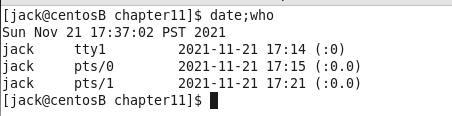
## 1.使用多个命令

到目前为止，你已经了解了如何使用shell的命令行界面提示符来输入命令和查看命令的结果。shell脚本的关键在于输入多个命令并处理每个命令的结果，甚至需要将一个命令的结果传给

另一个命令。shell可以让你将多个命令串起来，一次执行完成。如果要两个命令一起运行，可以

把它们放在同一行中，彼此间用分号隔开。



这个简单的脚本只用到了两个bash shell命令。date命令先运行，显示了当前日期和时间，后面紧跟着who命令的输出，显示当前是谁登录到了系统上。使用这种办法就能将任意多个命令串连在一起使用了，只要不超过最大命令行字符数255就行。

这种技术对于小型脚本尚可，但它有一个很大的缺陷：每次运行之前，你都必须在命令提示符下输入整个命令。可以将这些命令组合成一个简单的文本文件，这样就不需要在命令行中手动输入了。在需要运行这些命令时，只用运行这个文本文件就行了。

## 2. 创建shell脚本文件

要将shell命令放到文本文件中，首先需要用文本编辑器（参见第10章）来创建一个文件，然后将命令输入到文件中。

在创建shell脚本文件时，必须在文件的第一行指定要使用的shell。其格式为：

#!/bin/bash

在通常的shell脚本中，井号（#）用作注释行。shell并不会处理shell脚本中的注释行。然而，shell脚本文件的第一行是个例外，#后面的惊叹号会告诉shell用哪个shell来运行脚本（是的，你可以使用bash shell，同时还可以使用另一个shell来运行你的脚本）。

在指定了shell之后，就可以在文件的每一行中输入命令，然后加一个回车符。之前提到过，

注释可用#添加。例如：

#!/bin/bash

# This script displays the date and who's logged on

date

who

这就是脚本的所有内容了。可以根据需要，使用分号将两个命令放在一行上，但在shell脚本中，你可以在独立的行中书写命令。shell会按根据命令在文件中出现的顺序进行处理。

还有，要注意另有一行也以#开头，并添加了一个注释。shell不会解释以#开头的行（除了以

#!开头的第一行）。留下注释来说明脚本做了什么。

将这个脚本保存在名为test1的文件中，基本就好了。在运行新脚本前，还要做其他一些事。

现在运行脚本，结果可能会叫你有点失望。

$ test1

bash: test1: command not found

$

你要跨过的第一个障碍是让bash shell能找到你的脚本文件shell会通过PATH

环境变量来查找命令。快速查看一下PATH环境变量就可以弄清问题所在。

$ echo $PATH

/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin

:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin $

PATH环境变量被设置成只在一组目录中查找命令。要让shell找到test1脚本，只需采取以下两种作法之一：

A. Shell脚本目录添加到PATH环境中

B. 在提示符中用绝对或相对文件路径来引用shell脚本文件。

窍门 有些Linux发行版将$HOME/bin目录添加进了PATH环境变量。它在每个用户的HOME目录下提供了一个存放文件的地方，shell可以在那里查找要执行的命令。

在这个例子中，我们将用第二种方式将脚本文件的确切位置告诉shell。记住，为了引用当前

目录下的文件，可以在shell中使用单点操作符。

$ ./test1

bash: ./test1: Permission denied

$

现在shell找到了脚本文件，但还有一个问题。shell指明了你还没有执行文件的权限。快速查看一下文件权限就能找到问题所在。

$ ls -l test1

-rw-rw-r-- 1 user user 73 Sep 24 19:56 test1

$

在创建test1文件时，umask的值决定了新文件的默认权限设置。由于umask变量中被设成了022，所以系统创建的文件只有文件属主和属组才有读/写权限。

下一步是通过chmod命令赋予文件属主执行文件的权限。

$ chmod u+x test1

$ ./test1

Mon Feb 21 15:38:19 EST 2014

Christine tty2 2014-02-21 15:26

Samantha tty3 2014-02-21 15:26

Timothy tty1 2014-02-21 15:26

user tty7 2014-02-19 14:03 (:0)

