Shell 及其编程

在 Linux 中，Shell 即表示命令解释程序，又表示命令编程语言。Shell 具有程序语言的特性，包含变量、运算、函数和控制结构，可以组成复杂、灵活的命令流程，一方面可用于实现高级的操作系统功能，另一方面也可用作批处理作业的控制语言——描述作业的任务流程。为了叙述方便，我们称 Shell 程序为 Shell 脚本，称 Shell 命令解释程序为 Shell。Shell 脚本由 Shell 解释执行。Shell 的程序流程与《实验指导——Linux 用户管理》中图 2 基本相同，所不同的是：在提供命令界面时，它等待用户从终端输入 Shell 命令行；而在处理 Shell 脚本时，则是从脚本文件中读入 Shell 语句。

Linux 中有多种 Shell 可供用户选择，它们功能相似，语法稍有区别。其中主要有：

（1）Bourne Shell（B-Shell）

由 AT&T Bell 实验室 Slephen Bourne 主持开发的，是 UNIX 系统上最初的和最基本 Shell（简称 sh），其扩展版本 Bourne Again Shell（简称 bash）是 Linux 的默认 Shell。它的命令提示符是$。

（2）C Shell（C-Shell）

由加州大学 Berkeley 分校分校 Bill Joy 主持开发的主要运行于 BSD UNIX 系统上的 Shell（简称 csh），更多的考虑了用户界面的友好性。它的命令提示符是%。

（3）Korn Shell（K-Shell）

由 AT&T Bell 实验室的 David korn 开发的新 Shell（简称 ksh），结合了 C-Shell 与 B-Shell 的优点，与 B-Shell 兼容。它的命令提示符也是$。

下面以 bash 为例，学习 Shell 命令及其编程。

1.Shell 命令格式

Shell 命令的一般格式为：

cmd [options] [arguments]

其中，cmd 是命令名；options 是选择项；arguments 是参数。最简单的 Shell 命令只有命令名，复杂的 shell 命令可以有多个选择项和参数。选择项和参数都作为 Shell 命令执行时的输入，它们之间用空格分隔开。

Shell 通过提示符来提示用户键入命令行。每当接受一个命令行后，Shell 通常会创建新进程运行一个 Shell 的实例——称为子 Shell，由子 Shell 去执行此命令行。等命令执行完毕后，子 Shell 自动消亡，返回到原 Shell（即父 Shell），继续显示提示符。显示提示符的 Shell 习惯上称为当前 Shell。

Shell 允许在一个命令行中包含多个命令，各命令之间用“；”作为分隔符，按下回车键后，Shell 将顺序地执行每条命令。

Shell 命令中可以使用通配符：

*	匹配任何数目的字符。
?	匹配任何单字符。
[…]	匹配任何包含在括号里的单字符。

例如：

ls abc*	列出所有的名字以 abc 开头的文件清单。
ls a?c	列出所有的名字为 3 个字符并且以 a 开头、以 c 结尾的文件清单。
ls ab[1-3]	列出名字为 ab1、ab2、ab3 的文件清单。

2. 执行 Shell 命令

Shell 命令的执行方式有两种：前台方式和后台方式。前台方式下，命令在执行过程中可以通过终端与用户进行交互，接受用户从终端输入的数据，在终端上显示输出数据。而后台方式下，命令不能使用终端与用户进行交互。

在提示符下，输入命令行后，按下回车键后，就是使用前台方式执行命令。如果在命令行后添加一个结尾字符“&”，就是使用后台方式执行命令。这时系统会返回执行后台命令的进程 ID，以便用户对这个后台进程施加控制。

Shell 命令可以分为两种：内部命令和外部命令。内部命令在 Shell 内部实现，可以直接执行，如 echo、export、let、cd、pwd 等，外部命令需要调用相应的脚本文件或可执行目标文件，如 ls、rm、cat、more、gcc、date、chmod 等。Shell 查找执行文件的目录在环境变量 PATH（参见后面的环境变量小节）指定。如果需要执行的文件不在 PATH 指定的目录下，则需要指明文件的路径名。另外，脚本文件在执行前需要使用命令 chmod 将其转变成可执行文件。

例如：

chmod +x abc

将脚本文件 abc 转变成可执行文件。

运行一个脚本文件或应用程序的执行文件也可以使用两种不同的命令形式，一种是直接的命令形式，将执行文件名作为命令名使用；另一种是间接的命令形式，将执行文件名作为命令解释程序（bash）的参数使用。后者可以使用 Shell 的调试参数（参见后面的调试小节）。

例如：

```
$ ./abc                          ←直接命令形式，首位的$是提示符
$ bash  ./abc                    ←间接命令形式
```

