

第 27 章 shell 脚本基础

shell 脚本是一种简单而且功能强大的脚本语言。在 Linux 环境的编程过程中，shell 脚本能够很好地配合 C 语言的工作。创建一个 shell 脚本，再使用一系列 shell 命令的组合，可以快速而方便地完成相应工作。因此，shell 脚本无论对于 Linux 系统管理员还是 Linux 环境下的编程人员来说是很有实用价值的。本章将介绍编写 shell 脚本的基础知识。

27.1 编写最简单的 shell 脚本

shell 脚本的基本构成元素是 shell 命令，例如“ls”，“pwd”等命令都可以出现在 shell 脚本中。学习书写 shell 脚本需要从最简单的脚本开始写起，就像学习 C 语言要先写“hello word”程序一样。本节将介绍最基本的 shell 脚本的写法。

27.1.1 使用 shell 脚本的原因

shell 脚本在处理自动循环的后台任务方面功能很强大。如果用户需要处理一个大型的任务时，传统的做法是列出处理任务所需要的命令清单，一个一个地输入命令并且观察输出结果。如果命令的运行结果正确，则继续任务的下一步操作；否则再回到清单一步步观察。其流程如图 27-1 所示。

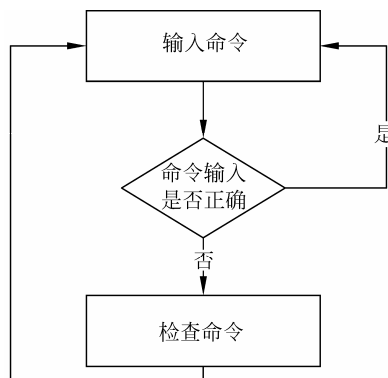


图 27-1 传统的 Linux 任务处理流程

当这个大型任务需要处理很多次时，需要将这些繁琐的命令输入若干次，这是令每一

个系统管理员或者编程者恼火的。不仅仅是因为工作量的巨大，而且重复的机器型劳动造成出错的概率也是很大的。使用 shell 脚本可以解决这个问题。

创建一个 shell 脚本，使用一系列 shell 命令的组合，再配以简单的 shell 脚本变量、条件判断、算术表达式和循环控制结构，可以快速地完成相应工作。这样的做法更有效率。执行 shell 脚本代替了在 shell 中依次输入每一个命令，而且大大降低了出错的可能，因为计算机的长处就是处理这种大量的重复性的工作。

因此使用 shell 脚本可以说是发挥了人和计算机的特长，不失为一种理想的解决问题的方案。使用 shell 脚本处理任务的执行流程如图 27-2 所示。

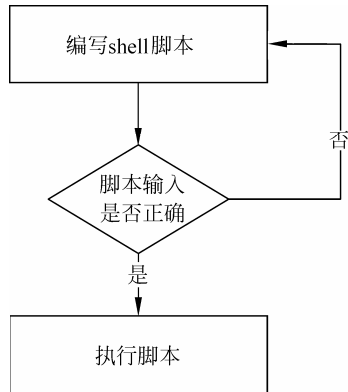


图 27-2 使用 shell 脚本处理任务的流程

27.1.2 脚本内容

shell 脚本本身并不复杂，它是一种解释型语言。在执行一个 shell 脚本时，解释器扫描一行脚本代码，之后执行这个代码。shell 脚本的第一行总是以一个固定的语句开头的。

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

这行语句表示这段脚本使用程序/bin/sh 解释器对其进行解释并且执行。这个解释器就是 shell。shell 脚本中的注释以“#”字符开头，解释器并不解释注释。与 C 语言不同的是，shell 中的注释会注视调用“#”字符以后的所有内容，直到这一行结束。在解释器程序扫描到注释时会跳过“#”之后所有的内容。因此，不要在“#”字符后面书写任何有用的否语句。

在第二行注释中写入脚本名是一个好习惯。下面实例演示了一个简单的 shell 脚本，这个脚本中没有命令，只有注释，因此什么都不做。

程序清单 27-1 test.sh

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

```
#test.sh
#第二行是脚本的名称。这是一个测试用的脚本，演示注释的使用
```

当一个 shell 脚本开始执行时，解释器程序从上到下扫描脚本中的命令和语句，并且执行每一个语句。

27.1.3 运行一段脚本

运行一个 shell 脚本前需要增加其执行权限，使其具有可执行的权限。在运行 shell 脚本时应该确保正确的脚本路径。下面实例演示了一个 shell 脚本的执行。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-2 test.sh

```
#!/bin/sh
#test.sh

#输出一个字符串 hello world
echo "hello world"
```

shell 脚本的后缀名是 “.sh”。echo 命令用于在屏幕上输出一行字符串，其作用有点类似于 C 语言中的 printf 函数。

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 chmod 命令增加 test.sh 文件的可执行权限。

```
$ chmod u+x test.sh
```

(3) 在 shell 中运行该 shell 脚本，只输入 shell 脚本的文件名如下：

```
$test.sh
```

(4) 输出错误信息。

```
bash : test.sh command not found
```

(5) 使用相对路径名执行该 shell 脚本如下：

```
$/test.sh
hello world
```

最简单的 shell 脚本已经运行成功了。在此总结 shell 脚本的编写流程如下：

- ☐ 在编辑器中编辑 shell 脚本文件。
- ☐ 将该文件保存为 “*.sh” 文件。
- ☐ 增加脚本文件的可执行权限。
- ☐ 使用相对路径名执行脚本文件。

