第6章 B-shell及编程

- shell对用户提出的运行程序请求进行解释执行。
- B-shell的特性，包括元字符、引号、shell的变量替换、命令替换、文件名生成，以及shell变量、流程控制、子程序。
- 使用shell可以编写脚本程序，在交互方式下也可以使用shell的流程控制编写复合命令。
- shell是面向命令处理的语言，shell提供的流程控制结构是通过对一些内部命令的解释实现的，另外一些灵活的功能通过shell替换实现。

6.1 启动B-shell

6.1.1 启动一个交互式B-shell

- 当登录shell不是B-shell（如csh）时，在命令提示符下键入sh命令，即进入了B-shell。
- 当sh作为登录shell被启动时，会自动执行用户主目录下的.profile文件中的命令。

6.1.2 #!/bin/sh：脚本文件的执行

- shell脚本（script）是预先定义好的一个命令序列，由需要执行的命令构成的文本文件。
- 当一个文件具有可执行属性，用户将它执行的时候，系统判断这个文件的格式，首先看它是不是一个经源程序编译链接后产生的，这种文件都满足系统的某一种特定格式。如果是，那么系统就加载这一程序，启动执行这一程序文件中的CPU指令。
- 否则，系统认为这是一个脚本文件，启动shell程序文件/bin/sh，运行/bin/sh文件中的CPU指令来解释执行脚本文件中的文本。
- UNIX的shell有自己的文本格式的脚本文件。为此，系统规定，如果脚本文件的第一行的头两个字符是#！，用这行后面的说明启动一个命令来解释这个脚本文件。

例如：用vi或者其他的文本编辑器，编辑下列的脚本文件lsdir，并且使用chmod u+x lsdir赋予文件可执行属性。

```
#!/bin/sh
if [ $# = 0 ]
then
dir=.
else
dir=$1
fi
find $dir -type d –print
```

【例6-1】使用#!为脚本文件自设定解释程序。

```
$ cat lsdir
#!/bin/od -c
if [ $# = 0 ]
then
dir=.
else
dir=$1
fi
find $dir -type d –print

$ chmod u+x lsdir
$ ./lsdir
0000000        #  !  /  b  i  n  /  o  d  -  c \n  i  f
0000020        [     $  #     =     0     ] \n  t  h  e  n
0000040        \n     d  i  r  =  .  \n  e  l  s  e \n
0000060        d  i  r  =  $  1 \n  f  i \n  f  i  n  d     $
0000100        d  i  r     -  t  y  p  e     d     -  p  r  i
0000120        n  t \n \n
```

脚本文件中第一行如果缺少了#!，那么，系统就会用默认的shell程序来解释执行脚本文件的文本。一般系统的默认shell是/bin/sh，Linux的默认shell是/bin/bash。

有三种方法可以执行脚本文件。

（1）sh < lsdir

（2）sh lsdir

sh lsdir /bin

（3）chmod u+x lsdir 给文件lsdir可执行属性

./lsdir

6.2 重定向与管道

6.2.1 输入重定向

1. 输入重定向自文件

用法：<文件

将标准输入重定向到一个磁盘文件，而不是从键盘输入。

【例6-2】 标准输入重定向的使用举例。

./myap < try.in

2. Here Document

用法：<<定界符

这种方法从shell脚本中获取数据，直到再次遇到定界符为止。UNIX把这种输入重定向的方法叫做“Here document”。

【例6-3】简单的Here document。

```
cat << TEXT
**********
* Hello! *
**********
TEXT
```

上述命令执行结果，在屏幕上显示：

```
**********
* Hello! *
**********
```

这里，<<符号后面是定界符TEXT，在shell输入下一个TEXT之前的这段文本，算作cat命令的标准输入。

【例6-4】 在Here document中进行命令替换和变量替换。

```
cat << TOAST
Hello! Time: `date`
My Home Directory  is  $HOME
Bye!
TOAST
```

上述命令执行结果为：

```
Hello! Time: Sat Jul 27 14:47:56 BEIJING 2004
My Home Directory is /usr/jiang
Bye!
```

【例6-5】 在Here document中禁止命令替换和变量替换。

```
cat << \TOAST
Hello! Time: `date`
My Home Directory  is  $HOME
Bye!
TOAST

cat << 'TOAST'
Hello! Time: 'date'
My Home Directory  is  $HOME
Bye!
TOAST
```

