第5章 Shell 程序设计

教学目标

- 了解 Shell 的功能和作用。
- 掌握 Shell 脚本的建立和执行方式。
- 掌握 Bash 各种变量的定义和运用。
- 掌握 Bash 程序设计常用控制语句的使用方法。
- 掌握 Bash 函数的定义及使用方法。
- 能阅读和编写 Shell 程序。

5.1 Shell 概述

5.1.1 Linux 中的 Shell

作为 Linux 操作系统的外壳，Shell 是用户与 Linux 操作系统交互的接口。交互式地解释和执行用户输入的命令是 Shell 的基本功能。同时，Shell 又是一种解释型的程序设计语言，支持绝大多数在高级语言中能见到的程序元素，如函数、变量、数组和程序控制结构。Shell 程序设计语言简单易学，可实现与相关 Linux 命令的有机结合，大大提高了编程效率。Linux 系统下的每个 Shell 程序称为一个脚本，是系统管理维护的重要手段。

由于 Linux 对 Shell 采用独立、自由、开放的方式，因此 Shell 的种类较多，常见的有 Bourne Shell(简称 sh)、C Shell(简称 csh)和 Korn Shell(简称 ksh)。

Bourne Shell 是 UNIX 默认的 Shell，在所有版本的 UNIX 中均提供。Bourne Shell 在 Shell 编程方面相当优秀，但在处理与用户的交互方面做得却不如其他几种 Shell。

C Shell 比 Bourne Shell 更适于编程，其语法与 C 语言很相似。Linux 为喜欢使用 C Shell 的人提供了 Tcsh。Tcsh 是 C Shell 的一个扩展版本，它支持命令行编辑、可编程单词补全、拼写校正、历史命令替换、作业控制等功能。

Korn Shell 集合了 C Shell 和 Bourne Shell 的优点，并且和 Bourne Shell 完全兼容。Linux 系统提供了 pdksh(ksh 的扩展)，它支持任务控制，可以在命令行上挂起、后台执行、唤醒或终止程序。

Bourne Again Shell(Bash)是 Linux 操作系统默认的 Shell，它是 Bourne Shell 的扩展。Bash 与 Bourne Shell 完全向后兼容，并在 Bourne Shell 的基础上增加了很多特性，如提供命令补全、命令编辑、命令别名和命令历史表等功能。它还包含很多 C Shell 和 Korn Shell 中的优点，既有灵活强大的编程接口，又有很友好的用户界面。

除了上述常用的 Shell 外，Linux 还提供一些流行的 Shell，如 ash、zsh 等。不同 Shell

之间的语法规则并不完全兼容，本章主要讲述基于 Bash 的程序设计。

5.1.2 Linux Bash 主要的特色功能

1. 命令补全

按 Tab 键，自动补全命令或程序名。

2. 使用通配符

“*”可替代多个字符，“?”可替代一个字符。

3. 历史命令

能自动跟踪用户每次输入的命令，并把输入的命令保存在历史列表缓冲区，用户可使用上下箭头键找出执行过的历史命令。

4. 别名功能

可使用 alias 和 unalias 命令给命令或可执行程序起别名和清除别名。

5.2 Shell 的启动与切换

用户登录时，系统会根据/etc/passwd 文件中该用户信息的配置内容启动默认的 Shell。如不特别指定和修改，该字段的内容在用户建立时通过读取/etc/default/useradd 的 SHELL 赋值得到。查看当前用户登录后默认使用的 Shell 种类，除了查看/etc/passwd 文件外，还可通过查看环境变量 SHELL 得到。

```
[root@localhost /]# echo $SHELL
/bin/bash
```

当然，用户也可临时改变或切换到非默认 Shell，可通过运行如下命令获取当前使用的 Shell 类别。

```
[root@localhost ~]# echo $0
-bash
```

5.2.1 启动新的 Shell

执行该新 Shell 对应的应用程序即可实现新 Shell 的启动，这些应用程序文件一般位于/bin 目录下。如以下命令可启动 Bourne shell。

```
[root@localhost /]# sh
sh-4.2#
```

启动成功后的 Shell 成为启动前 Shell 的子 Shell。

5.2.2 Shell 的退出

执行 exit 命令可退出当前 Shell，返回上一级父 Shell，在顶层 Shell 执行该命令，则退出系统。

```
sh-4.2# exit
exit
[root@localhost /]#
```

5.3 Shell 脚本的建立与执行

5.3.1 脚本的创建

Linux Shell 脚本类似 Windows 操作系统中的批处理文件，是能直接在 Shell 中解释执行的程序。Shell 脚本文件的建立方法与普通纯文本文件的建立方法相同。凡是能建立纯文本文件的编辑工具均可用来建立 Shell 脚本文件，如 Linux 中的 vi/vim、emacs、gedit，Windows 系统中的 Notepad、UltraEdit 等。例如，使用 vi，在当前目录（假定为/root/shellprog）创建一个文件名为 test01.sh 的脚本文件，其内容如下。

```
#!/bin/bash
#该脚本文件实现显示日期和登录系统的用户信息
date
who
```

在上述程序行中，以“#”开始的行是注释行，不会被执行。但“#!/bin/bash”行一般必须提供（如果指定的 Shell 与运行环境的 Shell 种类一致可省略），且最好位于脚本文件的第一行，它指明以后的脚本语句由哪种 Shell 来解释执行。注释之后的两句均为普通的能在 Shell 中运行的 Linux 命令。Bash 脚本中，多语句行也可通过分号续入同一格式行中，test01.sh 内容格式也可这样：

```
#!/bin/bash
#该脚本文件实现显示日期和登录系统的用户信息
date;who
```

5.3.2 Shell 脚本的执行

在 Linux Bash 中，Shell 脚本的执行有以下几种常用方式。

（1）将脚本文件的权限设置为可执行，在提示符下直接执行。

由于文本文件编辑工具产生的文件默认不具有可执行属性，如要脚本文件像其他可执行程序一样在 Shell 命令提示符中可直接执行，需要增加执行权限属性：

```
[root@localhost shellprog]# chmod a+x test01.sh
```

然后在 shell 命令提示符中输入：

```
[root@localhost shellprog]# ./test01.sh
```

当然，也可把增加执行属性之后的 test01.sh 复制到当前用户 PATH 环境变量包含的目录下，或把所在路径添加到 PATH 环境变量中，直接在 Shell 命令提示符下输入：

```
[root@localhost shellprog]# test01.sh
```

注：使用 which 命令进行搜索，或直接输入可执行程序或命令名，Shell 会到 PATH 环境变量包含的路径中去搜索，搜索到第一个匹配程序后，启动子进程执行它，执行完后又回到命令提示符状态。

（2）使用“.”或 source 命令执行脚本。

```
[root@localhost shellprog]#. test01.sh
[root@localhost shellprog]#source test01.sh
```

在 Bash 中，“.”和“source”命令后的参数会明确告诉 Shell，后面的第一个参数是脚本

程序。该脚本程序将直接在当前 Shell 环境中执行，不会启动子 Shell。

(3) 启动新的 Shell 执行脚本文件。

```
[root@localhost shellprog]# bash test01.sh
```

该方式是启动一个子 Shell 同时传入需解释执行的脚本文件名，执行完后子 Shell 退出。

5.3.3 开启脚本调试方式

在 Bash 中，如下脚本执行方式可打开调试跟踪功能。

```
bash -x 脚本文件名
```

一旦打开了调试功能，Bash 就会在执行程序的每一行时，以扩展的形式(会解析命令中的变量等)显示每一行，并在每条命令前面都加上一个加号(与输出内容相区分)。

例 5-1：

```
[root@localhost shellprog]# bash -x ./test01.sh
+ date
Fri Jan 24 10:45:36 CST 2020
+ who
root     tty1         2020-01-24 07:13
root     pts/0        2020-01-24 10:00 (192.168.150.1)
root     pts/1        2020-01-24 08:20 (192.168.150.1)
root     pts/2        2020-01-24 10:45 (192.168.150.1)
```

