第 3 章　变量和引用

在任何程序设计语言里面，变量都是一个非常重要的话题。变量是程序设计语言中不可缺少的元素。正确、恰当地使用变量可以增加程序的可读性，提高程序的健壮性和灵活性。本章将从变量的基础知识开始，依次介绍什么是变量、变量的赋值和替换，以及变量的引用。

本章主要涉及的知识点如下所述。

- 深入认识变量：主要介绍什么是变量、变量的命名、变量的类型、变量的有效范围，以及系统变量和用户自定义变量等。
- 变量的赋值和替换：主要介绍如何为变量赋值、如何取得变量的值、如何清空变量的值，以及如何进行变量替换。
- 引用：主要介绍什么是全引用、部分引用、命令替换和转义等。

3.1　深入认识变量

在程序设计语言中，变量是一个非常重要的概念。也是初学者在进行 Shell 程序设计之前必须掌握的一个非常基础的概念。只有理解变量的使用方法，才能设计出良好的程序。本节将介绍 Shell 中变量的相关知识。

3.1.1　什么是变量

顾名思义，变量就是程序设计语言中的一个可以变化的量，当然，可以变化的是变量的值。变量在几乎所有的程序设计语言中都有定义，并且其涵义也大同小异。从本质上讲，变量就是在程序中保存用户数据的一块内存空间，而变量名就是这块内存空间的地址。

在程序的执行过程中，保存数据的内存空间的内容可能会不断地发生变化，但是，代表内存地址的变量名却保持不变。

由于变量的值是在计算机的内存中，所以当计算机被重新启动后，变量的值将会丢失。因此，对于需要长久保存的数据，应该写入到磁盘中，避免存储在变量中。

3.1.2　变量的命名

对初学者来说，可以简单地认为，变量就是保存在计算机内存中的一系列的键值对。例如：

```
str="hello"
```

在上面的语句中，等号前面的部分就是键，等号后面的就是值。用户使用变量的目的

就是通过键来存取不同的值。

在程序设计语言中，一般将上述语句中的键称为变量名。因此，用户是通过变量名来对变量所代表的值进行存取。

在不同的程序设计语言中，对于变量名的要求有所不同。在 Shell 中，变量名可以由字母、数字或者下划线组成，并且只能以字母或者下划线开头。对于变量名的程序，Shell 并没有做出明确的规定。因此，用户可以使用任意长度的字符串作为变量名。但是，为了提高程序的可读性，建议用户使用相对较短的字符串作为变量名。

在一个设计良好的程序中，变量的命名有着非常大的学问。通常情况下，用户应该尽可能选择有明确意义的英文单词作为变量名，尽量避免使用拼音或者毫无意义的字符串作为变量名。这样的话，用户通过变量名就可以了解该变量的作用。

例如，下面的变量名都是非常好的选择：

```
PATH=/sbin
UID=100
JAVA_HOME="/usr/lib/jvm/jre-1.6.0-openjdk.x86_64/bin/../.."
SSHD=/usr/sbin/sshd
```

而下面的变量名则相对可读性较差：

```
a="123"
str1="hello"
```

因为程序的阅读者没有从变量名上面获取到任何有用的信息。当变量较多的时候，程序设计者本身也会搞不清楚这些变量的作用是什么，从而导致程序发生错误。

注意：在 Shell 语言中，变量名的大小写是敏感的，因此，大小写不同的两个变量名并不代表同一个变量。

3.1.3　变量的类型

与变量密切相关的有两个概念，其中一个是变量的类型，另外一个是变量的作用域。此处先介绍变量的类型，有效范围将在随后介绍。

根据变量类型确定的时间，可以将程序设计语言分为两类，分别是静态类型语言和动态类型语言。其中，静态类型语言是在程序的编译期间就确定变量类型的语言，例如 Java、C++和 PASCAL，在这些语言中使用变量时，必须首先声明其类型。动态设计语言是在程序执行过程中才确定变量的数据类型的语言。常见的动态语言有 VBScript、PHP 及 Python 等。在这些语言中，变量的数据类型根据第一次赋给该变量的值的数据类型来确定。

同样，根据是否强制要求类型定义，可以将程序设计语言分为强类型语言和弱类型语言。强类型语言要求用户在定义变量时必须明确指定其数据类型，例如 Java 和 C++。在强类型语言中，数据类型之间的转换非常重要。与之相反，弱类型语言则不要求用户明确指定变量的数据类型，例如 VBScript。用户可以将任意类型的数值赋给该变量。并且，变量的数据类型之间的转换也无需明确进行。

Shell 是一种动态类型语言和弱类型语言，即在 Shell 中，变量的数据类型无需显示地声明，变量的数据类型会根据不同的操作有所变化。准确地讲，Shell 中的变量是不分数据