上述两种情况，输出结果都是：

```
Hello! Time: 'date'
My Home Directory  is  $HOME
Bye!
```

6.2.2 输出重定向

1. 标准输出重定向

用法：>文件 >>文件

将标准输出重定向到一个磁盘文件。

例如：

ls -l > file1

ls -l >> file1

2．标准错误输出重定向

用法：2>文件

将标准错误输出重定向到文件。标准错误的重定向方法和csh不同。

【例6-6】 B-shell的标准错误输出重定向举例。

（1）cc myap.c -o myap 2> myap.err

将cc命令的stderr重定向到文件myap.err中。

（2）设try是程序员设计的某个应用程序。

try > try.out 2>try.err

try 1> try.out 2>try.err

将try程序执行后的stdout或stderr分别定向到两个不同的文件中。

3．指定文件描述符的输出重定向

用法：文件描述符>&文件描述符

【例6-7】 B-shell指定文件描述符的输出重定向。

myap >rpt 2>&1

或者：

myap 1>rpt 2>&1

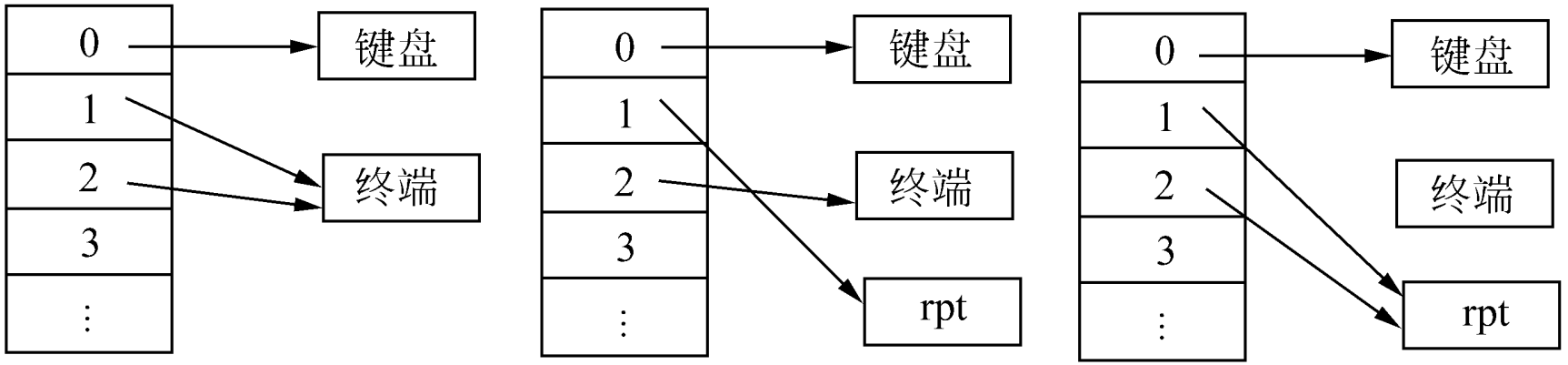

图6-1 指定文件描述符的输出重定向处理方式（一）

如果重定向的顺序反过来：

myap 2>&1 > rpt

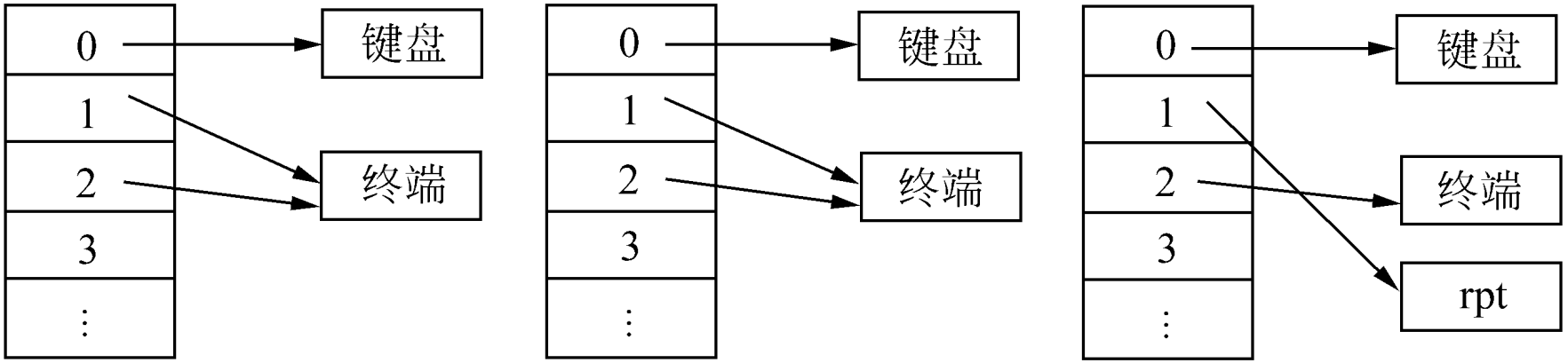

图6-2 指定文件描述符的输出重定向处理方式（二）

6.2.3 管道

【例6-8】 B-shell的管道处理举例。

（1）ls -l | grep '^d'

前一命令的stdout作后一命令的stdin。

（2）cc myap.c -o myap 2>&1 | more

前一命令的stdout+stderr作为下一命令的stdin。这里管道操作的优先级高，先完成cc的标准输出管道到下个命令的标准输入，然后，将文件描述符2重定向得和文件描述符1一样，就完成了将标准输出和标准错误输出都管道到下个命令的目的。

6.3 变　量

6.3.1 变量赋值和引用

- shell变量只有一种类型，存储字符串。
- shell变量可以赋值，内容可以被修改，也可以被引用。
- 变量名的第一个字符必须为字母，其余字符可以是字母、数字、下划线。
- 变量不需要事先定义，直接赋值就可以定义一个新变量，或者修改原变量的值。
- 引用时，在变量名前加$符，代表变量的内容。

- 在等号两侧不允许有多余的空格。
- 赋值时，等号右侧的字符串中含有空格或者制表符，换行符时，要用引号将打算赋值的字符串括起来。
- 引用时，在变量名前加$符，代表变量的内容。
- 引用一个未定义的变量，变量值为空字符串。

【例6-9】 shell变量的定义和引用举例。

定义一个名为addr的变量，存放IP地址字符串。

```
$ addr=20.1.1.254
$ echo  $addr
20.1.1.254
```

```
$ city="Beijing, China"
$ echo $city
Beijing, China
$ echo Connected to $proto Network
Connected to Network
$ proto=TCP/IP
$ echo Connected to $proto Network
Connected to TCP/IP Network
```

6.3.2 read：读用户的输入

内部命令read，可以从标准输入上读入一行，并将这行的内容赋值给一个变量。

【例6-10】 shell读取用户输入，并使用输入的信息。

```
$ read name
ccp.c
$ echo $name
ccp.c
$ ls -l $name
-rw-r--r--   1 jiang usr 32394 May 27 10:10 ccp.c
$
```

【例6-11】 在脚本程序中获取用户输入，并根据用户输入修改程序的配置文件。

假设有个应用程序**myap**运行时需要从文件**myap.cfg**中读取配置参数。使用下面的脚本文件，在程序安装时，根据用户的输入修改配置文件**myap.cfg**中的内容。

```
$ cat myap.cfg
ID 3098
SERVER 192.168.0.251
TCP-PORT 3450
TIMEOUT 10
LOG-FILE /usr/adm/myap.log
```

```
$ chmod u+x config.ap; cat config.ap
#!/bin/sh
echo 'Input IP address of server computer: \c'
read addr
ed myap.cfg > /dev/null <<TOAST
/SERVER
.d
i
SERVER $addr
.
w
q
TOAST
TIMEOUT 10
LOG-FILE /usr/adm/myap.log
$
```

$ ./config.ap

Input IP address of server computer: 202.112.67.213

$ cat myap.cfg

ID 3098

SERVER 202.112.67.213

TCP-PORT 3450

TIMEOUT 10

LOG-FILE /usr/adm/myap.log

$

6.3.3 环境变量和局部变量

- 创建的shell变量，默认为局部变量。
- 使用sh内部命令export可以将一个局部变量转换为环境变量。例如：export proto。
- 局部变量和环境变量是有区别的。在当前shell下启动的子进程只继承环境变量，不继承局部变量。
- 不附带任何参数的内部命令set会列出当前所有变量及其值，包括环境变量和局部变量。使用内部命令unset可以删除一个变量。例如：unset proto
- 与环境变量相关的重要的命令是env，env是一个外部命令，列出所有环境变量及其值。

【例6-12】观察shell的局部变量和环境变量的不同。

$ chmod u+x stat.report；cat stat.report

echo Connected to $proto Network

$ proto=AppleTalk

$ ./stat.report（启动一个子进程sh）

Connected to Networks

$ export proto

$ ./stat.report

Connected to AppleTalk Networks

【例6-13】在C语言程序中访问环境变量。

下面的程序例子使用getenv函数获取名为proto的环境变量的值。

```
int main(void)
{
    char *envstr;
    envstr = getenv("proto");
    if (envstr)
      printf("Protocol is  %s\n", envstr);
    else
      printf("Protocol is ??\n");
}
```

6.3.4 内置变量

在shell中有几个内置的变量，在shell脚本文件中可以直接使用这些内置变量，而且不允许对这些变量直接赋值。

B-shell的内置变量

内置变量	含　义
$?	最后一次执行的命令的返回码
$$	shell进程自己的PID
$!	shell进程最近启动的后台进程的PID
$#	命令行参数的个数（不包括脚本文件的名字在内）
$0	脚本文件本身的名字
$1 $2, …	第一个， 第二个，…，命令行参数
"$*"	"$1 $2 $3 $4..."，将所有命令行参数组织成一个整体，作为一个“单词”
"$@"	"$1" "$2" "$3"...，将多个命令行参数看作是多个“单词”

【例6-14】 位置变量$*与$@的区别。

```
$ cat arg.c
main(int argc, char **argv)
{
      int i;
      for (i = 0; i < argc; i++)
         printf("%d:[%s]\n", i, argv[i]);
}
$ cc arg.c -o arg
$ chmod u+x param; cat param
#!/bin/sh
echo $$
echo $#
echo "Usage: $0 arg1 arg2 ..."
./arg  "$@"
./arg  "$*"
$ ./param Copy Files to $HOME
19752
```

4

Usage: ./param arg1 arg2 ...

