Bourne Shell 及 shell 编程

作者：**何斌武**，hbwork@dlut.edu.cn,大连理工大学网络中心，April 1999.

URL: ftp://ftp.dlut.edu.cn/pub/PEOPLE/albin/

源码:

Bourne Shell

介绍：Bourne Shell 基础及其他很多有用的特性，shell 编程及组织。

主要内容：

.shell 基础　　基本介绍，环境，选项，特殊字符

.shell 变量　　用户定义变量，环境变量，位置变量(shell 参数)

.shell script 编程

　　条件测试，循环及重复控制

.shell 定制

1.shell 基础知识

　作者：Stephen Bourne 在 Bell 实验室开发

　建议：man sh　查看相关 UNIX 上的改进或特性

(1)shell 提示符及其环境

　/etc/passwd 文件

　提示符：$

　/etc/profile $HOME/.profile

(2)shell 执行选项

　-n　测试 shell script 语法结构，只读取 shell script 但不执行

　-x　进入跟踪方式，显示所执行的每一条命令，用于调度

　-a　Tag all variables for export

　-c "string"　从 strings 中读取命令

　-e　非交互方式

　-f　关闭 shell 文件名产生功能

-h locate and remember functions as defind
-i 交互方式
-k 从环境变量中读取命令的参数
-r 限制方式
-s 从标准输入读取命令
-t 执行命令后退出(shell exits)
-u 在替换中如使用未定义变量为错误
-v verbose,显示 shell 输入行

这些选项可以联合使用，但有些显然相互冲突，如-e 和-i.

(3)受限制 shell(Restircted Shell)
sh -r 或 /bin/rsh

不能执行如下操作：cd, 更改 PATH,指定全路径名，输出重定向，因此可以提供一个较
好的控制和安全机制。通常 rsh 用于应用型用户及拨号用户，这些用户通常是看不到提
示符的。通常受限制用户的主目录是不可写的。

不足：如果用户可以调用 sh,则 rsh 的限制将不在起作用，事实上如果用户在 vi 及 more
程序中调用 shell,而这时 rsh 的限制将不再起作用。

(4)用 set 改变 shell 选项
用户可以在$提示符下用 set 命令来设置或取消 shell 的选项。使用-设置选项，+取消相应
选项，大多数 UNIX 系统允许 a,e,f,h,k,n,u,v 和 x 的开关设置/取消。

set -xv
启动跟踪方式;显示所有的命令及替换，同样显示输入。
set -tu
关闭在替换时对未定义变量的检查。

使用 echo $-显示所有已设置的 shell 选项。

(5)用户启动文件 .profile
PATH=$PATH:/usr/loacl/bin; export PATH

(6)shell 环境变量
CDPATH 用于 cd 命令的查找路径
HOME /etc/passwd 文件中列出的用户主目录
IFS Internal Field Separator,默认为空格，tab 及换行符

MAIL　　/var/mail/$USERNAME　　mail 等程序使用
PATH
PS1，PS2　　默认提示符($)及换行提示符(>)
TERM　　终端类型，常用的有 vt100,ansi,vt200,xterm 等

示例：$PS1="test:";export PS1
test: PS1="\$";export PS1
$echo $MAIL
/var/mail/username

(7)保留字符及其含义

$　　shell 变量名的开始，如$var
|　　管道，将标准输出转到下一个命令的标准输入
#　　注释开始
&　　在后台执行一个进程
？　　匹配一个字符
*　　匹配 0 到多个字符(与 DOS 不同，可在文件名中间使用，并且含.)
$-　　使用 set 及执行时传递给 shell 的标志位
$!　　最后一个子进程的进程号
$#　　传递给 shell script 的参数个数
$*　　传递给 shell script 的参数
$@　　所有参数，个别的用双引号括起来
$?　　上一个命令的返回代码
$0　　当前 shell 的名字
$n　　(n:1-) 位置参数
$$　　进程标识号(Process Identifier Number, PID)
>file　　输出重定向
<file　　输入重定向
`command`　　命令替换，如　　filename=`basename /usr/local/bin/tcsh`
>>fiile　　输出重定向，append