5.4 Shell 程序设计语言基本语法

5.4.1 Shell 变量

变量与计算机的内存单元相对应，Linux Bash 中的变量可分为 Shell 环境变量和用户自定义的用户变量。

1. 用户变量

用户变量是指由用户在 Shell 或脚本程序中自定义和赋值使用的变量。变量名要求以字母或下画线开始的由字母、数字或下画线组成的字符串，大小写敏感。与强类型高级语言(C、Java 等)不同，Shell 中的用户变量无须声明和初始化。默认情况下，所有用户变量都被看作字符串类型并以字符串方式存储，Shell 会在操作时根据需要自动进行类型转换，未初始化的变量默认值为空串。用户变量默认作用域为当前 Shell，不能传导到子 Shell 中，类似高级语言中的局部变量，只是该局部范围为当前 Shell。Bash 中，也可通过 declare 关键字来设定变量的一些属性。

1) declare 基本格式

```
declare [-aixrp]变量名
```

2) 常用选项

-a：将指定变量定义为数组型。

-i：将指定变量定义为整数型。

-f：将指定变量定义为函数型。

-x：将指定变量变为环境变量。

-r：将指定变量设置成只读。

-p：显示指定变量的被声明的类型。

3）变量的赋值

在 Bash 中，变量的赋值语法格式如下。

```
变量名=字符串
```

注：

（1）在 Bash 中，可认为所有变量类型默认为字符串类型，其默认的初值为空串，故在 Bash 中，变量赋值和定义是同时进行的。

（2）赋值语句“=”两侧不能有空格。

（3）变量赋值等号右侧的字符串可以不用引号引起来，但如果右侧串中包含空格、制表符和换行符等，则应该使用引号引起来。故使用引号引起来一起进行字符串赋值是一个好习惯。

例 5-2：

```
str1=hello,world!
str2="good morning"
```

4）Bash 变量的引用

在 Bash 中，引用变量值的基本方式为

```
$变量名 或 ${变量名}
```

例 5-3：

```
[root@localhost shellprog]#str1=Hello
[root@localhost shellprog]#str2=${str1},world!
[root@localhost shellprog]#echo $str2
Hello,world!
```

例 5-3 第 3 行 echo 为 Bash 内置命令，可实现其后面给定的字符串在标准输出终端上的显示。默认执行该命令后，光标会自动换行；如需光标停留在显示内容后，则需使用 -n 选项参数。

“echo $str2”命令行语句执行完后显示的内容为：Hello,world!。如果对变量的引用出现在字符串开始或中间（如对 str2 赋值情形），则最好采用“${变量名}”格式。

5）命令替换

在 Bash 中，可以将命令的执行输出结果赋给变量，获取命令执行输出结果一般形式为

```
`命令表` 或 $(命令表)
```

注：格式`命令表`中使用的包围符号为倒（反）引号，非单引号，美式键盘中与“～”共键，命令表如包含多条命令，命令之间用分号分隔。

例 5-4：

```
[root@localhost shellprog]# userAndCurDir=`whoami;pwd`
[root@localhost shellprog]# echo $userAndCurDir
root /root/shellprog
```

6）Shell 中引号的使用

在 Shell 变量赋值和其他脚本语句中，会用到三种引号：双引号、单引号和倒引号。由

双引号(")引起来的字符,除 $、倒引号(`)、单引号(')、双引号(")、逻辑非(!)和反斜线(\)(这些特殊字符如需作普通字符对待,需使用反斜线字符(\)进行转义,如"\$"代表普通字符 $,而不是变量引用前置符)外,均作为普通字符对待。

例 5-5:

```
[root@localhost shellprog]# str1="Hello"
[root@localhost shellprog]# str2="world"
[root@localhost shellprog]# str3="${str1} ${str2}\!"
[root@localhost shellprog]# echo $str3
Hello world\!
```

在例 5-5 中,为了有效地分隔变量名字符串,可使用 {} 把变量名括起来引用变量值。

单引号引起来的字符或字符串都作为普通字符对待。

例 5-6:

```
[root@localhost shellprog]# str1="Hello"
[root@localhost shellprog]# str2="world"
[root@localhost shellprog]# str3='${str1} ${str2}\!'
[root@localhost shellprog]# echo $str3
${str1} ${str2}\!
```

倒引号(`)引起来的字符串被 Shell 解释为命令行列表,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。倒引号还可以嵌套使用。但应注意,嵌套使用时内层的倒引号必须用反斜线(\)将其转义。

7) 变量的删除

对于非只读变量,删除变量的命令格式如下。

```
unset 变量名
```

例 5-7:

```
[root@localhost ~]# a=20
[root@localhost ~]# echo $a
20
[root@localhost ~]# unset a
[root@localhost ~]# echo $a
```

执行 unset a 命令后,a 的内容为空,故例子中最后一条命令无内容输出。

2. 数组变量

Bash 支持一维数组,初始化时不需要定义数组大小。数组的下标是整数,从 0 开始编号。

1) Bash 数组变量声明并赋值的语法格式

```
数组变量名=(元素值1 元素值2 … 元素值n)
```

元素值之间一般以空格分隔(其他可用分隔符见 IFS 环境变量)。

2) 对数组元赋值的一般格式

```
数组变量名[下标]=值
```

也可以用 declare -a 显式声明一个变量为数组变量。

3) 数组变量值引用的一般格式

```
${数组变量名[下标]}
```

注:Bash 中,下标使用@或 * 可以获取数组中的所有元素值。另外,${#数组名

[*]}或${#数组名[@]}可获取数组中元素总个数，${#数组名[数字下标]}可获取指定下标元素的长度。

例 5-8：

```
[root@localhost shellprog]# animals=(dog pig cat)
[root@localhost shellprog]# animals[3]=sheep
[root@localhost shellprog]# echo "${animals[2]} ${animals[3]}"
cat sheep
[root@localhost shellprog]# echo ${animals[@]}
dog pig cat sheep
[root@localhost shellprog]# echo "${#animals[3]}"
5
[root@localhost shellprog]# echo "${#animals[2]}"
3
[root@localhost shellprog]# echo "${#animals[*]}"
4
```

3. 环境变量

Shell 中维护着一组称为环境变量的变量。这些变量用来记录特定的系统信息。如系统的名称、当前登录用户的用户名、用户 ID、主目录和执行文件搜索路径等，这些变量通常用大写字母作名字，在启动的子 Shell 或子进程中亦可见。

Bash 中常用的环境变量名及代表的意义如表 5-1 所示。

表 5-1　Bash 中常用的环境变量名及代表的意义

环境变量名	说　明
HOME	当前用户的主目录
PATH	命令搜索路径，以冒号隔开的目录路径列表
PS1	命令提示符
PS2	二级提示符，用来提示续行的输入，通常为“>”字符
IFS	内置环境变量，存放字段或命令参数分隔符，通常为空格、制表符和换行符
PWD	当前工作目录名称
TERM	当前用户控制终端的类型

环境变量的赋值、引用和删除与用户变量操作类似，如以下命令实现把当前路径加入环境变量 PATH 中：

```
[root@localhost shellprog]# PATH=$PATH:.
```

操作成功后，当前路径下的可执行程序直接输入程序文件名即可执行：

```
[root@localhost shellprog]# test01.sh
```

在 Bash 中，可通过 export 或 declare 命令把用户变量放入环境中，导出成为环境变量，从而可传递到启动的子 Shell 或子进程中。

1）导出用户变量基本格式

```
export 变量名
declare -x 变量名
```

2）取消导出基本格式

```
export -n 变量名
declare +x 变量名
```

例 5-9：

```
[root@localhost shellprog]# a=10
[root@localhost shellprog]# export b=20
[root@localhost shellprog]# bash
[root@localhost shellprog]# echo "a=$a b=$b"
a= b=20
[root@localhost shellprog]# exit
exit
[root@localhost shellprog]# declare -x a
[root@localhost shellprog]# bash
[root@localhost shellprog]# echo "a=$a b=$b"
a=10 b=20
```

