

# 第3章

## Linux Shell 编程

Shell 不仅仅只是一个命令解释程序，它还是一种高级程序设计语言。Shell 是一种解释型语言。Shell 程序常被称为命令脚本或脚本程序，由 Shell 命令加上程序控制结构组成，具有简洁、开发容易和便于移植的特点。众所周知，相比于编译型编程语言，解释型语言的执行效率较低，但编程效率却要高出十倍甚至百倍。使用 Linux Shell，通过简单执行一些文件系统级的高级操作即可迅速构建出系统维护所需的功能，因此 Shell 脚本程序是进行系统管理维护不可缺少的利器。本章主要介绍以下几方面的 Shell 编程知识。

- Shell 程序基础知识
- Shell 变量
- 控制结构
- Shell 函数
- Shell 内部命令
- Shell 程序调试

### 3.1 基础知识

读示例程序学习程序设计是最简洁有效的方法。为简明演示本章的示例，读者可先在自己家目录下建立一个“sh”目录专门放置 Shell 程序文件，为此输入以下命令。

```
jianglinmei@ubuntu:~/sh$ mkdir sh  
jianglinmei@ubuntu:~/sh$ cd sh,
```

下面直接开始第一个 Shell 程序。

#### 3.1.1 第一个 Shell 程序

第一个 Shell 程序——first.sh，如程序清单 3-1 所示。

程序清单 3-1 first.sh

```
#!/bin/bash  
cd /tmp  
echo "Hello, world!"
```

这是一个经典的程序入门示例，仅由三行代码构成，勿论代码是什么意思，先来看如何运行，

运行结果是什么。

### 3.1.2 如何运行 Shell 程序

运行 Linux Shell 程序的方法有三种。

- (1) 赋予程序文件可执行权限，直接运行。
- (2) 调用命令解释器（即 Shell）解释执行。
- (3) 使用 source 命令执行。

下面分别使用这三种方式运行第一个 Shell 程序，看结果是否会有不同。

采用第一种运行方式，操作如下。

```
jianglinmei@ubuntu:~/sh$ chmod a+x first.sh
jianglinmei@ubuntu:~/sh$ ./first.sh
Hello, world!
```

第一条命令为 first.sh 文件设置执行权限，第二条命令即执行它，屏幕显示出了预料之中的“Hello world!”。仔细观察命令行的提示信息，应发现命令执行前后，当前工作目录没有变化。

采用第二种运行方式，操作如下。

```
jianglinmei@ubuntu:~/sh$ bash first.sh
Hello, world!
jianglinmei@ubuntu:~/sh$
```

这里调用 Shell 解释器 bash 程序对 first.sh 解释执行，可以发现结果和第一种方式完全一样。

采用第三种运行方式，操作如下。

```
jianglinmei@ubuntu:~/sh$ source first.sh
Hello, world!
jianglinmei@ubuntu:/tmp$
```

这种方式的输出结果和前两种方式的结果相同，但仔细观察命令行的提示信息，可以发现当前工作目录变成了“/tmp”。

回头来看程序代码。

在本书第 1.3.3.1 小节介绍了目前 Shell 有多种不同的版本。常见的有 sh、ksh、csh、bash 和 zsh 等。不同的 Shell 的命令和语法不尽相同，那么如何得知用什么 Shell 来解释脚本程序呢？答案在程序的第一行。

命令行 Shell 在读取脚本第一行发现行首是“#!”时，会将剩余的字符串理解成一个 Shell 解释程序的绝对路径。随后，命令行 Shell 会启动一个新的 Shell 进程来执行脚本程序中的其余的每一行代码。因为本例中指定的 Shell 解释程序为“/bin/bash”，所以命令行 Shell 将启动新的 bash 进程来解释该脚本程序。鉴于 bash 是 Linux 中最常用的 Shell，本书仅介绍 bash 编程，所有脚本的解释程序都将指定为“/bin/bash”。

本书第 2 章介绍过 echo 命令，它的作用是向标准输出设备输出简单信息。程序清单 3-1 的第三行意即输出“Hello, world!”，从上述的输出结果可见，三种运行方式都得到了预期的输出。

程序清单 3-1 的第二行为：cd /tmp。其作用是将当前目录切换到“/tmp”，但是只有第三种方式达到了此目的，这是为什么呢？答案在下一小节。

### 3.1.3 Shell 的命令种类

Linux Shell 可执行的命令主要有三种：内部命令、Shell 函数和外部命令。

#### 1. 内部命令

内部命令是 Shell 解释器本身包含的命令，在文件系统中没有相应的可执行文件。例如，`cd` 命令和 `echo` 命令就是两个常见的 Shell 内部命令。还有，上面介绍的第三种运行 Shell 程序方式所用的 `source` 命令也是内部命令。命令行 Shell 在执行内部命令时，不需要创建新进程，当然也就不需要销毁进程。

`source` 命令也称为“点命令”，可以用点符号（“.”）代替 `source` 来执行命令，效果一样，例如：

```
jianglinmei@ubuntu:~/sh$ . first.sh
Hello, world!
jianglinmei@ubuntu:/tmp$
```

#### 2. Shell 函数

Shell 函数是以 Shell 语言书写的一系列程序代码，可以像其他命令一样被引用，本章后将详细介绍 Shell 函数。

#### 3. 外部命令

外部命令是独立于 Shell 的可执行程序，在文件系统中有相应的可执行文件。本书第 2 章介绍的大多数命令，如，`ls`、`find`、`locate`、`grep`、`ifconfig` 等，都是外部命令。命令行 Shell 在执行外部命令时，会创建一个当前 Shell 的复制进程来执行它，执行过程存在进程的创建和销毁。

Shell 执行外部命令的过程如图 3-1 所示，可描述如下。

- (1) 调用系统函数 `fork()` 创建一个命令行 Shell 的复制进程（子进程）。
- (2) 在子进程的运行环境中，查找并载入外部命令，以外部命令的程序代码取代该 Shell 子进程。此时，父 Shell 进程休眠并等待子进程执行完毕。
- (3) 子进程执行完毕后，父 Shell 进程被唤醒并继续从终端读取下一条命令。

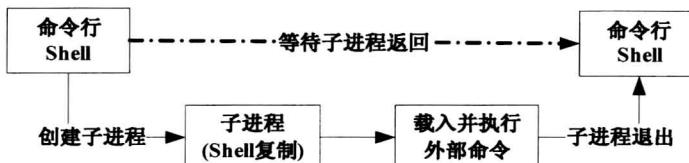


图 3-1 外部命令执行过程

在这一过程中，与回答上一小节最后一个问题关系密切的一个要点是：子进程改变的环境变量只影响本进程，而不会影响父进程。

为此，可以理解这三种运行方式执行第一个 Shell 程序的结果。因为 `source` 是 Shell 的内部命令，它执行的 `cd` 命令改变了命令行 Shell 本身的环境变量 (`$PWD`)，所以脚本程序执行完后当前目录改变了。而前两种运行 Shell 程序的方式均创建了一个子进程来执行脚本程序，其中的 `cd` 命令改变的只是子进程的环境变量 (`$PWD`)，并不影响作为父进程的命令行 Shell。而在子进程执行完毕后，活跃的仍是父进程，故当前目录并没有改变。

### 3.1.4 Shell 执行命令的顺序

交互式的 Shell 在获取用户输入的命令后，将按以下固定顺序寻找命令位置。

<b>别名</b>	使用 “alias command=...” 创建的命令
<b>关键字</b>	如: if, for, while 等
<b>函数</b>	Shell 语言书写的代码
<b>内部命令</b>	Shell 本身包含的命令, 如: cd、echo、source 等
<b>外部命令</b>	二进制可执行程序或脚本程序

由此可见，在同名时，别名的优先级最高而外部命令的优先级最低。使用 Shell 的内部命令“type”可以查看所要执行的命令是哪种类型。例如：

```
jianglinmei@ubuntu:/tmp$ type ls
ls 是 `ls --color=auto' 的别名
jianglinmei@ubuntu:/tmp$ type find
find 是 /usr/bin/find
jianglinmei@ubuntu:/tmp$ type pwd
pwd 是 shell 内嵌
```

从本例的输出可见：ls 是 “ls --color=auto” 的别名；find 是一个外部命令，其可执行文件是 “/usr/bin/find”；pwd 则是一个内部命令。

### 3.1.5 注释、退出状态和逻辑操作

Shell 程序中或 Shell 命令中，以 “#” 开头的文本表示**注释**，Shell 解释器将忽略 “#” 之后的所有内容。如果要将 “#” 作为普通字符对待，需在其之前加 “\” 进行转义，或使用引号对其进行引用。例如：

```
jianglinmei@ubuntu:~/sh$ # echo this will not show
jianglinmei@ubuntu:~/sh$ echo # this will not show

jianglinmei@ubuntu:~/sh$ echo \# this is common text
# this is common text
```

这里，第一条命令完全由 “#” 开头，实际上 Shell 并不解释任何命令，因此没有输出。第二条命令 echo 后的内容以 “#” 开头，所以输出是一个空行。最后一条命令的 “#” 被转义为普通字符，所以输出了完整的 “# this is common text”。

每一条 Shell 命令在退出时都会返回一个整数值给命令行 Shell，该返回值即代表 Shell 命令的**退出状态**。“退出状态”用于指示命令的运行情况：成功还是失败，如果失败了是什么原因导致的失败。一般约定以 0 表示成功，非零值表示失败。使用特殊变量 “\$?” 可以查看上一条命令的退出状态值。例如：

```
jianglinmei@ubuntu:~/sh$ echo this should succeed.
this should succeed.
jianglinmei@ubuntu:~/sh$ echo $?
0
jianglinmei@ubuntu:~/sh$ rm /tmp
rm: 无法删除 '/tmp' : 权限不够
jianglinmei@ubuntu:~/sh$ echo $?
```