其流程如图 27-3 所示。

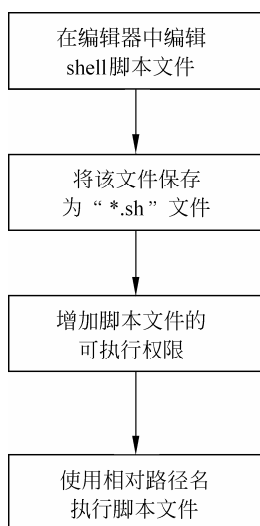


图 27-3 书写 shell 脚本的一般流程

27.2 shell 中的变量

shell 语言中可以使用变量，不过这里的变量不同于其他的高级语言中的变量。本小节将介绍在 shell 脚本中使用变量。

27.2.1 用户自定义变量

在编写 shell 脚本的过程中，用户可以使用自定义的变量来保存数据。这一点类似于 C 语言中的变量。下面程序段简单示意了 shell 中变量的定义：

```
$number=1
$a=2
$b=3
$c=$a+$b
$filename="test.txt"
$string="hello world"
```

在 shell 中使用变量时需要在变量名之前加上一个“\$”字符。“\$”字符作为 shell 脚本中的一个保留字，表示变量的开始。与 C 语言不同的是，shell 脚本中的变量在使用时并不需要提前定义，也就是说每一个变量都可以在程序员需要时就使用。

这样就大大增强了程序的灵活性，下面实例演示了计算变量 a 和变量 b 的和的简单程序，笔者分别给出了 C 语言版本和 shell 脚本的版本，读者可以通过对比的方式发现二者的不同。

程序清单 27-3 add.c 计算两个变量的和，C 语言版本

```
#include <stdio.h>
```

```

int main(void)
{
    int a, b;

    a = 1;
    b = 2;

    printf("the sum is : %d\n", a + b);

    return 0;
}

```

下面代码是一个 shell 版本的计算两个变量和的小程序，变量可以不声明而直接使用。

程序清单 27-4 add.sh 计算两个变量的和，shell 脚本版本

```

#!/bin/sh
#add.sh


$a=1
$b=2

$sum=$((a+b))      #计算两个变量的和，需要数值时应在表达式的外面加一个“$”符

echo "the res is $sum"

```

因为大部分 Linux 命令使用的是小写字母，而大多数的 shell 中的环境变量是使用大写字母表示的。因此，在 shell 脚本中出现的用户自定义变量最好使用小写字母，这样可以很好地区别 shell 中的环境变量。

 **注意：** shell 脚本中的变量名同样是对大小写敏感的。

因此，环境变量 \$PATH 和用户自定义变量 \$path 是不一样的。这种书写规则并不是强制性的，但这样的写法符合通用的 shell 变量名的书写规则。

shell 中变量的赋值是由左至右进行的，下面实例演示了在 shell 脚本中使用多重赋值。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-5 var.sh 演示 shell 脚本中变量的多重赋值

```

#!/bin/sh
#var.c 演示 shell 脚本中变量的多重赋值

x=2 y=$x

echo "x is $x, y is $y"

```

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 chmod 命令增加 var.sh 文件的可执行权限。

```
$ chmod u+x var.sh
```

(3) 在 shell 中运行该 shell 脚本如下：

```
$/var.sh
x is 2, y is 2
```

使用 `unset` 命令可以删除变量的赋值，使变量的值为空。如果用户需要清除一个变量的值可以这样做，其等效于将这个变量赋值为空。下面实例演示了使用 `unset` 命令清除一个变量的值。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-6 unset.sh 使用 unset 命令清除一个变量的值

```
#!/bin/sh
#unset.sh 使用 unset 清除变量的值

echo "before clearance"

string="hello world"
echo $string    #输出 string 变量的原来的值

echo "after clearance"

unset $string
echo $string    #输出清除了 string 变量后的值
```

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `unset.sh` 文件的可执行权限。

```
$ chmod u+x unset.sh
```

(3) 在 shell 中运行该 shell 脚本如下：

```
$/unset.sh
before clearance
hello world
after clearance
```

在输出 `string` 变量的值时其值已经变成了空值，因此打印了一个换行。

27.2.2 引用变量

在 shell 脚本中需要引用一个变量时可以有 3 种方法。这 3 种方法区别不大，通用的一点是需要引用在引用的变量名之前加上一个“\$”字符。

☐ 使用双引号引用变量

```
"$var"
```

☐ 使用大括号引用变量

```
${var}
```

❑ 直接引用变量

```
$var
```

使用双引号的方法来引用变量，可以阻止所有在引号中的特殊字符被重新解释。双引号在表示变量的值时也有很大的用处——能够阻止单词分割。如果一个字符串被双引号括起来时，那么这个字符串将被认为是一个整体。即使字符串中包含空格，里面的各个单词也不会被分开。下面实例演示了使用双引号来包括一个整个的字符串。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-7 str.sh 使用双引号定义字符串

```
#!/bin/sh
#str.sh #使用双引号包含字符串中的空格

string1=hello world
string2=" hello world"

echo $string1
echo $string2

exit 0
```