转义符及单引号：

$echo "$HOME $PATH"
/home/hbwork /opt/kde/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:
$echo '$HOME $PATH'
$HOME $PATH
$echo \$HOME $PATH
$HOME /opt/kde/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/hbwork/bin

其他：

$dir=ls

```
$$dir
$alias dir ls
$dir

ls > filelist
ls >> filelist
wc -l < filelist
wc -l filelist
sleep 5; echo 5 seconds reaches; ls -l
ps ax |egrep inetd
find / -name core -exec rm {} \; &
filename=`date "+%Y%m%d"`.log
```

2. shell 变量

变量：代表某些值的符号，如$HOME,cd 命令查找$HOME,在计算机语言中可以使用变量可以

进行多种运算和控制。

Bourne Shell 有如下四种变量：

.用户自定义变量

.位置变量即 shell script 之参数

.预定义变量（特殊变量）

.环境变量(参考 shell 定制部分)

(1)用户自定义变量（数据的存储）

```
$ COUNT=1
$ NAME="He Binwu"
```

技巧：因为大部分 UNIX 命令使用小写字符，因此在 shell 编程中通常使用全大写变量，

当然这并不是强制性的，但使用大写字符可以在编程中方便地识别变量。

变量的调用：在变量前加$

```
$ echo $HOME
/home/hbwork
$ WEEK=Satur
$ echo Today is $WEEKday
Today is
$echo Today is ${WEEK}day
Today is Saturday
```

Shell 变量赋值从右从左进行(Linux Shell/bash 从左向右赋值!)

```
$ X=$Y Y=y
$ echo $X
```

```
y
$ Z=z Y=$Z
$ echo $Y

$
```

使用 unset 命令删除变量的赋值

```
$ Z=hello
$ echo $Z
hello
$ unset Z
$ echo $Z

$
```

有条件的命令替换

在 Bourne Shell 中可以使变量替换在特定条件下执行，即有条件的环境变量替换。

这种变量替换总是用大括号括起来的。

.设置变量的默认值

在变量未赋值之前其值为空。Bourne Shell 允许对变量设置默认值，其格式如下：

${variable:-defaultvalue}

例：

```
$ echo Hello $UNAME
Hello
$ echo Hello ${UNAME:-there}
Hello there
$ echo $UNAME     #变量值并未发生变化

$ UNAME=hbwork
$ echo Hello ${UNAME:-there}
Hello hbwork
$
```

.另一种情况：改变变量的值，格式如下：

${variable:=value}

例：

```
$ echo Hello $UNAME
Hello
$ echo Hello ${UNAME:=there}
Hello there
```

```
$ echo $UNAME     #变量值并未发生变化
there
$
```

.变量替换中使用命令替换

```
$USERDIR=${$MYDIR:-`pwd`}
```

.在变量已赋值时进行替换 ${variable:+value}

.带有错误检查的有条件变量替换

```
${variable:?value}
例：
$ UNAME=
$ echo ${UNAME:?"UNAME has not been set"}
UNAME: UNAME has not been set
$ echo ${UNAME:?}
UNAME: parameter null or not set
```

(2)位置变量(Shell 参数)

在 shell script 中位置参数可用$1..$9 表示，$0 表示内容通常为当前执行程序的文件名。

.防止变量值被替换 readonly variable

.使用 export 命令输出变量，使得变量对子 shell 可用，当 shell 执行一下程序时，shell

将为其设置一个新的环境让其执行，这称之了 subshell. 在 Bourne Shell 中变量通常

被认为是本地变量，也就是说在对其赋值之外的 shell 环境之外是不认识此变量的。使

用 export 对 subshell 可用。

例：对变量 PS1 的 export 操作，shell 的提示符将发生变化。

```
$ PS1=`hostname`$
peony$sh
$ echo $PS1
$  <-输出结果
$ exit
peony$export PS1
peony$sh
peony$ echo $PS1
peony$  <-输出结果
peony$
```

3.Shell Script 编程

目的：使用 UNIX 所提供的最常用工具来完成所需复杂任务的强大功能。

(1)最简单的 Shell 编程

```
$ls -R / |grep myname |more
```

每天数据的备份：

```
$ cd /usr/yourname; ls * |cpio -o > /dev/rmt/0h
```

书写程序的目的是一次编程，多次使用（执行）！

```
$ cat > backup.sh
cd /home/hbwork
ls * | cpio -o > /dev/rmt/0h
^D
```