在例 5-9 中，第 1 行声明了普通用户变量 a 并赋值 10，第 2 行给用户变量 b 赋值 20 并放入环境中成为环境变量，第 3 行启动子 Bash，第 4 行在子 Bash 中显示两个变量的值(a 未定义，不可见，b 可见)，第 6 行退出子 Bash，第 8 行把 a 变成环境变量，重新启动子 Bash 显示两个变量值，发现在子 Bash 中，两者均可见。

例 5-9 中生成的环境变量为临时变量，用户退出系统后重新登录不会再生效。若要永久生效，可在环境文件.bashrc 和.bash_profile 中定义设置。

在 Bash 中，可用不带参数的 set、env、export、printenv 等命令查看当前用户的环境变量，set 命令还可查看声明定义的普通用户变量。

4. 位置参数

位置参数包括命令名和命令行参数。在 Shell 脚本中，按照在命令行上的出现位置确定其值的参数，不能通过赋值语句改变位置参数的值。在 Bash 中，可通过 set 命令修改除命令名之外的其他位置参数值。在脚本程序中，${0}表示命令名或程序名，${1}～${n}表示第 1～n 个参数，当 n＜10 时，大括号可省略。

如在命令行输入：

```
./test01.sh p1 p2 p3
```

在脚本文件 test01.sh 运行时，$0 为./test01.sh，$1 为 p1，$2 为 p2，$3 为 p3，大于 3 的位置参数值为空串。

在 Bash 中，$* 和$@表示除$0 之外的命令行上输入的所有位置参数值(在双引号内，两者引用结果有区别，“$*”为所有位置参数值组成的一个字符串，“$@”中各位置参数值字符串独立成串)，$# 表示命令行上输入的位置参数个数(不计$0)。

1) 初始化位置参数

可用 set 命令依次初始化位置参数，但$0 除外，$0 始终为命令名或脚本程序名。

例 5-10：

```
[root@localhost shellprog]# set --
[root@localhost shellprog]# set a.txt b.txt c.txt
[root@localhost shellprog]# echo "\$0=$0 \$1=$1 \$2=$2 \$3=$3 \$4=$4"
$0=-bash $1=a.txt $2=b.txt $3=c.txt $4=
```

上例中，set --用于清空位置参数值，第 2 行 set 命令使$1、$2、$3 分别为 a.txt、b.txt、c.txt，但$0 不变。

2) 移动位置参数

语法格式：

```
shift [n]
```

在 Bash 中，内置命令 shift 可用于重新分配位置参数的值。每执行一次 shift [n]命令，就会把位置参数（$0 除外）整体向左移 1 个（执行无参 shift 命令，默认移动 1 个占位）或 n 个位置。向左移动时，最左边指定个数的参数值将被丢弃。如执行一次无参 shift 命令，$1 将被丢弃，新的 $1 的值将被原 $2 值代替，新的 $2 值将被原 $3 值代替，以此类推。

下面的脚本程序 test02.sh 演示了位置参数的用法。

```
#!/bin/bash
echo  "Argument list : $@"
echo  "The total of argument is $#"
echo  "The script file name is $0"
echo  "The first argument is $1"
echo  "The ninth argument is $9"
shift
echo "The tenth argument is $9"
echo  '$@': $@,'$#': $#,'$1': $1
```

执行该脚本结果如下。

```
[root@localhost shellprog]# ./test02.sh p1 p2 p3 p4 p5 p6 p7 p8 p9 p10
Argument list :p1 p2 p3 p4 p5 p6 p7 p8 p9 p10
The total of argument is 10
The script file name is ./test02.sh
The first argument is p1
The ninth argument is p9
The tenth argument is p10
$@:p2 p3 p4 p5 p6 p7 p8 p9 p10, $#:9, $1:p2
```

5. 其他特殊参数

如表 5-2 所示，Bash 中还有一些特殊的参数代表特殊意义，其引用值只能由 Bash 动态设置，在脚本中可引用但不能通过赋值语句直接修改其值。

表 5-2　Bash 中一些特殊的参数引用代表的含义

参数名引用	含　义
$$	当前进程的进程号（PID）
$!	后台运行的最后一个进程的进程号（PID）
$?	最后一条命令的退出状态值，通常 0 表示执行成功，非 0 表示执行失败
$-	Bash 已设置的控制功能选项标识（通过 set 和 shopt 设置），如是否启用变量自动导出成环境变量，是否启用 Shell 调试功能等
$_	前一条已执行命令的第一个参数

6. 接收用户输入

使用 read 命令可实现从标准输入设备（键盘）读入数据赋给变量，基本格式如下。

```
read 变量 1 [变量 2 ]…
```

注：当命令行中给出的变量个数与给定数据个数相同，则依次对应赋值；若变量个数少于数据个数，则从左至右对应赋值，最后一个变量被赋予剩余的所有数据；若变量个数多于给定数据个数，则依次对应赋值，而没有数据与之对应的变量取空串。输入数据时，数据一般以空格或制表符分隔。

例 5-11：

```
[root@localhost shellprog]# read str1 str2
pig dog cat
[root@localhost shellprog]# echo "str1 = $str1 str2 = $str2"
str1 = pig str2 = dog cat
root@localhost shellprog]# read n1 n2 n3
10 20
[root@localhost shellprog]# echo "n1 = $n1 n2 = $n2 n3 = $n3"
n1 = 10 n2 = 20 n3 =
```

7. 参数置换变量

参数置换变量为另一种变量赋值方式，可以根据不同条件给变量赋不同的值，参数置换变量主要有以下 4 种方式。

```
name2 = ${name1:-word}
```

如果 name1 的值为空，name2 的值为给定的 word 字符串；否则 name2 的值等于 name1 的值，name1 的值不变。

```
name2 = ${name1:=word}
```

如果 name1 的值为空，则 name1 和 name2 的值均为给定的 word 字符串；否则 name2 的值等于 name1 的值，name1 的值不变。

```
name2 = ${name1:+word}
```

如果 name1 的值为空，则 name1 和 name2 的值均为空；否则 name2 的值为给定的 word 字符串，name1 的值不变。

```
name2 = ${name1:?word}
```

如果 name1 的值不为空，name2 的值等于 name1 的值，name1 的值不变；否则按以下格式显示。

```
Shell 脚本名:变量 1:word
```

并从当前 Shell 退出，name2 保持原值，这种方式主要用于出错提示。

8. 输入输出重定向

在 Bash 中执行一条命令时，系统会启动一个进程与之对应，该进程会自动打开三个标准文件：标准输入文件 stdin(默认对应键盘)、标准输出文件 stdout(默认对应终端屏幕)和标准错误输出文件 stderr(默认对应终端屏幕)。故在 Bash 中，默认情形下，输入从键盘读取，显示信息和出错信息输出到屏幕上。也可以通过 Bash 的重定向符“<”“<<”“>”“>>”进行命令和程序的输入/输出重定向，即替换默认的标准输入/输出设备。

1) 输入重定向

符号“<”可实现输入重定向，如可用读取普通文件内容代替从键盘上读取输入字符。输入重定向的格式为

```
命令 < 文件名
```

如默认情况下，read 从键盘读入值到变量中，可使用重定向符从文件读取值。

例 5-12：

```
[root@localhost shellprog]# cat input.txt
35 78
[root@localhost shellprog]# read a b < input.txt
```

```
[root@localhost shellprog]# echo "a = $a,b = $b"
a = 35,b = 78
```