在一条 Shell 命令中可以嵌入执行另一 Shell 命令。使用一对反单引号将需要嵌入执行的命令包括在其中，Shell 执行其中的命令，并将执行的结果替代该命令的字符串（包括反单引号）。需要注意的是，只允许嵌入一层。

例如：

```
$echo date
date                             ←显示字符串 date
$echo ｀date｀
Fri Jun 19 16:03:22 EST 2015     ←显示命令 date 的执行结果
$
```

3.输入/输出

Linux 的命令或脚本程序都具有三个标准的输入/输出端口。

- stdin　　标准输入端口，数据从终端的键盘上输入。
- stdout　　标准输出端口，数据输出到终端的屏幕上。
- stderr　　错误输出端口，错误信息输出到终端的屏幕上。

（1）输出转向

使用输出转向符“>”或“>>”可以将一条命令 command 的执行结果改送到一个文件中保存，而不是直接送到屏幕上显示。其命令格式为：

```
$command > file
```

或

```
$command >> file
```

“>”的功能是将输出送到 file 文件中，并覆盖原来的数据。“>>”的功能也是将输出转向

到 file 文件中，但输出的信息附加到该文件尾部。

例如：将文件 w1 和 w2 合并成一个文件 myfile。

```
$cat w1 > myfile
$cat w2 >> myfile
$
```

（2）输入转向

使用输入转向符“<”可以将一条命令 command 接收的输入数据改为从一个文件中读取，而不是接收键盘的输入数据。其命令格式为：

```
$command < file
```

“<”的功能是直接从 file 文件中获取输入数据，而不等待键盘的输入。

例如：显示文件 myfile 中包含的文本行数。

```
$wc -l < myfile
```

（3）错误输出转向

使用“2>”可以将一条命令的错误信息改送到一个文件中保存，而不是直接送到屏幕上显示。其命令格式为：

```
$command 2> file
```

“2>”的功能使命令 command 的错误信息就写入了文件 file 中。

例如：运行目标程序 myprog 并使其错误输出送到文件 err_file 中。

```
$myprog 2> err_file
```

（4）管道线

管道线“|”将前一条命令的标准输出端口与后一条命令的标准输入端口连接起来，即前一条命令执行中产生的结果数据通过标准输出端口送给后一条命令，作为该命令的输入数据，后一条命令则是通过标准输入端口来接受输入数据。使用管道线“|”可以将多个命令前后衔接起来，形成一个管道链：

命令 1 | 命令 2 | ... | 命令 n

管道链中的每一条命令都作为一个单独的进程运行，每一条命令的输出作为下一条命令的输入。需要注意的是管道线只能作用于标准输入/输出端口。

管道线实现了命令的组合，可以用于实现一些复合的高级功能，从而增强了 Shell 命令的灵活性。例如：下面的命令行

```
cat prog | more
```

实现了分页显示文件 prog 的内容。

如果希望能保存管道链中间某点的输出信息，使用 tee 命令就可满足这一要求。tee 命令的功能是将标准输入同时送到标准输出和指定的文件中。

例子：

```
cat prog | grep “string” | tee tmp | more
```

该命令在文件 prog 中查找所有包含字符串“string”的行，查找结果保存到文件 tmp 中，同时分页显示在屏幕上。

4.变量和环境变量

Shell 命令或脚本中可以使用变量，但这些变量只能是字符串变量，即变量的值表现为字符串。

（1）定义和引用

Shell 变量的定义和赋值可以同时进行，其格式为：

```
name=string                    ←在＝两边不能有空格
```

它定义了一个名字为 name 的变量，并且变量的值为字符串 string。在 Shell 命令或脚本中可以引用已定义变量的值，引用的格式为：

$name

引用的结果是用变量 name 的值 string 替代$name。例如：

ls $name

上面命令等价于：

ls string

如果引用一个未定义的变量时，其值视为空字符串。

（2）算术运算

当变量的值是整数的数值字符串时，可以使用 let 命令进行加（+）、减（-）、乘（*）、整除（/）和求余（%）的算术运算。例如：

```
$ name=10；let name=$name+12          ←在运算符两边不能有空格。
$ echo $name
22                                    ←变量 name 的值
```

（3）环境变量

Shell 变量的值仅限于定义变量的 Shell 进程使用，只有全局 Shell 变量的值可以传递给子 Shell 使用。全局 Shell 变量又称为环境变量，使用下面的命令：

export　name

将变量 name 指定为环境变量。例如：

```
$ name=string；export name
$ ./abc                               ←执行当前目录下的 abc 脚本文件
```