执行：

```
$ sh backup.sh
```

或：

```
$ chmod +x backup.sh
$ ./backup.sh
```

技巧：在 shell script 中加入必要的注释，以便以后阅读及维护。

(2)shell 是一个（编程）语言

同传统的编程语言一样，shell 提供了很多特性，这些特性可以使你的 shell script 编程更为有用，如：数据变量、参数传递、判断、流程控制、数据输入和输出，子程序及以中断处理等。

. 在 shell 编程中使用数据变量可以将程序变量更为通用，如在上面 backup.sh 中：

```
cd $WORKDIR
ls * | cpio -o > /dev/rmt/0h
```

. Shell 编程中的注释以#开头

. 对 shell 变量进行数字运算，使用 expr 命令

```
expr integer operator integer
```

其中 operator 为+ - * / %, 但对*的使用要用转义符\,如：

```
$expr 4 \* 5
20
$int=`expr 5 + 7`
$echo $int
12
```

(3)Shell 编程的参数传递，可通过命令行参数以及交互式输入变量(read)

restoreall.sh 对 backup.sh 程序的备份磁带进行恢复

```
$cat > restoreall.sh
cd $WORKDIR
cpio -i < /dev/rmt/0h
^D
```

restore1.sh:只能恢复一个文件

```
#restore1 --program to restore a single file
cd $WORKDIR
cpio -i $i < /dev/rmt/0h

$restore1 file1
```

恢复多个文件 restoreany ：

```
#restoreany --program to restore a single file
cd $WORKDIR
cpio -i $* < /dev/rmt/0h

$ restoreany file1 file2 file3
```

(4)条件判断

. if-then 语句,格式如下:

```
if command_1
then
        command_2
        command_3
fi
command_4
```

在 if-then 语句中使用了命令返回码$?,即当 command_1 执行成功时才执行 command_2 和 command_3,而 command_4 总是执行.

示例程序 unload: 在备份成功时删除原始文件,带有错误检查

```
cd $1
#备份时未考虑不成功的情况!
ls -a | cpio -o > /dev/rmt/0h
rm -rf *
```

改进如下:

#当使用了管道命令时,管理命令的返回代码为最后一个命令的返回代码
if ls -a | cpio -o > /dev/rmt/0h
then
rm -rf *
fi

. if-then-else 语句

```
if command_1
then
        command_2
else
        command_3
fi
```

技巧: 由于 shell 对命令中的多余的空格不作任何处理,一个好的程序员会用这一特性
对自己的程序采用统一的缩进格式,以增强自己程序的可读性.

. 使用 test 命令进行进行条件测试
格式: test conditions

test 在以下四种情况下使用: a. 字符比较 b.两个整数值的比较
c. 文件操作,如文件是否存在及文件的状态等
d. 逻辑操作,可以进行 and/or,与其他条件联合使用

a. 测试字符数据: shell 变量通常民政部下均作为字符变量

str1 = str2	二者相长,相同
str1 != str2	不同
-n string	string 不为空(长度不为零)
-z string	string 为空
string	string 不为空

例:

```
$ str1=abcd        #在含有空格时必须用引号括起来
$ test $str1=abcd
$ echo $?
0
$ str1="abcd "
$ test $str1=abcd
$ echo $?
1
```

Note: 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.

例:

```
$ str1="       "
$ test $str1
$ echo $?
1
$ test "$str1"
$ echo $?
0
$ test -n $str1
test: argument expected
$ test -n "$str1"
$ echo $?
0
$
```

b. 整数测试: test 与 expr 相同,可以将字符型变量转换为整数进行操作,expr 进行
整数的算术运算,而 test 则进行逻辑运算.

表达式	说明
int1 -eq int2	相等?
int1 -ne int2	不等?
int1 -gt int2	int1 > int2 ?
int1 -ge int2	int1 >= int2 ?
int1 -lt int2	int1 < int2 ?
int1 -le int2	int1 <= int2 ?