例 5-12 中,a,b 中的值使用“<”从 input.txt 文件中读取输入值。

符号“<<”可实现从命令行而不是文件重定向输入数据,其基本格式为

```
命令 << 开始标记符
输入行
...
输入行
结束标记符
```

标记符可自己指定,但开始的标记符和结束标记符必须相同。

例 5-13:

```
[root@localhost shellprog]# wc -l << EOF
This is added line
EOF
1
```

例 5-13 中,test.txt 中只有一行内容,通过“<<”从命令行给 wc 命令增加了计数行,但 test.txt 内容并未改变。

2) 输出重定向

符号“>”和“>>”可实现输出重定向,使默认输出不再到屏幕而是到某个文件,其基本格式为

```
命令 > 文件名
命令 >> 文件名
```

对于“>”而言,将重建(如果指定的文件不存在)或清空(如果指定的文件已存在)指定文件,把命令的输出结果重定向到指定的文件中。对于“>>”而言,仅追加内容到指定的文件中(如果指定的文件不存在,则创建文件,并存入命令输出结果;如果指定的文件存在,则在原文件尾部追加命令输出结果)。

例 5-14:

```
[root@localhost shellprog]# find ./ -name "*.sh" > findresult.txt
[root@localhost shellprog]# cat findresult.txt
./test15.sh
./test06.sh
./test09.sh
./case.sh
./test18.sh
./test19.sh
./test07.sh
./test12.sh
./test10.sh
./test17.sh
```

例 5-14 把查找结果存入文件而不是在屏幕上输出。

3) 基于文件描述符的重定向

Linux 系统中,启动的程序(称为进程)每打开一个文件,系统都会有一个唯一的整数与之对应,称为文件描述符。标准输入、标准输出和错误输出分别对应 0、1、2。在命令行中,重定向也可基于描述符进行操作。前面讲述的“>”“>>”“<”实际上等价于“1 >”“1 >>”“0 <”。因此,在 Bash 中还可进行如下重定向操作。

例 5-15：

```
[root@localhost shellprog]# ls a.txt
ls: cannot access a.txt: No such file or directory
[root@localhost shellprog]# ls a.txt 2>error.txt
[root@localhost shellprog]# cat error.txt
ls: cannot access a.txt: No such file or directory
```

例 5-15 中，通过“2>error.txt”把错误输出重定向到 error.txt 中。

例 5-16：

```
[root@localhost shellprog]# ls input.txt a.txt
ls: cannot access a.txt: No such file or directory
input.txt
[root@localhost shellprog]# ls input.txt a.txt >output1.txt
ls: cannot access a.txt: No such file or directory
[root@localhost shellprog]# cat output1.txt
input.txt
[root@localhost shellprog]# ls input.txt a.txt >output1.txt 2>error.txt
[root@localhost shellprog]# cat output1.txt
input.txt
[root@localhost shellprog]# cat error.txt
ls: cannot access a.txt: No such file or directory
[root@localhost shellprog]# ls input.txt a.txt >output2.txt 2>&1
[root@localhost shellprog]# cat output2.txt
ls: cannot access a.txt: No such file or directory
input.txt
```

例 5-16 中，“ls input.txt a.txt”命令行错误信息（本测试系统当前工作目录没有 a.txt）和标准输出信息均输出到屏幕。“ls input.txt a.txt >output1.txt”命令行把错误信息输出到屏幕，正常输出信息输出到 output1.txt 文件。“ls input.txt a.txt >output1.txt 2>error.txt”命令行把错误信息输出到 error.txt 文件，正常输出信息输出到 output1.txt 文件。“ls input.txt a.txt >output2.txt 2>&1”命令行错误信息和正常输出信息均输出到 output2.txt 文件，其中，“2>&1”表示文件描述符 2 使用文件描述符 1 的定向，由于文件描述符 1 已重定向到 output2.txt 文件，故两者均输出到同一文件。

4）/dev/null

Linux 提供了一个特殊的设备，向该设备写入任何信息都将如“泥牛入海，消于无形”。从该设备读入将得到一空串。因此，在一些情形下，可重定向到该设备以实现一些特殊的功能。

例 5-17：清空文件。

```
[root@localhost shellprog]# ls -l input.txt
-rw-r--r-- 1 root root 6 Feb 4 20:24 input.txt
[root@localhost shellprog]# cat /dev/null > input.txt
[root@localhost shellprog]# ls -l input.txt
-rw-r--r-- 1 root root 0 Feb 4 23:06 input.txt
```

9. 管道

管道由管道符号“|”分隔的多条命令组成。Bash 将管道符号前命令的标准输出连接到管道符号后面的标准输入。管道语法的基本格式如下。

命令 1|命令 2|…|命令 *n*

例 5-18：

```
root@localhost shellprog]# cat /etc/passwd|grep "stu*"|wc -l
9
```


例 5-18 命令实现了统计/etc/passwd 配置文件中有多少个以“stu”为前缀的用户名。

5.4.2 Shell 算术运算

算术运算包括数值运算、逻辑运算等。Bash 支持各种整数表达式的运算求值。浮点数运算求值需要 bc(Linux 支持的一款基本计算器，系统上没有 bc，可以使用包管理器下载安装)或者 awk 来实现。

Bash 中常见的运算符与 C 语言类似。

Linux Bash 算术运算有以下 4 种方式。

1. expr 命令

expr 是 Linux 中一个功能非常强大的命令，可用于四则运算和字符串操作。

例 5-19：

```
[root@localhost shellprog]# res=`expr 20 + 30`
[root@localhost shellprog]# echo $res
50
```

注意：操作数与运算符之间必须使用空格作为分隔符，由于 expr 命令中有的操作符在 Shell 中另有用途(如星号 * 等，需在前面加转义字符)，故使用该种方式进行运算时对输入格式要求很烦琐。

2. let 命令

基本语法格式：

```
let arg …
```

arg 是单独的算术表达式，与 C 语言中的表达式语法、优先级和结合性规则定义类似。除++、--和逗号外，所有的整型运算符都得到支持。变量也可以在算术表达式中直接使用名称访问，前面不需要带“$”符号。当表达式中有 Shell 的特殊字符时，必须用双引号括起来。如 let "v=6|5"，不用引号，则“|”被当作管道符号，而不是按位或操作。当使用 let 命令计算表达式的值时，若最后结果不为 0，则 let 命令的返回值为 0(表示“真”)；否则，返回值为 1(表示“假”)。这样，let 命令也可用于条件测试中。另外，let 命令中的操作符两侧不能有空格。

例 5-20：

```
[root@localhost shellprog]# let 2+3
[root@localhost shellprog]# echo $?
0
```

例 5-20 中，$? 为上次执行语句的返回值，2+3 表达式结果大于 0，故整个表达式语句的执行返回结果为 0(真)。

3. (())

对算术表达式而言，((算术表达式))可看作 let 命令的替换形式，遇特殊字符时，可不用双引号括起来。当需要取表达式的值时，可使用 $((算术表达式))获取表达式运算结果。

例 5-21：

```
let j=i*6+2 等价于((j=i*6+2)) 或 j=$((i*6+2))
[root@localhost shellprog]# ((i=1))
[root@localhost shellprog]# ((j=i*6+2))
[root@localhost shellprog]# echo $j
8
[[root@localhost shellprog]# j=$((i*6+2))
[root@localhost shellprog]# echo $j
8
[root@localhost shellprog]# ((j=2<3))
[root@localhost shellprog]# echo $j
1
```

4. 变量=$[ ]

Bash 支持$[ ]的方式把表达式的结果赋给变量，对方括号内的运算符也不需要做特殊处理。

例 5-22：

```
[root@localhost shellprog]# a=$[2|4]
[root@localhost shellprog]# echo $a
6
[root@localhost shellprog]# echo $[2*(3-1)]
4
```

