第二篇　Linux高级语言及管理编程

第5章　外 壳 编 程

在DOS 中，你可能会从事一些例行的重复性命令，此时你会将这些重复性的命令写成批处理命令，只要执行这个批处理命令就等于执行这些命令。在 Linux系统中也有类似的批处理命令，它的功能比起DOS的批处理命令更为强大，相对也较为复杂，已经和一般的高级语言不相上下。这些批处理命令在Linux中叫做外壳脚本（外壳Script ）。

外壳脚本是以文本方式储存的，而非二进制文件。所以外壳脚本必须在 Linux系统的外壳下解释执行。不同外壳的脚本大多会有一些差异，所以我们不能将写给 A 外壳的脚本用B 外壳执行。而在Linux系统中大家最常使用是Bourne 外壳以及C 外壳，所以本章结合这两个外壳的相同点和不同点来介绍外壳编程。

5.1　创建和运行外壳程序

5.1.1　创建外壳程序

你可以用任何的编辑器编辑外壳程序，只需将要执行的外壳或 Linux命令写入外壳程序即可。例如，假设你的系统在启动时挂接有一台 CD-ROM 驱动器，而你想更换驱动器中的CD ，并读取其中的内容。一种方法是：首先把想要读取的 CD 放入 CD-ROM 驱动器中，然后使用unmount 命令卸载CD-ROM 驱动器，最后再使用mount 命令挂接CD-ROM 驱动器。命令如下：

```
umount /dev/cdrom
mount -t iso9660 /dev/cdrom /cdrom
```

你可以创建一个包含这两个命令的文件名为 remount 的外壳程序，而不必在每次更换 CD 都重复执行这两个命令。

5.1.2　运行外壳程序

如何运行我们已经写好的外壳脚本呢？可以有四种方法，下面分别介绍这几种方法：

1. 可以把外壳脚本的权限设置为可执行，这样就可以在外壳提示符下直接执行。我们可以使用下列命令更改外壳脚本的权限：

chmod u+x filename	只有自己可以执行，其他人不能执行。
chmod ug+x filename	只有自己以及同一工作组的人可以执行，其他人不能执行。
chmod +x filename	所有人都可以执行。

而我们如何指定使用哪一个外壳来解释执行外壳脚本呢？几种基本的指定方式如下所述：

1) 如果外壳脚本的第一个非空白字符不是“ # ”，则它会使用Bourne 外壳。

2) 如果外壳脚本的第一个非空白字符是“ # ”，但不以“ #! ”开头时，则它会使用C 外壳。

3) 如果外壳脚本以“ #! ”开头，则“ #! ”后面所跟的字符串就是所使用的外壳的绝对路径

名。Bourne 外壳的路径名称为/bin/sh ，而C 外壳则为/bin/csh。

例如：

1) 如使用Bourne 外壳，可用以下方式：

```
echo enter filename
```

或者

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

2) 如使用C 外壳，可用以下方式：

```
# C 外壳Script
```

或者

```
#!/bin/csh
```

3) 如使用/etc/perl 作为外壳，可用以下方式：

```
#! /etc/perl
```

2. 第二种方法是执行外壳脚本想要执行的外壳，其后跟随外壳脚本的文件名作为命令行参数。例如，使用tcsh 执行上面的外壳脚本：

```
tcsh remount
```

此命令启动一个新的外壳，并令其执行remount 文件。

3. 第三种方法是在pdksh 和bash下使用. 命令，或在tcsh下使用source 命令。例如，在pdksh和bash下执行上面的外壳脚本：

```
.remount
```

或在tcsh下执行上面的外壳脚本：

```
source remount
```

4. 第四种方法是使用命令替换。

这是一个相当有用的方法。如果想要使某个命令的输出成为另一个命令的参数时，就可以使用这个方法。我们将命令列于两个 ` 号之间，而外壳会以这个命令执行后的输出结果代替这个命令以及两个 ` 符号。例如：

```
str='Current directory is `pwd`
echo $str
```