0:[./arg]

1:[Copy]

2:[Files]

3:[to]

4:[/usr/jiang]

0:[./arg]

1:[Copy Files to /usr/jiang]

6.3.5 shell的标准变量

系统登录成功后就由系统自动创建好的环境变量。

1. HOME

用户主目录的路径名。在C语言程序中可以用getenv("HOME")得到用户的主目录。

2. PATH

- 使用这个变量的值作为命令的查找路径。例如：

  PATH=/bin:/usr/bin:/etc，多个查找路径之间用冒号隔开。

- **PATH=./:/bin:/usr/bin:/etc**先搜索当前目录

  PATH=/:bin/usr/bin:/etc:./最后搜索当前目录

- 使用变量替换：

  PATH=./:$PATH

  PATH=$PATH:./

3. PS1和PS2

- 这两个变量设置B-shell提示符，分别设置B-shell的主提示符和副提示符。PS1="C>"，那么，主提示符变为C>。
- 副提示符用在这样的情况，按照shell的语法解释，一个命令在一行时输不完需要几行输入一个命令时，第2行及其他行的提示符用副提示符。
- B-shell一般的PS1="$"，PS2=">"。

【例6-15】 B-shell的主提示符和副提示符的用途。

```
$ msg='The echo command writes character strings
> to standard output. Strings are separated by spaces,
> a new-line character follows the last String.'
$ echo "$msg"
The echo command writes character strings
to standard output. Strings are separated by spaces,
a new-line character follows the last String.
$ cat <<TEXT
> *********
> Hello!
> *********
> TEXT
*********
Hello!
*********
$
```

4. TERM

终端类型名。许多全屏幕操作的软件（如**vi**），使用这一变量的值确定当前使用的终端的类型，去搜索终端库，以确定需要光标移动或其他控制操作时，主机该送往终端的转义字符序列的格式。

5. LANG或LANGUAGE

语种。有的命令会根据语种不同，给出相应的语种的提示信息，如：中文或英文。

6.4 替　换

- shell对脚本文件中的命令进行的替换工作有命令替换、变量替换和文件名生成。
- “替换”是shell提供灵活性的最重要手段。shell先替换用户输入或者脚本文件中的命令行，然后再执行命令。

6.4.1 文件名生成

shell将文件名通配符展开成多个文件名的工作，叫“文件名生成（Filename generation）”。

例如：命令ls *.c，shell会将*.c进行展开，展开后有多个文件时，按照字符串比较时从小到大的顺序排序。

6.4.2 变量替换

所谓变量替换就是将脚本文件中的$打头的单词，替换为变量值。

例如：

```
find $HOME –name "*.c" -print
echo Terminal type is $TERM
```

6.4.3 命令替换

用两个反撇号（`）括起来一个命令，用命令执行的标准输出代替两个反撇号标记出的字段，这就是所谓的“命令替换”。

反撇号是一个很少用到的符号，一般在键盘最左上角Esc键下方。

【例6-16】 shell命令替换的例子。

$ now=`date`（以命令date 的stdout替换`date`）

$ ./arg `date`（实际执行./arg Sun Dec 4 14:54:38 BEIJING 2004）

0:[./arg]

1:[Sun]

2.[Dec]

3:[4]

4:[14:54:38]

5:[BEIJING]

6:[2004]

$ frames=`expr 5 + 13`

$ echo $frames

18

$ count=10

$ count=`expr $count + 1`

$ echo $count

11

6.5 元 字 符

所谓“元字符” 是指在shell中有特殊含义的字符，shell对这些特殊字符进行特殊解释。

元 字 符	功 能
\	转义符，取消后继字符的特殊作用
空格，制表符	命令行参数的分隔符
回车	执行键入的命令
;	用于一行内输入多个命令
* [] ?	文件名通配符
$	引用 shell 变量
`	反撇号，用于命令替换
> < | >> <<	重定向与管道
&	后台运行
( )	用于定义 shell 函数，以及定义命令表
"	双引号，除$和`外，其他特殊字符的特殊含义被取消
'	单引号，对所括起的所有字符，不做特殊解释，特殊字符的特殊含义被取消

6.5.1 空格、制表符和转义符

和类似C语言一样的普通算法语言不同，空格和制表符在shell中起着很重要的作用，shell使用它做单词之间的分隔符。

另外，() > < | ; & 等除了它们自身的特殊含义外还同时起到单词分隔符的作用。

【例6-17】 shell中空格的作用。

下面的例子可以看出空格符在shell中的重要作用。

$ expr 33+44

33+44

$ expr 33 + 44

77

在普通的算法语言中，33+44和33 + 44没什么区别，但是，在shell中，它们的区别很大。

【例6-18】 shell中连续多个空格的作用。

```
$ expr   33 + 44
77
$ echo UNIX System V
UNIX System V
$ echo  UNIX System V
UNIX System V
```

6.5.2 回车和分号

- 回车的作用是标志一个命令输入的结束。按Enter键后，shell读取当前行，当前行的第一个单词是命令名，其余单词是该命令的参数，然后执行命令。
- UNIX允许把多个命令书写在一行，命令之间用分号分开，分号前后有无空格均可。 。

【例6-19】用分号串结两个命令。

```
$ date; who am i
Wed Jun 9 20:58:16 BEIJING 2004
jiang pts/2 Jun 09 20:58
```

6.5.3 文件名通配符

出现在命令行中的文件名通配符* [] ?会被shell展开成多个文件名。

6.5.4 美元符和反撇号

美元符$和配对使用的反撇号`分别用于变量替换和命令替换。

6.5.5 重定向和管道

重定向和管道功能给系统的使用带来了很多灵活性。可以使用它们组合多个命令以构造出更多功能。重定向符号和管道符号，都可以兼做命令分隔符。

【例6-20】 重定向和管道符的命令分隔符作用。

（1）./myap < my.in | ./myap2 > myap2.out

也可以写作

./myap<my.in|./myap2>myap2.out

（2）重定向符在可能会产生歧义的地方，应当使用空格。

列出文件2的属性，并存入文件2.list，使用下面的命令：

ls -l 2>2.list

shell会理解为执行ls -l并把标准错误输出重定向到文件2.list。应当使用：

ls -l 2 >2.list

- **B-shell除了对标准输入、标准输出和标准错误输出可以施行重定向之外，对除了0,1,2之外的其他文件描述符3~9也可以重定向。**
- **shell还有其他的几种重定向功能：<&和>&。**

例如：

- **./myap 8>8.list 将文件描述符8重定向到8.list。**
- **8>&1 文件描述符8重定向到文件描述符1的文件。**
- **5<&0 将文件描述符5作为输入文件，并且文件描述符5重定向得和文件描述符0一样指向同一文件。**

【例6-21】 shell对标准输入、标准输出和标准错误输出之外的其他文件描述符的重定向。

（1）将文件描述符8重定向到标准输出。

```
$ cat myap.c
main(int argc, char **argv)
{
      if (write(8, "Hello!\n", 7) < 0)
         perror("write fd8");
}
$ cc myap.c -o myap
$ ./myap 8 >`tty`
write fd8: Bad file number
$ ./myap 8>`tty`
Hello!
$
```

（2）将文件描述符5通过重定向归到标准输入。

```
$ cat myap.c
int main(void)
{
char buf[128];
int len;
len = read(5, buf, sizeof(buf));
if (len < 0)
   perror("read fd5");
else
   write(1, buf, len);
}
$ cc myap.c -o myap
$ ./myap
read fd5: Bad file number
$ ./myap 5<&0
Hello!
Hello!
```

6.5.6 启动程序后台执行

- &符作为后台启动程序的元字符。
- 在命令的结尾处加&符，那么，shell启动了这个命令进程之后，不等待命令运行结束，就立刻给出新的提示符，可以输入下个命令。如果这个命令的执行需要很长的时间，就会在后台运行。随后输入的命令执行的同时，后台程序也在运行。