从以上示例可以看到：第一条 echo 命令成功执行，其退出状态为 0；“rm /tmp”因权限不够执行失败，其退出状态为 1。

一般情况下，脚本程序中的各条命令是从上到下顺序执行的，不论上一条命令执行是否成功（退出状态为 0），下一条命令都能得到执行。各条命令可以分行书写，如程序清单 3-1 所示。也可以在一行中书写，但各命令之间要以“;”分隔开来，执行时按从左到右的顺序依次执行，如程序清单 3-2 所示。这两种书写方式的效果是一样的。

程序清单 3-2 second.sh

```
#! /bin/bash
cd /tmp; echo "Hello, world!"
```

除了使用“;”连接命令之外，还可以使用逻辑与（“&&”）运算符和逻辑或运算符（“||”）连接两条命令。这两个逻辑运算符均具有短路特性：对于逻辑与（“&&”）操作，只有当左边的命令执行成功（退出状态为 0）才会继续执行右边的命令，对于逻辑或（“||”）操作，则只有当左边的命令执行失败（退出状态为非 0 值）才会继续执行右边的命令。例如：

```
jianglinmei@ubuntu:~/sh$ ls one.sh && rm one.sh
ls: one.sh: 没有那个文件或目录
jianglinmei@ubuntu:~/sh$ ls first.sh || touch first.sh
first.sh
```

本示例第一条命令表达的含义是：如果 one.sh 可列出（文件存在）则删除之，但此处 one.sh 不存在，ls 执行失败，所以并不会执行 rm 命令去删除该文件。第二条命令表达的含义是：如果 first.sh 不可列出（文件不存在）则创建之，但此处 first.sh 已存在，故 touch 命令不执行。

除了逻辑与（“&&”）运算符和逻辑或运算符（“||”）以外，Shell 还支持逻辑非（“!”）运算符。逻辑非（“!”）运算符的作用是对退出状态值取反：成功则失败，失败则成功。例如：

```
[molin@XMUEDA sh]$ ! ls one.sh && echo no script
ls: one.sh: 没有那个文件或目录
no script
```

使用逻辑运算符组合起来的操作被称为“**逻辑操作**”。

### 3.1.6 复合命令

Linux Shell 中，可以用“{}”或“()”将多条命令括起来，使其在语法上成为一条命令，即形成一条复合命令（类似于 C 语言的复合语句）。复合命令的执行顺序是根据命令出现的先后次序，由左至右或由上到下执行。

复合命令中的各个命令之间必须用分号或换行符分隔开来。如果使用“{}”的话，还应注意：“{}”后应有至少一个空格，“}”前应有一个分号或换行符。

使用“{}”或“()”，它们的作用基本相同，唯有一点区别在于：用“{}”括起的命令在本 Shell 内执行，不产生新进程；用“()”括起的命令在一个子 Shell 内执行，命令行 Shell 会创建一个新的子 Shell 进程。这类似于使用 source 命令执行脚本和赋予脚本文件可执行权限并直接运行之间的区别。

下面举几个例子说明复合命令的使用。

## 1. 基本格式与用法

```
jianglinmei@ubuntu:~/sh$ { echo I am ; who ;} | head
I am
jianglinmei pts/0      2012-09-23 21:58 (10.8.18.212)
jianglinmei@ubuntu:~/sh$ (echo How man files in $PWD; ls | wc -w) | tail
How man files in /home/jianglinmei/sh
2
```

## 2. “{}” 和 “()” 的区别

```
jianglinmei@ubuntu:~/sh$ VAR1=1
jianglinmei@ubuntu:~/sh$ (VAR1=2; echo $VAR1)
2
jianglinmei@ubuntu:~/sh$ echo $VAR1
1
jianglinmei@ubuntu:~/sh$ { VAR1=2; echo $VAR1; }
2
jianglinmei@ubuntu:~/sh$ echo $VAR1
2
```

本例中，在“()”内将变量（变量将在本书下一节介绍）VAR1 设置为 2，但命令行 Shell 中显示 VAR1 的值仍为原始值 1。而在“{}”内将变量 VAR1 设置为 2，命令行 Shell 中显示 VAR1 的值即为 2。示例结果说明了两者的区别。

## 3.2 Shell 变量

### 3.2.1 变量的赋值与引用

和其他编程语言一样，Linux Shell 也使用变量来存储数据。

Shell 变量的名称应由字母、数字或下划线组成，并且只能以字母或下划线开头，大小写的意义是不同的，但是名称的长度没有限制。Shell 是一种弱类型的语言，变量存储的一切值都是字符串。但是必要的时候，只要是由数值构成的字符串，也可对其执行数值操作。

**变量赋值的方式为：**

变量名=变量值

注意：“=”两边不能有任何空格。当变量值中包含空格时，应为其加上单引号或双引号。在需要引用**变量**时，要在变量名前加上“\$”符号。例如：

```
jianglinmei@ubuntu:~/sh$ str="Hello, world"
jianglinmei@ubuntu:~/sh$ echo $str
Hello, world
```

Shell 变量本质上是一个键值对，即使用一个关键字来记录或引用一个值。例如上面示例中的“str=Hello, world!”，就是将字符串值“Hello, world!”赋予键 str。在 str 的作用范围内，均可使用 str 来引用“Hello, world!”，这个操作叫做**变量替换**。

和其他强类型的编程语言不同，Shell 变量不需要预先定义，或者说赋值即定义，而且可以引

用未赋过值的变量。在引用一个未事先赋过值的变量时，该变量值为一个空字符串。例如：

```
jianglinmei@ubuntu:~/sh$ echo $not_defined
```

本例输出一个空串。

在字符串中可以引用变量，使其值成为本字符串的一部分。例如：

```
jianglinmei@ubuntu:~/sh$ str='world!'
jianglinmei@ubuntu:~/sh$ echo Hello, $str
Hello, world!
```

在变量名后面紧跟一个由非空白字符开始的字符串时，为了使变量名和其后的字符串区分开来，应该用花括号“{}”将变量名括起来。例如：

```
jianglinmei@ubuntu:~/sh$ position=/usr/include/
jianglinmei@ubuntu:~/sh$ cat ${position}termio.h
/* Compatible <termio.h> for old `struct termio' ioctl interface.
   This is obsolete; use the POSIX.1 `struct termios' interface
   defined in <termios.h> instead. */

#include <termios.h>
#include <sys/ioctl.h>
```

此例中，如果引用时不使用花括号，即：

```
jianglinmei@ubuntu:~/sh$ cat $positiontermio.h
cat: .h: 没有那个文件或目录
```

此时，Shell 输出错误信息。因为 Shell 把\$positiontermio 看成了一个变量，而该变量的值为一个空串，因此整个字符串的值就是“.h”，所以提示“.h 文件不存在”。

使用 unset 命令可以将一个变量的值清除（即成为空串）。例如：

```
jianglinmei@ubuntu:~/sh$ str=has_a_value
jianglinmei@ubuntu:~/sh$ echo $str
has_a_value
jianglinmei@ubuntu:~/sh$ unset str
jianglinmei@ubuntu:~/sh$ echo $str
```

本例中，“unset str”后再输出\$str，结果为一个空串。

使用特殊变量引用“\${#变量名}”可以得到变量的长度，即字符数。例如：

```
jianglinmei@ubuntu:~/sh$ str="A 22 characters string"
jianglinmei@ubuntu:~/sh$ echo length of $str is ${#str}
length of A 22 characters string is 22
```

### 3.2.2 命令替换

Shell 允许将一个或多个命令的执行结果赋值给变量，这称为**命令替换**。有两种方式可以实现命令替换：使用反引号“`...`”或“\$(...)”。反引号或圆括号之间为一个或多个用“;”或逻辑运算符连接起来的命令。例如：

```
jianglinmei@ubuntu:~/sh$ str=`pwd; who`
jianglinmei@ubuntu:~/sh$ echo $str
/home/jianglinmei/sh jianglinmei pts/1 2012-09-24 08:27 (192.168.1.100)
```

```
jianglinmei@ubuntu:~/sh$ position=$(pwd||who)
jianglinmei@ubuntu:~/sh$ echo $position
/home/jianglinmei/sh
```

### 3.2.3 变量属性声明

在 Shell 中可以使用内部命令 `declare` 或 `typeset` ( 它们是完全相同的 ) 来限定变量的属性 ( 注: 命令 `declare` 在 bash 版本 2 之后才有 )。`declare` 使用一些选项来指定变量的属性, 常用选项如下。

-r	只读
-i	整数
-a	数组 ( 在下一小节介绍 )
-f	函数 ( 在本章 3.6 节介绍 )
-x	导出变量 ( 在本章 3.2.8 小节介绍 )

例如:

```
jianglinmei@ubuntu:~/sh$ declare -r SIZE=100
jianglinmei@ubuntu:~/sh$ SIZE=20
-bash: SIZE: 只读变量
```

本例中, 因 `SIZE` 被设为只读, 再对它赋值, Shell 就会报错。

再如:

```
jianglinmei@ubuntu:~/sh$ n=20
jianglinmei@ubuntu:~/sh$ n=n+30
jianglinmei@ubuntu:~/sh$ echo $n
n+30
jianglinmei@ubuntu:~/sh$ declare -i n
jianglinmei@ubuntu:~/sh$ n=20
jianglinmei@ubuntu:~/sh$ n=n+30
jianglinmei@ubuntu:~/sh$ echo $n
50
```

本例中, 同样是对变量 `n` 作 “`n=n+30`” 的操作, 但是在设置变量的属性为整数之前和之后, 其结果是不一样的。

### 3.2.4 数组变量

Linux Shell ( bash ) 支持一维数组变量。和普通变量一样, 使用数组变量也不需先定义或先赋值再使用, 在没有赋值的情况下, 数组元素的值就是空串, 并且 Shell 数组的元素个数没有限制。

与 C 语言类似, 存取数组的元素的方法是在数组名后加上下标。数组元素的下标由 0 开始编号, 放在一对方括号内, 如: `a[0]`。下标可以是整数或算术表达式, 其值应大于或等于 0。