结果如下：

```
Current directory is /users/cc/mgtsai
```

在上面的例子中，pwd 这个命令输出/users/cc/mgtsai，而后整个字符串代替原来的‘ pwd ’设定str 变量，所以str 变量的内容则会包括pwd 命令的输出。

5.2 使用外壳变量

就像其他的任何高级语言一样，在外壳脚本中使用变量也是十分重要的。

5.2.1 给变量赋值

在pdksh 和bash中，给变量赋值的方法是一样的，既在变量名后跟着等号和变量值。例如，想要把5赋给变量count，则使用如下的命令：

```
count=5      （注意，在等号的两边不能有空格）
```

在tcsh中，可以使用如下的命令：

```
set count = 5
```

因为外壳语言是一种不需要类型检查的解释语言，所以在使用变量之前无须事先定义，这

和C 或 Pascal语言不一样。这也说明你可以使用同一个变量来存储字符串或整数。给字符串赋值的方法和给整数赋值的方法一样。例如：

```
name=Garry              （在pdksh 和bash中）
set name = Garry        （在tcsh中）
```

5.2.2 读取变量的值

可以使用$读取变量的值。例如，用如下的命令将 count 变量的内容输出到屏幕上：

```
echo $count
```

5.2.3 位置变量和其他系统变量

位置变量用来存储外壳程序后面所跟的参数。第一个参数存储在变量 1中，第二个参数存储在变量2中，依次类推。这些变量为系统保留变量，所以你不能为这些变量赋值。同样，你可以使用$来读取这些变量的值。例如，你可以编写一个外壳程序 reverse，执行过程中它有两个变量。输出时，将两个变量的位置颠倒。

```
#program reverse，  prints the command line parameters out in reverse order
echo "$2" "$1"
```

在外壳下执行此外壳程序：

```
reverse hello there
```

其输出如下：

```
there hello
```

除了位置变量以外，还有其他的一些系统变量，下面分别加以说明：

• 有些变量在启动外壳时就已经存在于系统中，你可以使用这些系统变量，并且可以赋予新值：

$HOME	用户自己的目录。
$PATH	执行命令时所搜寻的目录。
$TZ	时区。
$MAILCHECK	每隔多少秒检查是否有新的邮件。
$PS1	在外壳命令行的提示符。
$PS2	当命令尚未打完时，外壳要求再输入时的提示符。
$MANPATHman	指令的搜寻路径。

• 有些变量在执行外壳程序时系统就设置好了，并且你不能加以修改：

$#	存储外壳程序中命令行参数的个数。
$?	存储上一个执行命令的返回值。
$0	存储外壳程序的程序名。
$*	存储外壳程序的所有参数。
“ $@ ”	存储所有命令行输入的参数，分别表示为(“ $1 ” “ $2 ” ...)。
$$	存储外壳程序的PID。
$!	存储上一个后台执行命令的PID。

5.2.4 引号的作用

在外壳编程中，各种不同的引号之间的区别是十分重要的。单引号（‘ ’）、双引号（“”）和

反斜杠（\）都用作转义。

• 这三者之中，双引号的功能最弱。当你把字符串用双引号括起来时，外壳将忽略字符串中的空格，但其他的字符都将继续起作用。双引号在将多于一个单词的字符串赋给一个变量时尤其有用。例如，把字符串hello there赋给变量greeting时，应当使用下面的命令：

```
greeting="hello there"              （在 bash 和pdksh环境下）
set greeting = "hello there"        （在 tcsh环境下）
```

这两个命令将hello there作为一个单词存储在greeting 变量中。如果没有双引号， bash 和pdksh将产生语法错，而tcsh则将hello 赋给变量greeting 。

• 单引号的功能则最强。当你把字符串用单引号括起来时，外壳将忽视所有单引号中的特殊字符。例如，如果你想把登录时的用户名也包括在 greeting变量中，应该使用下面的命令：

```
greeting="hello there $LOGNAME"          （在 bash 和pdksh环境下）
set greeting="hello there $LOGNAME"      （在 tcsh环境下）
```