例:

```
$ int1=1234
$ int2=01234
$ test $int1 -eq $int2
$ echo $?
0
```

c. 文件测试:检查文件状态如存在及读写权限等

-r filename	用户对文件 filename 有读权限?

-w filename 用户对文件 filename 有写权限?
-x filename 用户对文件 filename 有可执行权限?
-f filename 文件 filename 为普通文件?
-d filename 文件 filename 为目录?
-c filename 文件 filename 为字符设备文件?
-b filename 文件 filename 为块设备文件?
-s filename 文件 filename 大小不为零?
-t fnumb 与文件描述符 fnumb(默认值为 1)相关的设备是一个终端设备?

d. 测试条件之否定,使用!
例:

```
$ cat /dev/null > empty
$ test -r empty
$ echo $?
0
$ test -s empty
1
$ test ! -s empty
$ echo $?
0
```

e. 测试条件之逻辑运算
-a And
-o Or

例: $ test -r empty -a -s empty
$ echo $?
1

f. 进行 test 测试的标准方法
因为 test 命令在 shell 编程中占有很重要的地位,为了使 shell 能同其他编程语言一样
便于阅读和组织, Bourne Shell 在使用 test 测试时使用了另一种方法:用方括号将整个
test 测试括起来:

```
$ int1=4
$ [ $int1 -gt 2 ]
$ echo $?
0
```

例: 重写 unload 程序,使用 test 测试

```
#!/bin/sh
```

```
#unload - program to backup and remove files
#syntax: unload directory

#check arguments
if [ $# -ne 1 ]
then
        echo "usage: $0 directory"
        exit 1
fi

#check for valid directory name
if [ ! -d "$1" ]
then
        echo "$1 is not a directory"
        exit 2
fi

cd $1

ls -a | cpio -o > /dev/rmt/0h

if [ $? -eq 0 ]
then
        rm -rf *
else
        echo "A problem has occured in creating backup"
        echo "The directory will not be ereased"
        echo "Please check the backup device"
        exit 3
fi
# end of unload
```

在如上示例中出现了 exit, exit 有两个作用:一是停止程序中其他命令的执行,二
是
设置程序的退出状态

g. if 嵌套及 elif 结构

```
if command
then
        command
else
        if command
        then
```

```
                command
        else
                if  command
                then
                        command
                fi

        fi
fi
```

改进:使用 elif 结构

```
if  command
then
        command
elif    command
then
        command
elif    command
then
        command
fi
```

elif 结构同 if 结构类似,但结构更清淅,其执行结果完全相同.

/********抱歉，为了格式不乱，我用代码模式粘贴全文*********/

源码:

Bourne Shell 及 Shell 编程(2)

h.交互式从键盘读入数据

使用 read 语句，其格式如下：

```
read  var1  var2  ...  varn
```

read 将不作变量替换，但会删除多余的空格，直到遇到第一个换行符（回车），
并将输入值依次赋值给相应的变量。

例：

```
$ read  var1  var2  var3
Hello     my    friends
```

```
$ echo $var1 $var2 $var3
Hello my friends
$ echo $var1
Hello
$ read var1 var2 var3
Hello my dear friends
$ echo $var3
dear friends    <-输入多余变量时，输入值余下的内容赋给最后一个变量
$ read var1 var2 var3
Hello friends
$ echo $var3
                <- var3 为空
$
```

在 shell script 中可使用 read 语句进行交互操作：

```
...
#echo -n message 输出结果后不换行
echo -n "Do you want to continue: Y or N"
read ANSWER

if [ $ANSWER=N -o $ANSWER=n ]
then
        exit
fi
```

i. case 结构：结构较 elif-then 结构更清楚

比较 if-then 语句：

```
if [ variable1 = value1 ]
then
    command
    command
elif [ variable1 = value2 ]
then
    command
    command
elif [ variable1 = value3 ]
then
    command
    command
fi
```

相应的 case 结构：

```
case value in
    pattern1)
        command
        command;;
    pattern2)
        command
        command;;
    ...
    patternn)
        command;
 esac
```

* case 语句只执行第一个匹配模式

例：使用 case 语句建立一个菜单选择 shell script

```
#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
```