例如：在后台执行一个需要运行时间较长的排序操作，排序结果记入文件。

sort telnos > telnos1 &

6.5.7 括号

- 括号是shell的元字符，配对的括号之间的所有命令，会作为一个整体。
- shell在实现括号的功能时，会首先创建子shell进程，在子shell进程中执行括号内的命令。

【例6-22】 shell元字符括号的使用举例。

```
date;who>>users_on
(date;who)>>users_on
(ls -l;grep '^[^#]' data1)|wc –l
```

由于括号、分号、重定向符、管道符，都兼有单词分隔符的作用，所以上述命令，这些符号前后加不加空格都可以。

6.5.8 转义符

- shell中的反斜线，用作转义符，取消紧跟其后的元字符的特殊作用。
- 如果反斜线加在不是元字符的其他字符前面，那么，这个反斜线跟没有一样。。

【例6-23】 shell元字符\的使用举例。

（1）find / -size +100 \(-name core -o -name *.tmp \) -exec rm -f {} \;

这里括号、星号和分号之前都有反斜线，以阻止shell对它们的特殊解释，find命令看不到这些反斜线，它能得到的是真正的括号、星号、分号。

（2）空格是shell的很重要的元字符，空格前面加反斜线，就取消了空格作为单词分隔符的特殊作用。

ls -l > file\ list

这样会创建一个文件名中包含空格的文件。

vi 2\>\&1

会创建一个名为2>&1的文件。

（3）**echo UNIX\ \ System\ V**

在此，空格不再是单词分隔符，而是命令参数的有效组成部分，**echo**不是得到3个命令行参数，而是仅得到一个命令行参数，由14个字符组成的字符串。那么，**echo**把这14个字符打印出来，**UNIX**和**System**之间就有两个空格。

（4）反斜线加在其他元字符前面，元字符的特殊作用被取消。

- **echo *会打印出当前目录下的所有文件名，而echo *会打印一个星号。**
- **echo $HOME打印出当前用户的主目录，但是，echo \$HOME打印出的就是字符串$HOME。**

（5）非元字符前的反斜线的作用。

$ echo DOS Directory is C:\WINDOWS\DESKTOP

DOS Directory is C:WINDOWSDESKTOP

$ echo DOS Directory is C:\\WINDOWS\\DESKTOP

DOS Directory is C:\WINDOWS\DESKTOP

6.5.9 双引号和单引号

由于shell有这么多的元字符，如果需要的时候都用反斜线来转义，很不方便。

例如：打印10个星号，并询问(1+1)*2>3?那么需要的命令是：

```
echo \*\*\*\*\*\*\*\*\*\*
echo \(1+1\)\*2\>3\?
```

- **shell提供了单引号，在单引号内的所有字符都不再解释为元字符。下面的命令和前面的命令运行结果相同。**

  echo '********'**

  echo '(1+1)*2>3?'

- 双引号和单引号的用法差不多，只是在配对的双引号括起的内容中还保留了元字符（$）和反撇号（`），只允许变量替换和命令替换。

【例6-24】 单引号和双引号使用上的区别。

```
$ a=10
$ b=20
$ c=30
$ echo '($a+$b)*$c=?'
($a+$b)*$c=?
$ echo "($a+$b)*$c=?"
(10+20)*30=?
```

6.5.10 转义符与引号及反撇号

- 在双引号或者单引号内，都没有保留反斜线的元字符地位，所以，echo '\A'和echo "\A"的输出都是\A，而echo \A的输出却是A。
- shell允许在双引号括起的内容中使用\"代替双引号自身，允许\\代表反斜线自身，允许\$代表美元符自身，\`代表反撇号自身。其他情况下的反斜线保持原文不变。概括起来说，就是在配对的双引号括起来的内容中，只允许\"、\$、\`和\\这四个转义序列。

- **shell对单引号的处理与双引号不同，在配对的单引号内不允许任何转义。如果文本中有单引号就必须移到单引号括起的内容之外。**

【例6-25】 单引号处理，以及双引号括起的内容中容许的转义。

```
$ echo 'Don'\''t remove Peter'\''s DOS dir "C:\PETER"!'
Don't remove Peter's DOS dir "C:\PETER"!
$ echo "`who am i | awk '{print \$1}'`'s \$HOME is \"$HOME\""
jiang's $HOME is "/usr/jiang"
$ echo "\"pipeline\" is commands separated by |"
``pipeline'' is commands separated by |
```

- 同双引号问题类似的还有反撇号（`），反撇号内仅允许\`和\\转义序列，其他情况下，反斜线代表的是自己。
  - 在配对的反撇号内括起的内容中如果有反撇号，那么就用\`。
  - 反撇号内允许\\代表\自己。

【例6-26】 反撇号内的转义处理。

下面的一个实用例子是编写一段脚本程序，给出程序名字，终止系统中正在运行的进程。

终止一个正在运行的程序，一般的方法，首先使用ps命令列出当前的活动进程，然后，根据ps提供的进程PID，用kill命令终止这一进程。下面的例子终止程序myap的进程。

```
$ ps -e | grep myap
31650  pts/2  0:00 myap
$ kill 31650
$ ps -e | grep myap
$
```

- 为了挑选出期望终止的程序的PID，用awk命令。使用awk命令是：

```
ps -e | awk '/[0-9]:[0-9][0-9] myap$/{printf("%d ",$1)}'
```

- 为了编写可以灵活指定程序名字的脚本程序，使用脚本程序执行时得到的命令行参数$1

```
ps -e | awk "/[0-9]:[0-9][0-9] $1\$/{printf(\"%d \",\$1)}"
```

- 打印一个进程PID号不是最终目的，而是希望这个输出做kill的参数，应当使用“命令替换”功能。

```
kill `ps -e | awk "/[0-9]:[0-9][0-9] $1\\$/{printf(\\"%d \\",\\$1)}"`
```

```
$ cat k
PIDs=`ps -e | awk "/[0-9]:[0-9][0-9] $1\\$/{printf(\\"%d \\",\\$1)}"`
echo "kill $PIDs"
kill $PIDs
$ chmod u+x k
$ ps -e | grep myap
 27248  pts/2  0:00 myap
 31714  pts/2  0:00 myap
 36926  pts/2  0:00 myap
$ ./k myap
 kill 27248 31714 36926
$ ps -e | grep myap
$
```

6.6 条件判断

shell作为一种编程语言，程序的分支结构和循环处理是必需的要素。条件判定，使得程序的流程根据不同情况进行不同的处理。

6.6.1 条件

shell是一种命令语言，它设计得非常简练。shell变量只有字符串一种变量类型，就连加减乘除基本的算术运算，比较两个数大小这样的逻辑判断功能，都不提供。但是，它提供了命令替换等机制，使用这些机制，利用外部的命令，可以完成所需要的功能。这是一种“策略和机制分离”的方法。shell仅仅提供一种机制，但不提供解决问题的策略，所有策略问题“外包”给其他的命令，或者用户自己编写的应用程序。这种开放性设计，使系统有极大的灵活性，而且，大大简化了shell自身的设计。尽管这样会带来一些效率上的问题，对于shell这样面向命令处理的脚本语言来说，效率上的损失可以忍受。

为了提高效率，有些shell把脚本程序中某些常用的命令，如expr、test、true、false等，改进成内部命令，但这种改进是透明的，shell仍不失它一贯的风格。C语言本身也是这样一种思路，C语言连基本的输入输出语句都没有，它把这样的功能“外包”给了诸如printf、scanf、fgets这样的C语言基本要素之外的库函数。如果用户对这些函数不满意，完全可以编制自己的输入输出函数，它们和printf之类库函数在C语言中具有完全相同的地位。这种“策略和机制相分离”的方法，值得读者在设计其他软件系统时参考。