这将会把hello there root存储在变量greeting 中，如果你是以root 身份登录的话。但如果你在上面使用单引号，则单引号将会忽略 $符号的真正作用，而把字符串hello there $LOGNAME存储在greeting 变量中。

• 使用反斜杠是第三种使特殊字符发生转义的方法。反斜杠的功能和单引号一样，只是反斜杠每次只能使一个字符发生转义，而不是使整个字符串发生转义。请看下面的例子：

```
greeting=hello\ there       (在bash和pdksh环境下)
set greeting=hello\ there   (在tcsh环境下)
```

在命令中，反斜杠使外壳忽略空格，从而将hello there作为一个单词赋予变量greeting。

当你想要将一个特殊的字符包含在一个字符串中时，反斜杠就会特别地有用。例如，你想把一盒磁盘的价格$5.00赋予变量disk_price，则使用如下的命令：

```
disk_price=\$5.00       (在 bash 和pdksh环境下)
set disk_price = \$5.00  (在tcsh环境下)
```

如果没有反斜杠，外壳就会试图寻找变量5，并把变量5的值赋给disk_price。

5.3 数值运算命令

如果需要处理数值运算，我们可以使用 expr命令，下面列出可以使用的数值运算符及其用法：

```
expr expression
```

说明：

expression是由字符串以及运算符所组成的，每个字符串或是运算符之间必须用空格隔开。下面列出了运算符的种类及功能，运算符的优先顺序以先后次序排列，可以利用小括号来改变运算的优先次序。其运算结果输出到标准输出设备上。

: 字符串比较。比较的方式是以两字符串的第一个字母开始，以第二个字符串的最后一个字母结束。如果相同，则输出第二个字串的字母个数，如果不同则返回 0 。

* 乘法

/ 除法

% 取余数

+ 加法

- 减法

< 小于

<= 小于等于
= 等于
!= 不等于
>= 大于等于
> 大于
& AND运算
| OR运算

注意 当expression中含有*、(、)等符号时，必须在其前面加上\，以免被外壳解释成其他意义。

例如：

```
expr 2 \* \( 3 + 4 \)
```

输出结果为14。

test命令

在bash 和pdksh环境中，test命令用来测试条件表达式。其用法如下：

```
test expression
```

或者

```
[ expression ]
```

test命令可以和多种系统运算符一起使用。这些运算符可以分为四类：整数运算符、字符串运算符、文件运算符和逻辑运算符。

1) 整数运算符

int1 -eq int2	如果int1 和int2相等，则返回真。
int1 -ge int2	如果int1 大于等于int2，则返回真。
int1 -gt int2	如果int1 大于int2，则返回真。
int1 -le int2	如果int1小于等于int2，则返回真。
int1 -lt int2	如果int1小于int2，则返回真。
int1 -ne int2	如果int1 不等于int2，则返回真。

2) 字符串运算符

str1 = str2	如果str1 和str2相同，则返回真。
str1 != str2	如果str1 和str2不相同，则返回真。
str	如果str 不为空，则返回真。
-n str	如果str 的长度大于零，则返回真。
-z str	如果str 的长度等于零，则返回真。

3) 文件运算符

-d filename	如果filename 为目录，则返回真。
-f filename	如果filename 为普通的文件，则返回真。
-r filename	如果filename 可读，则返回真。
-s filename	如果filename 的长度大于零，则返回真。
-w filename	如果filename 可写，则返回真。
-x filename	如果filename 可执行，则返回真。

4) 逻辑运算符

! expr	如果expr 为假，则返回真。

expr1 -a expr2　　如果expr1 和expr2同时为真，则返回真。
expr1 -o expr2　　如果expr1 或 expr2有一个为真，则返回真。

tcsh中没有test命令，但它同样支持表达式。 tcsh支持的表达式形式基本上和 C语言一样。这些表达式大多数用在if 和while命令中。

tcsh表达式的运算符也分为整数运算符、字符串运算符、文件运算符和逻辑运算符四种。

1) 整数运算符

int1 <= int2　　如果int1小于等于int2，则返回真。
int1 >= int2　　如果int1 大于等于int2，则返回真。
int1 < int2　　如果int1小于等于int2，则返回真。
int1 > int2　　如果int1 大于int2，则返回真。