类型的，统一地按照字符串存储。但是根据变量的上下文环境，允许程序执行一些不同的操作，例如字符串的比较和整数的加减等。

【例 3-1】 演示 Shell 变量的数据类型，代码如下：

```
#-----------------------------/chapter3/ex3-1.sh------------------
#! /bin/bash

#定义变量 x，并且赋值为 123
x=123
#变量 x 加 1
let "x += 1"
#输出变量 x 的值
echo "x = $x"
#显示空行
echo
#替换 x 中的 1 为 abc，并且将值赋给变量 y
y=${x/1/abc}
#输出变量 y 的值
echo "y = $y"
#声明变量 y
declare -i y
#输出变量 y 的值
echo "y = $y"
#变量 y 的值加 1
let "y += 1"
#输出变量 y 的值
echo "y = $y"
#显示空行
echo
#将字符串赋给变量 z
z=abc22
#输出变量 z 的值
echo "z = $z"
#替换变量 z 中的 abc 为数字 11，并且将值赋给变量 m
m=${z/abc/11}
#输出变量 m 的值
echo "m = $m"
#变量 m 加 1
let "m += 1"
#输出变量 m 的值
echo "m = $m"

echo
#将空串赋给变量 n
n=""
#输出变量 n 的值
echo "n = $n"
#变量 n 加 1
let "n += 1"
echo "n = $n"
echo
#输出空变量 p 的值
echo "p = $p"
# 变量 p 加 1
let "p += 1"
echo "p = $p"
```

为了便于介绍，首先看一下以上程序的执行结果，如下：

```
01  [root@linux chapter3]# ./ex3-1.sh
02  x = 124
03
04  y = abc24
05  y = abc24
06  y = 1
07
08  z = abc22
09  m = 1122
10  m = 1123
11
12  n =
13  n = 1
14
15  p =
16  p = 1
```

在【例 3-1】中，第 5 行定义了变量 x，并且将一个整数值赋给该变量，所以变量 x 的数据类型为整数。第 7 行使用 let 语句将变量 x 的值加 1，从而变成了 124，输出结果的第 2 行正是程序的第 9 行的 echo 语句的执行结果。

代码的第 13 行将变量 x 的值中的数字 1 替换成了字符串 abc，并且将替换后的结果赋给变量 y，所以变量 y 实际上是一个字符串。执行结果的第 4 行是代码的第 15 行的 echo 语句的执行结果。

代码的第 17 行是使用 declare 语句声明整数变量 y，但是这个语句并不影响当前变量 y 的值，因此，执行结果的第 5 行是代码第 19 行的 echo 语句的执行结果。

代码的第 21 行是将变量 y 加 1，用户可能已经注意到，此时变量 y 的值为 abc24，这个值是一个含有字母和数字的字符串。为了能够执行加法运算，Shell 会自动进行数据类型的转换，如果遇到含有非数字的字符串，则该字符串将被转换成整数 0，所以，当执行加 1 运算之后，变量 y 的值就变成 1。执行结果的第 6 行是代码第 23 行的 echo 语句的输出。

代码的第 27 行将一个含有字母的字符串赋给变量 z，然后在第 29 行输出该变量的值，得到了结果的第 8 行。

代码的第 31 行将变量 z 的值中的字母 abc 替换为整数 11，并且将替换后的结果赋给变量 m，在第 33 行输出变量 m 的值，即执行结果的第 9 行。

代码的第 35 行将变量 m 的值加 1，由于此时 y 的值为 1122，这是一个完全由数字组成的字符串，所以 Shell 可以将其转换为整数，然后执行加法运算，得到结果为 1123。

代码的第 41 行～第 52 行分别测试了在空串以及没有定义的变量的情况下，执行加法运算的执行结果。从上面的执行结果可以得知，在这两种情况下，变量的值都会被转换为整数 0。

从上面的执行结果可以看到，Shell 中的变量非常灵活，可以参与任何运算。实际上，在 Shell 中，一切变量都是字符串类型的。

3.1.4　变量的定义

在 Shell 中，通常情况下用户可以直接使用变量，无需先进行定义，当用户第一次使用某个变量名时，实际上就同时定义了这个变量，在变量的作用域内，用户都可以使用该

变量。

【例 3-2】 演示通过直接使用变量来定义变量，代码如下：

```
#-----------------------------/chapter3/ex3-2.sh------------------
#! /bin/bash

#定义变量 a
a=1
#定义变量 b
b="hello"
#定义变量 c
c="hello world"
```

在上面的代码中，第 5 行定义了一个名称为 a 的变量，同时将一个数字赋给该变量。第 7 行定义了一个名称为 b 的变量，同时将一个字符串赋给该变量。第 9 行定义了一个变量 c，同时将一个包含空格的字符串赋给该变量。在 Shell 语言中，如果变量的值包含空格，则一定要使用引号引用起来。

尽管通过以上方式可以非常方便地定义变量，但是，对于变量的某些属性却不容易控制，例如，变量的类型和读写属性等。为了更好地控制变量的相关属性，bash 提供了一个名称为 declare 的命令来声明变量，该命令的基本语法如下：

```
declare attribute variable
```

其中，attribute 表示变量的属性，常用的属性有如下所述。

- -p：显示所有变量的值。
- -i：将变量定义为整数。在之后就可以直接对表达式求值，结果只能是整数。如果求值失败或者不是整数，就设置为 0。
- -r：将变量声明为只读变量。只读变量不允许修改，也不允许删除。
- -a：变量声明为数组变量。但这没有必要。所有变量都不必显式定义就可以用做数组。事实上，在某种意义上，所有变量都是数组，而且赋值给没有下标的变量与赋值给下标为 0 的数组元素相同。
- -f：显示所有自定义函数，包括名称和函数体。
- -x：将变量设置成环境变量，这样在随后的脚本和程序中可以使用。