user pts/0 2014-02-21 15:21 (:0.0) $

成功了！现在万事俱备，只待执行新的shell脚本文件了。

## 3. 显示信息

大多数shell命令都会产生自己的输出，这些输出会显示在脚本所运行的控制台显示器上。很多时候，你可能想要添加自己的文本消息来告诉脚本用户脚本正在做什么。可以通过echo命令来实现这一点。如果在echo命令后面加上了一个字符串，该命令就能显示出这个文本字符串。

$ echo This is a test

This is a test

$

注意，默认情况下，不需要使用引号将要显示的文本字符串划定出来。但有时在字符串中出

现引号的话就比较麻烦了。此时就需要使用引号引起

**1.例子**

$ echo Let's see if this'll work

Lets see if thisll work

$

**echo命令可用单引号或双引号来划定文本字符串。如果在字符串中用到了它们，你需要在**

**文本中使用其中一种引号，而用另外一种来将字符串划定起来**。

$ echo "This is a test to see if you're paying attention"

This is a test to see if you're paying attention

$ echo 'Rich says "scripting is easy".'

Rich says "scripting is easy".

$

所有的引号都可以正常输出了。

可以将echo语句添加到shell脚本中任何需要显示额外信息的地方。

$ cat test1

#!/bin/bash

# This script displays the date and who's logged on

echo The time and date are:

date

echo "Let's see who's logged into the system:"

who

$

当运行这个脚本时，它会产生如下输出。

$ ./test1

The time and date are:

Mon Feb 21 15:41:13 EST 2014

Let's see who's logged into the system:

Christine tty2 2014-02-21 15:26

Samantha tty3 2014-02-21 15:26

Timothy tty1 2014-02-21 15:26

user tty7 2014-02-19 14:03 (:0)

user pts/0 2014-02-21 15:21 (:0.0)

$

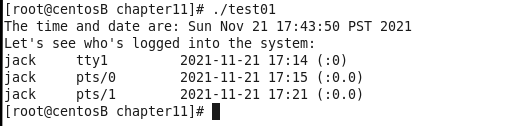
但如果想把文本字符串和命令输出显示在同一行中，该怎么办呢？可以用echo语句

的-n参数。只要将第一个echo语句改成这样就行：

echo -n "The time and date are: "

你需要在字符串的两侧使用引号，保证要显示的字符串尾部有一个空格。命令输出将会在紧

接着字符串结束的地方出现。现在的输出会是这样：



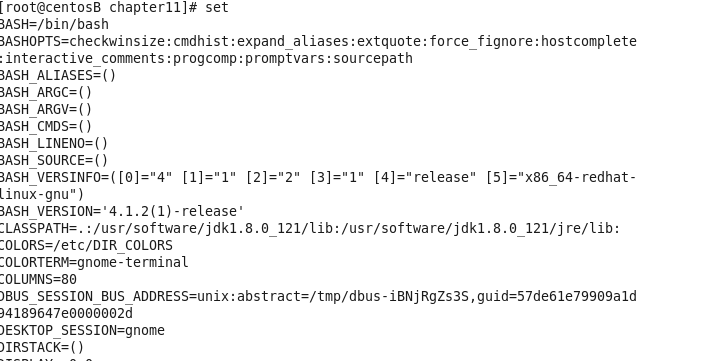
echo命令是shell脚本中与用户交互的重要工具。你会发现在很多地方都能用到它，尤其是需要显示脚本中变量的值的时候。我们下面继续了解这个。