可以使用赋值语句对数组元素**直接赋值**, 其一般格式如下。

数组名 [ 下标 ] = 值

例如:

```
jianglinmei@ubuntu:~/sh$ student[0]=Alice
jianglinmei@ubuntu:~/sh$ student[1]=Bob
```

```
jianglinmei@ubuntu:~/sh$ student[2]=Tom
```

引用数组元素值的一般格式为：

```
 ${数组名[下标]}
```

例如：

```
jianglinmei@ubuntu:~/sh$ echo ${student[0]}
Alice
```

数组的各个元素可以利用上述方式逐个赋值，也可以**组合赋值**。定义数组并为其赋值的一般格式是：

```
数组名= ( 值 1 值 2 ... 值 n )
```

其中，各个值之间应以空格分开。例如：

```
jianglinmei@ubuntu:~/sh$ ARR=(How nice a day it is.)
jianglinmei@ubuntu:~/sh$ echo ${ARR[1]} ${ARR[3]}
nice day
```

引用没有带下标的数组名相当于引用下标为 0 的数组元素。以下示例说明：ARR 就等价于 ARR[0]。

```
jianglinmei@ubuntu:~/sh$ echo ${ARR} = ${ARR[0]}
How = How
jianglinmei@ubuntu:~/sh$ echo ${#ARR} = ${#ARR[0]}
3 = 3
```

例子中，\${#ARR[0]} 意为下标 0 处数组元素的长度。

有两个特殊的变量引用 “\${数组名[\*]}” 和 “\${数组名[@]}”，它们表示引用数组中的所有非空元素。例如：

```
jianglinmei@ubuntu:~/sh$ week=(Mon Tue Wed)
jianglinmei@ubuntu:~/sh$ week[3]=Thur
jianglinmei@ubuntu:~/sh$ week[5]=Sat
jianglinmei@ubuntu:~/sh$ echo ${week[*]}
Mon Tue Wed Thur Sat
jianglinmei@ubuntu:~/sh$ echo ${week[@]}
Mon Tue Wed Thur Sat
```

利用命令 unset 可以取消一个数组的定义。例如：

```
jianglinmei@ubuntu:~/sh$ unset week
jianglinmei@ubuntu:~/sh$ echo ${week[*]}
```

unset 后 \${week[\*]} 的结果为空。

相应地，特殊变量引用 “\${#数组名[\*]}” 和 “\${#数组名[@]}”，它们表示所引用数组中的所有非空元素的个数。例如：

```
jianglinmei@ubuntu:~/sh$ week=(Mon Tue Wed)
jianglinmei@ubuntu:~/sh$ week[6]=Sun
jianglinmei@ubuntu:~/sh$ echo ${#week[*]}
4
```

```
jianglinmei@ubuntu:~/sh$ echo ${#week[@]}
4
```

### 3.2.5 变量引用操作符

使用\$加花括号“\${...}”除简单地引用变量的值外，还可以进行更多的高级操作，如：字符串替换和模式匹配替换。

- 字符串替换

#### 1. \${varname:-word}

含义：如果 varname 存在且非空串，则返回 varname 的值，否则返回 word。

作用：如果变量未定义，则取默认值。

例如：

```
jianglinmei@ubuntu:~/sh$ unset str
jianglinmei@ubuntu:~/sh$ echo ${str:-"blank"}
blank
jianglinmei@ubuntu:~/sh$ str="some content"
jianglinmei@ubuntu:~/sh$ echo ${str:-"blank"}
some content
```

#### 2. \${varname:=word}

含义：如果 varname 存在且非空串，则返回 varname 的值，否则将 varname 的值设为 word，并返回 word。

作用：如果变量未定义，则取默认值。

例如：

```
jianglinmei@ubuntu:~/sh$ unset str
jianglinmei@ubuntu:~/sh$ echo ${str:="blank"}
blank
jianglinmei@ubuntu:~/sh$ echo $str
blank
jianglinmei@ubuntu:~/sh$ str="some content"
jianglinmei@ubuntu:~/sh$ echo ${str:-"blank"}
some content
```

#### 3. \${varname:+word}

含义：如果 varname 存在且非空串，则返回 wor 的值，否则返回空串。

作用：测试变量是否存在。

例如：

```
jianglinmei@ubuntu:~/sh$ unset str
jianglinmei@ubuntu:~/sh$ echo ${str:+not blank}

jianglinmei@ubuntu:~/sh$ str="some content"
jianglinmei@ubuntu:~/sh$ echo ${str:+not blank}
not blank
```

#### 4. \${varname:?message}

含义：如果 varname 存在且非空串，则返回 varname 的值，否则输出 message，并退出当前脚本程序。

作用：用于捕捉变量未定义导致的错误。

例如：

```
jianglinmei@ubuntu:~/sh$ cat novar.sh
#!/bin/bash
echo ${str:?no parameter}
echo "last sentence"
jianglinmei@ubuntu:~/sh$ bash novar.sh
novar.sh: 行 2: str: no parameter
```

本例，首先输出脚本程序 novar.sh 的内容，其中含两条 echo 语句，但因 str 未定义，使用 bash 执行该脚本时，仅输出了“no parameter”就退出了。

注：以上四种字符串替换格式中，每个冒号都是可选的。如果省略冒号，则判断“varname 是否存在”，而不论是否非空。例：

```
jianglinmei@ubuntu:~/sh$ str=
jianglinmei@ubuntu:~/sh$ echo ${str?"not exist"}

jianglinmei@ubuntu:~/sh$ unset str
jianglinmei@ubuntu:~/sh$ echo ${str?"not exist"}
not exist
```

- 模式匹配替换

1. \${varname#pattern}

含义：如果 pattern 匹配 varname 的头部，则删除**最短**匹配部分，并返回剩余部分，varname 本身不变。

例如：

```
jianglinmei@ubuntu:~/sh$ filepath=/home/alice/major.minor.ext
jianglinmei@ubuntu:~/sh$ echo ${filepath#/*/}
alice/major.minor.ext
jianglinmei@ubuntu:~/sh$ echo $filepath
/home/alice/major.minor.ext
```

2. \${varname##pattern}

含义：如果 pattern 匹配 varname 的头部，则删除**最长**匹配部分，并返回剩余部分，varname 本身不变。

例如：

```
jianglinmei@ubuntu:~/sh$ filepath=/home/alice/major.minor.ext
jianglinmei@ubuntu:~/sh$ echo ${filepath##*/}
major.minor.ext
```

3. \${varname%pattern}

含义：如果 pattern 匹配 varname 的尾部，则删除**最短**匹配部分，并返回剩余部分，varname 本身不变。

例如：

```
jianglinmei@ubuntu:~/sh$ filepath=/home/alice/major.minor.ext
jianglinmei@ubuntu:~/sh$ echo ${filepath%.*}
/home/alice/major.minor
```

#### 4. \${varname%%pattern}

含义：如果 pattern 匹配 varname 的尾部，则删除最长匹配部分，并返回剩余部分，varname 本身不变。

例如：

```
jianglinmei@ubuntu:~/sh$ filepath=/home/alice/major.minor.ext
jianglinmei@ubuntu:~/sh$ echo ${filepath%.*}
/home/alice/major
```

#### 5. \${varname/pattern/string}或\${varname//pattern/string}

含义：如果 pattern 匹配 varname 的某个子串，则将 varname 的最长匹配部分替换为 string，并返回替换后的串，varname 本身不变。如果模式以“#”开头，则意为必须匹配 varname 的首部，如果模式以“%”开头，则意为必须匹配 varname 的尾部。如果 string 为空串，匹配部分将被删除。如果 varname 为“@”或“\*”，操作将被依次用于每个位置参数（参见下一小节），并且扩展为结果列表。注：第一种格式仅替换第一次匹配的子串，第二种格式会替换所有匹配的子串。

例如：

```
jianglinmei@ubuntu:~/sh$ filepath=/home/alice/major.minor.ext
jianglinmei@ubuntu:~/sh$ echo ${filepath/alice/tom}
/home/tom/major.minor.ext
```

再如：

```
jianglinmei@ubuntu:~/sh$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:$JAVA_HOME
/bin
jianglinmei@ubuntu:~/sh$ echo -e ${PATH//:/\n}
/usr/local/sbin\n/usr/local/bin\n/usr/sbin\n/usr/bin\n/sbin\n/bin\n/usr/games\n$JAVA_HOME
/bin
jianglinmei@ubuntu:~/sh$ echo -e ${PATH//:/"\n"}
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
/usr/games
$JAVA_HOME/bin
```

本例将环境变量 PATH 中的所有（因为使用的是第二种格式）冒号“：“替换成换行符。

### 3.2.6 位置参数和特殊变量

位置参数也称位置变量，是运行 Shell 脚本程序时，命令行 Shell 传递给脚本的参数，以及在 Shell 脚本程序中调用函数时传递给函数的参数。这位置变量的名称很特别，是以 0, 1, 2……这些整数命名的。

位置变量的数字与参数出现的具体位置相对应：0 对应命令名（脚本名），1 对应第一个实参，2 对应第二个实参……依此类推。相应地，使用 \$0, \$1, \$2……引用这些位置变量。如果，位置变量的数字是由两个或更多个数字构成，则一般应用一对花括号把数字括起来，如：\${10}, \${11}。命令行实参与脚本中位置变量的对应关系如下所示。

cmd	p1	p2	p3	p4 ...	p10	p11
\$0	\$1	\$2	\$3	\$4 ...	\$(10)	\$(11)

除了这些数字形式的变量之外，Shell 中几个有特殊含义的 Shell 变量（如表 3-1 所示），它们的值只能由 Shell 根据实际情况进行赋值，而不允许用户重新设置。其中一些特殊变量常与位置变量一起使用。