2) 字符串运算符

str1 == str2　　如果str1 和str2相同，则返回真。
str1 != str2　　如果str1 和str2不相同，则返回真。

3) 文件运算符

-r file　　如果file可读，则返回真。
-w file　　如果file可写，则返回真。
-x file　　如果file可执行，则返回真。
-e file　　如果file存在，则返回真。
-o file　　如果当前用户拥有file ，则返回真。
-z file　　如果file 长度为零，则返回真。
-f file　　如果file 为普通文件，则返回真。
-d file　　如果file 为目录，则返回真。

4) 逻辑运算符

exp1 || exp2　　如果exp1 为真或exp2 为真，则返回真。
exp1 && exp2　　如果exp1 和exp2同时为真，则返回真。
! exp　　如果exp 为假，则返回真。

5.4 条件表达式

bash、pdksh和tcsh 都有两种条件表达方法，即if 表达式和case 表达式。

5.4.1 if 表达式

bash、pdksh和tcsh 都支持嵌套的if...then...else 表达式。bash 和pdksh 的if 表达式如下：

```
if [ expression ]
then
commands
    elif [ expression2 ]
then
commands
else
commands
fi
```

elif 和else 在if 表达式中均为可选部分。elif是else if的缩写。只有在if表达式和任何在它之

前的elif表达式都为假时，才执行elif。fi关键字表示if表达式的结束。

在tcsh中，if表达式有两种形式。第一种形式为：

```
if (expression1) then
commands
else if (expression2) then
commands
else
commands
endif
```

tcsh的第二种形式是第一种形式的简写。它只执行一个命令，如果表达式为真，则执行，如果表达式为假，则不做任何事。其用法如下：

```
if (expression) command
```

下面是一个bash 或 pdksh 环境下if 表达式的例子。它用来查看在当前目录下是否存在一个叫.profile的文件：

```
if [ -f .profile ]
then
echo "There is a .profile file in the current directory."
else
echo "Could not find the .profile file."
fi
```

在tcsh环境下为：

```
#
if ( { -f .profile } ) then
echo "There is a .profile file in the current directory."
else
echo "Could not find the .profile file."
endif
```

5.4.2 case 表达式

case表达式允许你从几种情况中选择一种情况执行。外壳中的 case表达式的功能要比Pascal或C语言的case 或switch语句的功能稍强。这是因为在外壳中，你可以使用 case表达式比较带有通配符的字符串，而在Pascal 和C语言中你只能比较枚举类型和整数类型的值。

bash 和pdksh的case表达式如下：

```
case string1 in
str1)
commands;;
str2)
commands;;
*)
commands;;
esac
```

在此，将string1 和str1、str2比较。如果str1 和str2中的任何一个和strings1相符合，则它下面的命令一直到两个分号(;;)将被执行。如果str1 和str2中没有和strings1相符合的，则星号（*）下面的语句被执行。星号是缺省的case条件，因为它和任何字符串都匹配。

tcsh的选择语句称为开关语句。它和C语言的开关语句十分类似。

```
switch (string1)
case str1:
```

```
statements
breaksw
case str2:
statements
breaksw
default:
statements
breaksw
endsw
```

在此，string1和每一个case关键字后面的字符串相比较。如果任何一个字符串和 string1相匹配，则其后面的语句直到 breaksw将被执行。如果没有任何一个字符串和 string1匹配，则执行default后面直到breaksw的语句。