参数 variable 表示变量名称。

注意：declare 命令又写做 typeset。

【例 3-3】 演示使用不同的方法声明变量，导致变量在不同的环境下表现出不同的行为，代码如下：

```
#-----------------------------/chapter3/ex3-3.sh------------------
#! /bin/bash

定义变量 x，并将一个算术式赋给它
x=6/3
echo "$x"
#定义变量 x 为整数
declare -i x
echo "$x"
```

```
10  #将算术式赋给变量 x
11  x=6/3
12  echo "$x"
13  #将字符串赋给变量 x
14  x=hello
15  echo "$x"
16  #将浮点数赋给变量 x
17  x=3.14
18  echo "$x"
19  #取消变量 x 的整数属性
20  declare +i x
21  #重新将算术式赋给变量 x
22  x=6/3
23  echo "$x"
24  #求表达式的值
25  x=$[6/3]
26  echo "$x"
27  #求表达式的值
28  x=$((6/3))
29  echo "$x"
30  #声明只读变量 x
31  declare -r x
32  echo "$x"
33  #尝试为只读变量赋值
34  x=5
35  echo "$x"
```

以上程序的执行结果如下：

```
01  [root@linux chapter3]# ./ex3-3.sh
02  6/3
03  6/3
04  2
05  0
06  ./ex3-3.sh: line 15: 3.14: syntax error: invalid arithmetic operator
(error token is ".14")
07  0
08  6/3
09  2
10  2
11  2
12  ./ex3-3.sh: line 32: x: readonly variable
13  2
```

下面对比执行结果分析一下【例 3-3】中的代码。第 5 行使用通常的方法定义了一个变量 x，并且将一个算术式作为初始值赋给该变量。第 6 行输出变量 x 的值。前面已经讲过，Shell 中将所有的数据都看做是字符串来存储的，所以在程序执行的时候，Shell 并不将 6/3 当成一个将被求值的算术式，而是作为一个普通的字符串，所以第 6 行直接输出了这个算术式本身，得到了结果的第 2 行。

代码的第 8 行使用 declare 语句声明了变量 x 为整数，但是程序并没对变量 x 重新赋值，所以第 9 行的 echo 语句的执行结果仍然得到算术式本身，即结果的第 3 行。

代码的第 11 行对变量 x 重新赋值，将前面的算术表达式赋给它。因为当变量被声明为整数之后，可以直接参与算术运算，所以第 12 行的 echo 语句中输出了算术式的值，即结果的第 4 行。

第 14 行尝试将一个字符串值赋给整数变量 x，并且在第 15 行使用 echo 语句输出 x 的值。在 Shell 中，如果变量被声明成整数，把一个结果不是整数的表达式赋值给它时，就会变成 0。因此，结果第 5 行中的 0 是代码第 15 行的 echo 语句的输出。

第 17 行将一个浮点数赋给变量 x，因为 bash 并不内置对浮点数的支持，所以得到了执行结果中的第 6 行的错误消息，此时，变量 x 的值变为 0，即结果的第 7 行中的 0。

第 20 行取消变量 x 的整数类型属性，第 22 行重新将算术式赋给变量 x，并且在第 23 行使用 echo 语句输出变量 x 的值。由于此时变量 x 已经不是整数变量，所以不能直接参数算术运算。因此，变量 x 的值仍然得到了算术式本身，即结果的第 8 行。

在 Shell 中，为了得到算术式的值，可以有两种方法，其中一种就是使用方括号，即第 25 行中的方式。结果的第 9 行正是此时变量 x 的值。另外一种是使用圆括号，即第 28 行中的方式。从执行结果可以得知，这两种方式都可以得到用户所期望的结果。

第 31 行使用-r 选项声明了一个只读变量，第 43 行尝试为该变量重新赋值，从而得到了结果中第 12 行的错误消息。此时变量 x 的值仍然是 2，所以才有结果中的第 13 行。

3.1.5　变量和引号

在 Shell 编程中，正确理解引号的作用非常重要。Shell 语言中一共有 3 种引号，分别为单引号（' '）、双引号（" "）和反引号（` `）。这 3 种引号的作用是不同的，其中单引号括起来的字符都作为普通字符出现；由双引号括起来的字符，除“$”、“\”、“`”和“"”这几个字符仍是特殊字符并保留其特殊功能外，其余字符仍作为普通字符对待；由反引号括起来的字串被 Shell 解释为命令，在执行时，Shell 首先执行该命令，并以它的标准输出结果取代整个反引号（包括两个反引号）部分。关于单引号和双引号的作用，将在后面的引用部分介绍，下面举例说明一下反引号的使用方法。

【例 3-4】 演示反引号使用方法，代码如下：

```
01  #------------------------------/chapter3/ex3-4.sh------------------
02  #! /bin/bash
03
04  #输出当前目录
05  echo "current directory is `pwd`"
```

在上面的代码中，第 5 行的语句中包含一个由反引号引用起来的 Shell 命令 pwd。以上命令的执行结果如下：

```
[root@linux chapter3]# ./ex3-4.sh
current directory is /root/chapter3
```

从上面的执行结果可以得知，代码的第 5 行在执行的过程中会首先执行 pwd 命令，用该命令的执行结果取代命令所在的位置，然后执行 echo 语句。

注意：反引号是键盘左上角的波浪号“～”下面的那个符号。

3.1.6　变量的作用域

接下来讨论 Shell 语言中的与变量密切相关的另外一个概念，即变量的作用域。与其他程序设计语言一样，Shell 中的变量也分为全局变量和局部变量两种。下面分别介绍这两

种变量的作用域。

1．全局变量

通常认为，全局变量是使用范围较大的变量，它不仅限于某个局部使用。在 Shell 语言中，全局变量可以在脚本中定义，也可以在某个函数中定义。在脚本中定义的变量都是全局变量，其作用域为从被定义的地方开始，一直到 Shell 脚本结束或者被显式地删除。