表 3-1

特殊变量

特殊变量	含    义
\$#	命令行上参数的个数，但不包含 Shell 脚本名本身
\$*	以一个单字符串显示向脚本程序传递的所有参数，不含\$0
\$@	从参数 1 开始，显示向脚本程序传递的所有参数，不含\$0。如果放在双引号中进行扩展，则“\$@”与“\$1”“\$2”“\$3”...等效
\$?	上一条命令执行后的返回值（即“退出状态”），一个 10 进制整数，见 3.1.5 小节的介绍
\$\$	运行脚本的当前进程的进程号
\$!	上一个后台命令对应的进程号
\$-	由当前 Shell 设置的执行标志名组成的字符串

下面以 Shell 程序——posvar.sh（如程序清单 3-3 所示）为例，说明各个与位置参数相关的变量的用法。

程序清单 3-3

```
#!/bin/bash
echo 'Parameter number:' $#
echo 'All digit variables:' $0 $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11}
echo '$*: '$*
echo '$@: '$@
```

程序清单 3-3 中，第一行使用特殊变量“\$#”输出命令行传递的参数个数，第二行依次输出各位置参数的值，后面两行分别用特殊变量“\$\*”和“\$@”输出命令行传递的所有参数。运行该脚本，命令如下。

```
jianglinmei@ubuntu:~/sh$ chmod u+x posvar.sh
jianglinmei@ubuntu:~/sh$ ./posvar.sh 1 2 3 4 5 6 7 8 9 10 11
Parameter number: 11
All digit variables: ./posvar.sh 1 2 3 4 5 6 7 8 9 10 11
$*: 1 2 3 4 5 6 7 8 9 10 11
$@: 1 2 3 4 5 6 7 8 9 10 11
```

本例首先为脚本赋予可执行权限，然后向其传递了 11 个参数（参数之间以空白字符分隔），从运行结果可以清晰地看出各变量的含义与作用。

不加双引号的\$@与\$\*的作用是一样的，加上双引号的“\$@”与“\$\*”在特殊场合作用会有所不同，“\$@”表示的是引用所有参数，“\$\*”表示引用所有参数连接在一起（中间以空格分隔）的字符串。如果用于数组，“\${数组名[@]}”表示引用数组整体的各个元素，“\${数组名[\*]}”表示引用所有数组元素连接在一起（中间以空格分隔）的字符串。

位置参数的值不能由用户直接设置，但可以使用 set 命令间接地设置除\$0 以外的位置变量的值。以程序清单 3-4 为例。

---

程序清单 3-4 setposvar.sh

---

```
#! /bin/bash
set learning linux program
echo $0 $1 $2 $3
```

程序清单 3-4 的第二行 set 命令按顺序设置了\$1、\$2、\$3 的值，随后第三行输出所有变量的值。运行该脚本的命令如下。

```
jianglinmei@ubuntu:~/sh$ bash setposvar.sh
setposvar.sh learning linux program
```

Shell 内置了一个 shift 命令，用于向左移动位置参数，亦即原来的\$2 的值赋给\$1（原\$1 的值永远丢失），原来的\$3 的值赋给\$2，原来的\$4 的值赋给\$3，依此类推。结果是参数的个数少了一个，\$#的值会减一。以程序清单 3-5 所列脚本为例。

---

程序清单 3-5 shiftposvar.sh

---

```
#! /bin/bash
set learning linux program
echo "parameter number: $#, there are:"
echo $1 $2 $3
shift
echo "after shifted, parameter number: $#, there are:"
echo $1 $2 $3
```

在命令行运行该脚本，如下所示。

```
jianglinmei@ubuntu:~/sh$ bash shiftposvar.sh one two three
parameter number: 3, there are:
one two three
after shifted, parameter number: 2, there are:
two three
```

shift 命令后也可跟一个大于 1 的整数参数，如：shift 2 表示向左移动两位，此时\$1 和\$2 的值均将丢失，参数个数减 2。

### 3.2.7 read 命令

Linux Shell 提供了 read 命令用于从键盘上读取数据并赋值给指定的变量。利用 read 命令可编写用户相交互式的脚本程序。read 命令的一般格式是：

```
read 变量 1 [变量 2 ...]
```

注意，输入数据时，数据项间应以空格或制表符作为分隔符，变量个数和数据个数之间可能出现下面三种情况。

(1) 变量个数与给定数据个数相同，则依次对应赋值。例如：

```
jianglinmei@ubuntu:~/sh$ read a b c
Linux Shell programming
jianglinmei@ubuntu:~/sh$ echo $a : $b : $c
Linux : Shell : programming
```

(2) 变量个数少于数据个数，则从左至右对应赋值，但最后一个变量被赋予剩余的所有数据。

例如：

```
jianglinmei@ubuntu:~/sh$ read a b c
Linux shell programming is very funny!
jianglinmei@ubuntu:~/sh$ echo $a : $b : $c
Linux : shell : programming is very funny!
```

(3) 变量个数多于给定数据个数，则依次对应赋值，而没有数据与之对应的变量取空串。

例如：

```
jianglinmei@ubuntu:~/sh$ read a b c
Linux shell
jianglinmei@ubuntu:~/sh$ echo $a : $b : $c
Linux : shell :
```

### 3.2.8 export 语句

用户可以在脚本或命令行上定义一些变量并予以赋值，包括改变环境变量的值。在同一 Shell 中，变量值是可见的；但是在子 Shell 中，父 Shell 的变量不可见。例如：

```
jianglinmei@ubuntu:~/sh$ cat child.sh
#!/bin/bash
echo $str
jianglinmei@ubuntu:~/sh$ str="parent shell variable"
jianglinmei@ubuntu:~/sh$ bash child.sh

jianglinmei@ubuntu:~/sh$ source child.sh
parent shell variable
```

从上面示例可以看出，使用 bash 执行脚本 child.sh 时，变量 str 的值为空串。

通常，在命令行上输入的命令都是由相应的进程执行的，即父进程创建子进程，子进程完成该命令的功能。然而，子进程执行时的环境与父进程的环境往往不同。一个进程在自己的环境中定义的变量是局部变量，仅限于自身范围，不能自动传给其子进程。就是说，子进程只能继承父进程的公用区和转出区中的数据，而每个进程的数据区和栈区是私有的，不能继承，如图 3-2 所示。

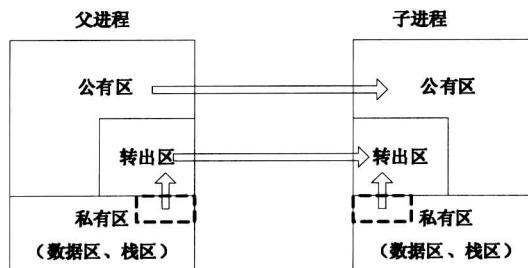


图 3-2 父子进程的继承关系

从图 3-2 中看出，父进程定义的变量对其子进程的运行环境不产生任何影响。为了使其各个子进程能继承父进程中定义的变量，就必须用 export (转出) 命令将这些变量送入进程转出区。

export 命令的一般格式是：

```
export [变量名]
```

例如：

```
jianglinmei@ubuntu:~/sh$ cat child.sh
#!/bin/bash
echo $str
jianglinmei@ubuntu:~/sh$ str="parent shell variable"
jianglinmei@ubuntu:~/sh$ export str
jianglinmei@ubuntu:~/sh$ bash child.sh
parent shell variable
```

另外，在同一 `export` 命令行上可以有多个变量名，例如：

```
jianglinmei@ubuntu:~/sh$ export TERM PATH SHELL HOME
```

利用不带参数的 `export` 命令可以显示本进程利用 `export` 命令所输出的全部变量。此外，也可以利用 `env` 命令列出所有的环境变量，包括本进程及以前的“祖先进程”所输出的变量。`export` 与 `env` 在输出格式上是不同的。例如：

```
jianglinmei@ubuntu:~/sh$ export
declare -x HISTTIMEFORMAT="%Y-%m-%d %H:%M:%S "
declare -x HOME="/home/jianglinmei"
declare -x JAVA_HOME="/usr/lib/jvm/java-6-openjdk/"
..... (省略若干环境变量)
jianglinmei@ubuntu:~/sh$ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_fignore:hist
append:interactive_comments:login_shell:progcomp:promptvars:sourcepath
BASH_ALIASES=()
..... (省略若干环境变量)
```

最后，应说明的是：子 Shell 中可以读取但无法改变父 Shell 中 `export` 出来的变量的值。

## 3.3 控制结构

Shell 具有一般高级程序设计语言所具有的条件控制结构、循环控制结构和函数定义与调用功能。条件控制就是条件测试与执行，即根据条件测试的结果执行不同的代码。条件控制结构有 `if` 语句、`case` 语句和条件测试命令。循环控制用于控制某些代码的重复行为，是众多高级语言不可缺少的元素。循环控制结构有 `for` 循环、`while` 循环、`until` 循环和跳出循环命令。

### 3.3.1 条件测试

#### 3.3.1.1 test 命令

条件测试命令 `test` 用于评估表达式的值以便进行条件控制。`test` 命令有两种书写格式。

`test 表达式`

或

`[ 表达式 ]`

`test` 命令评估“表达式”参数，如果表达式的值为“真”，其退出状态为 0（即成功），否则退

出状态为非零值（即失败）。注意，使用方括号格式时，“[”右边和“]”左边各需至少一个空格。另外，如果在 test 语句中使用了 Shell 变量，为避免歧义，最好用双引号将变量括起来。test 的基本使用方法如下。

```
jianglinmei@ubuntu:~/sh$ test "3" \> "2";echo $?
0
jianglinmei@ubuntu:~/sh$ [ "2" \> "3" ];echo $?
1
```

本例执行条件测试命令，并以“\$?”输出测试结果，0 为成功（真），1 为失败（假）。

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 4 类：文件属性测试运算符、字符串测试运算符、数值测试运算符和逻辑运算符。