当前 Shell 将创建一个子 Shell 执行 abc 脚本文件，由于 name 已经成为环境变量，在 abc 中可以引用 name 的值 string。

当用户登录 Linux 系统时，系统已经建立多个环境变量，可以使用命令 env 查看。其中几个主要的环境变量有：

- LOGNAME　用户的注册名。
- HOME　定义了用户的主目录的路径。
- PATH　定义了执行一个命令时所要查找的路径。
- CPATH　定义了 C 编译器查找.h 文件时所要搜索的路径。
- LPATH　定义了 ld 连接库函数时所要查找的路径。

不仅脚本程序可以使用环境变量的值，应用程序中也可以通过调用函数 getenv()来引用它们。

（4）特殊变量

另外，Shell 还实现了一些特殊变量供脚本程序使用：

$1，$2，$3，……	表示命令名后面的参数 1，参数 2，参数 3，……
$0	命令名
$#	命令的参数个数
$*	所有的命令参数
$$	执行命令的进程 ID

5.Shell 控制结构

在 Shell 脚本中可以使用与过程式语言（如 Pascal 和 C）类似的控制结构、函数以及注释。注释以字符“#”开始，至行尾结束。

（1）判断结构

if [条件表达式]
then 命令表 1
else 命令表 2
fi

其中，命令表是一组命令序列，各个命令间用“;”或换行符分隔。当条件表达式的值为真时，执行命令表 1，否则执行命令表 2。可以没有 else 部分。条件表达式有三种：

a）字符串比较

str1 = str2	如果字符串 str1 和 str2 相等，则为真。
str1 != str2	如果字符串 str1 和 str2 不相等，则为真。
-z str	如果字符串 str 长度为 0，则为真。
-n str	如果字符串 str 长度不为 0，则为真。

b）数值比较

N1 -eq N2	如果数值 N1 等于 N2，则为真。
N1 -ne N2	如果数值 N1 不等于 N2，则为真。
N1 -gt N2	如果数值 N1 大于 N2，则为真。
N1 -lt N2	如果数值 N1 小于 N2，则为真。
N1 -ge N2	如果数值 N1 大于等于 N2，则为真。
N1 -le N2	如果数值 N1 小于等于 N2，则为真。

c）文件特性测试

-r file	如果文件 file 存在且可读，则为真。
-w file	如果文件 file 存在且可写，则为真。
-x file	如果文件 file 存在且可执行，则为真。
-d file	如果文件 file 是目录文件，则为真。
-c file	如果文件 file 是字符特别文件，则为真。
-b file	如果文件 file 是块特别文件，则为真。

条件表达式还可以用逻辑操作符连接起来，构成组合条件表达式。

!	逻辑“非”操作
&&	逻辑“与”操作
||	逻辑“或”操作

注意，条件表达式中的比较对象、比较运算符、逻辑操作符以及“[”、“]”之间都必须留有空格。

（2）循环结构

有三种循环结构，它们分别是：

a）for 循环结构：

for 循环变量 in 名字表
do
命令表
done

其中，名字表是以空格分隔的字符串序列。Shell 依次将名字表中的字符串赋给循环变量并执行命令表，直到处理完名字表中的所有字符串为止。

例如：copyfile 脚本程序拷贝当前工作目录下的所有文件到 backup 目录下。如果 backup 目录不存在，则创建 backup 目录。

```
$cat copyfile                    ←显示 copyfile 脚本文件
#copy all files in the current directory to the ./backup
```

```
if [ -d $HOME/backup ]
then
  echo "backup directory is existed"
else
  mkdir $HOME/backup;
  echo "make backup directory now"
fi
total=`ls`                          #将当前目录下所有的文件名作为字符串赋值给 total
for file in $total
do
   if [ -f $file ]
   then
      cp $file $HOME/backup/
      echo "$file copied"
   fi
done
echo "***Backup completed***"
$
$ ./copyfile                              ←执行 copyfile 脚本文件
work1 copied
work2 copied
work3 copied
***Backup completed***
$
```

b）while 循环结构

while　表达式
do
　　命令表
done

当表达式为真时，Shell 重复执行命令表，直到表达式为假时结束循环。while 语句的退出状态为命令表中被执行的最后一条命令的退出状态。如果未执行命令表中的任何命令，退出状态为 0。