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `str.sh` 文件的可执行权限。


```
$ chmod u+x str.sh
```

(3) 在 shell 中运行该 shell 脚本如下：

```
$/str.sh
hello
hello world
```

27.2.3 为表达式求值

当需要在 shell 脚本中为一个数学表达式求值时，则应该使用 `expr` 命令。该命令用于对一个表达式求值。通过给定的运算符连接操作数，并对操作数求值。

 **注意：**操作数和运算符之前必须有空格隔开。

`expr` 命令中可以使用数学运算符。例如，数字比较操作、整数运算操作或者逻辑操作等。下面实例演示了 `expr` 变量对表达式的求值。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-8 expr.sh 演示使用 expr 命令求表达式的值

```
#!/bin/sh
#expr.sh 使用 expr 命令计算表达式的值

#在数学表达式中使用加法操作
expr 3 + 2
echo $?

#在数学表达式中使用求余操作
expr 3 % 2
echo $?

#在数学表达式中使用乘法操作时乘法符号必须被转义
expr 3 \* 2
echo $?

exit 0
```

(2) 编写好脚本后, 增加 shell 脚本文件的可执行权限。可以使用 chmod 命令增加 expr.sh 文件的可执行权限。


```
$ chmod u+x expr.sh
```

(3) 在 shell 中运行该 shell 脚本如下:

```
$/expr.sh
5
1
6
```

在 shell 脚本中也可以使用双括号代替 expr 命令计算表达式的值。其形式如下:

```
$( (表达式) )
```

 **注意:** 数学表达式有两对括号。

下面实例使用双括号的方法代替 expr 命令计算表达式的值。

(1) 在 vi 编辑器中编辑该脚本如下:

程序清单 27-9 brace.sh 演示使用双括号求表达式的值

```
#!/bin/sh
# brace.sh 使用双括号计算表达式的值

#在数学表达式中使用加法操作
sum=$(( 3 + 2 ))
echo $sum

#在数学表达式中使用求余操作
mod=$(( 3 % 2 ))
```



```
echo $mod

#在数学表达式中使用乘法操作
mul=$((3 \* 2))
echo $mul

exit 0
```

(2) 编写好脚本后, 增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `brace.sh` 文件的可执行权限。

```
$ chmod u+x brace.sh
```

(3) 在 shell 中运行该 shell 脚本如下:

```
$. /expr.sh
5
1
6
```

27.2.4 变量的类型

shell 脚本中的变量很特别, 因为在这里变量并不区分类型。这一点与 C 语言以及其他的一些编程语言完全不同, 也容易使初学者感到困惑。本质上, shell 脚本中的变量都是字符串, 在对变量的解释上则依赖于 shell 脚本中的变量定义, 如图 27-4 所示。

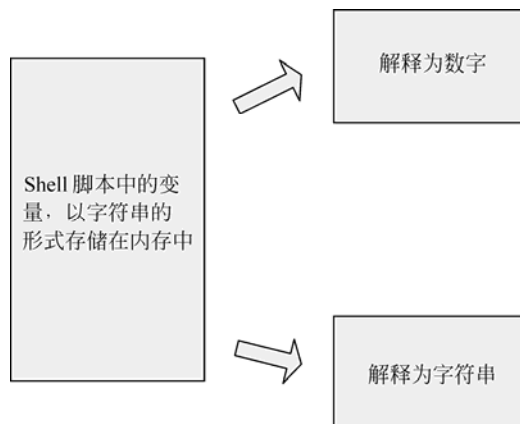


图 27-4 shell 脚本中变量的存储

例如, 整型变量也是一个字符串, shell 允许比较操作和整数操作。其中的关键因素是, 这个整型变量中的值是否只有数字。如果只有数字则在进行整数操作时将其解释为一个整型; 如果还包含其他的字符, 则只能将其解释为字符串。

下面实例演示了 shell 脚本中的整型变量和字符串变量的使用。该脚本首先使用一个整型变量 `a` 进行加法运算, 之后使用一个值中包含字母的变量 `b` 进行演示。该变量转换为了字符串的形式, 即使使用 `declare` 命令也不能更改其类型。本脚本还演示了未初始化的变量以及值为空的变量的算术运算。其执行流程如图 27-5 所示。

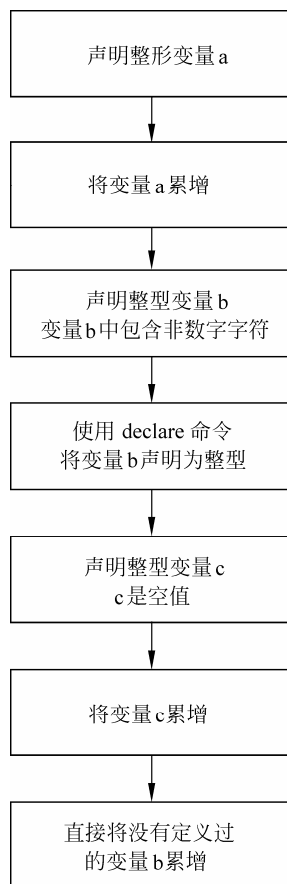


图 27-5 整型变量与字符串变量转换存储的程序流程图

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-10 int_str.sh 演示 shell 脚本中的整型变量和字符串变量

```

#!/bin/bash
# int_str.sh 演示整型变量和字符串变量之间的微妙差别

a=1024          #整型变量 a，将其赋值为 1024
$a=$a+1         #累增变量 a 的值
echo "the value of a is :$a"

b=102a          #在整型数据中混杂一个字母，把变量 b 从整型变为字符串
echo " the value of b is :$b"

declare -i b     #使用 declare 命令，明确指定变量 b 是一个整型变量
echo " the value of b is :$b"

$b=$b+1         #累增变量 b 的值
echo " the value of b is :$b"