5.4.3 条件测试命令

在 Bash 中，if、while、until 等语句中的控制条件是以命令的执行是否成功来判断的，无法直接接收条件表达式，但可通过如下几种命令测试方式处理条件表达式。

方式 1：

```
test expression
```

方式 2：

```
[ expression ]
```

方式 3：

```
[[ expression ]]
```

方式 4：

```
(( expression ))
```

使用时，应注意以下几点。

(1) 如果 test 语句中使用 Shell 变量，为避免造成歧义，最好用双引号将变量括起来。

(2) 方括号或圆括号与表达式之间、运算符前后以空格分隔。

(3) 在 Bash 条件测试表达式中，真值对应 0，假值对应 1，刚好与条件表达式的值相反，如$((2<3))为 0，test 2 < 3 的执行结果则为 1。

(4) 用于表达式分组的圆括号前后应有转义字符，即使用“\(”和“\)”。

(5) 为了与重定向功能区分，在方式 1 和方式 2 中，“<”和“>”符号前需添加转义字符。

test 或[ ]是 Bash 中最常用的测试命令，两者完全等价，它们可以和多种运算符一起使用。这些运算符从功能上可分为 4 类：数值测试、字符串比较测试、文件测试和逻辑运算。

1. 数值测试

表 5-3 为常用数值测试表达式形式及功能。

表 5-3　常用数值测试表达式比较功能描述

比较形式	功　能	比较形式	功　能
n1 -eq n2	测试整数 n1 是否等于 n2	n1 -le n2	测试整数 n1 是否小于或等于 n2
n1 -ne n2	测试整数 n1 是否不等于 n2	n1 -gt n2	测试整数 n1 是否大于 n2
n1 -lt n2	测试整数 n1 是否小于 n2	n1 -ge n2	测试整数 n1 是否大于或等于 n2

例 5-23：

```
[root@localhost ~]# [ 100 -gt 200 ]
[root@localhost ~]# echo $?
1
[root@localhost ~]# a=20
[root@localhost ~]# test "$a" -gt 10
[root@localhost ~]# echo $?
0
```

2. 字符串比较测试

表 5-4 为常用字符串测试表达式形式及功能，两个字符串的比较是基于 ASCII 码逐个比较的。另外，对于“>”和“<”比较符，为了和重定向功能区分，作为字符串比较符要做转义。

表 5-4　字符串测试表达式形式及功能描述

测试表达式形式	功　能
str	测试字符串 str 是否为空
str1 = str2	测试字符串 tr1 是否等于字符串 str2
str1 != str2	测试字符串 str1 是否不等于字符串 str2
-n str	测试字符串 str 的长度是否不为 0
-z str	测试字符串 str 的长度是否为 0
str1 < str2	测试字符串 str1 是否小于字符串 str2
str1 > str2	测试字符串 str1 是否大于字符串 str2

例 5-24：

```
[root@localhost ~]# [ text = TEXT ]
[root@localhost ~]# echo $?
1
[root@localhost ~]# test "this" \< "that"
[root@localhost ~]# echo $?
1
[root@localhost ~]# [ $PWD ]
[root@localhost ~]# echo $?
0
```

注：对[[ ]]方式而言，“=”可做字符串的模式匹配判断。即使用[[字符串=模式]]格式，左边字符串匹配右边模式则返回真。另外，如果一个字符串变量可能为空或未定义，最好使用-n 和-z 判断，而不是直接使用字符串变量。

3. 文件测试

Bash 中可对文件或目录的状态进行判断测试，常用的文件测试表达式形式及功能描述

如表 5-5 所示，表中的“文件名”为 Linux 系统中广义的文件名，文件类型可以是普通文件(f)、目录文件(d)、字符设备文件(c)、块设备文件(b)、管道文件(p)等。

表 5-5　文件测试表达式形式及功能描述

测试表达式形式	功　　能
-r 文件名	检查文件是否存在且对当前用户可读
-w 文件名	检查文件是否存在且对当前用户可写
-x 文件名	检查文件是否存在且对当前用户可执行
-f 文件名	检查文件是否存在且是一个普通文件
-e 文件名	检查文件是否存在
-d 文件名	检查文件是否存在且是目录
-p 文件名	检查文件是否存在且是命名管道文件
-b 文件名	检查文件是否存在且是块设备文件
-c 文件名	检查文件是否存在且是字符设备文件
-h 文件名	检查文件是否存在且是符号链接文件
-s 文件名	检查文件是否存在且非空
-O 文件名	检查文件是否存在且其属主为当前用户
-G 文件名	检查文件是否存在且属组与当前用户的初始组相同
-g 文件名	检查文件是否存在且设置了 SGID
-u 文件名	检查文件是否存在且设置了 SUID
-k 文件名	检查文件是否存在且设置了 SBIT
文件 1 -nt 文件 2	检查文件 1 是否比文件 2 新
文件 1 -ot 文件 2	检查文件 1 是否比文件 2 旧

例 5-25：

```
[root@localhost shellprog]# [ -d /etc ]
[root@localhost shellprog]# echo $?
0
[root@localhost shellprog]# [ -u /bin/passwd ]
[root@localhost shellprog]# echo $?
0
```

4. 逻辑运算

测试表达式可以通过逻辑运算符组合起来使用，Shell 中能使用的逻辑运算符和功能见表 5-6。

表 5-6　测试表达式中的逻辑运算符

逻辑运算符	功　　能
!	逻辑非，置于逻辑表达式之前，置反表达式真假值
-a	逻辑与，置于两个逻辑表达式之间，仅当两个逻辑表达式为真时，结果才为真
-o	逻辑或，置于两个逻辑表达式之间，其中任一个逻辑表达式为真，结果为真

注：逻辑运算符和被组合的表达式之间应有空格。[[]]和(())方式中的逻辑与、逻辑或使用符号 &&、‖。另外，也可以用 && 和 ‖ 对多个测试命令结果进行组合。

例 5-26：

```
[root@localhost shellprog]# [ ! 1 -le 2 ]
[root@localhost shellprog]# echo $?
```

```
1
[root@localhost shellprog]# [ -O test01.sh -a -w test01.sh ]
[root@localhost shellprog]# echo $?
1
[root@localhost shellprog]# s1=5
[root@localhost shellprog]# s2=11
[root@localhost shellprog]# test "$s1" -gt 0 -o "$s2" -lt 10
[root@localhost shellprog]# echo $?
0
[root@localhost shellprog]# test \("$s1" -gt 0 \) -a \( "$s2" -lt 10 \)
[root@localhost shellprog]# echo $?
1
[root@localhost shellprog]# [ 2 -gt 3 ] || [ -f test01.sh ]
[root@localhost shellprog]# echo $?
1
[root@localhost shellprog]# [ 2 -gt 3 ] && [ -f test01.sh ]
[root@localhost shellprog]# echo $?
1
```

5. 特殊条件测试

在条件和循环语句中还可使用以下三个特殊的条件测试命令。

:表示什么都不做,返回值为 0。

true 表示总为真,返回值为 0。

false 表示总为假,返回值为 1。

例 5-27:

```
[root@localhost shellprog]# true
[root@localhost shellprog]# echo $?
0
[root@localhost shellprog]# false
[root@localhost shellprog]# echo $?
1
[root@localhost shellprog]# :
[root@localhost shellprog]# echo $?
0
```

5.4.4 命令执行操作符

多条命令可以在一行中出现,它们可以顺序执行(命令间以“;”隔开),前面命令的执行结果不会影响后面命令的执行,最终返回结果为最后一条命令的执行返回结果。

同一行中的多条命令也可以通过逻辑与操作符“&&”、逻辑或操作符“||”连接在一起。与 C 之类的高级语言类似,与操作符“&&”连接的两条命令,若前一条执行不成功(返回值非 0),不会执行后一条;或操作符“||”连接的两条命令,若前一条执行成功(返回值 0),后一条不会执行。