在上例中，*指默认匹配动作。此外，case 模式中也可以使用逻辑操作，如下所示
：

```
 pattern1 | pattern2 )    command
                              command ;;
```

这样可以将上面示例程序中允许用户输入数字或每一个大写字母。

```
case "$CHOICE" in
        1|R)    echo "Restore";;
        2|B)    echo "Backup";;
        3|U)    echo "Unload";;
        *)      echo "Sorry $CHOICE is not a valid choice

                exit 1
esac
```

(5)循环控制

<1> while 循环:

格式：

```
while   command
do
        command
        command
        command
        ...
done
```

例： 计算 1 到 5 的平方

```
#!/bin/sh
#
#Filename: square.sh
int=1

while [ $int -le 5 ]
do
        sq=`expr $int \* $int`
        echo $sq
        int=`expr $int + 1`
done
echo "Job completed"

$ sh square.sh
1
4
9
16
25
```

```
Job  completed
```

<2> until 循环结构：

格式：

```
until  command
do
        command
        command
        ....
        command
done
```

示例：使用 until 结构计算 1-5 的平方

```
#!/bin/sh

int=1

until  [  $int  -gt  5  ]
do
        sq=`expr  $int  \*  $int`
        echo  $sq
        int=`expr  $int  +  1`
done
echo  "Job  completed"
```

<3> 使用 shift 对不定长的参数进行处理

在以上的示例中我们总是假设命令行参数唯一或其个数固定，或者使用$*将整个命令

行参数传递给 shell script 进行处理。对于参数个数不固定并且希望对每个命令参数

进行单独处理时则需要 shift 命令。使用 shift 可以将命令行位置参数依次移动位置，

即$2->$1，$3->$2. 在移位之前的第一个位置参数$1 在移位后将不在存在。

示例如下：

```
#!/bin/sh
#
#Filename:  shifter
```

```
until [ $# -eq 0 ]
do
    echo "Argument is $1 and `expr $# - 1` argument(s) re
main"
    shift
done


$ shifter 1 2 3 4
Argument is 1 and 3 argument(s) remain
Argument is 2 and 2 argument(s) remain
Argument is 3 and 1 argument(s) remain
Argument is 4 and 0 argument(s) remain
$
```

使用 shift 时，每进行一次移位，$#减 1，使用这一特性可以用 u
ntil 循环对每个参
数进行处理，如下示例中是一个求整数和的 shell script:

```
#!/bin/sh
# sumints - a program to sum a series of integers
#

if [ $# -eq 0 ]
then
    echo "Usage: sumints integer list"
    exit 1
fi

sum=0

until [ $# -eq 0 ]
do
    sum=`expr $sum + $1`
    shift
done
echo $sum


$ sh sumints 324 34 34 12 34
438
$
```

使用 shift 的另一个原因是 Bourne Shell 的位置参数变量为$1~$9, 因此通过位置变量

只能访问前 9 个参数。但这并不等于在命令行上最多只能输入 9 个参数。此时如果想访问

前 9 个参数之后的参数，就必须使用 shift.

另外 shift 后可加整数进行一次多个移位，如：

```
shift 3
```

<4>. for 循环

格式：

```
for var in arg1 arg2 ... argn
do
        command
        ....
        command
done
```

示例：

```
$ for letter in a b c d e; do echo $letter;done
a
b
c
d
e
```

对当前目录下的所有文件操作：

```
$ for i in *
  do
        if [ -f $i ]
        then
                echo "$i is a file"
        elif    [ -d $i ]
                echo "$i is a directory"
        fi
  done
```

求命令行上所有整数之和：

```
#!/bin/sh
```

```
sum=0

for INT in $*
do
        sum=`expr $sum + $INT`
done

echo $sum
```

<6> 从循环中退出： break 和 continue 命令

break	立即退出循环
continue	忽略本循环中的其他命令，继续下一下循环

在 shell 编程中有时我们要用到进行无限循环的技巧，也就是说这种循环一直执行
碰
到 break 或 continue 命令。这种无限循环通常是使用 true 或 false 命令开始的。UNIX
系统中的 true 总是返加 0 值，而 false 则返回非零值。如下所示：

```
#一直执行到程序执行了 break 或用户强行中断时才结束循环
while true
do
        command
        ....
        command
done
```