- shell条件判定只提供了一种机制，条件判断的惟一依据是判定一条命令是否执行成功。判断方法是根据命令执行的返回码，返回0，就算是条件成立，返回非0的任意值，都算条件不成立。
- 所谓的“命令执行的返回码”是由命令自身的行为决定的。
  - cmp命令比较两个文件是否相同，两文件相同时，cmp命令返回码为0；否则，不为0。其返回码可以用于shell的条件判断。
- 用管道线连接在一起的若干命令，shell仅采用最后一个命令执行的返回码。

【例6-27】 在C语言程序中不同的分支结束程序时给出不同的返回码。

下边是源程序myap.c的一个框架。

```
1 int main(int argc, char *argv[])
2 {
3   ……
4  if (……) {
5       ……
6       exit(4);
7  }
8  ……
9  if (……) {
10      ……
11      exit(19);
12 }
13 ……
14 if (……) {
15      ……
16      exit(0);
17 }
18 ……
19 return 0;
20 }
```

在shell中，有一个内置变量$?，它是上个命令执行结束后的返回码的值。

【例6-28】 设当前目录下有目录xyz并且不存在一个名为abc的文件或目录。

```
$ ls -d xyz
xyz
$ echo $?
0
$ ls -d abc
abc: not found
$ echo $?
2
```

6.6.2 最简单的条件判断

最简单的条件判断仅含有一个分支，条件成立或者不成立时执行相应的命令。具体做法是用&&或||连接两个命令。

- **命令1 &&命令2**

若命令1执行成功（返回码为0）则执行命令2，否则不执行命令2。

语义短路

- **命令1||命令2**

若命令1执行失败（返回码不为0）则执行命令2，否则不执行命令2。

【例6-29】利用ls命令的返回码进行条件判断。

```
$ ls -d xyz && echo FOUND
xyz
FOUND
$ ls -d xyz > /dev/null && echo FOUND
FOUND
$ ls -d abc || echo No dir \'abc\'
ls: abc not found
No dir 'abc'
$ ls -d abc 2> /dev/null > /dev/null || echo No dir \'abc\'
No dir 'abc'
```

6.6.3 命令true与命令false

- 命令true和命令false不是shell中的关键字。
- 许多UNIX系统存在/bin/true和/bin/false两个命令文件。
  - true命令的返回码总为0，除此之外不做任何操作。

    ```
    int main(void) { return 0; }
    ```

  - false命令的返回码总不为0。
- shell中有个内部命令，名字为冒号(:)，冒号命令和true命令有相同的效果。

6.6.4 命令test与命令[

- UNIX系统自带的命令test，可以提供一些常用的条件判断。
- 命令[与test功能等价，和test命令不同的是，命令[要求其最后一个命令行参数必须为右方括号。
- 早期的UNIX中的确依靠两个程序文件/bin/test和/bin/[，在Linux中就是符号连接。

例如：下面两个命令的执行结果完全相同。注意，空格是必不可少的。

```
test -r /etc/motd
[ -r  /etc/motd ]
```

test命令主要提供了以下的判断功能：

1. 文件特性检测

用于文件特性检测的参数如下：

-f 普通文件

-d 目录文件

-s size>0

-r 可读

-w 可写

-x 可执行

例如：

```
test -r /etc/motd  && echo readable
[ -r /etc/motd ] && echo readable
```

2. 字符串比较

用于字符串比较的参数如下：

-z *str1*	***str1*串长度等于0 (zero)**
-n *str1*	***str1*串长度不等于0 (non-zero)**
str1* = *str2	***str1*与*str2*串相等**
str1* != *str2	***str1*串与*str2*串不等**

- **等号和不等号两侧的空格是必不可少的。**
- **判断两个字符串相等或不等时，如果两个字符串之一有可能是空字符串，应当使用双引号将必要的内容括起来。**

例如：test $# = 0 && echo "No argument“

[-n "$name"] || echo "empty string."

3. 整数比较

用于整数比较的参数如下：

-eq	equal	=
-gt	greater than	>
-ge	greater or equal	≥
-ne	not equal	≠
-lt	less than	<
-le	less or equal	≤

例如： test `ls | wc -l` -ge 1000 && echo "Too many files"

4. 逻辑运算

用于逻辑运算的参数如下：

!	NOT（非）
-o	OR（或）
-a	AND（与）

例如：判断$cmd是一个具有可执行属性的普通文件。需要判断它不是目录文件，并且又具有可执行属性。

```
[ ! -d $cmd -a -x $cmd ]
```

【例6-31】条件判断命令中空格符的重要作用。

本例说明脚本文件中必需的空格不可省略。这也是shell中使用[……]结构时经常出现的错误。

设有一脚本文件t1，当给它的命令行参数正好是两个时，打印出一条信息。但是，实际执行的结果却是，给定了9个命令行参数，这条信息照样打印出来。

```
$ cat t1
echo "count=$#"
[ $#=2 ] && echo There are 2 files.
$ ./t1 *.c
count=9
There are 2 files.
```

6.6.5 { }与( )

当使用&&或||时，需要在条件分支中完成多个动作，执行若干个命令，就需要使用类似复合语句的构造，在shell中使用大括号。书写规则为

{ *list*;}

左大括号后面必须有一个空格，右大括号前面必须有分号，或者由换行符代替。*list*是由一个或者多个命令构成的命令表。

【例6-32】 在B-shell中使用大括号实现复合语句的构造。

```
DIR=/usr/include/sys/netinet
pwd
[ -d $DIR ] && {
cd $DIR
echo "Current Directory is `pwd`"
echo "`ls -l *.h | wc -l` files (*.h)"
}
pwd
```

执行结果如下：

```
/usr/jiang
Current Directory is /usr/include/sys/netinet
27 files (*.h)
/usr/include/sys/netinet
```

- 括号也有将多个命令合成一个整体的功能。括号是shell的元字符，所以书写格式上不必要像大括号那样一定要单独作为一个独立命令的行首单词。书写规则为： (*list*)
- 大括号括起的一组命令是在shell进程中执行，
- 括号括起的一组命令，却是在子shell中执行。shell会首先创建子shell进程，然后，在这一子shell中执行命令，括号内的命令执行完毕之后，子shell就会终止，返回到shell。
- 同样执行效果的前提下，用{}会比()执行效率更高些。

【例6-33】 {}与()的不同之处。

将上例中的{}改成()。那么，执行结果会有所不同，脚本程序最后一行的输出不同。

/usr/jiang

Current Directory is /usr/include/sys/netinet

27 files (*.h)

/usr/jiang

shell点命令(.)：在执行脚本文件的命令之前增加句点和空格，那么，脚本文件的命令就在当前shell中执行，而不是启动一个新的子shell进程来解释脚本文件中的命令。

【例6-34】点命令，以及shell与子shell的区别。

```
$ pwd
 /usr/jiang
 $ cat cdn
 echo $$
 cd /usr/include/sys/netinet
 pwd
 $ ./cdn
 4650
```

```
/usr/include/sys/netinet
$ echo $$
6626
$ pwd
/usr/jiang
$ . ./cdn
6626
/usr/include/sys/netinet
$ pwd
/usr/include/sys/netinet
$
```

【例6-35】 使用{}时，多行合并为一行书写的例子。

```
[ -f core ] && {
echo "Remove core file"
rm -f  core
}
```