- 文件属性测试

有关文件测试运算符的形式和功能如表 3-2 所示。

表 3-2 文件测试运算符

参 数	功 能
-b 文件名	若文件存在并且是块设备文件，则测试条件为真
-c 文件名	若文件存在并且是字符设备文件，则测试条件为真
-d 文件名	若文件存在并且是目录文件，则测试条件为真
-e 文件名	若文件存在，则测试条件为真
-f 文件名	若文件存在并且是普通文件，则测试条件为真
-g 文件名	若文件存在并且设置了 SETGID 位，则测试条件为真
-h 文件名	若文件存在并且是一个符号连接，则测试条件为真
-L 文件名	同-h
-p 文件名	若文件存在并且是一个命名的 FIFO 文件（即命名管道），则测试条件为真
-r 文件名	若文件存在并且是用户可读的，则测试条件为真
-S 文件名	若文件存在并且是一个 socket，则测试条件为真
-s 文件名	若文件存在并且文件的长度大于 0（即非空），则测试条件为真
-t 文件描述字	若文件被打开且其文件描述字是与终端设备相关的，则测试条件为真。默认的“文件描述字”是 1
-u 文件名	若文件存在并且设置了 SETUID 位，则测试条件为真
-w 文件名	若文件存在并且是用户可写的，则测试条件为真
-x 文件名	若文件存在并且是用户可执行的，则测试条件为真
-O 文件名	若当前用户是文件的所有者，则测试条件为真
-G 文件名	若当前用户的组 ID 匹配文件的组 ID，则测试条件为真
文件 1 -nt 文件 2	若文件 1 比文件 2 新，则测试条件为真
文件 1 -ot 文件 2	若文件 1 比文件 2 旧，则测试条件为真

- 字符串测试

有关字符串测试运算符的形式和功能如表 3-3 所示。注意，双目运算符两侧应有至少一个空格。

表 3-3

字符串测试运算符

参 数	功 能
-z str	如果字符串 str 的长度为 0, 即空串, 则测试条件为真
-n str	如果字符串 str 的长度大于 0, 即非空串, 则测试条件为真
str	如果字符串 str 不是空字符串, 则测试条件为真
sl = s2	如果 sl 等于 s2, 则测试条件为真, “=” 也可以用 “==” 代替
s1 != s2	如果 s1 不等于 s2, 则测试条件为真
s1 < s2	如果按字典顺序 s1 在 s2 之前, 则测试条件为真
s1 > s2	如果按字典顺序 s1 在 s2 之后, 则测试条件为真

### ● 数值测试

有关数值测试运算符的形式和功能如表 3-4 所示。

表 3-4

数值测试运算符

参 数	功 能
n1 -eq n2	如果整数 n1 等于 n2, 则测试条件为真
n1 -ne n2	如果整数 n1 不等于 n2, 则测试条件为真
n1 -lt n2	如果 n1 小于 n2, 则测试条件为真
n1 -le n2	如果 n1 小于或等于 n2, 则测试条件为真
n1 -gt n2	如果 n1 大于 n2, 则测试条件为真
n1 -ge n2	如果 n1 大于或等于 n2, 则测试条件为真

### ● 逻辑测试

有关逻辑测试运算符的形式和功能如表 3-5 所示。

表 3-5

逻辑测试运算符

参 数	功 能
! 表达式	如果表达式的值为假, 则测试条件为真
表达式 1 -a 表达式 2	如果表达式 1 和表达式 2 的值都为真, 则测试条件为真
表达式 1 -o 表达式 2	如果表达式 1 和表达式 2 的值有任一个为真, 则测试条件为真
\(表达式)	圆括号前面加上转义符 “\”, 使圆括号失去“在一个子 Shell 内执行复合命令”的作用。圆括号将由逻辑运算的组合起来的复合表达式括起来, 使之成为一个整体, 用于改变运算优先级。例如: [(“\$a” -ge 0) -a \ (“\$b” -le 100)]
	逻辑表达式中, 圆括号的优先级最高, 条件测试运算符优先级高于“!”运算符, “!”运算符的优先级高于“-a”运算符, “-a”运算符高于“-o”

注意, 表 3-5 中的逻辑测试运算符用于连接测试表达式, 逻辑操作符“&&”和“||”用于连接两个命令。例如:

```
jianglinmei@ubuntu:~/sh$ [ \(`a` = "$HOME" -o 3 -lt 4 \) ]; echo $?
0
jianglinmei@ubuntu:~/sh$ [ "a" = "$HOME" ] || [ 3 -lt 4 ]; echo $?
0
```

- 特殊条件测试

除以上条件测试外，在控制结构中还常用下列三个特殊条件测试语句。

- |           |                   |
|-----------|-------------------|
| (1) :     | 表示不做任何事情，其退出值为 0。 |
| (2) true  | 表示总为真，其退出值总是 0。   |
| (3) false | 表示总为假，其退出值是 255。  |

### 3.3.1.2 let 命令

test 命令非常强大，但只能执行算术比较运算且书写烦琐。为此，bash 提供了专门执行整数算术运算的命令是 let，其语法格式为：

```
let 算术表达式...
或
((算术表达式))
```

这里的算术表达式使用 C 语言中表达式的语法、优先级和结合性，可以执行 C 语言中常见的算术、逻辑和位操作。除++，--和逗号“,”之外，所有整型运算符都得到支持。此外，还提供了方幂运算符“\*\*”。

命名的参数在算术表达式中可直接用名称访问，前面不用带“\$”符号，也不需要对算术表达式中的操作符进行转义。算术运算的操作数只能是整数（按长整数进行求值）。除 0 会产生错误，但不会溢出。

如果算术表达式求值为 0，则设置退出状态为 1；如果求值为非 0 值，则退出状态为 0。例如：

```
jianglinmei@ubuntu:~/sh$ let x=2 y=2**3 z=y*3;echo $? $x $y $z
0 2 8 24
jianglinmei@ubuntu:~/sh$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y $w
0 3 8 16
jianglinmei@ubuntu:~/sh$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y $w
0 4 8 13
```

表 3-6 列出了在算术表达式中可用的运算符及其优先级和结合性。

表 3-6 算术表达式中的运算符

优 先 级	运 算 符	结 合 性	功 能
1	-	右结合	取表达式的负值
	+	右结合	取表达式的正值
2	!	右结合	逻辑非
	~	右结合	按位取反
3	**		方幂
4	*	左结合	乘
	/	左结合	除
	%	左结合	取模
5	+	左结合	加
	-	左结合	减
6	<<	左结合	左移若干二进制位
	>>	左结合	右移若干二进制位

续表

优先级	运算符	结合性	功能
7	>	左结合	大于
	$\geq$	左结合	大于或等于
	<	左结合	小于
	$\leq$	左结合	小于或等于
8	$=$	左结合	相等
	$\neq$	左结合	不相等
9	$\&$	左结合	按位与
10	$\wedge$	左结合	按位异或
11	$ $	左结合	按位或
12	$\&\&$	左结合	逻辑与
13	$\ $	左结合	逻辑或
14	$:$	右结合	条件计算
15	$=$	右结合	赋值
	$+=$ 和 $-=$	右结合	运算且赋值
	$*=$ 和 $/=$	右结合	
	$%=$ 和 $\&=$	右结合	
	$^=$ 和 $ =$	右结合	
	$>=$ 和 $<=$	右结合	

表达运算符优先级是由高到低排列的，即 1 级最高，15 级最低。同级运算符在同一个表达式中出现时，其执行顺序由结合性决定。

表达式中可以使用括号来改变运算符的操作顺序，即在运算时要先计算括号内的表达式。

当表达式中有 Shell 的特殊字符时，必须用双引号将其括起来。例如，“let ”val=a|b””。如果不括起来，Shell 会把其中的 “|” 看成管道符，将其左右两边看成不同的命令，因而无法正确执行。但是使用 “(())” 形式时，即使表达式中有 Shell 的特殊字符时，也不必用双引号将其括起来。例如：

```
jianglinmei@ubuntu:~/sh$ let "v = 6 | 5"; echo $v
7
jianglinmei@ubuntu:~/sh$ let v = 6 | 5
-bash: let: =: 语法错误: 期待操作数 (错误符号是 "=")
5: 找不到命令
jianglinmei@ubuntu:~/sh$ ((v = 6 | 5)); echo $v
7
```

使用 “\$((算术表达式))” 形式，可以返回算术表达式的确切值（而不是 let 命令的退出码），并将返回值赋值给其他变量。例如：

```
jianglinmei@ubuntu:~/sh$ v=$((6+9)); echo $?; echo $v
0
15
```

### 3.3.1.3 “[ ]” 测试

同 “(( ))”一样，利用复合命令 “[ [ ] ]” 可以对文件名和字符串使用更自然的语法，其中的特殊字符不用转义。在 “[ [ ] ]” 中，允许用括号和逻辑操作符 “&&” 和 “||” 把 test 命令支持的测试组合起来。例如：

```
jianglinmei@ubuntu:~/sh$ [[ (-d "$HOME") && (-w "$HOME") ]] && echo "home is a writable directory"
home is a writable directory
```

在使用 = 或 != 操作符时，复合命令 “[ [ ] ]” 还能在字符串上进行模式匹配。例如：

```
jianglinmei@ubuntu:~/sh$ [[ "abc def .d,x--" == a[abc]*\ ?d* ]]; echo $?
0
jianglinmei@ubuntu:~/sh$ [[ "abc def c" == a[abc]*\ ?d* ]]; echo $?
1
jianglinmei@ubuntu:~/sh$ [[ "abc def d,x" == a[abc]*\ ?d* ]]; echo $?
1
```

## 3.3.2 if 语句

if 语句用于条件控制，其一般语法结构为：

```
if 测试条件 1
then
    命令组 1
[elif 测试条件 2]
then
    命令组 2
[else
    命令 x]
fi
```

其中，if、then、elif、else 和 fi 是关键字，方括号括起的是可选部分，elif 语句可以有任意多个，命令组 n (n 为 1, 2, 3...) 只有在相应的测试条件 n (n 为 1, 2, 3...) 成立时才执行，命令 else 语句中的命令 x 只在所有的测试条件都不满足时才执行。习惯上可以将 then 关键字与 if 写在同一行，此时，then 之前应有一个分号，格式如下。

```
if 测试条件 1; then
    命令组 1
[elif 测试条件 2]; then
    命令组 2
[else
    命令 x]
fi
```

if 语句唯一可测试的内容是命令退出状态，也就是说，测试条件是一条或多条命令，多条命令可由分号、换行符分隔或由逻辑操作符连接。如果测试条件是多条命令的话，则以最后一条得到执行的命令的退出状态为准。以程序清单 3-6 为例。

程序清单 3-6 if.sh

```
#!/bin/bash
```

```

echo 'type in the user name.'
      read user
if grep $user /etc/passwd > /tmp/null && who | grep $user
then
      echo "$user has logged in the system."
      cp /tmp/null ~/me.tmp
      rm /tmp/null
else
      echo "$user has not logged in the system."
fi

```

执行该脚本如下。

```

jianglinmei@ubuntu:~/sh$ source if.sh
type in the user name.
jianglinmei
jianglinmei pts/0    2012-09-25 10:23 (10.8.18.212)
jianglinmei has logged in the system.
jianglinmei@ubuntu:~/sh$ source if.sh
type in the user name.
alice
alice has not logged in the system.