【例 3-5】 演示全局变量的使用方法，代码如下：

```
#-----------------------------/chapter3/ex3-5.sh------------------
#! /bin/bash

#定义函数
func()
{
   #输出变量 x 的值
   echo "$v1"
   #修改变量 x 的值
   v1=200
}
#在脚本中定义变量 x
v1=100
#调用函数
func
#输出变量 x 的值
echo "$v1"
```

在上面的代码中，第 5 行～第 11 定义了名称为 func()的函数，第 8 行在函数内部输出全局变量 v1 的值，第 10 行修改全局变量 v1 的值为 200，第 13 行在脚本中，即函数外面定义了变量 v1，该变量是全局变量。第 15 行调用函数 func()，第 17 行重新输出修改后的变量 v1 的值。

该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-5.sh
100
200
```

在上面的执行结果中，100 是第 8 行的 echo 语句的输出，从执行结果可以得知，在函数 func()内部可以访问全局变量 v1。200 是第 17 行的 echo 语句的输出，这是因为程序的第 10 行在函数 func()内部修改了变量 v1 的值。从【例 3-5】的执行结果可以得知，在脚本中定义的变量为全局变量，不仅可以在脚本中直接使用，还可以在函数内部直接使用。

除了在脚本中定义全局变量之外，在函数内部定义的变量默认情况下也是全局变量，其作用域为从函数被调用时执行变量定义的地方开始，一直到 Shell 脚本结束或者被显式地删除为止。

【例 3-6】 演示在函数内部定义全局变量的方法，代码如下：

```
#-----------------------------/chapter3/ex3-6.sh------------------
#! /bin/bash

#定义函数
func()
```

```
06  {
07     #在函数内部定义变量
08     v2=200
09  }
10  #调用函数
11  func
12  #输出变量的值
13  echo "$v2"
```

在上面的代码中，第 5 行～第 9 行定义了名称为 func()的函数，其中在第 8 行定义了一个名称为 v2 的变量。第 11 行调用 func()函数，第 13 行输出变量 v2 的值。

该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-6.sh
200
```

之所以会得到 200，是因为在 Shell 中，默认情况下，函数内部定义的变量也属于全局变量。因此，在代码的第 8 行定义的变量 v2 在函数外部仍然可以使用。

注意：函数的参数是局部变量。

2．局部变量

与全局变量相比，局部变量的使用范围较小，通常仅限于某个程序段访问，例如函数内部。在 Shell 语言中，可以在函数内部通过 local 关键字定义局部变量，另外，函数的参数也是局部变量。

【例 3-7】 演示使用 local 关键字定义局部变量。本例对【例 3-6】的代码稍作改动，在定义变量 v2 的时候使用 local 关键字，代码如下：

```
01  #-----------------------------/chapter3/ex3-7.sh------------------
02  #! /bin/bash
03
04  #定义函数
05  func()
06  {
07     #使用 local 关键字定义局部变量
08     local v2=200
09  }
10  #调用函数
11  func
12  #输出变量的值
13  echo "$v2"
```

该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-7.sh

```

从上面的执行结果可以得知，由于在函数内部使用 local 关键字显式地定义了局部变量，所以在函数外面不能获得该变量的值。第 13 行的 echo 语句仅仅输出了空值。

注意：关于函数的详细介绍请参见本书第 6 章。

如果用户在函数外面定义了一个全局变量，同时在某个函数内部又存在相同名称的局部变量，则在调用该函数时，函数内部的局部变量会屏蔽函数外部定义的全局变量。也就是说，在出现同名的情况下，函数内部的局部变量会优先被使用。

【例 3-8】 演示全局变量和局部变量的区别。在本例中，定义两个名称都为 v1 的变量，其中一个为全局变量，另外一个为局部变量，然后分别比较这两个变量的不同的行为，代码如下：

```
01  #-----------------------------/chapter3/ex3-8.sh------------------
02  #! /bin/bash
03
04  #定义函数
05  func()
06  {
07     #输出全局变量 v1 的值
08     echo "global variable v1 is $v1"
09     #定义局部变量 v1
10     local v1=2
11     #输出局部变量 v1 的值
12     echo "local variable v1 is $v1"
13  }
14  #定义全局变量 v1
15  v1=1
16  #调用函数
17  func
18  #输出全局变量 v1 的值
19  echo "global variable v1 is $v1"
```

在上面的代码中，第 5 行～第 13 行定义了函数 func()。由于函数内部的局部变量是在第 10 行定义的，所以第 8 行的 echo 语句输出的是全局变量 v1 的值。第 10 行使用 local 关键字定义相同名称的局部变量 v1，并且赋值为 2。第 12 行输出了局部变量 v1 的值。第 15 行定义了全局变量 v1，并且赋值为 1。第 17 行调用函数 func()。第 19 行输出全局变量 v1 的值。

该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-8.sh
global variable v1 is 1
local variable is 2
global variable is 1
```

从上面的执行结果可以得知，在函数 func()中，第一次输出的是全局变量 v1 的值，第二次输出的是局部变量 v1 的值。尽管在函数内部修改了变量 v1 的值，但是这只影响到局部变量 v1，所以在函数外部输出的仍然是全局变量的值。

注意： Shell 变量中的符号“$”表示取变量的值。只有在取值的时候才使用，定义和赋值时无需使用符号“$”。另外，实际上 Shell 中变量的原型为${var}，而常用的书写形式$var 只是一个简写。在某些情况下，简写形式会导致程序执行错误。