下面是bash 或 pdksh环境下case表达式的一个例子。

它检查命令行的第一个参数是否为-i 或e。如果是-i，则计算由第二个参数指定的文件中以i开头的行数。如果是 -e，则计算由第二个参数指定的文件中以 e开头的行数。如果第一个参数既不是 -i也不是 -e，则在屏幕上显示一条的错误信息。

```
case $1 in
-i)
count='grep ^i $2 | wc -l '
echo "The number of lines in $2 that start with an i is $count"
;;
-e)
count='grep ^e $2 | wc -l'
echo "The number of lines in $2 that start with an e is $count"
;;
* )
echo "That option is not recognized"
;;
esac
```

此例在tcsh 环境下为：

```
# remember that the first line must start with a # when using tcsh
switch ( $1 )
case -i | i:
set count = 'grep ^i $2 | wc -l'
echo "The number of lines in $2 that begin with i is $count"
breaksw
case -e | e:
set count = 'grep ^e $2 | wc -l'
echo "The number of lines in $2 that begin with e is $count"
breaksw
default:
echo "That option is not recognized"
breaksw
endsw
```

5.5 循环语句

外壳中提供了几种循环语句，最为常用的是for表达式。

5.5.1 for 语句

bash 和pdksh中有两种使用for 语句的表达式。

第一种形式是：

```
for var1 in list
do
commands
done
```

在此形式时，对在list 中的每一项，for语句都执行一次。List可以是包括几个单词的、由空格分隔开的变量，也可以是直接输入的几个值。每执行一次循环，var1都被赋予list中的当前值，直到最后一个为止。

第二种形式是：

```
for var1
do
statements
done
```

使用这种形式时，对变量var1中的每一项，for语句都执行一次。此时，外壳程序假定变量var1中包含外壳程序在命令行的所有位置参数。

一般情况下，此种方式也可以写成：

```
for var1 in "$@"
do
statements
done
```

在tcsh中，for循环语句叫做foreach。其形式如下：

```
foreach name (list)
commands
end
```

下面是一个在bash 或 pdksh环境下的例子。

此程序可以以任何数目的文本文件作为命令行参数。它读取每一个文件，把其中的内容转换成大写字母，然后将结果存储在以.caps作为扩展名的同样名字的文件中。

```
for file
do
tr a-z A-Z < $file >$file.caps
done
```

在tcsh环境下，此例子可以写成：

```
#
foreach file ($*)
tr a-z A-Z < $file >$file.caps
end
```

5.5.2 while 语句

while 语句是另一种循环语句。当一个给定的条件为真时，则一直循环执行下面的语句直到条件为假。在bash 和pdksh环境下，使用while 语句的表达式为：

```
while expression
do
statements
```

```
done
```

而在tcsh中，while 语句为：

```
while (expression)
statements
end
```

下面是在bash 和pdksh中while语句的一个例子。程序列出所带的所有参数，以及他们的位置号。

```
count=1
while [ -n "$*" ]
do
echo "This is parameter number $count $1"
shift
count='expr $count + 1'
done
```

其中shift命令用来将命令行参数左移一个。

在tcsh中，此例子为：

```
#
set count = 1
while ( "$*" != "" )
echo "This is parameter number $count $1"
shift
set count = 'expr $count + 1'
end
```

5.5.3 until 语句

until 语句的作用和while语句基本一样，只是当给定的条件为假时，执行 until 语句。until 语句在bash 和pdksh中的写法为：

```
until expression
do
commands
done
```

让我们用until语句重写上面的例子：

```
count=1
until [ -z "$*" ]
do
echo "This is parameter number $count $1"
shift
count='expr $count + 1'
done
```

在应用中，until语句不是很常用，因为until语句可以用while语句重写。

5.6 shift 命令

bash、 pdksh和tcsh都支持shift 命令。shift 命令用来将存储在位置参数中的当前值左移一个位置。例如当前的位置参数是：

```
$1 = -r $2 = file1 $3 = file2
```