```

在本例中，jianglinmei 是已登录用户，所以执行了测试条件下所有命令后执行 if 分支的命令。alice 是系统中不存在的用户，所以测试条件中的 grep 命令失败，who 命令没有执行，然后执行了 then 分支的命令。

### 3.3.3 case 语句

case 语句允许进行多重条件选择。其一般语法格式是：

```

case 字符串 in
模式字符串 1)     命令
...
命令; ;
模式字符串 2)     命令
...
命令; ;
...
模式字符串 n)     命令
...
命令; ;
esac

```

case 语句的执行过程是，用“字符串”的值依次与各模式字符串进行比较，如果发现同某一个匹配，那么就执行该模式字符串之后的各个命令，直至遇到两个分号为止。如果没有任何模式字符串与该字符的值相符合，则不执行任何命令。以程序清单 3-7 为例。

程序清单 3-7 case.sh

```

#!/bin/bash
echo "please chose either 1,2 or3"
echo "[1]ls -l $1"

```

---

```

echo "[2]cat $1"
echo "[3]quit"
read response
case $response in
1)    ls -l $1;;
2)    cat $1;;
3)    echo "good bye"
esac

```

---

执行该脚本如下。

```

jianglinmei@ubuntu:~/sh$ source case.sh first.sh
please chose either 1,2 or3
[1]ls -l first.sh
[2]cat first.sh
[3]quit
1
-rwx--x-wx 1 jianglinmei jianglinmei 42 2012-09-22 15:18 first.sh
jianglinmei@ubuntu:~/sh$ source case.sh first.sh
please chose either 1,2 or3
[1]ls -l first.sh
[2]cat first.sh
[3]quit
2
#!/bin/bash
cd /tmp
echo "Hello, world!"
jianglinmei@ubuntu:~/sh$ source case.sh first.sh
please chose either 1,2 or3
[1]ls -l first.sh
[2]cat first.sh
[3]quit
3
good bye

```

在使用 case 语句时应注意以下几点。

(1) 每个模式字符串后面可有一条或多条命令，其中最后一条命令必须以两个分号（即`;;`）结束。

(2) 模式字符串中可以使用通配符，如程序清单 3-8 所示。

---

程序清单 3-8 case\_pattern.sh

```

#!/bin/bash
case $1 in
-f)    echo "find first.sh"
       find ~ -name "first.sh";;
-l)    echo "ls first.sh"
       ls -l first.sh;;
*)    echo "quit";;
esac

```

---

执行该脚本如下。

```

jianglinmei@ubuntu:~/sh$ source case_pattern.sh -f
find first.sh
/home/jianglinmei/sh/first.sh

```

```
jianglinmei@ubuntu:~/sh$ source case_pattern.sh -others
quit
```

(3) 如果一个模式字符串中包含多个模式，那么各模式之间应以竖线 (|) 隔开，表示各模式是“或”的关系，即只要给定字符串与其中一个模式相配，就会执行其后的命令表。如程序清单 3-9 所示。

程序清单 3-9 case\_multi\_pattern.sh

```
#!/bin/bash
read choice
case $choice in
time|date)      echo "the time is `date`.";;
dir|path)       echo "current directory is `pwd`.";;
*)              echo "bad argument.";;
esac
```

执行该脚本如下。

```
jianglinmei@ubuntu:~/sh$ source case_multi.sh
dir
current directory is /home/jianglinmei/sh.
jianglinmei@ubuntu:~/sh$ source case_multi.sh
date
the time is 2012 年 09 月 27 日 星期四 14:50:55 CST.
```

(4) 各模式字符串应是唯一的，不应重复出现，并且要合理安排它们的出现顺序。例如，不应将“\*”作为头一个模式字符串。因为“\*”可以与任何字符串匹配，它若第一个出现，就不会再检查其他模式了。

(5) case 语句以关键字 case 开头，以关键字 esac ( case 倒过来写 ) 结束。

(6) case 的退出状态 (返回值) 是整个结构中最后执行的那个命令的退出状态，若没有执行任何命令，则退出状态为零。

### 3.3.4 while 语句

Shell 中有三种用于循环的语句，即 while 语句、until 语句和 for 语句。

while 语句的一般格式是：

```
while 测试条件
do
    命令表
done
```

可以把关键字 do 与 while 写在同一行，此时测试条件应以“;”结束，形式如下。

```
while 测试条件; do
    命令表
done
```

while 语句的执行过程是：先进行条件测试，如果结果为真，则执行循环体 (关键字 do 和 done 之间的命令表)，然后再做条件测试……直到测试条件为假时，才终止 while 语句的执行。测试条件的使用方式和 if 语句一样，可以是一组命令或 3.4.1 小节介绍的所有条件测试。以程序清单 3-10

为例。

程序清单 3-10 while.sh

```
#!/bin/bash
while [ $1 ]
do
    if [ -f $1 ]; then
        echo -e "\ndisplay:$1"
        cat $1
    else
        echo "$1 is not a file name."
    fi
    shift
done
```

执行该脚本如下。

```
jianglinmei@ubuntu:~/sh$ source while.sh first.sh posvar.sh

display:first.sh
#!/bin/bash
cd /tmp
echo "Hello, world!"

display:posvar.sh
#!/bin/bash
echo 'Parameter number:' $#
echo 'All digit variables:' $0 $1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ${11}
echo '$*: '$*
echo '$@: '$@"
```

这段程序对各个给定的位置参数，首先判断其是否为普通文件，若是，则显示其内容，否则，显示它不是文件的信息。每次循环处理一个位置参数\$1，利用 shift 命令可把后续位置参数左移。

### 3.3.5 until 语句

until 语句的一般形式是：

```
until 测试条件
do
    命令表
done
```

可以把关键字 do 与 until 写在同一行，此时测试条件应以 “;” 结束，形式如下。

```
until 测试条件; do
    命令表
done
```

until 语句与 while 语句很相似，只是测试条件不同，即当测试条件为假时，才执行循环体中的命令表，直到测试条件为真时终止循环。以程序清单 3-11 为例。

程序清单 3-11 until.sh

```
#!/bin/bash
```

---

```

until [ -z "$2" ]; do
    cp $1 $2
    shift 2
done

if [ -n "$1" ]; then
    echo "bad parameter!"
fi

```

---

执行该脚本如下。

```

jianglinmei@ubuntu:~/sh$ source until.sh if.sh if2.sh while.sh while2.sh
jianglinmei@ubuntu:~/sh$ ls *2.sh
if2.sh while2.sh

```

程序清单 3-11 中，如果第二个位置参数不为空，就将文件 1 复制给文件 2，然后将位置参数左移两个位置。接着重复上面过程，直至没有第二个位置参数为止。退出 until 循环后，测试第一个位置参数，如果不为空，则显示参数不对。

### 3.3.6 for 语句

for 语句主要有两种使用方式：一种是值表方式，另一种是算术表达式方式。

#### 3.3.6.1 值表方式

值表方式的一般格式是：

```

for 变量 [in 值表]
do
    命令表
done

```

也可将所有命令写在一行，此时要注意各命令应以“;”结束。格式如下：

```
for 变量 [in 值表]; do 命令表; done
```

for 循环的循环变量的值取自给出的值表。其中，用方括号括起来的部分表示可省略。如果省略值表，则表示变量的值取自从“\$1”起的所有位置变量。以程序清单 3-12 为例。

程序清单 3-12 for.sh

---

```

#!/bin/bash
for day in Monday Wednesday Friday Sunday
do
    echo $day
done

```

---

执行该脚本如下。

```

jianglinmei@ubuntu:~/sh$ source for.sh
Monday
Wednesday
Friday
Sunday

```

程序清单 3-12 所示脚本的执行过程是，循环变量 day 依次取值表中各字符串（各字符串之间

以空格分隔), 即第一次将“monday”赋给 day, 然后进入循环体, 执行其中的命令——显示出 Monday。第二次将“wednesday”赋给 day, 然后执行循环中的命令, 显示出 Wednesday。依次处理, 当 day 把值表中各字符串都取过一次之后, 下面 day 的值就变为空串, 从而结束 for 循环。因此, 值表中字符串的个数就决定了 for 循环执行的次数。

又如:

```
jianglinmei@ubuntu:~/sh$ week=(Mon Tue Wed)
jianglinmei@ubuntu:~/sh$ for i in "${week[@]}"; do echo $i; done
Mon
Tue
Wed
```

for 语句的值表也可以是文件正则表达式, 格式为:

```
for 变量 in 文件正则表达式
do
    命令表
done
```

其执行过程是, 变量的值依次取当前目录下(或给定目录下)与正则表达式相匹配的文件名, 每取值一次, 就进入循环体执行命令表, 直至所有匹配的文件名取完为止, 退出 for 循环。例如:

```
jianglinmei@ubuntu:~/sh$ for file in *.sh; do wc -w $file; done
24 case_multi.sh
24 case_patter.sh
34 case.sh
26 cond_test.sh
7 first.sh
13 for.sh
63 if.sh
28 posvar.sh
11 setposvar.sh
25 shiftposvar.sh
11 test.sh
24 until.sh
29 while.sh
```