上面所示的循环也可以使用 until false，如下：

```
until false
do
        command
        ....
        command
done
```

在如下 shell script 中同时使用了 continue,break 以及 case 语句中的正规表达式用
法：

```
#!/bin/sh
```

```
# Interactive program to restore, backup, or unload
# a directory

echo "Welcome to the menu driven Archive program"

while true
do
# Display a Menu
    echo
    echo "Make a Choice from the Menu below"
    echo _
    echo "1   Restore Archive"
    echo "2   Backup directory"
    echo "3   Unload directory"
    echo "4   Quit"
    echo

# Read the user's selection
    echo -n "Enter Choice: "

    read CHOICE

    case $CHOICE in
        [1-3] ) echo
                # Read and validate the name of the directo
ry

                echo -n "What directory do you want? "
                read WORKDIR

                if [ ! -d "$WORKDIR" ]
                then
                    echo "Sorry, $WORKDIR is not a director
y"
                    continue
                fi

                # Make the directory the current working dir
ectory
                cd $WORKDIR;;

            4) :;;      # :为空语句，不执行任何动作
            *) echo "Sorry, $CHOICE is not a valid choice"
```

```
                continue
        esac

        case "$CHOICE" in
            1) echo "Restoring..."
               cpio -i </dev/rmt/0h;;

            2) echo "Archiving..."
               ls | cpio -o >/dev/rmt/0h;;

            3) echo "Unloading..."
               ls | cpio -o >/dev/rmt/0h;;

            4) echo "Quitting"
               break;;
        esac

    #Check for cpio errors

        if [ $? -ne 0 ]
        then
            echo "A problem has occurred during the process"
            if [ $CHOICE = 3 ]
            then
                echo "The directory will not be erased"
            fi

            echo "Please check the device and try again"
            continue
        else
            if [ $CHOICE = 3 ]
            then
                rm *
            fi
        fi
    done
```

(6)结构化编程：定义函数

同其他高级语言一样，shell 也提供了函数功能。函数通常也称之为子过程(subroutine)

,

其定义格式如下：

```
funcname()
{
        command
        ...
        command;   #分号
}
```

定义函数之后，可以在 shell 中对此函数进行调用，使用函数定义可以将一个复杂的程序
分
为多个可管理的程序段，如下所示：

```
# start program

setup ()
{  command list ; }

do_data ()
{  command list ; }

cleanup ()
{  command list ; }

errors ()
{  command list ; }

setup
do_data
cleanup
# end program
```

技巧：

. 在对函数命名时最好能使用有含义的名字，即函数名能够比较准确的描述函数所
完成
的任务。

. 为了程序的维护方便，请尽可能使用注释

使用函数的另一个好处就是可以在一个程序中的不同地方执行相同的命令序列(函数),
如下所示：

```
iscontinue()
```

```
{
    while true
    do
            echo -n "Continue?(Y/N)"
            read ANSWER

            case $ANSWER in
                [Yy])    return 0;;
                [Nn])    return 1;;
                *) echo "Answer Y or N";;
            esac
    done
}
```

这样可以在 shell 编程中调用 iscontinue 确定是否继续执行：

```
if iscontinue
then
    continue
else
    break
fi
```