例如：count 脚本程序计算一个整数序列的和。

```
$cat count                          ←显示 count 脚本文件
if [ $1 -gt $2 ]
then
   echo "arg1 must be smaller than arg2"
   exit
fi
i=$1
sum=0
while [ $i -le $2 ]
do
   let sum=$sum+$i
```

```
    let i=$i+1
done
echo $sum
$
$ ./count 1 10                    ←执行 count 脚本文件
55
```

c）until 循环结构

until　表达式
do
**　命令表**
Done

until 语句与 while 语句的循环判断方式正好相反，当表达式为假时，Shell 重复执行命令表，直到表达式为真时结束循环。

例如：test 是使用 until 循环语句改写 count 程序的结果。

```
$cat test                         ←显示 test 脚本文件
if [ $1 -gt $2 ]
then
   echo "arg1 must be smaller than arg2"
   exit
fi
i=$1
sum=0
until [ $i -gt $2 ]
do
   let sum= $sum+$i
   let i=$i+1
done
echo $sum
$
$ ./test 1 10                     ←执行 test 脚本文件
55
```

在循环结构中可以使用 break 和 continue 语句：

break [n]

break 语句用于从包含该语句的最近一层循环语句中跳出一层，如果指定了 n，则跳出 n 层循环。

continue [n]

continue 语句并不跳出循环，而是重新开始下一轮循环。如果指定了 n，则重新开始第 n 个内层循环。

（3）多分支结构

case　变量　in
模式 1)　命令表 1
;;
模式 2)　命令表 2
;;

……

模式 n)　命令表 n

;;

esac

模式是字符串或包含通配符的字符串。Shell 将变量的值依次与各个模式比较，当找到一个完全匹配的模式后，执行该模式的命令表，执行完后从 case 语句跳出。每个命令表都以“;;”结束，表示跳出开关语句。

（4）函数

定义函数的语法为：

function-name

{

命令表

}

function-name 是函数名，花括号内是函数体。函数必须先定义后调用，函数的调用方式与脚本的执行方式相同：

function-name　[arg ...]

方括号内是调用函数时使用的参数，通过特殊变量$1、$2、...传递给函数。

例如：　tjfile 脚本程序统计指定目录下（包括子目录）的可执行文件数目。

```
$cat tjfile                          ←显示 tjfile 脚本文件
sum=0
function count {                     # 定义函数
    cd $1
    for name in `ls`
    do
        if [ -d $name ]
        then
            count $name              # 遇到子目录，递归调用函数
        elif [ -x $name ]
        then
            let sum+=1
        fi
    done
   cd   ../
}
count $1                             # 调用函数
echo There are $sum executable files.
$
$ ./tjfile   /bin
There are 76 executable files.
$
```

6.调试

调试一个 Shell 脚本程序通常采用以下方法。

（1）在脚本程序中使用输出语句 echo，跟踪关键点的执行情况。这类似于在 C 程序中

使用 printf()来调试程序。

（2）在脚本程序中使用 set 语句，设置一些调试参数，使 Shell 在执行脚本程序的过程中自动输出有关信息。有下列调试参数：

e	如果一个命令失败，就中止该命令程序的执行；
n	读入各命令行但不执行它们；
u	当引用了未定义的变量时，输出报错信息；
v	执行各命令行时，显示其没有进行变量值替代的原始内容；
x	执行各命令行时，显示经过变量值替代后的命令行的内容，用前缀符“+”标示进行了“替代”的命令行。

使用 set 语句设置调试参数：

set -调试参数

使用 set 语句停止调试参数的使用：

set +调试参数

（3）在执行 Shell 脚本程序时指定上述调试参数。

$bash -调试参数 脚本文件名

这等价于在命令文件的第一行插入“set -调试参数”语句。在 set 和 bash 语句中还可以同时设置多个调试参数。

7.作业控制

作业是由一组相关的程序执行任务组成。Shell 脚本能够很好地描述作业的运行流程，使用 Shell 命令描述作业任务，使用 Shell 控制结构描述任务之间的关系及执行流程，使用 Shell 变量和管道线实现任务之间的数据交换。作业通常以后台方式运行。

Linux 还提供了简单的作业控制命令，可以指定运行作业的时间。

（1）at 命令

常用的 at 命令格式为：

at -f fname TIME

Shell 将在指定的时间 TIME 开始执行名为 fname 的脚本文件或其他执行文件。TIME 的表示方法很灵活，读者可以使用帮助命令 man 了解它们。

例如，想在 3 天后的下午 4 点钟运行脚本文件 abc。

at -f abc 4pm + 3 days

想在星期一临晨 1 点运行该作业。

at -f abc 1am MON

如果需要每周的星期一都运行该作业，则在脚本文件 abc 的最后加入上面的命令。

（2）batch 命令

batch 命令可以指定作业在系统负载较轻（即 CPU 利用率低于 80％）的时候运行。常用的 batch 命令格式为：

batch -f fname