该语句统计当前目录下所有以.sh 结尾的文件的字数。

for 语句的值表还可以是全部的位置参数, 格式为:

```
for 变量 [in $*]
do
    命令表
done
```

其执行过程是, 变量依次取位置参数的值, 然后执行循环体中的命令表, 直至所有位置参数取完为止。以程序清单 3-13 为例。

---

### 程序清单 3-13 for\_pos.sh

```
#!/bin/bash

#display files under a given directory
# $1-the name of the directory
```

```

# $2-the name of files
dir=$1;shift
if [ -d $dir ]; then
    cd $dir
    for name; do
        if [ -f $name ]; then
            cat $name
            echo "end of ${dir}/${name}"
        else
            echo "invalid file name:${dir}/${name}"
        fi
    done
else
    echo "bad directory name:$dir"
fi

```

执行该脚本如下。

```

jianglinmei@ubuntu:~/sh$ source for_pos.sh . first.sh
#!/bin/bash
cd /tmp
echo "Hello, world!"
end of ./first.sh

```

执行这个 Shell 脚本时，如果第一个位置参数是合法的目录，那么就把后面给出的各个位置参数所对应的文件显示出来。若给出的文件名不正确，则显示出错信息。如果第一个位置参数不是合法的目录，则显示目录名不对。

### 3.3.6.2 算术表达式方式

for 语句的算术表达式的一般格式是：

```

for ((e1; e2; e3)); do 命令表; done
或者
for ((e1; e2; e3)); do
    命令表
done

```

其中， $e_1$ 、 $e_2$ 、 $e_3$  是算术表达式。它的执行过程与 C 语言中 for 语句相似。

- (1) 先按算术运算规则计算表达式  $e_1$ 。
- (2) 接着计算  $e_2$ ，如果  $e_2$  值不为 0，则执行命令表中的命令，并且计算  $e_3$ ；然后重复 (2)，直至  $e_2$  为 0，退出循环。

$e_1$ 、 $e_2$ 、 $e_3$  这三个表达式中任何一个都可以缺少，但彼此间的分号不能缺少。在此情况下，缺少的表达式的值就默认为 1（注意和命令测试不同，此处 1 表示“真”，0 表示“假”）。

整个 for 语句的返回值是命令表中最后一条命令执行后的返回值。如果任一算术表达式非法，那么该语句失败。

以程序清单 3-14 为例。

程序清单 3-14 for\_math.sh

```

#!/bin/bash

for ((i=1; i<=$1; i++)); do

```

---

```

for ((j=1; j<=i; j++)); do
    echo -n "* "
done
echo ""
done

```

---

执行该脚本如下。

```

jianglinmei@ubuntu:~/sh$ source for_math.sh 5
*
*
*
*
*
*
*
```

本例打印给定行数的\*号，第一行打印 1 个，第二行打印 2 个，依此类推。行数由用户在命令行上输入。

### 3.3.7 break、continue 和 exit

#### 3.3.7.1 break 命令

break 命令的作用是退出循环体。其语法格式是：

```
break [n]
```

其中，n 为一整数，表示要跳出几层循环，默认值是 1，即只跳出一层循环。执行 break 命令时，是从包含它的那个循环体中向外跳出。以程序清单 3-15 为例。

程序清单 3-15 break.sh

```

#!/bin/bash

num=$1
while true; do
    echo -n "$num "
    if ((--num == 0)); then
        break;
    fi
done
echo ""
```

---

执行该脚本如下。

```

jianglinmei@ubuntu:~/sh$ bash break.sh 10
10 9 8 7 6 5 4 3 2 1
```

程序清单 3-15 所示脚本首先读取第一个参数的值，然后输出该数值到 1 之间的所有数字。该脚本中 while 的测试条件总为真，它的唯一出口点就是执行 break 命令。

#### 3.3.7.2 continue 命令

continue 命令的作用是跳过本次循环中在它之后的循环体语句，回到本层循环的开头，进行下一次循环。其语法格式是：

```
continue [n]
```

其中，n 表示从包含 continue 语句的最内层循环体向外跳出几层循环，默认值为 1。以程序清

单 3-16 为例。

程序清单 3-16 continue.sh

```
#!/bin/bash

num=$1
for ((i=0; i<num; i++)); do
    if ((i % 2 == 0)); then
        continue;
    fi
    echo -n "$i "
done
echo ""
```

执行该脚本如下。

```
jianglinmei@ubuntu:~/sh$ source continue.sh 10
1 3 5 7 9
```

程序清单 3-16 所示脚本首先读取第一个参数的值，然后输出 0 到该数值之间的所有奇数数字（跳过偶数），其中 “ $i \% 2$ ” 表示求  $i$  除以 2 的余数。

### 3.3.7.3 exit 命令

`exit` 命令的功能是立即退出正在执行的 Shell 脚本，并设定退出状态（返回值）。其语法格式是：

```
exit [n]
```

其中，`n` 是设定的退出状态（返回值）。如果未显式给出 `n` 值，则退出状态为最后一个命令的执行状态。

### 3.3.7.4 select 语句

`select` 语句通常用于菜单的设计，它自动完成接收用户输入的整个过程，包括显示一组菜单项及读取用户的选择。

`select` 语句的语法格式为：

```
select identifier [in word...]
do
    命令表
done
```

如果省略 [`inword`]，那么参数 `identifier` 就以位置参数 (`$1, $2...`) 作为给定的值。下面以程序清单 3-17 为例说明 `select` 的用法。

程序清单 3-17 continue.sh

```
#!/bin/bash

PS3="Choice? "
select choice in query add delete update exit
do
    case "$choice" in
        query) echo "Call query routine"; break;;
        add) echo "call add routine"; break;;
        delete) echo "Call delete routine"; break;;
        update) echo "Call update routine"; break;;
    esac
done
```

```

    exit)      echo "call exit routine"; break;;
    esac
done
echo "You input $REPLY; your choice is: $choice"

```

执行该脚本如下。

```

jianglinmei@ubuntu:~/sh$ source select.sh
1) query
2) add
3) delete
4) update
5) exit
Choice? 3 (用户输入 3)
Call delete routine
You input 3; your choice is: delete

```

本例中，执行 select 命令时，会列出用序号 1 到 n（本例中为 5）标记的菜单，序号与 in 之后给定的字（word）一一对应，然后给出提示（PS3 的值），并接收用户的选择（输入一个数字），并将该数据赋值给环境变量 REPLY。如果输入的数据是 1 到 n 中的一个值，那么参数 identifier（本例中为 choice）就置为该数字所对应的字。如果未输入数据，则重新显示该选择清单，该参数置为 null。对于每个选择都执行关键字 do 至 done 之间的命令，直至遇到 break 命令。

## 3.4 Shell 函数

同大多数高级语言一样，在 Shell 脚本中可以定义并调用函数。函数的定义格式为：

```

[function] 函数名()
{
    命令表
}

```

函数必须先定义，后使用。函数定义之后可被调用任意多次。调用函数时，直接使用函数名，不必带圆括号，就像使用一般命令一样。调用函数不会创建新的进程，而是在本 Shell 脚本所属的进程中执行。

Shell 脚本可利用位置变量向函数传递数据，函数中所用的位置参数 \$1, \$2 等对应函数调用语句中的实参。另外，在函数体内可以访问脚本中任何定义在函数外面的变量（全局变量），但是不能访问其他函数内用 local 关键字定义的局部变量。

通常，函数中的最后一个命令执行之后，函数即退出。也可利用 return 命令在任意位置退出函数。return 命令的语法格式是：

```
return [n]
```

其中，n 值是退出函数时的退出状态（返回值），如果未指定 n，则退出状态取最后一个命令的退出状态。

下面以程序清单 3-18 为例说明函数的用法。

## 程序清单 3-18 function.sh

```
#!/bin/bash

output()
{
    echo "-----"
    echo $a $b $c
    echo $1 $2 $3
    echo "-----"
}

input()
{
    local y

    echo "Please input value of x and y"
    read x y
}

a="Working directory"
b="is"
c=`pwd`

output You are welcome

x=
input
echo "Value of x is $x, value of y is $y"
```

执行该脚本如下。

```
jianglinmei@ubuntu:~/sh$ source function.sh
-----
Working directory is /home/jianglinmei/sh
You are welcome
-----
Please input value of x and y
10 20
Value of x is 10, value of y is
```

程序清单 3-18 所示脚本中，`output` 分别输出了全局变量 `a`、`b` 和 `c` 的值，以及位置变量`$1`、`$2` 和`$3` 的值，这是典型的脚本向函数传递数据的方式。`input` 函数读取两个数分别存放到变量 `x` 和 `y` 中，由于 `y` 是一个由 `local` 声明的局部变量，在函数体外不具有可见性，因此最后输出变量 `y` 的值为空。

## 3.5 Shell 内部命令

Shell 程序本身定义了一些命令，称为 Shell 内部命令，这些命令均在本 Shell 进程内执行。本书前面已经介绍过许多内部命令，如：

```
:、.、 source、break [n]、continue [n]、cd、echo、type、exit [n]、export、pwd、read、return
```

[n]、set、shift [n]、test、bg、fg 和 kill 等。

下面简要介绍另外一些内部命令。

#### 1. eval 命令

eval 命令的格式是：

```
eval [参数...]
```

eval 命令会首先扫描参数，所有参数被读取并连接成一个字符串，然后 eval 再将该字符串当成命令来执行。例如：

```
jianglinmei@ubuntu:~/sh$ var="wc -l first.sh"
jianglinmei@ubuntu:~/sh$ eval $var
3 first.sh
```

执行命令“eval \$var”时，首先进行变量替换——将\$var 替换成“wc -l first.sh”，然后再执行该命令，从而得到上述结果。

#### 2. exec 命令

exec 命令的格式是：

```
exec [arg...]
```

它在本 Shell 中执行由参数 arg 指定的命令，该命令将替代本 Shell 进程，即执行命令后命令行 Shell 将不复存在，命令退出整个 Shell 就退出了。例如：

```
jianglinmei@ubuntu:~/sh$ exec vi
```