另外,如果希望把多条命令的输出结果汇集在一起形成一个输出流,作为下一管道的输入,则可使用{}和()将命令组括起来达到该效果。

5.4.5 if 语句

和 C 语言中的 if 语句一样,Bash 中的 if 语句用于实现程序控制的条件分支选择,其基本语法形态有三种。

1. 形态一

```
if 命令
then
  命令表
fi
```

2. 形态二

```
if 命令
then
  命令表 1
else
  命令表 2
fi
```

3. 形态三

```
if 命令 1
then
  命令表 1
elif 命令 2
then
  命令表 2
…
else
  命令表 n
fi
```

注：在 if 语句中，用于判断的命令语句可分为普通命令语句和条件测试命令语句两种形式。对普通命令语句(也包括可执行的脚本程序)以命令的执行成功与否作为判断的条件。如命令执行成功，其返回值为 0，判断条件为真；如果命令执行不成功，其返回值不等于 0，判断条件为假。如果命令语句形式的判断条件由多条命令组成，那么判断条件只取决于最后一条命令的执行退出状态码。当然，多条命令退出状态也可进行 &&(与)、‖(或)和!(非)操作。条件测试命令语句的描述见 5.4.3 节。

形态三中的省略号表示可嵌套，else 语句和 elif 语句的区别在于 else 语句必须和一个 fi 语句配套，而多个嵌套的 elif 语句只需一个 fi 语句即可。

例 5-28：test03.sh 实现查询用户 zhangsan 是否存在系统中，存在则给出提示。

```
#!/bin/bash
if grep ^zhangsan /etc/passwd
then
echo "user zhangsan exists!"
fi
```

例 5-29：test04.sh 实现从键盘终端输入用户名，并判断该用户是否已在线登录，已登录在线则发一个问候信息，否则给出未登录提示。

```
#!/bin/bash
echo "type in the user name,please!"
read user
if who | grep $user
then
  echo "user $user has logged in!"
  echo "hello! $user!" | write $user
else echo "$user has not logged in the system!"
fi
```

例 5-30：test05.sh 实现从命令行输入用户名，查询输入的用户名是否存在，并给出相应的提示。

```
#!/bin/bash
if [ -n "$1" ]
then
 if cat /etc/passwd|grep ^$1
  then
    echo "User $1 exists!"
 else
   echo "User $1 not exists!"
 fi
else
   echo " No user name entered"
fi
```

test05.sh 测试执行过程及显示结果如下。

```
[root@localhost shellprog]# chmod a+x test05.sh
[root@localhost shellprog]# ./test05.sh
 No user name entered
[root@localhost shellprog]# ./test05.sh root
root:x:0:0:root:/root:/bin/bash
User root exists!
[root@localhost shellprog]# ./test04.sh zhangsan1
User zhangsan1 not exists!
```

例 5-31：test06.sh 实现从键盘上读取一字符，然后根据字符的值做进一步的判断。

```
#!/bin/bash
echo -n "Please input the answer:(y/Y/N/n)"
read answer
if [ $answer = y -o $answer = Y ]
then
echo "The answer is right."
elif [ $answer = n -o $answer = N ]
then
echo "The answer is wrong."
else
echo "Bad input!Please input the answer:(y/Y/N/n)"
fi
```

test06.sh 测试执行结果如下。

```
[root@localhost shellprog]# source test06.sh
Please input the answer:(y/Y/N/n)y
The answer is right.
[root@localhost shellprog]# source test06.sh
Please input the answer:(y/Y/N/n)f
Bad input!Please input the answer:(y/Y/N/n)
```

5.4.6 case 语句

和 C 语言等高级语言类似，Bash 中的 case 语句用于进行多重条件选择，其语法格式如下。

```
case 待测试的变量值或字符串 in
模式字符串 1[|模式字符串 2 ]…) 命令列表;;
模式字符串 1[|模式字符串 2 ]…) 命令列表;;
```

```
...
*) 默认命令列表;;
esac
```

注：

(1) 每个模式字符串后面可有一条或多条命令，最后一条命令必须以两个分号结束，模式字符串中可以使用通配符。

(2) 如果一个模式字符串中包含多个模式，那么各模式之间应以竖线(|)隔开，表示各模式是“或”的关系。

(3) 各模式字符串不重复出现。

(4) case 语句以关键字 case 开头，以关键字 esac 结束。

(5) case 的退出(返回)值是整个结构中最后执行的那个命令的退出值。若没有执行任何命令，则退出值为零。

(6) *)表示默认匹配，一般放在最后。

case 语句的执行过程为：用“变量值或字符串”去依次与各模式字符串比较。如果发现同某一个匹配，则执行该模式字符串之后的命令列表，直到遇到两个分号为止。如果没有任何模式字符串与之匹配，则执行默认命令列表(带 * 模式)或终止。

例 5-32：tes07.sh 实现了 test06.sh 的功能。

```
#!/bin/bash
echo -n "Please input the answer:(y/Y/N/n)"
read answer
case "$answer" in
y|Y) echo "The answer is right!";;
n|N) echo "The answer is wrong";;
*) echo "Bad Input!Please input the answer:(y/Y/N/n)";;
  esac
```

5.4.7 for 语句

LinuxBash 中，for 循环语句的基本格式如下。

```
for 变量名 in 列表值
do
  命令表
done
```

在执行命令列表之前，会先检查所有执行的列表值中指定的值是否未被使用，若未被使用，则给变量赋值并执行命令列表，直到列表中的值全部被使用后退出(即遍历列表中的值)。遍历完成后，变量的值为最后一次迭代的值。

列表值的指定有如下几种方式。

1. 直接列举方式

需遍历的值表在 in 后一一列举(值之间用空格隔开)。

例 5-33：test08.sh 实现输出 1～10 的平方值。

```
#!/bin/bash
 for p in 1 2 3 4 5 6 7 8 9 10
 do
  echo $(( p*p))
 done
```

2. 从变量中读取列表值

可将一系列的值存储在一个变量中，然后遍历整个列表。

例 5-34：test09.sh 实现从变量 citylist 依次遍历值。

```
#!/bin/bash
citylist = "Beijing Chengdu Hangzhou Shanghai Wuhang Shenzhen"
 for city in $citylist
 do
   echo let's go to $city!
 done
```

测试执行结果如下。

```
[root@localhost shellprog]# source test09.sh
let's go to Beijing!
let's go to Chengdu!
let's go to Hangzhou!
let's go to Shanghai!
let's go to Wuhan!
let's go to Shenzhen!
```

3. 从命令执行结果读取值

生成的列表值由执行命令得到的输出结果产生。

例 5-35：test10.sh 实现读取当前目录下目录项的功能。

```
#!/bin/bash
 for item in`ls`
 do
   echo $item
 done
 #end
```

4. 使用通配符读取目录下的文件列表

在文件名或路径中使用通配符。

例 5-36：在 test11.sh 中 for 命令会自动遍历当前目录下的.sh 文件，显示文件名及内容。

```
#!/bin/bash
for file in *.sh
do
  if test -f $file
  then
   echo "------------------------------------------------------"
   echo $file
   echo "------------------------------------------------------"
   cat $file
 fi
done
```

5. 当列表值为 $@时，简化 for 控制语句格式

```
for 变量
do
  命令列表
done
```

例 5-37：test12.sh 可列举显示命令行接收的参数值列表。

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

```
for item
do
echo $item
done
#end
```

tesh12.sh 测试执行结果如下。

```
[root@localhost shellprog]# bash -x ./test12.sh n1 n2 n3
+ for item in '"$@"'
+ echo n1
n1
+ for item in '"$@"'
+ echo n2
n2
+ for item in '"$@"'
+ echo n3
n3
```

