes|y)
      echo "see you tonight"
      ; ;
no|n)
      echo "sorry,I have no time"
      ; ;
*)
      echo "what's your means?"
```

```
    ;;
esac
```

7.11.4 判断文件类型

编写 Shell 程序，判断用户输入文件路径，显示其文件类型（普通，目录，链接，其它文件类型）

```
#! /bin/bash
#文件名:/root/bin/filetype.sh
read -p "请输入一个文件路径:" file
#判断文件是否存在
'ls $file &> /dev/null'
#若存在,判断文件类型并输出
if [ $? -eq 0 ]; then
    style='ls -ld $file | head -c1'
    case $style in
    -)
      echo "这是一个普通文件"
      ;;
    d)
      echo "这是一个目录文件"
      ;;
    l)
      echo "这是一个链接文件"
      ;;
    *)
      echo "这是其他类型文件"
      ;;
    esac
#若不存在,提示并退出
else
    echo "该文件不存在"
    exit 2
fi
```

7.11.5 打印九九乘法表

```
#! /bin/bash
# 文件名:/root/bin/jiujiu.sh
```

```
# 判断 i 的值是否在 1- 9
for i in {1. . 9}; do
#    判断 j 的值是否在 1- $ i
for j in 'seq 1  $ i'; do
#         若在,则打印 i*j 的值
          echo -en " $ i* $ j =  $ [ $ i* $ j] \t"
          done
          echo
done
```

7. 11. 6　输入正整数 n，计算 1+…+n 的和

```
#!  /bin/bash
#文件名:/root/bin/sum. sh
#定义变量
sum=0
read -p "请输入一个正整数:" num
#判断 num 是否是正整数
if [[ $ num  = ~ ^[[:digit:]]+ $ ]]; then
#      若是,当 i 在 1- $ num 时, 输出 sum 值
       for i in 'seq 1  $ num'; do
           let sum+= $ i
       done
           echo "sum= $ sum"
#若不是,提示输出正整数
else
       echo "请输入一个正整数!"
fi
```

7. 11. 7　计算 100 以内所有正奇数之和

```
#!  /bin/bash
# 文件名:/root/bin/sum2. sh
# 定义变量
i=1
sum=0
#当 i<100 时,执行下面语句
while [ $ i -le 100 ]; do
#当 i 为奇数时,另 sum=sum+I, i=i+1
```

```
    while [ $[i%2] -eq 1 ]; do
      let sum+= $i
    let i+=1
done
#     当 i 不为奇数时,i=i+1
      let i+=1
done
#输出结果
echo "sum= $sum"
```

7.11.8 循环输出 1-10

```
#! /bin/bash
#文件名:/root/bin/printnum.sh
#定义变量
i=1
#当 i>10 时,退出循环
until [ $i -gt 10 ]; do
#     输出 i 的值,i=i+1
      echo $i
      let i+=1
done
```

7.11.9 检查软件包是否安装

```
#! /bin/bash
#文件名:/root/bin/softinstall.sh
if rpm -q sysstat &>/dev/null; then
    echo "sysstat is already installed."
else
    echo "sysstat is not installed!"
fi
```

7.11.10 判断用户输入的是否为数字

```
#! /bin/bash
#文件名:/root/bin/number.sh
if [[ $1 = ~^[0-9]+ $ ]]; then
    echo "Is Number."
else
```

```
    echo "No Number. "
fi
```

7.11.11　连续输入 5 个 100 以内的数字，统计和、最小和最大

```
#! /bin/bash
#文件名:/root/bin/sum2. sh

COUNT=1
SUM=0
MIN=0
MAX=100
while [ $COUNT -le 5 ]; do
    read -p "请输入 1-10 个整数:" INT
    if [[ ! $INT =~ ^[0-9]+$ ]]; then
        echo "输入必须是整数!"
        exit 1
    elif [[ $INT -gt 100 ]]; then
        echo "输入必须是 100 以内!"
        exit 1
    fi
    SUM=$(($SUM+$INT))
    [ $MIN -lt $INT ] && MIN=$INT
    [ $MAX -gt $INT ] && MAX=$INT
    let COUNT++
done
echo "SUM: $SUM"
echo "MIN: $MIN"
echo "MAX: $MAX"
```

小结

Shell 既可以解释和执行用户输入的命令，也可以用来进行程序设计。Shell 程序基本语法较为简单，主要由开头部分、注释部分以及语句执行部分组成。像高级程序设计语言一样，Shell 也提供说明和使用变量的功能。

Shell 提供了用来控制程序和执行流程的命令，包括条件分支和循环结构，用户可以用这些命令创建非常复杂的程序。在 Shell 程序中循环控制语句可以使用 for 语句、while 语句以及 until 语句。for 循环语句对一个变量的可能的值都执行一个命令序列。while 语句

是用命令的返回状态值来控制循环的。until 循环语句是另外一种循环结构，它和 while 语句相类似。

习题

7-1　简述 Shell 的功能。

7-2　简述 Shell 程序的组成。

7-3　简述 Shell 程序创建过程。

7-4　简述 Shell 程序中 if 条件语句和 case 条件语句的区别。

上机练习

7-1　查看当前系统下用户 Shell 定义的环境变量的值。

7-2　定义变量 v1 的值为 25，并将其显示在屏幕上。

7-3　定义变量 v2 的值为 160，并使用 test 命令比较其值是否大于 150。

7-4　创建一个简单的 Shell 程序，其功能为显示计算机主机名以及显示系统日期和时间。

7-5　使用 for 语句创建一个 Shell 程序，其功能为 1+2+3+4+5+. . . . . +125。

7-6　使用 while 语句创建一个 Shell 程序，其功能为计算 1～10 的平方和。