第6部分
Shell概述及其应用

2024年6月30日

第6部分　Shell概述及其应用

6.1 Shell语言概述

6.2 Shell特殊字符

6.3 Shell变量

6.4 Shell表达式

6.5 Shell控制结构

6.6 Shell函数

6.1 Shell语言概述

6.1.1 Shell语言的特点

与其他编程语言相比，Shell语言具有如下特点：

(1) Shell是一种解释性语言。

(2) Shell是基于字符串的语言。

(3) Shell是命令级语言。

6.1.2 Shell语言的功能

1 命令解释功能：将用户可读的命令转换成计算机可理解的命令，并控制命令的执行。

2 输入/输出重定向：Linux系统将键盘作为标准输入、显示器作为标准输出。在默认情况下，Shell将命令的输入流定向为键盘、输出流（包括错误流）定向为显示器。当这些定向不能满足用户需求时，用户可以在命令中用符号“>”或“<”重新定向，如定向到文件。

3 管道线处理：利用管道将一个命令的输出送入另一个命令，实现多个命令组合完成复杂命令的功能。

4 系统环境设置：用Shell命令设置环境变量，维护用户的工作环境。

5 程序设计语言：Shell命令本身可以作为程序设计语言，将多个Shell命令组合起来，编写能实现系统或用户所需功能的程序。

6.1.3 Shell程序

Shell具有程序设计功能，如果在使用过程中需要用重复、复杂命令完成某项工作，就可以利用Shell编程实现。这样用户在定义自己的命令同时也实现了系统Shell命令的扩充。使用Shell脚本的最初动机也是为省去手动输入命令的麻烦，Shell脚本将一系列的命令放在一个文件中，就不必每次手动输入同样的命令。

Shell程序也称Shell脚本(script)，是由一系列Shell命令为基本元素构成的文本文件。把用户键入的Shell命令按照控制结构组织到一个文本文件中，批量交给Shell去执行。

6.1.4 Shell程序的建立与执行

同一般的结构化程序不同，Shell程序是按行解释，不需要编译成目标程序，也不需要链接成可执行的目标码，只要Shell脚本是可执行的即可。

Shell脚本是文本文件，因此可以用任何文本编辑器(如vi、emacs等)建立和编辑脚本。Shell脚本文件的名称没有限定的扩展名，通常不带扩展名或带“.sh”扩展名。

Shell脚本的执行方式主要有3种

(1) 将脚本作为可执行文件执行：

```
$ chmod  a+x  hello
$ ./hello
```

(2) 启动一个Shell子进程来执行脚本文件：

```
$ bash hello                    #或bash < hello
```

(3) 让当前Shell进程执行脚本文件：

```
$ . hello                       #注意.后面的空格
```

例6.1 简单Shell程序及执行：

```
$ cat hello                          #hello程序
# This is a shell script to say hello.
echo Hello World!
echo -n “Today is ”
date “+%A, %B %d, %Y.”
```

```
$ . hello                            #运行hello程序
Hello World!
Today is Saturday, October 13, 2007.
```

6.2 Shell特殊字符

Shell定义了一些特殊的字符，称为元字符(meta-characters)，它们对Shell有特殊的含义。Shell在读入命令行后，要先对命令行进行扫描，找出元字符并进行相应的替换或处理，以确定要执行的程序和它的参数及执行方式等。

6.2.1 通配符

通配符用于描述命令中的文件名参数。当Shell在命令的参数中遇到带有通配符的文件名模式时，它将目录中的所有文件与该模式进行匹配，并用匹配的文件名替换参数中的文件名模式。表6-1列出了常用的通配符。

表 6-1 常用的文件名通配符

符号	格 式	含 义
*		匹配任何字符串，包括空字符串
?		匹配任何单个字符
[]		匹配方括号内列出的某个单个字符
,	[字符 1,字符 2,...]	指定多个匹配的字符
-	[开始字符-结束字符]	指定匹配的字符范围
!	[!字符]	指定不匹配的字符