3.1.7　系统变量

Shell 语言的系统变量主要在对参数判断和命令返回值判断时使用，包括脚本和函数的

参数，以及脚本和函数的返回值。Shell 语言中的系统变量并不多，但是十分有用，特别是在做一些参数检测的时候。表 3-1 列出了常用的系统变量。

表 3-1　Shell 中常用的系统变量

变　量	说　明
$*n*	*n* 是一个整数，从 1 开始，表示参数的位置，例如$1 表示第 1 个参数，$2 表示第 2 个参数等
$#	命令行参数的个数
$0	当前 Shell 脚本的名称
$?	前一个命令或者函数的返回状态码
$*	以“参数 1 参数 2……”的形式将所有的参数通过一个字符串返回
$@	以“参数 1”“参数 2”……的形式返回每个参数
$$	返回本程序的进程 ID（PID）

【例 3-9】 演示常用系统变量的使用方法。在本例中通过 Shell 系统变量来获取不同的信息，代码如下：

```
01  #-----------------------------/chapter3/ex3-9.sh------------------
02  #! /bin/bash
03
04  #输出脚本的参数个数
05  echo "the number of parameters is $#"
06  #输出上一个命令的退出状态码
07  echo "the return code of last command is $?"
08  #输出当前脚本名称
09  echo "the script name is $0"
10  #输出所有的参数
11  echo "the parameters are $*"
12  #输出其中的几个参数
13  echo "\$1=$1;\$2=$2;\$11=$11"
```

在上面的代码中，第 5 行使用变量$#获取当前脚本的参数个数，第 7 行使用变量$?获取上一个脚本或者命令的返回状态码，第 9 行通过变量$0 获取当前脚本的名称，第 11 行通过变量$*以一个字符串的形式返回所有的参数值，第 13 行通过变量$n 输出其中几个参数的值。

该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-9.sh a b c d e f g h i j k l m n
the number of parameters is 14
the return code of last command is 0
the script name is ./ex3-9.sh
the parameters are a b c d e f g h i j k l m n
$1=a;$2=b;$11=a1
```

在执行 ex3-9.sh 脚本文件时，为该程序提供了 14 个参数，所以代码第 5 行的输出结果为 14。同时由于代码第 5 行的 echo 语句执行成功，所以代码第 7 行的输出结果为 0。代码第 9 行的 echo 语句输出了当前脚本文件的名称为 ex3-9.sh。第 11 行通过变量$*返回所有的参数的值。第 13 行分别通过变量$1、$2 和$11 获取第 1 个、第 2 个和第 11 个参数的值。从上面的执行结果可以得知，第 1 个和第 2 个参数的值都已经正确获取，但是第 11 个参数

的值却输出了“a1”，而非希望得到的 k。为什么会得到这种错误的结果呢？

原来在 Shell 语言中，使用$n 的形式获取位置参数时，Shell 通常的变量名只是一位数字，即 1～9。因此，在上面的程序中，使用变量$11 获取第 11 个参数的值时，Shell 会将“$1”作为变量名，导致了获取结果实际上是第 1 个参数的值。而最后的“1”则是变量名$11 中的第 2 个“1”。这个数字会直接与前面的字符串连接在一起。

为了能够使 Shell 正确地知道哪些部分是变量名，用户可以使用大括号来界定变量名。下面将变量名$11 中的数字 11 用大括号括起来，如下：

```
echo "\$1=$1;\$2=$2;\$11=${11}"
```

此时再次执行【例 3-9】的脚本文件，就可以得到正确的结果，如下所示。

```
[root@linux chapter3]# ./ex3-9.sh a b c d e f g h i j k l m n
the number of parameters is 14
the return code of last command is 0
the script name is ./ex3-9.sh
the parameters are a b c d e f g h i j k l m n
$1=a;$2=b;$11=k
```

注意：【例 3-9】中的反斜线“\”称为转义字符，用于将一些 Shell 中的特殊字符转换为普通字符，例如“$”或者“’”等。

3.1.8　环境变量

Shell 的环境变量是所有 Shell 程序都可以使用的变量。Shell 程序在运行时，都会接收一组变量，这组变量就是环境变量。环境变量会影响到所有脚本的执行结果。表 3-2 列出了常用的 Shell 变量。

表 3-2　常用的 Shell 环境变量

变量	说　明
PATH	命令搜索路径，以冒号为分隔符。注意与 Windows 下不同的是，当前目录不在系统路径里
HOME	用户主目录的路径名，是 cd 命令的默认参数
COLUMNS	定义了命令编辑模式下可使用命令行的长度
HISTFILE	命令历史文件
HISTSIZE	命令历史文件中最多可包含的命令条数
HISTFILESIZE	命令历史文件中包含的最大行数
IFS	定义 Shell 使用的分隔符
LOGNAME	当前的登录名
SHELL	Shell 的全路径名
TERM	终端类型
TMOUT	Shell 自动退出的时间，单位为秒，若设为 0 则禁止 Shell 自动退出
PWD	当前工作目录

除了表 3-2 列出的一些环境变量之外，用户还可以使用 set 命令列出所有的环境变量，如下：