## 4. 使用变量

### 1 环境变量

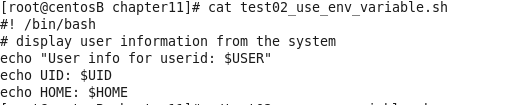
你已经看到过Linux的一种变量在实际中的应用。Linux系统的环境变量。也可以在脚本中访问这些值。shell维护着一组环境变量，用来记录特定的系统信息。比如系统的名称、登录到系统上的用户名、用户的系统ID（也称为UID）、用户的默认主目录以及shell查找程序的搜索路径。可以用set命令来显示一份完整的当前环境变量列表。

$ set

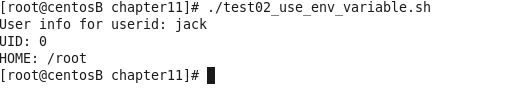


在脚本中，你可以在环境变量名称之前加上美元符（$）来使用这些环境变量。下面的脚本

演示了这种用法。



$USER、$UID和$HOME环境变量用来显示已登录用户的有关信息。脚本输出如下：



使用变量

注意，echo命令中的环境变量会在脚本运行时替换成当前值。另外，在第一个字符串中可

以将$USER系统变量放置到双引号中，而shell依然能够知道我们的意图。但采用这种方法也有一个问题。看看下面这个例子会怎么样。

$ echo "The cost of the item is $15"

The cost of the item is 5

显然这不是我们想要的。只要脚本在引号中出现美元符，它就会以为你在引用一个变量。在

这个例子中，脚本会尝试显示变量$1（但并未定义），再显示数字5。要显示美元符，你必须在它

前面放置一个反斜线。

$ echo "The cost of the item is \$15"

The cost of the item is $15

看起来好多了。反斜线允许shell脚本将美元符解读为实际的美元符，而不是变量。

**说明 你可能还见过通过${variable}形式引用的变量。变量名两侧额外的花括号通常用来帮**

**助识别美元符后的变量名。**

### 2 用户变量

除了环境变量，shell脚本还允许在脚本中定义和使用自己的变量。定义变量允许临时存储数

据并在整个脚本中使用，从而使shell脚本看起来更像一个真正的计算机程序。

用户变量可以是任何由字母、数字或下划线组成的文本字符串，长度不超过20个。用户变量区分大小写，所以变量Var1和变量var1是不同的。使用等号将值赋给用户变量。在变量、等号和值之间不能出现空格。这里有一些给用户变量赋值的例子。

var1=10

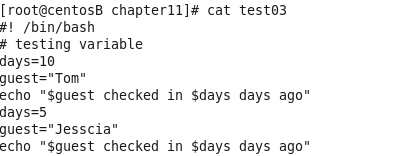
var2=-57

var3=testing

var4="still more testing"

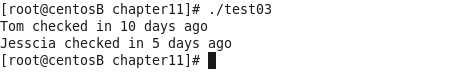
shell脚本会自动决定变量值的数据类型。在脚本的整个生命周期里，shell脚本中定义的变量会一直保持着它们的值，但在shell脚本结束时会被删除掉。

与系统变量类似，用户变量可通过美元符引用。



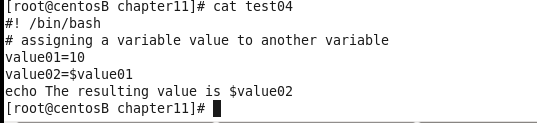
运行脚本会有如下输出。

$ chmod u+x test3



变量每次被引用时，都会输出当前赋给它的值。重要的是要记住，引用一个变量值时需要使

用美元符，而引用变量来对其进行赋值时则不要使用美元符。通过一个例子你就能明白我的意思。



在赋值语句中使用value1变量的值时，仍然必须用美元符。这段代码产生如下输出。

$ chmod u+x test4



要是忘了用美元符，使得value2的赋值行变成了这样：

value2=value1

那你会得到如下输出：

$ ./test4

The resulting value is value1

$

没有美元符，shell会将变量名解释成普通的文本字符串，通常这并不是你想要的结果。

### 3 命令替换[重点]

shell脚本中最有用的特性之一就是可以从命令输出中提取信息，并将其赋给变量。把输出赋

给变量之后，就可以随意在脚本中使用了。这个特性在处理脚本数据时尤为方便。

有两种方法可以将命令输出赋给变量,如下所示

反引号字符（`）

 $()格式

要注意反引号字符，这可不是用于字符串的那个普通的单引号字符。由于在shell脚本之外很少用到，因为这是许多shell脚本中的重要组件。提示：在美式键盘上，它通常和波浪线（~）位于同一键位。