6.2.2 输入/输出重定向

输入/输出重定向的作用是改变命令的输入/输出环境。当Shell在命令行中遇到输入/输出重定向，它将对命令的标准输入/输出文件作相应的更改，然后再执行命令。表6-2列出了常用的输入/输出重定向。

表 6-2 常用的输入/输出重定向

符号	格 式	含 义
<	命令 < 文件	标准输入重定向
>	命令 > 文件	标准输出重定向
2>	命令 2> 文件	标准错误输出重定向
&>	命令 &> 文件	标准输出合并重定向
>>	命令 >> 文件	标准输出附加重定向

例6.2 编写exam文件

```
$ cat exam
 echo “this is the example of  in/out”
 cat hello>hello1
 cat hello1
$ . exam
```

6.2.3 命令执行控制符

命令执行控制符用于控制命令的执行方式，指示Shell何时该执行这个命令以及在何处(前台、后台)执行这个命令。表6-3列出了常用的命令执行控制符。

表 6-3 常用的命令执行控制符

符号	格 式	含 义
;	命令 1; 命令 2	顺序执行命令 1 和命令 2
&&	命令 1 && 命令 2	“逻辑与”执行，若命令 1 执行成功则执行命令 2；否则不执行命令 2
||	命令 1 || 命令 2	“逻辑或”执行。若命令 1 执行成功则不执行命令 2；否则继续执行命令 2
&	命令&	后台执行命令

1. 顺序执行

“;”是顺序执行符，它将两个或多个命令组合在一个命令行中，指示Shell顺序执行这些命令。

例6.3 转到上一级目录，显示目录的路径名和目录的文件列表：

```
$ cd ..; pwd; ls
```

2. 条件执行

“&&”是逻辑与执行符，它将两个或多个命令组合在一个命令行中，指示Shell依次执行这些命令直到某个命令失败为止。

“||”是逻辑或执行符，它将两个或多个命令组合在一个命令行中，指示Shell依次执行这些命令直到某个命令成功为止。

例6.4 将文件file1复制到file2，如果成功则删除file1：

```
$ cp file1 file2 && rm file1
```

将文件file1复制到file2，如果失败则执行chmod命令：

```
$ cp file1 file2 || chmod o+r file1
```

3. 后台执行

“&”是后台执行符，它指示Shell将该命令放在后台执行。后台执行的命令不占用终端与用户交互，因此Shell在执行后台命令后可以立即返回提示符。

例6.5 后台执行压缩文件：

```
$ gzip hello &
```

在后台顺序执行两命令，20秒后跳出提示信息“Tea is ready”：

```
$ ( sleep 20; echo Tea is ready ) &
```

6.2.4 命令组合符

命令组合符的作用是指示Shell将多个命令组合在一起执行。组合的目的是对这些命令统一进行某种操作，如管道、后台运行、输入/输出重定向等。

命令的组合形式有两种：{ 命令; 命令; … } 和 (命令; 命令; …)。

例6.6 将两命令的输出送到mydoc，mydoc的第1行是Report，后面是file的内容：

```
$ ( echo Report; cat file ) > mydoc
```

6.2.5 命令替换符

当一个字符串被括在反撇号“`”中时，该字符串将先被Shell作为命令解释执行，然后用命令执行后的输出结果替换‘字符串’。

例6.7 命令替换符的用法：

```
$ echo Today is `date '+%A'`      #替换后为echo Today is Thursday

Today is Thursday
```

6.2.6 其他元字符

表 6-4 其他元字符

符号	含义
#	注释符，其后的内容被忽略
$	变量引用符
空格	分隔符，分隔命令名、选项和参数

6.2.7 元字符的引用

需要引用元字符的原始含义而不是它的特殊含义时，必须用引用符对它进行转义，消除其特殊含义。当Shell遇到引用符时，它将该引用符作用范围内的字符看作是普通字符。常用的引用符有3种，即转义符、单引号和双引号。表6-5列出了它们的含义。