```
[root@linux chapter3]# set | more
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:force_fignore:hos
tcomplete:interactive_comments:login_shell:progcomp:promptvars:sourcepa
th
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSINFO=([0]="4" [1]="1" [2]="2" [3]="1" [4]="release" [5]=
"x86_64-redhat-linux-gnu")
BASH_VERSION='4.1.2(1)-release'
COLORS=/etc/DIR_COLORS
COLUMNS=235
CVS_RSH=ssh
DIRSTACK=()
EUID=0
GROUPS=()
G_BROKEN_FILENAMES=1
HISTCONTROL=ignoredups
HISTFILE=/root/.bash_history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/root
HOSTNAME=linux
…
```

用户如果想要使用环境变量，则可以通过相应的变量名来获取。

【例 3-10】 演示通过环境变量来获取与当前 Shell 有关的一些环境变量的值，代码如下：

```
01  #-----------------------------/chapter3/ex3-10.sh-------------------
02  #! /bin/bash
03
04  #输出命令搜索路径
05  echo "commands path is $PATH"
06  #输出当前的登录名
07  echo "current login name is $LOGNAME"
08  #输出当前用户的主目录
09  echo "current user's home is $HOME"
10  #输出当前的 Shell
11  echo "current shell is $SHELL"
12  #输出当前工作目录
13  echo "current path is $PWD"
```

在上面的代码中，第 5 行输出当前用户所设置的命令搜索路径，第 7 行输出当前用户的登录名，第 9 行输出当前用户的主目录，第 11 行输出当前 Shell 的全路径，第 13 行输出当前的工作路径。

该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-10.sh
commands path is /usr/lib64/qt-3.3/bin:/usr/local/sbin:/usr/local/
bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin:/usr/pgsql-9.2/bin
current login name is root
current user's home is /root
```

```
current shell is /bin/bash
current path is /root/chapter3
```

注意：按照惯例，Shell 中的环境变量全部使用大写字母表示。

3.2　变量赋值和清空

在了解了变量的基础知识之后，接下来再介绍一下 Shell 语言中变量的赋值和变量的销毁。

3.2.1　变量赋值

在 Shell 语言中，通常情况下变量并不需要专门的定义和初始化语言。一个没有初始化的 Shell 变量被认为是一个空字符串。用户可以通过变量的赋值操作来完成变量的声明并赋予一个特定的值。并且可以通过赋值语句为一个变量多次赋值，以改变其值。

在 Shell 中，变量的赋值使用以下语法：

```
variable_name=value
```

其中，varibale_name 表示变量名，关于变量名的命名规则，前面已经介绍过了，不再重复说明。value 表示将要赋给变量的值。一般情况下，Shell 中将所有普通变量的值都看做字符串。如果 value 中包含空格、制表符和换行符，则必须用单引号或者双引号将其引起来。双引号内允许变量替换，而单引号则不可以。

中间的等于号“=”称为赋值符号，赋值符号的左右两边不能直接跟空格，否则 Shell 会将其视为命令。

例如，下面都是一些正确的赋值语句：

```
v1=Linux
v2='RedHat Linux'
v3="RedHat Linux $HOSTTYPE"
v4=12345
```

此外，Shell 允许只包含数字的变量值参与数值运算，例如上面的 v4。另外，在上面的 v3 变量值中，包含一个特殊符号“$”，这个符号的作用是取某个变量的值，将在后面的变量替换中介绍。

3.2.2　引用变量的值

当变量赋值完成之后，就需要使用变量的值。在 Shell 中，用户可以通过在变量名前面加上“$”来获取该变量的值。实际上，在前面的许多例子中，我们已经多次使用了这个符号来获取变量的值，为了使用户更加清楚 Shell 中变量值的引用方法，在此进行详细介绍。

【例 3-11】 演示 Shell 变量的引用方法，代码如下：

```
01  #-----------------------------/chapter3/ex3-11.sh------------------
02  #!/bin/bash
03
```

```
04   v1=Linux
05   v2='RedHat Linux'
06   v3="RedHat Linux $HOSTTYPE"
07   v4=12345
08
09   #输出变量 v1 的值
10   echo "$v1"
11   #输出变量
12   echo "$v2"
13   #输出变量 v3 的值
14   echo "$v3"
15   #输出变量 v4 的值
16   echo "$v4"
```

在上面的代码中，第 4 行～第 7 行分别定义了 4 个变量，并且分别各自赋予了不同的初始值。第 10 行～第 16 分别输出变量 v1～v4 的值。该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-11.sh
Linux
RedHat Linux
RedHat Linux x86_64
12345
```

可以得知，上面的程序分别输出了 4 个变量的值。其中，v3 中的环境变量$HOSTTYPE 被具体的值取代。

另外，在 Shell 中，字符串是可以直接连接在一起的，如果对【例 3-11】进行修改，将最后的变量 v4 的输出语句，即代码的第 13 行改成以下代码：

```
echo "$v4abc"
```

则程序在执行时会出现错误，如下：

```
[root@linux chapter3]# ./ex3-11.sh
Linux
RedHat Linux
RedHat Linux x86_64

```

从上面的执行结果可以得知，ex3-11.sh 脚本文件只正确输出了前 3 个变量的值，而最后一个变量的值变成了空字符串。

注意：在上面的执行结果中，变量 v4 的值也输出了，只是变成了空字符串，并非没有输出。

出现这种情况的原因在于，Shell 在进行解释代码时，遇到“$v4abc”这个字符串之后，并不知道具体的变量名到底是什么，因此，它会将整个字符串作为一个变量名来使用。在本程序中，变量 v4abc 当然是没有定义的了。所以导致 ex3-11.sh 在最后只输出了一个空行。

为了能够使 Shell 正确地界定变量名，避免混淆，用户在引用变量时可以使用大括号将变量名括起来，如下：