** shell 函数与 shell 程序非常相似，但二者有一个非常重要的差别：shell 程序是由子 shell 执行的，而 shell 函数则是作为当前 shell 的一部分被执行的，因此在当前 shell 中可以改变函数的定义。此外在任意 shell(包括交互式的 shell)中均可定义函数，如：

```
$ dir
dir: not found
$ dir () { ls -l ;}
$ dir
total 5875
-rw-r--r--    1 hbwork          100 Nov 10 23:16 doc
-rw-r--r--    1 hbwork      2973806 Nov 10 23:47 ns40docs.zip
-rw-r--r--    1 hbwork      1715011 Nov 10 23:30 ns840usr.pdf
-rw-r--r--    1 hbwork      1273409 Sep 23  1998 radsol21b6.tar.Z
-rw-r--r--    1 hbwork         7526 Nov 10 23:47 wget-log
```

```
-rw-r--r--    1 hbwork          1748 Nov 13 21:51 wget-log.1
$ unset dir
$ dir () {
> echo "Permission   Link Owner Group   File_SZ    LastAccess
FileName"
> echo "-------------------------------------------------------"
> ls -l $*;
> }

    $ dir
    Permission   Link Owner Group   File_SZ    LastAccess FileN
ame
    -------------------------------------------------------
    total 5875
    -rw-r--r--    1 hbwork           100 Nov 10 23:16 doc
    -rw-r--r--    1 hbwork       2973806 Nov 10 23:47 ns40docs.zip

    -rw-r--r--    1 hbwork       1715011 Nov 10 23:30 ns840usr.pdf

    -rw-r--r--    1 hbwork       1273409 Sep 23  1998 radsol21b6.t
ar.Z
    -rw-r--r--    1 hbwork          7526 Nov 10 23:47 wget-log
    -rw-r--r--    1 hbwork          1748 Nov 13 21:51 wget-log.1
```

通常情况下，shell script 是在子 shell 中执行的，困此在此子 shell 中对变量所作的

修改对父 shell 不起作用。点(.) 命令使用 shell 在不创建子 shell 而由当前 shell 读取

并执行一个 shell script, 可以通过这种方式来定义函数及变量。此外点(.)命令最

常用的功能就是通过读取.profile 来重新配置初始化 login 变量。如下所示：

```
$ . .profile
```

(csh 相对于.命令的是 source 命令).

(7)使用 And/Or 结构进行有条件的命令执行

<1> And ，仅当第一个命令成功时才有执行后一个命令,如同在逻辑与表达式中如果前面的

结果为真时才有必要继续运算，否则结果肯定为假。

格式如下：

```
command1 && command2
```

例：rm $TEMPDIR/* && echo "File successfully removed"

等价于

```
if rm $TEMPDIR/*
then
        echo "File successfully removed"
fi
```

<2>Or, 与 AND 相反，仅当前一个命令执行出错时才执行后一条命令

例： rm $TEMPDIR/* || echo "File not removed"

等价与：

```
if rm $TEMPDIR/*
then
        command
else
        echo "File not removed"
fi
```

<3> 混合命令条件执行

command1 && command2 && command3

当 command1, command2 成功时才执行 command3

command1 && command2 || comamnd3

仅当 command1 成功，command2 失败时才执行 command3

当然可以根据自己的需要进行多种条件命令的组合，在此不多讲述。

(8) 使用 getopts 命令读取 unix 格式选项

UNIX 格式选项指如下格式的命令行参数：

command -options parameters

使用格式：

getopts option_string variable

具体使用方法请参考 getopts 的在线文档(man getopts).

示例如下：

```
#newdate
if [ $# -lt 1 ]
then
      date
else
    while getopts mdyDHMSTjJwahr OPTION
    do
        case $OPTION
        in
            m) date '+%m ';;   # Month of Year
            d) date '+%d ';;   # Day of Month
            y) date '+%y ';;   # Year
            D) date '+%D ';;   # MM/DD/YY
            H) date '+%H ';;   # Hour
            M) date '+%M ';;   # Minute
            S) date '+%S ';;   # Second
            T) date '+%T ';;   # HH:MM:SS
            j) date '+%j ';;   # day of year
            J) date '+%y%j ';;# 5 digit Julian date
            w) date '+%w ';;   # Day of the Week
            a) date '+%a ';;   # Day abbreviation
            h) date '+%h ';;   # Month abbreviation
            r) date '+%r ';;   # AM-PM time
            \?) echo "Invalid option $OPTION";;
        esac
    done
fi

$ newdate -J
94031
$ newdate -a -h -d
Mon
Jan
31
$ newdate -ahd
Mon
Jan
31
$
```