执行shift 命令：

```
shift
```

位置参数将会变为：

```
$1 = file1 $2 = file2
```

你也可以指定shift 命令每次移动的位置个数。下面的例子将位置参数移动两个位置：

```
shift 2
```

下面是一个应用shift 命令的例子。此程序有两个命令行参数，一个代表输入文件，另一个代表输出文件。程序读取输入文件，将其中的内容转换成大写，并将结果存储在输出文件中。

```
while [ "$1" ]
do
if [ "$1" = "-i" ] then
infile="$2"
shift 2
elif [ "$1" = "-o" ]
then
outfile="$2"
shift 2
else
echo "Program $0 does not recognize option $1"
fi
done
tr a-z A-Z <$infile >$outfile
```

5.7 select 语句

select 语句只存在于pdksh中，在bash 或 tcsh中没有相似的表达式。select 语句自动生成一个简单的文字菜单。其用法如下：

```
select menuitem [in list_of_items]
do
commands
done
```

其中方括号中是select语句的可选部分。

当select语句执行时， pdksh为在list_of_items中的每一项创建一个标有数字的菜单项。list_of_items可以是包含几个条目的变量，就像choice1 choice2，或者是直接在命令中输入的选择项，例如：

```
select menuitem in choice1 choice2 choice3
```

如果没有list_of_items，select语句则使用命令行的位置参数，就像for表达式一样。

一旦你选择了菜单项中的一个， select语句就选中的菜单项的数字值存储在变量 menuitem 中。然后你可以利用do中的语句来执行选中的菜单项要执行的命令。

下面是select语句的一个例子。

```
select menuitem in pick1 pick2 pick3
do
echo "Are you sure you want to pick $menuitem"
read res
if [ $res = "y" -o $res = "Y" ]
then
break
fi
done
```

5.8 repeat 语句

repeat 语句只存在于tcsh中，在pdksh 和bash中没有相似的语句。repeat 语句用来使一个单一的语句执行指定的次数。repeat 语句如下：

```
repeat count command
```

下面给出repeat 语句的一个例子。它读取命令行后的一串数字，并根据数字在屏幕上分行输出句号。

```
#
foreach num ($*)
repeat $num echo -n "."
echo ""
end
```

任何repeat 语句都可以用while 或 for语句重写。repeat语句只是更加方便而已。

5.9 子函数

外壳语言可以定义自己的函数，就像在C 或其他语言中一样。使用函数的最大好处就是程序更为清晰可读。下面是如何在bash 和pdksh中创建一个函数：

```
fname () {
shellcommands
}
```

在pdksh中也可以使用如下的形式：

```
function fname {
shellcommands
}
```

使用函数时，只须输入以下的命令：

```
fname [parm1 parm2 parm3 ...]
```

tcsh 外壳中不支持函数。

你可以传递任何数目的参数给一个函数。函数将会把这些参数视为位置参数。请看下面的例子。

此例子包括四个函数：upper ()、lower ()、print ()和usage_error ()，他们的任务分别是：将文件转换成大写字母、将文件转换成小写字母、打印文件内容和显示出错信息。 upper ()、lower ()、print ()都可以有任意数目的参数。如果将此例子命名为 convert，你可以在外壳提示符下这样使用该程序：convert -u file1 file2 file3。

```
upper () {
shift
for i
do
tr a-z A-Z <$1 >$1.out
rm $1
mv $1.out $1
shift
done; }
lower () {
shift
for i
do
```

```
tr A-Z a-z <$1 >$1.out
rm $1
mv $1.out $1
shift
done; }
print () {
shift
for i
do
lpr $1
shift
done; }
usage_error () {
echo "$1 syntax is $1 <option> <input files>"
echo ""
echo "where option is one of the following"
echo "p — to print frame files"
echo "u — to save as uppercase"
echo "l — to save as lowercase"; }
case $1
in
p | -p) print $@;;
u | -u) upper $@;;
l | -l) lower $@;;
*) usage_error $0;;
esac
```

虽然外壳语言功能强大而且简单易学，但你会发现在有些情况下，外壳语言无法解决你的问题。这时，你可以选择Linux系统中的其他语言，例如C 和C++、gawk、以及Perl等。