```
echo "${v4}abc"
```

那么以上程序就会得到正确的结果，如下：

```
[root@linux chapter3]# ./ex3-11.sh
Linux
RedHat Linux
RedHat Linux x86_64
12345abc
```

实际上，本书非常建议用户使用大括号来将变量名进行明确地界定，因为这是一种非常正式的、完整的书写方法，而用户通常使用省略大括号的形式只是一种简写。

3.2.3　清除变量

当某个 Shell 变量不再需要时，可以将其清除。当变量被清除后，其所代表的值也会消失。清除变量使用 unset 语句，其基本语法如下：

```
unset variable_name
```

其中，参数 varibale_name 表示要清除的变量的名称。

【例 3-12】 演示 Shell 变量清除方法，并且观察在清除前后变量值的变化，代码如下：

```
01  #-----------------------------/chapter3/ex3-12.sh------------------
02  #! /bin/bash
03
04  #定义变量 v1
05  v1="Hello world"
06  #输出 v1 的值
07  echo "$v1"
08  #清除变量
09  unset v1
10  echo "the value of v1 has been reset"
11  #再次输出变量的值
12  echo "$v1"
```

在上面的代码中，第 5 行定义了一个名称为 v1 的变量，第 7 行输出该变量的值。第 9 行使用 unset 语句将该变量清除，然后在第 12 行再次输出该变量的值。

该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-12.sh
Hello world
the value of v1 has been reset
```

从上面的执行结果可以得知，在执行第 9 行的 unset 语句之后，变量 v1 已经被清除掉了，因此，第 12 行的 echo 语句仅仅输出了空值。

3.3　引用和替换

通常对于弱类型的程序设计语言来说，变量的功能都相对比较单薄。但是对于 Shell 来说，变量的功能却非常强大。为了加强变量的功能，Shell 对变量的使用方法进行了极大地扩展。本节将介绍其中的引用和替换。

3.3.1　引用

所谓引用，是指将字符串用引用符号包括起来，以防止其中的特殊字符被 Shell 解释为其他涵义。特殊字符是指除了字面意思之外还可以解释为其他意思的字符。例如，在 Shell 中“$”符号的本身涵义是美元符号，其 ASCII 码值为十进制 36。除了这个涵义之外，前面已经讲过，“$”符号还可以用了获取某个变量的值，即变量替换。星号“*”也是一个特殊的字符，星号可以用来作为通配符使用。

【例 3-13】 演示星号通配符的使用方法，命令如下：

```
[root@linux chapter3]# ll ex*
-rwxr-xr-x  1   root        root    179     Jan  7 11:51        ex3-10.sh
-rwxr-xr-x  1   root        root    114     Jan  7 15:49        ex3-11.sh
-rwxr-xr-x  1   root        root    100     Jan  7 16:15        ex3-12.sh
…
```

在上面的 ll 命令中，参数 ex*表示列出以 ex 开头的所有的文件。但是，如果使用双引号将其引用起来，则其涵义会发生变化，如下：

```
[root@linux chapter3]# ll "ex*"
ls: cannot access ex*: No such file or directory
```

从上面的执行结果可以得知，当参数 ex*被双引号引用起来之后，ll 命令会将其作为一个普通的文件名来对待。但是当前目录中并不存在名称为 ex*的文件，所以，会给出没有该文件或者目录的提示信息。

对比这两次的执行结果，可以发现，其中起作用的是双引号，正是它改变了星号的意义。而这正是引用的目的。

在 Shell 中，一共有 4 种引用符号，如表 3-3 所示。

表 3-3　常用引用符号

引用符号	说　明
双引号	除美元符号、单引号、反引号和反斜线之外，其他所有的字符都将保持字面意义
单引号	所有的字符都将保持字面意义
反引号	反引号中的字符串将被解释为 Shell 命令
反斜线	转义字符，屏蔽后的字符的特殊意义

注意：在 Linux 中，ll 命令是 ls -l 命令的别名。

3.3.2　全引用

在 Shell 语句中，当一个字符串被单引号引用起来之后，其中所有的字符，除单引号本身之外，都将被解释为字面意义，即字符本身的涵义。这意味着被单引号引用起来的所有字符都将被解释为普通的字符，因此，这种引用方式称为全引用。

【例 3-14】 演示全引用的使用方法，代码如下：

```
01  #------------------------------/chapter3/ex3-14.sh------------------
02  #! /bin/bash
03
```

```
04   #定义变量 v1
05   v1="chunxiao"
06   #输出含有变量名的字符串
07   echo 'Hello, $v1'
```

在上面的代码中，第 5 行定义了一个字符串变量 v1，第 7 行使用 echo 语句输出一个含有变量名 v1 的用单引号引用起来的字符串。前面已经讲过，单引号表示全引用，因此第 7 行会原封不动地输出单引号中的字符串。该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-14.sh
Hello, $v1
```

3.3.3　部分引用

对于单引号来说，被其引用起来的所有的字符都将被解释为字面意义。而对于双引号来说，情况会有所不同。如果用户使用双引号将字符串引用起来，则其中所包含的字符除美元符号（$）、反引号（`），以及反斜线（\）之外的所有其他的字符，都将被解释为字面意义，这称为部分引用。也就是说，在部分引用中，“$”、“`”和“\”仍然拥有特殊的涵义。例如，“$”符号仍然可以用来引用变量的值。

【例 3-15】 演示部分引用的使用方法，本例将【例 3-14】中的单引号改为双引号，代码如下：