C 语言风格的 for 命令

基本格式为：

```
for ((e1;e2;e3))
do
  命令表
done
```

其中，e1、e2、e3 是算术表达式，其执行过程与 C 语言中 for 语句中对应的表达式类似。整个 for 语句的返回值是命令表中最后一条命令执行后的状态值。

例 5-38：test13.sh 实现的功能和 test08.sh 相同。

```
#!/bin/bash
for ((p=1; p<10; p++))
do
 echo $(( p*p))
done
```

5.4.8 while 语句

while 语句的一般形式为

```
while 命令
do
  命令列表
done
```

与 C 语言中的 while 语句类似，如命令的执行结果返回值为真，则执行一遍命令列表中的命令并回到测试条件处继续执行命令；若为真则继续执行，如此循环往复，直到测试条件为假才退出循环。测试条件处的命令可以是普通命令，也可以是测试条件命令。

例 5-39：test14.sh 实现使用 while 语句计算 1～10 的平方。

```
#!/bin/bash
let int=1
while [ $int -le 7 ]
do
  let sq=$int*$int
  echo $sq
  let int=$int+1
```

```
done
```

例 5-40：test15.sh 实现读取文件 score.txt 内容并显示。

```
#/bin/bash
while read item
do
  echo $item
done < score.txt
```

假定 score.txt 的文件内容如下。

```
zhangsan 540
lisi 600
wangwu 700
```

test15.sh 执行情况如下。

```
[root@localhost shellprog]# source test15.sh
zhangsan 540
lisi 600
wangwu 700
```

注：在 Bash 的循环语句结束处(done 关键词后)可使用重定向符，对循环体中的输入/输出进行重定向。

5.4.9 until 语句

在 Linux Bash 中，until 语句的基本格式如下。

```
until 命令
do
  命令列表
done
```

until 语句与 while 语句很相似，只是当命令执行完成后的退出码(测试命令的测试结果)为假时继续执行循环体内的命令，直到为真时退出循环体。

例 5-41：test16.sh 用于计算 1+2+…+1000 的值。

```
#!/bin/bash
let i=1
let sum=0
until [ $i -gt 1000 ]
do
  let sum+=i
  let ++i
done
echo $sum
```

5.4.10 break 语句和 continue 语句

1. break 语句

该语句可以实现跳出循环体功能，若需跳出多层循环，可在 break 语句后给出要跳出的层数，其基本格式如下。

```
break [n]
```

2. continue 语句

该语句用于跳过循环体中在它之后的语句，回到本层循环的开始处，进行下一次循环。

在嵌套循环中，continue 后可跟一数字，用于指定内层循环体向外跳转的嵌套层数。其基本格式如下。

```
continue [n]
```

例 5-42：test17. sh 列举了 continue 和 break 的用法。该例首先产生或清空文件 scores. txt，并输入"Name:Course:score"行内容；紧接着根据交互情况输入成绩信息到文件 scores. txt 中；最后循环完成或输入'x'字符跳出循环，显示 scores. txt 中的内容。

```
#!/bin/bash
echo "Name:Course:score" > scores.txt
for x in zhangsan lisi wangwu
do
    for y in Chinese Mathmatics English
do
echo "Please input score to $x:$y:"
read score
echo "$x:$y:$score" >> scores.txt
echo "input 'n' to do next process"
echo "       's' to skip the other processes in current level"
echo "       'x' to terminate all jobs"
read ops
if [ $ops = n ]
then
echo "do next process"
      continue
elif [ $ops = s ]
then
    echo "skip the other processess in current level"
    continue 2
 elif [ $ops = x ]
      then
            echo "terminate all process"
            break 2
          fi
        done
done
cat scores.txt
```

测试运行该脚本程序结果如下。

```
[root@localhost shellprog]#source test17.sh
Please input score to zhangsan:Chinese:
89
input 'n' to do next process
       's' to skip the other processes in current level
       'x' to terminate all jobs
n
do next process
Please input score to zhangsan:Mathmatics:
96
input 'n' to do next process
       's' to skip the other processes in current level
       'x' to terminate all jobs
s
skip the other processess in current level
Please input score to lisi:Chinese:
89
```

```
input 'n' to do next process
      's' to skip the other processes in current level
      'x' to terminate all jobs
x
terminate all process
Name:Course:score
zhangsan:Chinese:89
zhangsan:Mathmatics:96
lisi:Chinese:89
```

5.4.11 exit 语句

在 Linux Bash 中，exit 语句的功能为退出正在执行的 Shell 脚本程序，并设定退出值。其语法格式如下。

```
exit [n]
```

n 是设定的退出值，若未给出，则脚本程序退出值为最后一条命令的执行状态。

5.4.12 函数

1. 函数定义

与其他绝大多数高级语言一样，Linux Bash 也提供了函数功能，函数定义的一般格式如下。

```
函数名() {
  命令列表
}
```

或

```
function 函数名() {
  命令列表
}
```

函数体中，可使用 break 语句终止函数的执行，也可使用 return 语句退出函数并返回指定的整数退出状态码(函数执行后的默认状态码为函数体执行路径中最后一条命令的执行完成状态码)。函数中也可使用 local 命令定义仅在函数和子函数中可见的局部变量。另外，export -f 可把定义的函数导出为环境变量，unset 命令可取消函数的定义。

2. 函数的使用

Bash 中函数的使用方法与外部命令一样，只要直接使用函数名即可。在使用函数时，一样可以传入参数。函数处理参数的方式与脚本文件处理命令行参数的方法是一样的。在函数中，$n 代表传入函数的第 n 个参数值。同时也可以使用 shift 命令来移动函数参数。

例 5-43：test18.sh 为求最小值功能的脚本程序。

```
#!/bin/bash
 getMin()
  {
   local min
    while [ $1 ]
     do
       if [ $min ]
        then
           if [ $1 -lt  $min ]
```

```
            then
                min = $1
            fi
        else
            min = $1
        fi
        shift
    done
    return $min
}
  getMin $@
  echo "The minimum value of $@ is : $?"
```

上述 Shell 脚本文件首先定义一个求最小值的函数 getMin，当执行该脚本文件时，根据脚本文件后的输入参数值调用 getMin 求得其中的最小值并予以显示。

注意：使用 return 返回最小值，此最小值不能大于退出码的最大值(255)，当然也可用共享用户变量的方式代替 return 返回最小值方式。

该脚本测试运行情况如下。

```
[root@localhost shellprog]# source test18.sh 34 56 12
The minimum value of 34 56 12 is :12
```

5.5 Shell 综合编程举例

5.5.1 批量添加用户

超级管理员可能需要批量添加用户账号，如批量添加 student1～student60，初始密码为 123456 的一批用户。例 5-44 实现了该功能，脚本程序启动运行时需要三个参数：用户名前缀、需添加的总数和初始密码。

例 5-44：test19.sh

```
let i=1;
if [ $# -le 3 ]
then
if [ `id -u` != 0 ]; then exit 1; fi
 tempfilestr="passwddate +%s"
while [ $i -le $2 ]
do
adduser "$1$i"
if [ $? ] ; then
echo "$1${i} is added successfully!"
 echo "$1${i}:$3" >>$tempfilestr
 else
  echo "$1$i exists!"
fi
 let i=$i+1
done
chpasswd<$tempfilestr
rm -rf $tempfuilestr
else
echo "Parameter input error!"
fi
cat /etc/passwd | grep $1
```

上述代码行 tempfilestr="passwd`date + %s`"也可使用 Linux 创建临时文件的特殊命令 mktemp 实现：

```
tempfilestr=`mktemp passwdXXXXXX`
```