命令替换允许你将shell命令的输出赋给变量。尽管这看起来并不那么重要，但它却是脚本编程中的一个主要组成部分。要么用一对反引号把整个命令行命令围起来：

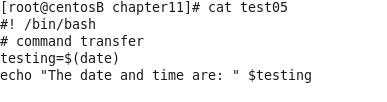
testing='date'

要么使用$()格式：

testing=$(date)

shell会运行命令替换符号中的命令，并将其输出赋给变量testing。注意，赋值等号和命令

替换字符之间没有空格。这里有个使用普通的shell命令输出创建变量的例子。



变量testing获得了date命令的输出，然后使用echo语句显示出它的值。运行这个shell脚

本生成如下输出。

$ chmod u+x test5



这个例子毫无吸引人的地方（也可以干脆将该命令放在echo语句中），但只要将命令的输出

放到了变量里。

下面这个例子很常见，它在脚本中通过命令替换获得当前日期并用它来生成唯一文件名。

#!/bin/bash

# copy the /usr/bin directory listing to a log file

today=$(date +%y%m%d)

ls /usr/bin -al > log.$today

today变量是被赋予格式化后的date命令的输出。这是提取日期信息来生成日志文件名常用

的一种技术。+%y%m%d格式告诉date命令将日期显示为两位数的年月日的组合。

$ date +%y%m%d

140131

$

这个脚本将日期值赋给一个变量，之后再将其作为文件名的一部分。文件自身含有目录列表

的重定向输出。运行该脚本之后，应该能在目录中看到一个新文件。

-rw-r--r-- 1 user user 769 Jan 31 10:15 log.140131

目录中出现的日志文件采用$today变量的值作为文件名的一部分。日志文件的内容是

/usr/bin目录内容的列表输出。如果脚本在明天运行，日志文件名会是log.140201，就这样为新的一天创建一个新文件。

警告 **命令替换会创建一个子shell来运行对应的命令。子shell（subshell）是由运行该脚本的shell所创建出来的一个独立的子shell（child shell）。正因如此，由该子shell所执行命令是无法使用脚本中所创建的变量的。**

在命令行提示符下使用路径./运行命令的话，也会创建出子shell；要是运行命令的时候

不加入路径，就不会创建子shell。如果你使用的是内建的shell命令，并不会涉及子shell。

在命令行提示符下运行脚本时一定要留心！

## 5 重定向输入和输出

有些时候你想要保存某个命令的输出而不仅仅只是让它显示在显示器上。bash shell提供了几个操作符，可以将命令的输出重定向到另一个位置（比如文件）。重定向可以用于输入，也可以用于输出，可以将文件重定向到命令输入。

### 5.1 输出重定向

最基本的重定向将命令的输出发送到一个文件中。bash shell用大于号（>）来完成这项功能：

command > outputfile

之前显示器上出现的命令输出会被保存到指定的输出文件中。

$ date > test6

$ ls -l test6

-rw-r--r-- 1 user user 29 Feb 10 17:56 test6

$ cat test6

Thu Feb 10 17:56:58 EDT 2014

$

重定向操作符创建了一个文件test6（通过默认的umask设置），并将date命令的输出重定向

到该文件中。如果输出文件已经存在了，重定向操作符会用新的文件数据覆盖已有文件。

$ who > test6

$ cat test6

user pts/0 Feb 10 17:55

$

现在test6文件的内容就是who命令的输出。

有时，你可能并不想覆盖文件原有内容，而是想要将命令的输出追加到已有文件中，比如你正在创建一个记录系统上某个操作的日志文件。在这种情况下，可以用双大于号（>>）来追加数据。