```
01   #-----------------------------/chapter3/ex3-15.sh------------------
02   #! /bin/bash
03
04   #定义变量
05   v1="chunxiao"
06   #输出变量的值
07   echo "Hello, $v1"
```

以上代码的执行结果如下：

```
[root@linux chapter3]# ./ex3-15.sh
Hello, chunxiao
```

从上面的执行结果可以得知，echo 语句中的变量名已经被变量的值取代。

3.3.4　命令替换

所谓命令替换，是指在 Shell 程序中，将某个 Shell 命令的执行结果赋给某个变量。在 bash 中，有两种语法可以进行命令替换，分别使用反引号和圆括号，如下：

```
'shell_command'
$(shell_command)
```

以上两种语法是等价的，用户可以根据自己的习惯来选择使用。

【例 3-16】 演示反引号的使用方法，代码如下：

```
01   #-----------------------------/chapter3/ex3-16.sh------------------
02   #! /bin/bash
03
04   #变量替换
05   v1='pwd'
```

```
06   #输出变量的值
07   echo "current working directory is $v1"
```

在上面的代码中，第 5 行将 Shell 命令 pwd 的执行结果赋给变量 v1，然后在第 7 行输出该变量的值。

该程序的执行结果如下：

```
[root@linux chapter3]# ./ex3-16.sh
current working directory is /root/chapter3
```

从上面的执行结果可以得知，命令 pwd 的执行结果已经成功地替换了变量名 v1。

注意：Shell 会将反引号中的字符串当做 Shell 命令，如果输入了错误的命令，则会出现“command not found”的错误提示。

在上面的代码中，使用了反引号，如果使用圆括号，则对代码进行相应的修改，如下：

```
01   #! /bin/bash
02
03   v1=$(pwd)
04   echo "current working directory is $v1"
```

以上代码的执行结果与例【3-15】完全相同。

3.3.5　转义

顾名思义，转义的作用是转换某些特殊字符的意义。转义使用反斜线表示，当反斜线后面的一个字符具有特殊的意义时，反斜线将屏蔽该字符的特殊意义，使得 Shell 按照该字符的字面意义来解释。

例如，我们已经知道，$符号是一个特殊的符号，通常加在变量名的前面，用来获取变量的值。在下面的两个命令中，第一个命令可以获得变量的值，而第二个命令则直接输出了变量名，如下：

```
[root@linux chapter3]# echo $SHELL
/bin/bash
[root@linux chapter3]# echo \$SHELL
$SHELL
```

为什么会得到这样的结果呢？这是因为在第二个命令的$符号前面使用了转义字符“\”，从而使得紧跟在后面的$符号失去了其特殊的作用，变成了一个普通的字符，即仅仅表示$符号本身而已。

3.4　小　　结

本章详细介绍了 Shell 语言中的变量和引用的相关知识，主要内容包括变量涵义、变量的命名规则、变量的定义、变量的作用域、系统变量和环境变量、变量赋值和清除、全引用和部分引用、命令替换，以及转义等。重点在于掌握好变量的定义方法、变量的作用域、常用的系统变量和环境变量的使用方法，以及全引用和部分引用。在下一章中，将介绍条件测试和判断语句。

第 4 章　条件测试和判断语句

作为一个实用的、能够解决实际问题的 Shell 程序，必须能够根据执行过程中的各种实际情况来做出正确的选择。实际上，这也是各种程序设计语言都必须解决的一个问题。

Shell 提供了一系列的条件测试来处理程序执行过程中的各种情况，并做出进一步的操作。本章将介绍各种条件测试的基本语法，以及 Shell 程序的基本流程控制语句判断语句的使用方法。

本章主要涉及的知识点有如下所述。

- 条件测试：主要介绍 Shell 程序中的文件、变量、字符串数值，以及逻辑等条件测试。
- 条件判断语句：介绍基本的 if、if else，以及 if elif 语句的使用方法。
- 多条件判断语句 case：主要介绍 case 语句的基本语法，以及使用 case 语句来解决一些实际问题。
- 运算符：主要介绍 Shell 中常用的运算符的使用方法，例如算术运算符、位运算符，以及自增、自减运算符等。

4.1　条 件 测 试

为了能够正确处理 Shell 程序运行过程中遇到的各种情况，Shell 提供了一组测试运算符。通过这些运算符，Shell 程序能够判断某种或者几个条件是否成立。条件测试在各种流程控制语句，例如判断语句和循环语句中发挥了重要的作用，所以，了解和掌握这些条件测试是非常重要的。本节将介绍常见的条件测试。

4.1.1　条件测试的基本语法

在 Shell 程序中，用户可以使用测试语句来测试指定的条件表达式的条件的真或者假。当指定的条件为真时，整个条件测试的返回值为 0；反之，如果指定的条件为假，则条件测试语句的返回值为非 0 值。对于熟悉其他的程序设计语言，例如 Java 或者 C 的用户来说，这一点非常重要，因为在这些语言中，通常情况下，条件表达式的值为真时，整个表达式的值为非 0 值；而当条件表达式的值为假时，这个表达式的值为 0。

注意：在 Shell 程序中，条件测试中的指定条件为真时，条件测试的返回值为 0。这主要是为了保持与 Shell 程序的退出状态一致。当某个 Shell 程序成功执行后，该进程会返回一个 0 值；而如果该程序执行错误，则会返回一个非 0 值。

条件测试的语法有两种，分别是 test 命令和[命令，下面对这两种语法进行介绍。