```

```

c=""          #c 变量的值是一个空值
echo " the value of c is :$c"

$c=$c+1      #使用加法运算符操作一个值为空的变量
echo " the value of c is :$c"

$d=$d+1      #使用加法运算符操作没有声明的变量
echo " the value of d is :$d"

exit 0

```

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `int_str.sh` 文件的可执行权限。

```
$ chmod u+x int_str.sh
```

(3) 在 shell 中运行该 shell 脚本如下：

```

$ ./int_str.sh
the value of a is : 1025
the value of b is : 102a
the value of b is : 102a
the value of b is : 1
the value of c is :
the value of c is : 1
the value of d is : 1

```

shell 脚本不区分变量的类型有好处也有坏处。好处是使 shell 脚本程序变得更加灵活，编写代码的难度也更小了；坏处是这种方法产生错误的机率很高，而且会使程序员养成不好的编程习惯。因此一个 C 语言的程序员转向 shell 编程会觉得很容易，一旦习惯了 shell 脚本的编程模式，再转回到 C 语言下进行开发就会觉得很难了。

这就好像一个在月球居住久了的人回到地球上后不习惯于地球的重力一样。因此，在使用 shell 脚本之前最好先确认自己良好的编程习惯已经养成，并且在使用 shell 脚本的时候时时告诫自己——这种方法只适用于 shell 脚本的编程。

27.2.5 操作自定义变量

当用户定义了一个变量后，除了引用这个变量外，还可以做一些其他操作。这些操作可以方便用户使用自定义变量，同时有利于对 shell 脚本的调试。

在 shell 脚本中，变量未赋值之前其值为空。在引用该变量时，允许用户对变量设置默认值，其使用方法如下：

```
${var:-defaultvalue}
```

`var` 代表设置默认值的变量名称，而 `defaultvalue` 代表该变量的默认值。如果变量没有被赋值，则使用设置的默认值代替原来的空值；如果变量被设置了一个值，则使用新设置的值。如图 27-6 所示为 shell 脚本中变量默认值的使用。

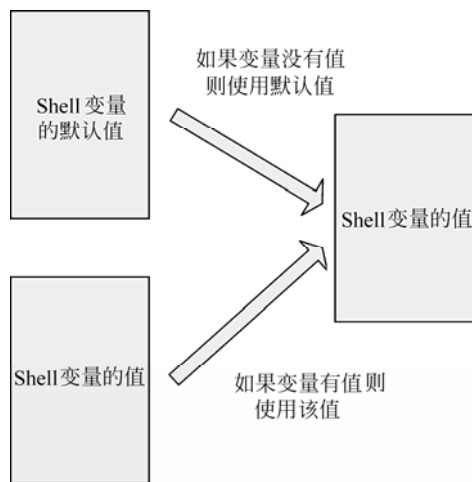


图 27-6 变量默认值的示意图

下面实例演示了设置 shell 脚本中变量的默认值。该脚本首先设置一个没有值的变量，在设置了变量的默认值之后输出。接下来将该变量赋值，再次输出该变量的值。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-11 default.sh 设置变量的默认值

```
#!/bin/sh
#default.sh 设置变量的默认值

echo linux $version

echo "set default-value"

echo linux ${version:-2.6.12} #设置变量的默认值

echo the value is : $version #变量本身的值并没有改变

$version="new version"      #将变量赋值

$ echo linux ${version:-2.6.12}#再次输出该变量的值
```

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `default.sh` 文件的可执行权限。

```
$ chmod u+x default.sh
```

(3) 在 shell 中运行该 shell 脚本如下：

```
$default.sh
linux
set default-value
linux 2.6.12
the value is :
linux new version
```

27.2.6 位置变量

在运行一个 shell 脚本时可以向脚本传递命令行参数，这些命令行参数可以在 shell 脚本内部被引用到。根据每一个命令参数的位置，在 shell 中可以使用 \$1 至 \$9 来表示。\$0 表示当前执行进程的文件名，也就是程序的执行文件名。

下面实例演示了使用位置变量输出传递给 shell 脚本的命令行参数。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-12 args.sh 输出传递给 shell 脚本的命令行参数

```
#!/bin/sh
#args.sh 输出该 shell 脚本的命令行参数

#输出所有的 10 个命令行参数，每个参数使用位置变量表示
echo No.0 $0
echo No.1 $1
echo No.2 $2
echo No.3 $3
echo No.4 $4
echo No.5 $5
echo No.6 $6
echo No.7 $7
echo No.8 $8
echo No.9 $9
```

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 chmod 命令增加 args.sh 文件的可执行权限。

```
$ chmod u+x args.sh
```

(3) 在 shell 中运行该 shell 脚本如下：

```
$/args.sh arg1 arg2 arg3
No.1 ./args.sh
No.2 arg1
No.3 arg2
No.4 arg3
No.5
No.6
No.7
No.8
No.9
No.10
```