该代码行会在当前目录下产生唯一的名字以 passwd 为前缀的临时文件(该文件仅对文件主有读写权限)，并把文件名字符串赋给变量 tempfilestr。

chpasswd 可用于设置指定用户的密码，可从规范的文本文件批量读入并更改。

test19.sh 测试执行情况如下。

```
[root@localhost shellprog]# source test19.sh student 5 123456
student1 is added successfully!
student2 is added successfully!
student3 is added successfully!
student4 is added successfully!
student5 is added successfully!
student1:x:1202:1206::/home/student1:/bin/bash
student2:x:1203:1207::/home/student2:/bin/bash
student3:x:1204:1208::/home/student3:/bin/bash
student4:x:1205:1209::/home/student4:/bin/bash
student5:x:1206:1210::/home/student5:/bin/bash
```

5.5.2 信号测试

例 5-45：test20.sh

```
#!/bin/bash
trap 'echo PROGRAM TERMINATE' SIGTERM
trap 'echo PROGRAM INTERRUPTED' INT
trap 'echo PROGRAM HUP' SIGHUP
trap 'echo PROGRAM QUIT' SIGQUIT
trap 'echo PROGRAM TSTP' SIGTSTP
while true
do
  echo "Program running."
# sleep 1
done
```

例 5-45 中，trap 用于设置信号处理程序，设置好后，当信号触发时可执行相应的功能。

5.5.3 启动脚本/etc/profile 分析

例 5-46：/etc/profile

```
pathmunge () {
    case ":${PATH}:" in
        *:"$1":*)
            ;;
        *) if [ "$2" = "after" ] ; then
                PATH=$PATH:$1
            else
                PATH=$1:$PATH
            fi
    esac
}
if [ -x /usr/bin/id ]; then
    if [ -z "$EUID" ]; then
```

```
        # ksh workaround
        EUID=`/usr/bin/id -u`
        UID=`/usr/bin/id -ru`
    fi
    USER="`/usr/bin/id -un`"
    LOGNAME=$USER
    MAIL="/var/spool/mail/$USER"
fi
# Path manipulation
if [ "$EUID" = "0" ]; then
    pathmunge /usr/sbin
    pathmunge /usr/local/sbin
else
    pathmunge /usr/local/sbin after
    pathmunge /usr/sbin after
fi
HOSTNAME=`/usr/bin/hostname 2>/dev/null`
if [ "$HISTCONTROL" = "ignorespace" ] ; then
    export HISTCONTROL=ignoreboth
else
    export HISTCONTROL=ignoredups
fi
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL
if [ $UID -gt 199 ] && [ "`/usr/bin/id -gn`" = "`/usr/bin/id -un`" ]; then
    umask 002
else
    umask 022
fi
for i in /etc/profile.d/*.sh /etc/profile.d/sh.local ; do
    if [ -r "$i" ]; then
        if [ "${-#*i}" != "$-" ]; then
            . "$i"
        else
            . "$i" >/dev/null
        fi
    fi
done
unset i
unset -f pathmunge
```

例 5-46 为 Linux 启动过程中执行的一个 Shell 脚本文件，下面详细描述该脚本文件实现的功能。

```
pathmunge () {
    case ":${PATH}:" in
        *:"$1":*)
            ;;
        *) if [ "$2" = "after" ] ; then
                PATH=$PATH:$1
            else
                PATH=$1:$PATH
            fi
    esac
}
```

上述代码定义了一个函数 pathmunge，该函数实现在系统环境变量 PATH 中添加搜索路径。

pathmunge 函数的调用格式为 pathmunge srchpath [after]。srchpath 为添加的具体

路径名，after为可选项，使用该选项表示添加在环境变量PATH尾部，否则添加在首部。pathmunge函数首先根据输入的要添加的搜索路径名（$1）判断是否已存在环境变量PATH中，如不存在，则根据是否有after选项（$2）添加路径到环境变量PATH的尾部或首部。

```
if [ -x /usr/bin/id ]; then
    if [ -z "$EUID" ]; then
        # ksh workaround
        EUID=`/usr/bin/id -u`
        UID=`/usr/bin/id -ru`
    fi
    USER="`/usr/bin/id -un`"
    LOGNAME=$USER
    MAIL="/var/spool/mail/$USER"
fi
```

紧接着，通过运行“id -u”和“id -ru”命令获取实际用户id和有效用户id（系统启动时这两者一般是相同的），并赋给变量EUID、UID、USER和LOGNAME，并设置登录用户的系统邮箱文件（存储在MAIL变量中）。

```
if [ "$EUID" = "0" ]; then
    pathmunge /usr/sbin
    pathmunge /usr/local/sbin
else
    pathmunge /usr/local/sbin after
    pathmunge /usr/sbin after
fi
HOSTNAME=`/usr/bin/hostname 2>/dev/null`
if [ "$HISTCONTROL" = "ignorespace" ] ; then
    export HISTCONTROL=ignoreboth
else
    export HISTCONTROL=ignoredups
fi
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL
```

以上代码行判断当前是否是以root用户（root用户标识为0）身份登录，从而决定需添加的命令搜索路径。通过运行hostname命令获取网络主机名并赋给变量HOSTNAME。最后通过export设定PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL为全局可访问环境变量。

```
if [ $UID -gt 199 ] && [ "`/usr/bin/id -gn`" = "`/usr/bin/id -un`" ]; then
    umask 002
else
    umask 022
fi
```

以上代码行通过UID的值范围不同设定不同权限掩码。

```
for i in /etc/profile.d/*.sh /etc/profile.d/sh.local ; do
    if [ -r "$i" ]; then
        if [ "${-#*i}" != "$-" ]; then
            . "$i"
        else
            . "$i" >/dev/null
        fi
    fi
done
```

以上代码行继续执行/etc/profile.d下的扩展名为.sh的脚本和/etc/profile.d下的sh.local脚本。

习　　题

一、填空题

1. Linux用户登录后，默认启动的Shell信息位于配置文件_____信息行的最后一个字段。

2. Linux Bash中修改命令提示符可通过更改环境变量_____的值实现，可执行文件路径位于环境变量_____中。

3. Linux Bash中命令补全功能，可通过按_____键实现。

4. Linux Bash中取消变量的定义使用的命令为_____。

5. 对Shell脚本进行调试可使用_____命令开启_____选项(也可启动Shell时使用该选项将Shell设置成跟踪模式)。

二、选择题

1. 在Bash中表示管道操作的符号是(　　)。

A. ‖　　B. |　　C. >>　　D. //

2. 在Linux Bash命令模式匹配中，代表任意一个字符串的通配符为(　　)。

A. *　　B. #　　C. @　　D. ?

3. 在Linux Bash脚本文件中，注释行以(　　)字符开始。

A. !　　B. #　　C. ?　　D. //

三、简答题

1. 在Linux Bash中，有哪几种启动运行脚本程序的方式？

2. 在Linux Bash中，单引号、双引号和倒引号的作用有什么区别？

3. 简述Linux Bash中管道和重定向的概念及用法。

四、Bash编程题

1. 编写Bash脚本程序，求1+2+…+1000，并把结果打印出来。

2. 试编写Bash脚本程序，批量删除用户user01～user100。

实验　批量用户添加

一、实验目的

1. 掌握脚本文件的建立和执行方法。

2. 掌握循环分支语句的用法。

二、实验内容

以超级管理员身份批量添加一批用户，用户名前缀、需要添加的用户起始编号、添加的总数和初始密码均从命令行中接收。

三、实验操作提示

1. 以 root 用户登录系统或使用 su 命令切换到 root 用户。

2. 用 vi 工具创建批量用户产生的脚本文件。该脚本文件的内容可参考本章 test19.sh (test19.sh 用户名后缀编码默认从 1 开始,本实验要求从命令行中接收)。

3. 使用 chmod 更改该脚本文件的执行属性(使用 source 命令执行脚本可省掉这一步)。

4. 执行脚本文件。

5. 通过查看/etc/passwd 文件确认操作生效与否。

6. 可用一个用户做登录测试,确认密码和用户是否生效。