启动 vi 后，“:q”退出 vi，将发现并不会返回到 Shell。

#### 3. readonly 命令

readonly 命令的格式是：

```
readonly [name...]
```

readonly 命令标记给定的 name（变量名）是只读的，如果没有给出参数，则列出所有只读变量的清单。使用 readonly 标记变量等价于使用 declare -r 标记变量。

#### 4. trap 命令

trap 命令的格式是：

```
trap [arg] [n]...
```

其中，arg 是当 Shell 收到信号 n 时所读取并执行的命令。当设置 trap 时，arg 被扫描一次。在 trap 被执行时，arg 也被扫描一次。所以通常用单引号把 arg 对应的部分括起来。trap 命令可用来设定接收到某个信号所完成的动作，忽略某个信号的影响或者恢复信息产生时系统预设的动作。

trap 命令按信号码顺序执行。允许的最高信号码是 16。试图对当前 Shell 已忽略的信号设置 trap 无效；试图对信号 11（内存故障）设置 trap，则产生错误。trap 命令有以下几种常见的用法。

（1）为某些信号另外指定处理方式。例如：

```
jianglinmei@ubuntu:~$ trap 'echo "breaking signal got" > ~/exit.txt' 0 1 2 3 15
```

本示例设置，当 Shell 脚本接收到信号 0（从 Shell 退出）、信号 1（挂起）、信号 2（中断）、信号 3（退出）或信号 15（过程结束）时，都将执行由单引号括起来的命令，即在家目录下创建

一个 exit.txt 文件，其中记录了接到信号的提示和时间。

执行该命令后，如果退出 Shell，再次登录进去，可在家目录下找到该文件，内容如下。

```
jianglinmei@ubuntu:~$ cat exit.txt
breaking signal at 2012年 09月 28日 星期五 08:35:53 CST
```

(2) 指定 arg 为空串以忽略信号，例如：

```
jianglinmei@ubuntu:~$ trap '' 0 1 2 3 15
```

本示例设置忽略所有 0 1 2 3 15 号信号。

(3) 不指定 arg，把信号的动作恢复成原来系统默认的动作。例如：

```
jianglinmei@ubuntu:~$ trap 0 1 2 3 15
```

## 5. set 命令

set 命令的功能主要有三个：显示已定义的全部变量、设置位置参数的值、设置 Shell 脚本的执行选项（标志项）。前二者本书前面已有介绍，下面介绍最后一个功能。

set 命令设置执行选项标志的一般格式是：

set -标志字符

或

set +标志字符

其标志字符前使用“-”表示打开该标志项，标志字符前使用“+”表示关闭该标志项。常用的标志项以下几种。

- a 对被修改或被创建的变量自动标记，表明要被转出 (export) 到后继命令环境中
- e 当一个简单命令以非零状态终止时，将立即退出 Shell。如果执行失败的命令是 while 或 until 循环、if 语句、由 && 或 || 连接的命令行的一部分，则不退出 Shell
- f 禁止路径名扩展，即禁用文件通配符
- h 打开命令行历史
- n 读命令但不执行。用来检查脚本的语法，交互式运行时不能开启
- x 使 Shell 对以后各命令行在完成参数替换且执行该行命令之前，先显示该行的内容。在重显命令行的行首有一个“+”号，随后才是执行该命令行的结果
- v 使 Shell 对以后各命令行都按原样先在屏幕上显示出来，然后才对命令行予以执行，并显示相应结果

## 6. wait 命令

wait 命令的格式是：

```
wait [n]
```

wait 命令等待进程 ID 为 n 的进程终止，并报告终止状态。如果没有指定 n，则等待所有当前活动的子进程终止。wait 命令的返回码始终是 0。

# 3.6 Shell 程序调试

任何人都无法保证程序编写完后就完全正确，因此不管采用什么编程语言，都会要面临程序

调试的问题。由于不像 C、Java 等高级语言具有专门的调试工具、甚至有集成的开发环境，Shell 程序只能通过一些很原始的方法进行调试，因此其调试可能会更加困难一些。不过幸运的是，一般 Shell 程序不会太长，这也降低了调试的复杂度。一般 Shell 脚本不能正常运行的原因可能有三种：运行环境问题、语法错误和逻辑错误。

### 1. 运行环境问题

可能的情况有：

- 使用非 bash Shell 运行按 bash 语法书写的脚本。为防此类错误，应遵守在脚本的第一行指定 bash 解释器的约定。
- PATH 环境变量中没有包括“.”（即当前目录），这是默认情况，所以要直接运行当前目录下的脚本，应在脚本名前加上“./”，或将“.”加入到 PATH 环境变量。

### 2. 语法错误

语法错误是编写程序时违反了所用编程语言的规则而造成的。它是在写脚本时最容易犯的错误，也是最容易修改的一类错误。这类错误包括：命令格式错误、特殊符号未转义错误、拼写错误、括号、引号不成对错误等。

出现语法错误时，Shell 无法解释代码，会显示出错信息，指明出了什么错误和出错的大致位置。

### 3. 逻辑错误

逻辑错误表现在程序能运行，但运行的结果和程序员预达到的目的不符合。此类错误是程序调试要解决的主要问题。

调试 Shell 程序的常用方法有两种。一是在使用 echo 或 printf 输出提示（如变量值），二是使用 set 命令打开“-x”或“-v”选项将 Shell 设置成跟踪模式。下面以程序清单 3-19 为例说明使用 set 命令“-x”选项进行调试的方法。

程序清单 3-19 debug.sh

```
#!/bin/bash

if [ $# -eq 0 ]; then
    echo 'Usage: debug -n'
    exit 1
fi

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

执行该脚本如下。

```
jianglinmei@ubuntu:~/sh$ set -x
jianglinmei@ubuntu:~/sh$ source debug.sh 1 2 3
+ source debug.sh 1 2 3
++ '[' 3 -eq 0 ']'
++ sum=0
++ '[' 3 -eq 0 ']'
++ (( sum=0+1 ))
```

```

++ shift
++ '[' 2 -eq 0 ']'
++ (( sum=1+2 ))
++ shift
++ '[' 1 -eq 0 ']'
++ (( sum=3+3 ))
++ shift
++ '[' 0 -eq 0 ']'
++ echo 6
6
jianglinmei@ubuntu:~/sh$ set +x
+ set +x

```

从执行结果可以看出，当 Shell 设置了“-x”选项后，debug.sh 脚本中的每一条可执行命令都被解析替换成最终的命令行并显示了出来。通过对原始命令和解析后的命令可以较容易地发现一些逻辑错误。

## 3.7 小结

本章系统地介绍了 Linux 环境下 Shell ( bash ) 编程方法。在基础知识部分介绍了 Shell 脚本程序的构成、运行方法和多 Shell 命令的组合方法。Shell 变量的使用是 Shell 编程的最基本知识，Shell 变量是弱类型的变量，其使用以及计算方法很灵活也有很多特别之处，其中一些特殊变量需要读者去记忆。Shell 编程语言的控制结构和大多数的高级语言类似，重要的区别在于其中的条件测试。由于 Shell 语言中的每条语句其实都是由命令构成，而且变量实际上都是字符串，这使得其条件测试较为复杂，读者应仔细区分每一种条件测试方法的测试运算符的作用。Shell 中函数的使用较为简单，需要注意的是局部变量的定义方法和参数的传递方法。本章还介绍了 Shell 中的一些内部命令的使用，其中有些命令有助于完成一些复杂的程序功能。本章最后介绍了 Shell 调试的方法，这是编写程序必须具备的基本能力。

## 3.8 习题

- (1) 如何指定 Shell 脚本的解释器？有哪几种运行 Shell 脚本程序的方法？其区别是什么？
- (2) 有几种主要 Shell 命令？它们运行时有何区别？
- (3) 如何查看命令类型？同名不同类型的命令的执行顺序是怎样的？
- (4) Shell 有整型变量吗？如何指定变量的属性？如何进行数值运算？
- (5) 命令的退出状态有何作用？有哪些将命令连接在一起的方法？它们的区别是什么？
- (6) 有哪两种复合命令，其区别是什么？
- (7) Shell 变量必须先定义后使用吗？引用变量的语法是怎样的？
- (8) 如何给数组元素赋值，有几种赋值方法？如何引用数组的某个元素？如何引用数组的所有元素？如何取得数组元素个数？
- (9) 有哪些变量引操作符，它们分别能完成什么功能？
- (10) 什么时候要以及为什么要用 export 语句？

- (11) 有哪几类条件测试方法？使用 test 命令进行条件测试有哪几类测试操作符？  
(12) 什么时候应使用 let 命令进行条件测试而不是使用 test 命令，其好处是什么？  
(13) 各控制结构如果要把多个关键字写在同一行应注意什么？for 语句各部分的执行顺序是怎样的？  
(14) set 命令的“-x”选项有何作用？  
(15) 请编程实现：判断任一参数给出的是不是字符设备文件，如果是则将其拷贝到 /dev 目录下。  
(16) 请编写程序，分别用 while、until 和 for 循环计算 1 到位置参数\$1 所给出的数之间所有是 3 的倍数的数之和。  
(17) 请编程实现：打印边长为 n 的由“\*”号组成的等边三角形。形如：

```
*  
* *  
* * *  
* * * *  
* * * *
```

其中变量 n 的值通过命令行参数传入。

- (18) 设/tmp 路径下有 1000 个文件，文件名的格式为 filename\_YYYYMMDD\_xxx.dat（其中 YYYYMMDD 为 8 位数字表示的日期，xxx 为三位数字表示的序列号），例如：backup\_20040108\_089.dat。请编程实现：将这些文件名依次改名为 YYYYMMDD\_filename\_yyy.dat，其中 yyy=1000-xxx。例如，将 backup\_20040108\_089.dat 改名为 20040108\_backup\_911.dat。