写成一行应当为：

```
[ -f core ] && { echo "Remove core";rm -f core;}
```

6.6.6 条件结构if

条件结构if可以提供多个分支。条件if的语法是：

if ***condition***

then ***list***

elif ***condition***

then ***list***

else

list

fi

list可以是多个命令构成的命令表；***condition***仍然是一个命令，根据返回码判定为条件满足或者不满足。

- 实现条件if的关键字是if、then、elif、else、fi。表现为一个独立命令的首个单词，作为命令名。shell把它们处理成内部命令，然后再赋以shell的特殊解释，用来进行流程控制的。

- 在多个命令构成*list*时，没有必要再由多个命令构成的*list*的开始和结尾用大括号或者括号括起来，因为，在then和elif之间夹着的多个命令算作一个分支要执行的程序块，或者叫分程序。在else和fi之间夹着的多个命令算作另一个分支要执行的程序块。

【例6-36】 if结构的使用举例，条件满足和不满足分别执行不同的命令。

将系统中现有文件errfile合并到文件errlog的尾部，合并时加入合并的日期。

```
$ cat errmonitor
LOGFILE=./errlog
date>>$LOGFILE
if test -r errfile
   then
        cat errfile>>$LOGFILE
        rm errfile
else
        echo "No error">>$LOGFILE
fi
```

6.6.7 case结构

case结构是基于模式匹配基础上的多条件分支结构，在很多情况下比使用if结构更简练。

case *word* in

**　　*pattern1*) *pat1_list*;;**

**　　*pattern2*) *pat2_list*;;**

esac

其中esac是case四个字母的反序。case和esac是关键字，shell是通过把它们解释为内部命令的方式实现流程控制。

（1）模式描述时，使用**shell**的文件名匹配规则，这样使用起来更方便。

（2）;;是一个整体，不可分隔，不能在两分号间加空格，也不能用两个连续的空行代替它（这一点如同&&和||）。在右括号和;;之间可以夹着多个命令定义的一个程序块，这个程序块可以有多个命令，也没必要用大括号或者括号括起来。

（3）可以使用竖线罗列出多个模式。

（4）当***word***可以与多个模式匹配时，只执行它所遇到的第一个命令表。

【例6-37】 case结构的使用举例。

```
case "$1" in
START|start)
    （一段程序）
    ;;
STOP|stop)
    （一段程序）
    ;;
*)
    echo "Usage: $O [start|stop]"
    ;;
esac
```

6.7 循 环 结 构

6.7.1 while结构

循环结构while的语法是：

while ***condition***

do ***list***

done

- 其中，while、do、done是关键字，必须以独立命令行的首个单词的身份出现。do和done之间的一段程序算作循环体。
- while结构是在条件满足的前提下，循环执行do和done框起来的循环体内的命令。

【例6-38】 shell脚本程序使用while结构的例子。

每隔10s检查文件lockfile是否可读，并打印出这个文件的属性，直到这个文件被其他的任务删除后循环才退出。

```
$ cat waitlock
while test -r lockfile
do
    ls -l lockfile
    sleep 10
done
```

书写这一脚本程序时，行与行合并时应注意的问题见上一节。下面的写法是错误的：

```
while test -r lockfile do
    ls -l lockfile
    sleep 10
done
```

下面的写法是正确的：

```
while [ -r lockfile ];do ls -l lockfile; sleep 5;done
```

【例6-39】交互式使用while结构。

使用ftp命令从远程计算机传输文件到本地文件mydata，如果线路速度不是很快，用下面的命令监视文件mydata的增长速度。

```
while true;do ls -l mydata;sleep 10;done
```

shell的内部命令冒号命令(:)，冒号命令执行起来和true命令有相同的效果，所以上述命令也可以改写为下面的形式：

```
While :;do ls -l mydata;sleep 10;done
```

6.7.2 expr：计算表达式的值

- 命令expr用来求表达式的值，它是独立于shell之外的外部命令。
- 在B-shell编程中，B-shell本身没有提供数学运算和字符串运算的能力，所有这些运算都是借助于命令expr完成的。

1. 算数运算和关系运算

- expr支持算术运算加减乘除，取余数，以及数值比较的关系运算。要求操作数是包含数字0~9的字符串，前面还可以带负号。
- expr将字符串转化为整数后运算，对数字执行的算术运算，显示之前再转换回字符串。
- 作为运算符的符号，必须作为单独的一个命令行参数，运算符两侧的空格是必不可少的。
- 考虑到shell的元字符，应该转义的地方必须加反斜线转义，以确保expr命令会得到它所期望的符号，而不是被shell作特殊解释。

表6-3 expr的运算符

算 符	运 算	算 符	运 算
*	乘法运算	<=	小于或等于
/	除法运算	=	等于
%	取余数运算	!=	不等于
+	加法运算	>=	大于或等于
-	减法运算	>	大于
<	小于		

- expr的运算优先级和C语言一样。乘除法优先级最高，其次加减法，然后是关系运算。关系运算的结果是expr打印1或者0。也可以使用括号。
- expr返回值与系统最后退出命令刚好相反，成功返回1，任何其他值为无效或错误。

【例6-40】 shell中使用expr命令时不可漏掉shell必需的转义符。

假设a，b，c是三个shell变量。求a*(b+c)，正确的写法为：

```
expr $a \* \( $b + $c \)
```

判断x是否大于20。正确的写法为：

```
[ `expr $x \> 20` = 1 ] && echo OK
```

使用时注意转义符和必须有的空格。

【例6-41】 每秒一次倒计数到0。

```
$ cat count
count=10
[ $# = 0 ] || count=$1
while [ $count -ge 0 ]
do
      echo "$count \c"
      count=`expr $count - 1`
      sleep 1
done
$ ./count 15
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
$
```

2. 字符串运算

用法：**expr *string* : *pattern***

- 这是一种字符串匹配运算。用正则表达式***pattern***去匹配字符串***string***，从最左字符开始，尽量匹配，看能匹配多长。最终打印出匹配的长度值，不匹配时打印**0**。这里的运算符冒号，必须单独作为一个命令行参数传递给**expr**命令。

【例6-42】 获取正则表达式与字符串的匹配长度。

```
$ expr 123 : "[0-9]*"
3
$ expr A123 : "[0-9]*"
0
$ expr 123A : "[0-9]*"
3
$ expr 123A : "[0-9]*$"
0
```

2. 字符串运算

用法：**expr *string* : *pattern***

- 这是一种字符串匹配运算。用正则表达式***pattern***去匹配字符串***string***，从最左字符开始，尽量匹配，看能匹配多长。最终打印出匹配的长度值，不匹配时打印**0**。这里的运算符冒号，必须单独作为一个命令行参数传递给**expr**命令。
- 在正则表达式中有部分文字用**\(**和**\)**括起来，这两个符号在匹配时不起作用，那么，字符串能与正则表达式匹配时，打印括号内能匹配的部分；否则打印空字符串。

【例6-43】 抽取正则表达式可匹配的字符串中的字符串片段。

$ tty

/dev/tty6

$ expr `tty` : .* 返回字符串长度

9

$ expr `tty` : '/dev/tty\(.*\)' 获取终端文件名字中的编号

6

$ termno=`expr \`tty\` : '/dev/tty\\(.*\\)'` 注意反撇号和转义符的关系

$ echo $termno

6

$ expr `tty` : /dev/tty\\\(.*\\\) 只使用转义符的格式

6

$ termno=`expr \`tty\` : /dev/tty\\\\\\(.\\*\\\\\\)`

$ echo $termno

6

$ pwd

/usr/include/arpa

$ expr `pwd` : '.*/\([^/]*\)$' 截取路径名的最后一个分量

arpa

在上述的例子中，按照反撇号和转义符的关系（参见6.5.10小节），那么，

`expr \`tty\` : /dev/tty\\\\\\(.\\*\\\\\\)`

就是用下列命令的执行结果进行命令替换。

expr `tty` : /dev/tty\\\(.*\\\)