(4) 再次在 shell 中运行该 shell 脚本，并且增加命令行参数的个数。

```
$/args.sh arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9 arg10
No.1 ./args.sh
```

```
No.2 arg1
No.3 arg2
No.4 arg3
No.5 arg4
No.6 arg5
No.7 arg6
No.8 arg7
No.9 arg8
No.10 arg9
```

由此可知，空余的位置变量会被设置为空值，而多余的命令行参数则不能被保存在位置变量中。位置变量的作用是很大的，活用位置变量将大大提高 shell 脚本的功能。

27.2.7 重新分配位置变量

使用 shift 命令会重新分配位置参数，该操作把所有的位置参数都向左移动一个位置。也就是说原来的 \$2 变成了 \$1，而 \$9 变成了 \$8，如图 27-7 所示。

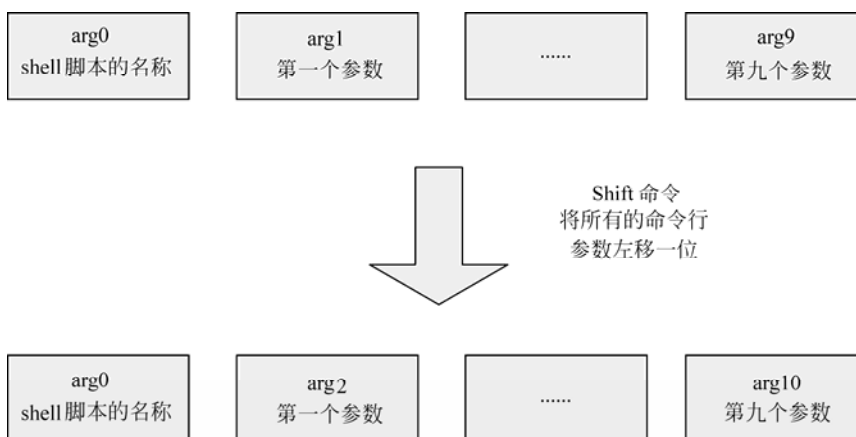


图 27-7 重新分配位置变量

原来的 \$1 就消失了，但是 \$0 (shell 脚本文件的名称) 是不会改变的。如果传递了大量的位置参数到脚本中，那么 `shift` 命令允许用户访问超过位置变量所能表示的命令行参数。

下面实例演示了使用 `shift` 命令访问到第 10 个命令行参数以后的命令行参数。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-13 shift.sh 访问第 10 个以后的命令行参数

```
#!/bin/sh
#shift.sh 访问第 10 个以后的命令行参数

#输出所有的 10 个命令行参数，每个参数使用位置变量表示
echo No.0 $0
echo No.1 $1
echo No.2 $2
```

```

echo No.3 $3
echo No.4 $4
echo No.5 $5
echo No.6 $6
echo No.7 $7
echo No.8 $8
echo No.9 $9

ehco shifting #移位操作，将第 11 个命令行参数 ($10) 移动到$9 的位置，$1 就丢失了

echo No.0 $0
echo No.1 $1
echo No.2 $2
echo No.3 $3
echo No.4 $4
echo No.5 $5
echo No.6 $6
echo No.7 $7
echo No.8 $8
echo No.9 $9

```

(2)编写好脚本后,增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `shift.sh` 文件的可执行权限。

```
$ chmod u+x shift.sh
```

(3) 在 shell 中运行该 shell 脚本如下:

```

$ ./shift.sh arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9 arg10
No.1 ./args.sh
No.2 arg1
No.3 arg2
No.4 arg3
No.5 arg4
No.6 arg5
No.7 arg6
No.8 arg7
No.9 arg8
No.10 arg9
shifting
No.1 ./args.sh
No.2 arg2
No.3 arg3
No.4 arg4
No.5 arg5
No.6 arg6
No.7 arg7
No.8 arg8
No.9 arg9
No.10 arg10

```

27.3 退出状态

`exit` 命令用于结束一个 shell 脚本的运行，就像 C 语言中的调用“`exit(0)`”，或者在 `main` 函数中“`return 0;`”一样。shell 脚本在结束运行时也返回一个值，并且这个值会传递给调用脚本的父进程。这个父进程通常就是 shell，但是有些时候一些其他的用户程序也会调用 shell 脚本。父进程接收到这个值（其实这个值作为子进程结束状态的一部分）后，做下一步的处理，如图 27-8 所示。