表 6-5 引 用 符

符 号	含 义
\	转义符，消除其后面的单个元字符的特殊含义
“...”	消除双引号中的大部分元字符的特殊含义，不能消除的字符有：$、`、”、\
‘...’	消除单引号中的所有元字符(’本身除外)的特殊含义，即单引号内的内容原样不动

例6.8 在命令行中引用元字符：

```
$ echo \\\*                          #第1个和第3个\字符是转义符
  \*
$ echo “* is a wildcard.”   #消除*字符的特殊含义
  * is a wildcard.
$ echo ‘The prompter is “$”’   #消除双引号字符的特殊含义
  The prompter is “$”
$ echo “Don’t do that!”                  #消除单引号字符的特殊含义
  Don’t do that!
$ echo “Name  ID  Age  Class”       #消除空格符的特殊含义
  Name  ID  Age  Class
$ echo Name  ID  Age  Class       #未转义的空格被看作是分隔符
  Name ID Age Class
```

6.3 Shell变量

Shell提供了定义和使用变量的功能。用户可以将一些常用的数据存放在Shell变量中，并在命令行中引用这些变量。使用变量可以定制Shell的行为，方便Shell的使用和编程。

6.3.1 变量的定义与使用

变量是具有名字的一块内存空间，用于保存程序中要用到的数据。Shell是基于**字符串**的编程语言，Shell的变量只能存储字符串，因此Shell变量只有两种类型，即**字符串**和**字符串数组**。

1. 定义变量

在Shell中，对变量的定义与赋值是同时完成的。有3种方式可以为变量赋值：

(1) 用赋值命令，格式是：

变量名=字符串

注意：变量的名字必须以字母或下划线开头，可以包括字母、数字和下划线。赋值号“=”两边不能有空格。字符串中含有空格，应用引号将字符串括起。

例6.9 用变量赋值命令定义变量：

```
$ nodehost=beijing.WEB
$ user="zhang ming"
$ path=/bin:/usr/bin:/etc/bin
$ count=10
```

(2) 用read命令，从标准输入读入字符串赋给变量，格式是：　　read变量名 [变量名…]

例6.10　定义3个变量并为它们输入值：

```
$ read usera userb userc
  joe  zhao  ming
```

执行该read命令时，它将等待用户的输入。用户为每个变量输入一个字符串值，中间用空格分开。

(3) 在for命令中定义变量，用于进行循环控制。

2. 引用变量

- 引用变量即是将变量的值(字符串)，替换在发生引用的位置。

  引用变量的方法是在变量名前加引用字符“$”，格式是：

  $变量名 或 ${变量名}

例6.11 在命令中引用变量：

```
$  dir=/home/jhm/backup
$  echo $dir          #实际执行echo /home/jhm/backup
   /home/jhm/backup
$ cd $dir/newdir   #实际执行cd /home/jhm/backup/newdir
$ pwd
 /home/jhm/backup/newdir
$ echo $ dir          #加了空格后这里的$被看作普通字符
$ dir
```

这里不能加空格

例6.12 引用变量的方法：

```
$ echo $dir_1     # dir_1变量未定义，实际执行echo
  (空串)
$ echo ${dir}_1  #实际执行echo /home/jhm/backup_1
 /home/jhm/backup_1
$ str="This is a string"
$ echo "${str}ent test of variables"
  This is a stringent test of variables
$ echo "$strent test of variables"        #实际执行echo "test of
                                            variables"
 test of variables
```

3. 设置只读变量

为了防止变量的值被修改(也就是被重新赋值)，可以用readonly命令将变量设置为只读的。readonly命令的格式是：

readonly 变量名 [变量名…]

例6.13 设置只读变量：

```
$ readonly dir
$ dir=/home/jhm/project #对只读变量进行赋值
bash: dir: readonly variable      (赋值失败)
```

4. 清除变量

用unset命令清除变量。清除后的变量变为未定义变量，引用其值将得到空字符串。注意：只读变量是不能被清除的。

unset命令的格式是：

unset变量名[变量名…]

例6.14 清除变量：

```
$ unset dir
$ echo $dir

$
```