进一步，按照shell的元字符处理方式，expr的第一个参数是tty命令的执行结果，第二个参数是由冒号独立组成的字符串，expr实际可以得到的第三个参数是字符串/dev/tty\(.*\)，由expr命令内部自行解释第三个参数中的正则表达式。

6.7.3 for结构

for结构的循环，循环体被执行多次。要求给出一个由多个单词构成的表格。每次循环，循环控制变量取值是表格中的一个单词。语法为：

for ***name*** **in** ***word1 word2 ...***

do ***list***

done

其中，***name***是循环控制变量，在循环体内用***$name***引用变量的值。

for循环的另一种格式是：

for *name*

do *list*

done

这种格式没有指定循环控制变量的取值表，那么，**系统就会用shell的位置变量中的$1，$2，…来作为循环控制变量的取值表**。相当于：

for *name* in $1 $2 ...

do *list*

done

【例6-44】使用for循环的例子。

这是SCO UNIX中一段开机时系统自动执行的脚本程序。检索/etc/rc.d目录下所有的直属或一级子目录中的所有可执行文件，执行它们。

```
if [ -d /etc/rc.d ]
then
        for cmd in  /etc/rc.d/*/* /etc/rc.d/*
        do
            [ ! -d $cmd -a -x $cmd ] && $cmd
        done
fi
```

【例6-45】将所有的命令行参数逆序显示出来。脚本程序中使用for循环。

```
$ cat rev1
list=""
for arg
do
        list="$arg $list"
done
echo "$list"
$ ./rev1 aa bb cc
cc bb aa
```

【例6-46】 给出一组程序名，终止这些程序文件启动的所有进程。

```
$ cat k
for name
do
    echo "$name: \c"
    PID=`ps -e|awk "/[0-9]:[0-9][0-9] $name\\$/{printf(\\"%d\\",\\$1)}"`
    if [ -n "$PID" ]
    then
      echo kill $PID
      kill $PID
    else
      echo No process
    fi
done
```

$ k myap findkey sortdat

myap: kill 20608 27336 28072 29720

findkey: kill 36994 37948

sortdat: No process

6.7.4 break与continue

shell的内部命令break和continue用在循环结构for和while中使用，与C语言中的beak和continue流程控制功能类似。

【例6-47】 break使用的例子。

将命令行参数逆序输出，脚本程序中使用了跳出循环的内部命令break。

```
$ cat rev2
count=$#
cmd=echo
while true
do
        cmd="$cmd \$$count"
        count=`expr $count - 1`
        [ $count -eq 0 ] && break
done
eval $cmd
$ ./rev2 aa bb cc
cc bb aa
```

【例6-48】 continue使用的例子。

将命令行参数逆序输出，脚本程序中使用了提前终止循环体的内部命令continue。

```
$ cat rev3
count=$#
cmd=echo
while true
do
    cmd="$cmd \$$count"
    count=`expr $count - 1`
    [ $count -gt 0 ] && continue
    eval $cmd
    exit 0
done
$ ./rev3 aa bb cc dd
dd cc bb aa
```

【例6-49】从脚本程序中输入一个有效的IP地址。

```
$ cat getip
ADDR=192.168.0.112
while true
do
     echo "Please Input IP Address [$ADDR]: \c"
     read addr
    [ "$addr" = "" ] && addr=$ADDR
    if [ `expr "$addr" :
    "[0-9][0-9]*\.[0-9][0-9]*\.[0-9][0-9]*\.[0-9][0-9]*$"` -gt 0 ]
    then
      break
    else
      echo " **** Invalid IP Address !"
fi
done
echo "IP Address is $addr"
```

```
$ ./getip
Please Input IP Address [192.168.0.112]:
IP Address is 192.168.0.112
$ ./getip
Please Input IP Address [192.168.0.112]: abcd
**** Invalid IP Address !
Please Input IP Address [192.168.0.112]: 123.12.12.190
IP Address is 123.12.12.190
$ ./getip
Please Input IP Address [192.168.0.112]: 257.222.340.231
IP Address is 257.222.340.231
$
```

6.8 函　数

从UNIX System V 2.0开始，才允许使用shell函数。

语法：*name*() { *list*;}

- 在调用函数时，引用函数的名字，可以附加上0到多个参数，在函数体内部以位置变量$1，$2，…或$*，$@方式引用函数的参数。
- 在函数体内部可以使用内部命令return，使函数有返回码，返回码0代表成功，非零表示失败。
- 函数体内一个函数不能调用它自己。shell函数不允许递归调用。函数体的执行不创建新的子shell进程，它和脚本文件的其他部分一样，在同一个shell进程中执行。
- shell脚本中可以使用#号作注释，如果#号出现在一个词的首部，那么，从#号至行尾的所有字符被忽略。

【例6-50】使用shell函数的例子。

这个例子是一个广域网通信适配卡驱动程序安装脚本的一部分。

这个通信适配卡安装之前要求输入硬件的中断号，I/O基地址和通信速率。然后，列出操作员的输入，等待确认后才执行安装操作。

安装操作在41~85行之间，被省略。

第4~16行的函数get_answer打印出一条消息，然后，强制输入y或者n，否则继续要求用户输入。函数体内使用了$1引用调用函数get_answer的命令行参数，用return使得shell函数像普通命令一样有返回码，供条件判断使用。

第20~34行的函数get_val有四个参数。第一个参数$1存放用户输入值的shell变量的名字，第二个参数$2是输入之前给用户的提示信息，第三个参数$3是用户直接按下回车时的默认值，最后一个参数$4是允许取值的有效值列表。函数get_val强制用户输入一个有效值列表中的有效值，否则，就给出有效值列表做提示，并进一步要求用户重新输入。函数体内使用了while循环和for循环结构。第29行的break命令含有参数2，可以跳出两层循环。

脚本程序执行时从第36行开始执行。由于脚本文件前面有了函数说明，shell记下了函数名字get_val和get_answer作为内部命令，在执行命令get_val和get_answer时，shell都转去执行函数体，而不是到磁盘上寻找这样名字的命令文件。

$ awk '{print NR,$0}' WanCom

```
 1  # Shell function to read a Y/N response
 2  # Usage: get_answer <message>
 3  #
 4  get_answer()
 5  {
 6      while  true
 7      do
 8          echo "$1? (y/n) \c"
 9          read yn
10           case $yn in
11              [yY]) return 0;;
12              [nN]) return 1;;
13                 *) echo "Please answer y or n" ;;
14           esac
15       done
16  }
```

```
17  # Shell function to get a value
18  # Usage: get_val <var_name> <message> <default_val> <list>
19  #
20  get_val()
21  {
22    while true
23    do
24      echo "$2 [$3] : \c"
25      read val
26      [ "$val" = "" ] && val=$3
27      for i in $4
28      do
29          [ "$val" = "$i" ] && break 2
30      done
31      echo "**** Invalid choice $val, must be in $4"
32    done
33    eval "$1=$val"
34  }
```

```
35  #  main program
36  get_val INTR "Interrupt Number" 10 "2 3 4 5 7 10 11 12 14 15"
37  get_val PORT "I/O Base Address" 320 "200 210 220 230 300 310 320 330"
38  get_val BAUD "Baud Rate" 9600 "2400 9600 14400 33600 64000"

39  echo "Interrupt $INTR, I/O base address $PORT, Baud rate is $BAUD"
40  get_answer "Do you want to install WanCom adapter driver" && {
41                  echo "Please Wait ...\c"
                    ……
85  }
$ ./WanCom
Interrupt Number [10] : 22
**** Invalid choice 22, must be in 2 3 4 5 7 10 11 12 14 15
Interrupt Number [10] : 14
I/O Base Address [320] :
Baud Rate [9600] : 33.6
**** Invalid choice 33.6, must be in  2400 9600 14400 33600 64000
Baud Rate [9600] : 33600
```

Interrupt 14, I/O base address 320, Baud rate is 33600

Do you want to install WanCom adapter driver? (y/n) y

Please Wait ...