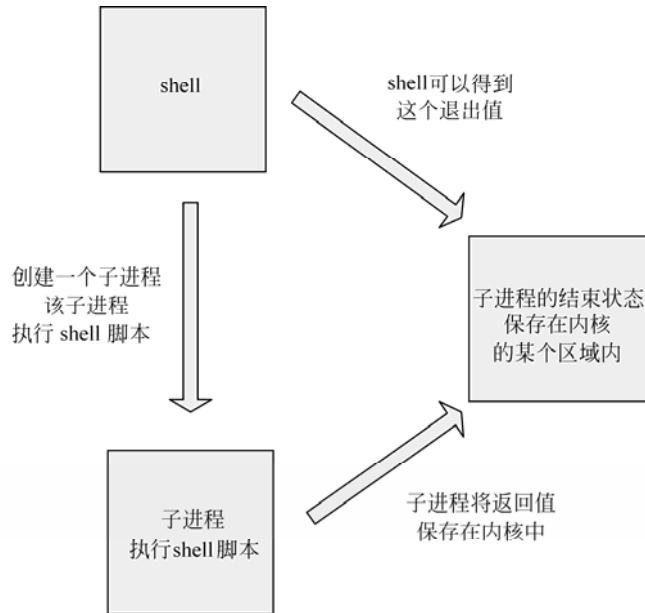


图 27-8 shell 脚本退出状态值传递给父进程示意图

每个 shell 命令在结束执行时都会返回一个退出状态码。成功执行了该命令则返回 0；否则，不成功的命令将返回一个非零值。非零值通常都被解释成一个错误码，从惯例的角度来讲返回 0 代表正常。如果返回一个非零值，则表示进程出现了异常。

这时候需要根据退出状态码找到进程退出的原因。通常来讲程序员和系统管理员之间应当约定一个退出状态码的协议，这样便于系统管理员帮助程序员发现错误。

注意：退出状态码必须是十进制数，范围必须是 0 至 255。

当脚本以不带参数的 `exit` 命令来结束时，脚本的退出状态码就由脚本中最后执行的命令来决定。也就是 `exit` 之前的命令。下面实例演示了 shell 脚本的退出状态码的使用，该脚本调用一个用户编写的简单程序。读者可以对比使用 `exit` 命令退出和不使用该命令退出的两个版本的 shell 脚本的区别。

(1) 在 vi 编辑器中编辑测试程序。

程序清单 27-14 test.c 指定退出状态码

```
#include <stdio.h>

int main(void)
{
    printf("running\n");

    return 25; /* 该程序的退出状态码是 25 */
}
```

(2) 在 shell 中编译该程序。

```
$gcc test.c-o test
```

(3) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-15 exit.sh 测试不同的退出状态码

```
#!/bin/sh
#exit.sh 测试不同进程的退出状态码

./test #执行 test 程序
```

(4) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 chmod 命令增加 exit.sh 文件的可执行权限。

```
$ chmod u+x exit.sh
```

(5) 在 shell 中运行该 shell 脚本如下：

```
$/exit.sh
running
```

(6) 检查 shell 脚本的退出状态码。

```
$echo $?
25
```

(7) 在 vi 编辑器中修改该脚本，添加 exit 退出命令。

程序清单 27-16 exit.sh 测试不同的退出状态码

```
#!/bin/sh
#exit.sh 测试不同进程的退出状态码

./test #执行 test 程序

exit 35 #添加一个 exit 退出命令
```

(8) 再次执行该 shell 脚本如下：

```
$/exit.sh
```

```
running
```

(9) 检查该脚本的退出状态码。

```
$echo $?
35
```

\$?是一个 shell 中的内置变量，代表着最后一次运行进程的退出状态码。在 shell 脚本中不写 exit 命令的效果和下面代码等价。

```
exit $?
```

在 shell 脚本的结尾处使用 exit 命令，但是不加任何参数，和上面所述的两种情况类似。

27.4 条件测试

在 shell 脚本中，条件测试使用专门的命令来进行。这一点对于一些高级语言的程序员来说有些难以适应。了解 shell 中繁琐的条件测试是很必要的，否则会对分支的逻辑产生模糊概念。本小节将介绍 shell 脚本中的条件测试命令。


27.4.1 测试文件状态

test 命令一般有两种格式。

```
test condition
```

或

```
[ condition ]
```

 **注意：**使用 “[]” 时，要在条件和 “[]” 之间加上空格。

测试文件状态的条件表达式很多，但是最常用几个如下所示。

- ☐ -d 文件是否为目录；
- ☐ -s 文件是否长度大于 0；
- ☐ -f 文件是否是普通文件；
- ☐ -L 文件是否是符号链接；
- ☐ -u 文件是否设置了 suid 位；
- ☐ -r 文件是否可读；
- ☐ -w 文件是否可写；
- ☐ -x 文件是否可执行。

当条件测试的返回状态是 0 时，则表示测试成功，否则失败。下面实例测试 test.txt 文件是否可读、可写和可执行的。该脚本中使用两种条件测试的方法，读者可以对照理解。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-17 file.sh 测试 test.txt 文件是否可读可写可执行

```
#!/bin/sh
# file.sh 测试 test.txt 文件是否可读可写可执行
#测试 test.txt 文件的读权限，使用 “[ ]” 的方法
[ -r test.txt ]
echo $?

#测试 test.txt 文件的写权限，使用 “[ ]” 的方法
[ -w test.txt ]
echo $?

#测试 test.txt 文件的执行权限，使用 “[ ]” 的方法
[ -x test.txt ]
echo $?

#测试 test.txt 文件的读权限，使用 “test 命令” 的方法
test -r test.txt
echo $?

#测试 test.txt 文件的写权限，使用 “test 命令” 的方法
test -w test.txt
echo $?

#测试 test.txt 文件的执行权限，使用 “test 命令” 的方法
test -x test.txt
echo $?

exit 0
```

(2)编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `file.sh` 文件的可执行权限。

```
$ chmod u+x file.sh
```

(3) 使用 `ls` 命令查看 `test.txt` 的权限信息。

```
$ls -l test.txt
-rw-r--r--    1   root          0   Feb    7   04:50   test.txt
```