示例程序：复制程序

```
# Syntax: duplicate [-c integer] [-v] filename
```

```
#       where integer is the number of duplicate copies
#       and -v is the verbose option

COPIES=1
VERBOSE=N


while getopts vc: OPTION
do
    case $OPTION
    in
        c) COPIES=$OPTARG;;
        v) VERBOSE=Y;;
        \?) echo "Illegal Option"
              exit 1;;
    esac
done

if [ $OPTIND -gt $# ]
then
    echo "No file name specified"
    exit 2
fi


shift `expr $OPTIND -1`

FILE=$1
COPY=0

while [ $COPIES -gt $COPY ]
do
    COPY=`expr $COPY + 1`
    cp $FILE ${FILE}${COPY}
    if [ VERBOSE = Y ]
    then
        echo ${FILE}${COPY}
    fi
done


$ duplicate -v fileA
fileA1
$ duplicate -c 3 -v fileB
```

fileB1
fileB2
fileB3

4. Shell 的定制

通常使用 shell 的定制来控制用户自己的环境，比如改变 shell 的外观(提示符)以及增强
自己的命令。

(1)通常环境变量来定制 shell

通常改变环境变量可以定制 shell 的工作环境。shell 在处理信息时会参考这些环境变量
，改变环境变量的值在一定程度上改变 shell 的操作方式，比如改变命令行提示符。

.使用 IFS 增加命令行分隔符

默认状态下 shell 的分隔符为空格、制表符及换行符，但可以通过改变 IFS 的值加入自己
的分隔符。如下所示：

```
$ IFS=":"
$ echo:Hello:my:Friend
Hello my Friend
```

(2)加入自己的命令及函数

如下程序：

```
#Directory and Prompt change program
#Syntax: chdir directory

if [ ! -d "$1" ]
then
        echo "$1 is not a directory"
        exit 1
fi

cd $1
PS1=`pwd`$
export PS1

$ chdir /usr/home/teresa
$
```

但此程序在执行时系统提示符并不会改变，因为此程序是在子 shell 中执行的。因此其变
量
对当前 shell 并无影响，要想对当前 shell 起作用，最好是将此作为函数写在自己的.profile
中
或建立自己的个人函数文件.persfuncs

```
#Personal function file persfuncs

chdir()
{
#Directory and Prompt change program
#Syntax: chdir directory
if [ ! -d "$1" ]
then
        echo "$1 is not a directory"
        exit 1
fi

cd $1
PS1=`pwd`$
export PS1;
}
```

再执行：

```
$ . .persfuncs
$ chdir temp
/home/hbbwork/temp$
```

也可在自己的.profile 文件中用 . .persfuncs 调用.persfuncs.

说明：在 bash/tcsh 中已经使用别名，相对而言别名比此方法更为方便。

5. 有关 shell 的专门讨论

(1)shell 程序的调试

切记：程序员（人）总是会犯错误的，而计算机是不会错的。

使用-x 进行跟踪执行，执行并显示每一条指令。

(2)命令组

用小括号将一组命令括起来，则这些命令会由子 shell 来完成；而{command_list;}则在
当
前 shell 中执行。这两者的主要区别在于其对 shell 变量的影响，子 shell 执行的命令不会
影响当前 shell 中的变量。

```
$ NUMBER=2
$ (A=2;B=2;NUMBER=`expr $A + $B`; echo $NUMBER)
4
$ echo $NUMBER
2
$ { A=2;B=2;NUMBER=`expr $A + $B`; echo $NUMBER; }
4
$ echo $NUMBER
4
```

总结：

在本章中讲述了 Bourne Shell 的基本知识,使用 shell 变量，shell script 基础。这些概
念
对于理解学习 Korn Shell, csh 以及其他 script 编程都是非常有用的。

很多 OS 都有不少语言及一些 script 功能，但很少有象 UNIX SHELL 这样灵活强大的 script
脚
本语言能力。

对于系统管理员或程序员来说，熟练地使用 shell script 将对日常工作(系统维护及管理
)
非常有用，如果你想作一个合格的系统管理员，强烈建议你进一步深入的了解和使用
shell.

另外，对于系统管理员来说，PERL 也是一个必不可少的 script 编程语言，尤其是对于处
理
文本格式的各种文件，PERL 具有 shell, awk, sed, grep 等的功能，但使用起来更为灵活
，
功能也更强大。大家可以参考“Perl By Examples"来学习和使用 PERL。

（完）