...

$

6.9 shell开关和位置变量

set命令后不跟任何参数时，列出shell的所有变量，包括局部变量和环境变量。set是内部命令，在B-shell和C-shell中用法会有区别。

6.9.1 set：设置B-shell内部开关

内部命令set可以用来设置一些shell开关，以影响shell的某些行为。常用的这些开关有：

-x　在每执行一条命令时，先打印出这个命令及命令参数，为区别于正常的shell输出，还在前面冠以+号

+x 取消上述设置

-u 当引用一个未赋值的变量时，产生一个错误

+u 当引用一个未赋值的变量时，认为是一个空串

【例6-51】 shell脚本程序中-x开关的作用。

脚本程序中打开-x开关，可以观察到程序执行的流程。

```
$ cat chmod1
set -x
echo "$*"
for i
do
    [ -f $i ] && {
    chmod a+r $i
    echo "$i is readable"
    }
done
$ ./chmod1 a*
+ echo a1 aa8 abcd
a1 aa8 abcd
+ [ -f a1 ]
```

```
+ chmod a+r a1
+ echo a1 is readable
a1 is readable
+ [ -f aa8 ]
+ [ -f abcd ]
+ chmod a+r abcd
+ echo abcd is readable
abcd is readable
```

本例中第一行的**set -x**也可以省略，使用命令**sh -x chmodl a***也能达到相同的效果。或者，将脚本文件**chmod1**的第一行修改为**#!/bin/sh -x**也可以。

【例6-52】交互式shell中-x开关的作用。

交互式shell中打开-x开关，可以观察shell的命令替换，变量替换，文件名生成，以及转义符的作用。每次shell在真正执行一个命令之前都把要执行的命令显示出来。

```
$ set -x
$ tty
+ tty
/dev/tty6
$ termno=`expr \`tty\` : /dev/tty\\\\\\(.\\*\\\\\\)`
+ tty
+ expr /dev/tty6 : /dev/tty\(.*\)
termno=6
$ echo $termno
+ echo 6
6
$ find $HOME -size +100 \( -name \*.c -o -name xxxx \) -exec rm -i {} \;
+ find /usr/jiang -size +100 ( -name *.c -o -name xxxx ) -exec rm -i {} ;
```

set -u开关也非常有用。一般情况下，程序员不会引用一个未赋值的shell变量。凡是出现了这种情况，往往是因为程序中引用变量时的变量名拼写错误。设置-u开关，一旦引用一个未赋值的shell变量，shell脚本程序就会立刻停下来，可以提前发现程序中的错误。

【例6-53】 shell中-u开关的使用。

可以检查出引用变量时的变量名拼写错误。

```
$ name=CSPTF
$ echo Connecting to $nmae Network
Connecting to Network
$ echo Connecting to $NAME Network
Connecting to Network
$ set -u
$ echo Connecting to $nmae Network
nmae: 0402-009 Parameter is not set.
$ echo Connecting to $NAME Network
NAME: 0402-009 Parameter is not set.
$ echo Connecting to $name Network
Connecting to CSPTF Network
$
```

6.9.2 set：设置shell位置变量

set命令后边跟多个参数，可以修改shell的位置变量。位置变量就是$1，$2， $3，…，以及$#，$@和$*。位置变量还会影响未罗列出循环控制变量取值表的for结构循环。

【例6-54】使用set命令设置shell的位置变量。

使用set命令可以重新设置shell的位置变量，先前的位置变量全部被新的值代替。

```
$ echo $#
0
$ date
Sun Jul 28 11:00:40 BEIJING 2004
$ set `date`
$ echo $1 $2 $3 $4
Sun Jul 28 11:00:40
$ ls -l /etc/motd
-rw-r--r-- 1 root staff 316 Jan 5 08:42 /etc/motd
$ set `ls -l /etc/motd`
sh:-rw-r-r-: bad option(s)
$ set -- `ls -l /etc/motd`
$ echo $9:$5  $1
/etc/motd:316  -rw-r--r--
```

6.9.3 shift：位置变量的移位

除了set命令外，内部命令shift，也可以影响位置变量。它的功能是使位置变量“移位”。例如：$#为4，$1，$2，$3，$4分别为aa，bb，cc，dd。那么，执行shift命令后，$#变为3，而$1，$2，$3分别变成bb，cc，dd。

shift命令还可以跟一个整数做参数，说明“移位”几个位置，上例中，如果执行shift 2命令，那么，$#变为2，而$1，$2分别变成cc，dd。

shift命令也影响位置变量的$*和$@。

【例6-55】 逐个打印源程序文件。

打印多个源程序文件，每打印一个文件之前列出文件名，打印文件时每行带上行号。

```
$ cat prt
while  [ $# -gt 0 ]
do
    echo ====================
    echo FILE NAME: $1
    echo ====================
    awk '{printf("2d %s\n",NR,$0)}' $1
    shift
done
$ ./prt makefile *.[ch]
```

【例6-56】 给出若干个程序名，终止这些程序文件启动的所有进程。

获取程序启动的进程PID的方法，在前面的“元字符”6.5.10小节中介绍过。这里的脚本程序中使用了位置变量和shift命令。

```
$ cat k
while [ $# != 0 ]
do
    echo "$1: \c"
    PID=`ps -e | awk "/[0-9]:[0-9][0-9] $1\\$/{printf(\\"%d \\",\\$1)}"`
    if [ -n "$PID" ]
    then
        echo kill $PID
        kill $PID
    else
```

```
        echo No process
    fi
    shift
done
$ ./k myap findkey sortdat
myap: kill 26506 38020
findkey: kill 31542
sortdat: No process
$
```

【例6-57】 软件安装时调整操作系统内核的部分参数。

下面的一段脚本程序，在SCO UNIX系统中安装某个软件包时，调整操作系统内核参数。第一行和第二行列出了相应的参数和期望的配置值。例如：要求MSGMNB参数取值至少32768, NQUEUE参数取值至少64。

命令configure的格式：

configure -y 参数名

configure 参数名=参数值 参数名=参数值 ……

第一种格式，打印出指定名字的内核参数的当前取值，第二种格式，调整指定名字的内核参数为指定的参数值，并且可以一次调整多个参数。这个命令是SCO UNIX专用的，在其他UNIX中没有通用性。

脚本程序使用set命令和shift命令对位置变量的影响。内核参数当前取值已经满足要求的参数不再调整。

```
 1 PARA="MSGMNB MSGTQL MSGSEG NBLK4096 NBLK2048 NBLK1024 NQUEUE"
 2 VAL=" 32768  600    16384   16      128      100      64"
 3 CHANGE=

 4 cd /etc/conf/cf.d
 5 set $VAL
 6 for i in $PARA; do
```

```
 7      x=`./configure -y $i`
 8      if [ $x -lt $1 ]
 9      then
10          echo "Adjusting parameter $i from $x to $1"
11          CHANGE="$i=$1 $CHANGE"
12      fi
13      shift
14  done

15  if [ -n "$CHANGE" ]
16  then
17       ./configure $CHANGE
18  fi
```