(4) 在 shell 中运行该 shell 脚本如下：

```
0
0
1
0
0
1
```

运行结果说明 `test.txt` 文件只有可读写的权限，但是没有可执行的权限。下面实例测试传入脚本的文件名的文件是否是普通文件、目录文件和符号链接。该脚本使用位置变量作为文件名进行测试。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-18 test.sh 测试文件的种类

```
#!/bin/sh
# test.sh 测试文件的种类

#第一个文件是一个普通文件
[ -f $1]
echo $?

#第二个文件是一个目录文件
[ -d $2]
echo $?

#第二个文件是一个符号链接
[ -l $3]
echo $?

#运行的 shell 脚本本身也是一个普通文件
[ -f $0]
echo $?

exit 0
```

(2)编写好脚本后,增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `test.sh` 文件的可执行权限。

```
$ chmod u+x test.sh
```

(3) 使用 `mkdir` 命令创建一个目录 `dir`。

```
$mkdir dir
```

(4) 使用 `symlink` 命令创建一个符号链接。

```
$symlink link test.txt
```

(5) 在 shell 中运行该 shell 脚本如下：

```
$./test.sh test.txt dir link
0
0
0
0
```

试验结果说明每一步的测试都是成功的,每个文件的类型都和预期的一样。

27.4.2 测试时使用逻辑操作符

与 C 语言类似,shell 脚本中同样提供三种逻辑操作完成此功能。逻辑运算符通常和分支语句配合使用,实现程序执行流程的不同。

- ❑ -a 逻辑与，两个操作数均为真，结果为真，否则为假。
- ❑ -o 逻辑或，两个操作数一个为真，结果为真，否则为假。
- ❑ ! 逻辑非，条件为假，结果为真，否则为假。

下面实例比较两个文件，检查两个文件是否同时可写。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-19 and.sh 检查两个文件是否同时可写

```
#!/bin/sh
# and.sh 检查两个文件是否同时可写

[ -w $1 -a $0 ] #检查两个文件是否同时可写，其中一个是 shell 脚本本身
echo $?

exit 0
```

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 chmod 命令增加 and.sh 文件的可执行权限。

```
$ chmod u+x and.sh
```

(3) 使用 ls 命令查看两个文件。

```
$ls -l test.txt and.sh
-rw-r--r--    1  root        0  Feb   7   04:50  test.txt
-rwxr--r--    1  root        0  Feb  15   04:52  and.sh
```

(4) 在 shell 中运行该 shell 脚本如下：

```
$/and.sh test.sh
0
```

下面的例子测试两个文件是否有一个具有可执行权限。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-20 or.sh 检查两个文件是否有一个可写执行

```
#!/bin/sh
# or.sh 检查两个文件是否有一个可写执行

[ -x $1 -o $0 ] #检查两个文件是否同时可写，其中一个是 shell 脚本本身
echo $?

exit 0
```

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 chmod 命令增加 or.sh 文件的可执行权限。

```
$ chmod u+x or.sh
```

(3) 使用 ls 命令查看两个文件。

```
$ls -l test.txt and.sh
-rw-r--r--    1   root        0   Feb    7   04:50   test.txt
-rwxr--r--    1   root        0   Feb   15   04:52   and.sh
```

(4) 在 shell 中运行该 shell 脚本如下：

```
$/and.sh test.sh
0
```

27.4.3 字符串测试

字符串测试是很重要的，特别在测试用户输入或比较变量时其重要性尤为突出。在 shell 脚本中字符串的比较没有特定的函数。由于所有的变量均以字符串的形式存储，因此 shell 脚本中的字符串可以直接进行比较。这一点和 C 语言是不同的。

在 C 语言中字符串常量是以地址的形式存储，因此，比较两个字符串实际是在比较两个字符串的首地址。而 shell 脚本中的字符串存储的是字符串的内容，因此，比较的是两个字符串的内容，如图 27-9 所示。

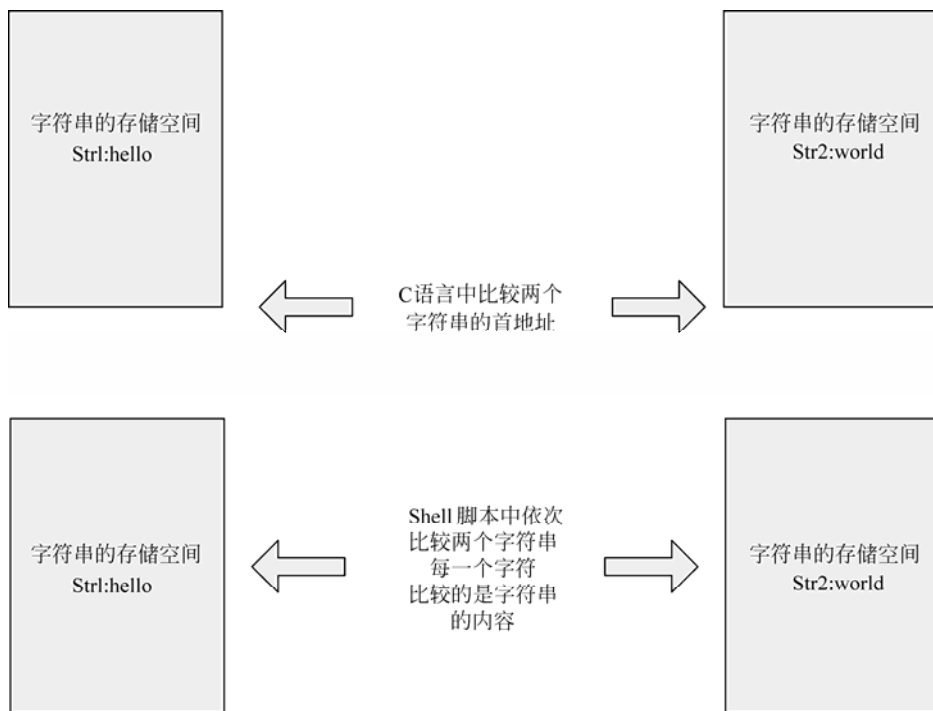


图 27-9 C 语言中的字符串比较和 shell 脚本中的字符串比较

字符串测试有以下 5 种格式：

```
test "string"
test operator "string"
test "string1" operator "string2"
```

```
[ operator "string" ]
[ "string1" operator "string2" ]
```

其中 `string1` 和 `string2` 代表需要测试的字符串，而 `operator` 代表字符串测试的操作符。这个操作共有以下 4 种可能的情况。

- ❑ `=` 两个字符串相等。
- ❑ `!=` 两个字符串不等。
- ❑ `-z` 字符串是空串。
- ❑ `-n` 字符串是非空串。

下面实例演示了测试字符串。该实例测试三个字符串，其中有一个字符串 `$str3` 是空串。实例中使用“`[]`”和“`test` 命令”两种方法进行测试。

(1) 在 vi 编辑器中编辑该脚本。

程序清单 27-21 str.sh 测试字符串

```
#!/bin/sh
#str.sh 演示测试字符串的方法

str1="hello"
str2="hello"
str3="" #str3 是一个空串

#测试字符串 str1 和 str2 相等
test $str1=$str2
echo $?

#测试字符串 str3 是空串
test -z $str3
echo $?

#测试字符串 str2 是非空串
test -n $str2
echo $?

#测试字符串 str1 和 str2 不相等
test $str1!= $str2
echo $?

echo "using [ ]"          #使用 “[ ]” 进行数据测试

#测试字符串 str1 和 str2 相等
[ $str1=$str2 ]
echo $?

#测试字符串 str3 是空串
[ -z $str3 ]
echo $?
```

```
#测试字符串 str2 是非空串
[ -n $str2 ]
echo $?

#测试字符串 str1 和 str2 不相等
[ $str1!= $str2 ]
echo $?

exit 0
```

(2) 编写好脚本后, 增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `str.sh` 文件的可执行权限。

```
$ chmod u+x str.sh
```

(3) 在 shell 中运行该 shell 脚本如下:

```
$. /str.sh
0
0
0
1
using [ ]
0
0
0
1
```

27.4.4 测试数值


在 shell 脚本中, 数值也是以字符串的形式进行存储的。因此在比较数值的时候, 或多或少的带有字符串比较的影子。测试数值可以使用许多操作符, 其格式如下:

```
"number" operator "number"
```

或者

```
[ "number" operator "number" ]
```

其中 `number` 代表要测试的数据, 而 `operator` 代表测试的操作符。

 **注意:** 数据两边的双引号是不能省略的。这代表将数据化为字符串进行测试。

`operator` 有 6 种情况, 如下所示。

- ☐ `-eq` 两个数值相等。
- ☐ `-ne` 两个数值不相等。
- ☐ `-gt` 第一个数大于第二个数。
- ☐ `-lt` 第一个数小于第二个数。
- ☐ `-le` 第一个数小于等于第二个数。
- ☐ `-ge` 第一个数大于等于第二个数。

下面实例演示了测试数值的方法。笔者同样使用 “[]” 和 “test 命令” 两种方法演示测试数值。

(1) 在 vi 编辑器中编辑该脚本如下：

程序清单 27-22 number.sh 测试两个数据

```
#!/bin/sh
# number.sh 演示测试两个数据

#测试两个数据相等
test "1" -eq "2"
echo $?

#测试两个数据不相等
test "1" -ne "2"
echo $?

#测试比较两个数据的大小
test "1" -gt "2"
echo $?

test "1" -lt "2"
echo $?

#测试两个数据的大小，包括等于
test "2" -ge "2"
echo $?

test "2" -le "2"
echo $?

echo "using [ ]"          #使用 “[ ]” 进行数据测试

#测试两个数据相等
[ "1" -eq "2" ]
echo $?

#测试两个数据不相等
[ "1" -ne "2" ]
echo $?

#测试比较两个数据的大小
[ "1" -gt "2" ]
echo $?

[ "1" -lt "2" ]
echo $?

#测试两个数据的大小，包括等于
[ "2" -ge "2" ]
```

```
echo $?  
  
[ "2" -le "2" ]  
echo $?  
  
exit 0
```

(2) 编写好脚本后，增加 shell 脚本文件的可执行权限。可以使用 `chmod` 命令增加 `number.sh` 文件的可执行权限。

```
$chmod u+x number.sh
```

(3) 在 shell 中运行该 shell 脚本如下：

```
$. /number.sh  
1  
0  
0  
1  
0  
0  
using [ ]  
1  
0  
0  
1  
0  
0
```