$ date >> test6

$ cat test6

user pts/0 Feb 10 17:55

Thu Feb 10 18:02:14 EDT 2014

test6文件仍然包含早些时候who命令的数据，现在又加上了来自date命令的输出。

### 5.2 输入重定向

输入重定向和输出重定向正好相反。输入重定向将文件的内容重定向到命令，而非将命令的

输出重定向到文件。

输入重定向符号是小于号（<）：

command < inputfile

一个简单的记忆方法就是：在命令行上，命令总是在左侧，而重定向符号“指向”数据流动

的方向。小于号说明数据正在从输入文件流向命令。

这里有个和wc命令一起使用输入重定向的例子。

$ wc < test6

2 11 60

$

wc命令可以对对数据中的文本进行计数。默认情况下，它会输出3个值：

 文本的行数---2行

文本的词数---11

 文本的字节数

通过将文本文件重定向到wc命令，你立刻就可以得到文件中的行、词和字节的计数。这个例子说明test6文件有2行、11个单词以及60字节。

还有另外一种输入重定向的方法，称为内联输入重定向（inline input redirection）。这种方法

无需使用文件进行重定向，只需要在命令行中指定用于输入重定向的数据就可以了。乍看一眼，这可能有点奇怪，但有些应用会用到这种方式。

内联输入重定向符号是远小于号（<<）。除了这个符号，你必须指定一个文本标记来划分输

入数据的开始和结尾。任何字符串都可作为文本标记，但在数据的开始和结尾文本标记必须一致。

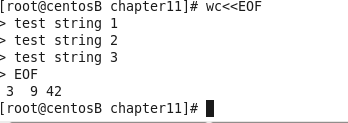
command << marker

data

marker

在命令行上使用内联输入重定向时，shell会用PS2环境变量中定义的次提示符

来提示输入数据。下面是它的使用情况。



次提示符会持续提示，以获取更多的输入数据，直到你输入了作为文本标记的那个字符串。

wc命令会对内联输入重定向提供的数据进行行、词和字节计数。

## 6管道

有时需要将一个命令的输出作为另一个命令的输入。这可以用重定向来实现，只是有些笨拙。

$ rpm -qa > rpm.list

$ sort < rpm.list

abrt-1.1.14-1.fc14.i686

abrt-addon-ccpp-1.1.14-1.fc14.i686

abrt-addon-kerneloops-1.1.14-1.fc14.i686

[...]

rpm命令通过Red Hat包管理系统（RPM）对系统上安装的软件包进行管理。配合-qa选项使用时，它会生成已安装包的列表，但这个列表并不会遵循某种特定的顺序。如果你在查找某个或某组特定的包，想在rpm命令的输出中找到就比较困难了。通过标准输出重定向，rpm命令的输出被重定向到了文件rpm.list。命令完成后，rpm.list保存着系统中所有已安装的软件包列表。接下来，输入重定向将rpm.list文件的内容发送给sort命令，

该命令按字母顺序对软件包名称进行排序。

这种方法的确管用，但仍然是一种比较繁琐的信息生成方式。我们用不着将命令输出重定向

到文件中，可以将其直接重定向到另一个命令。这个过程叫作管道连接（piping）。

和命令替换所用的反引号（`）一样，管道符号在shell编程之外也很少用到。该符号由两个

竖线构成，一个在另一个上面。然而管道符号的印刷体通常看起来更像是单个竖线（|）。在美式键盘上，它通常和反斜线（\）位于同一个键。管道被放在命令之间，将一个命令的输出重定向到另一个命令中：

command1 | command2

**不要以为由管道串起的两个命令会依次执行。Linux系统实际上会同时运行这两个命令，在**

**系统内部将它们连接起来。在第一个命令产生输出的同时，输出会被立即送给第二个命令。数据传输不会用到任何中间文件或缓冲区。**

现在，可以利用管道将rpm命令的输出送入sort命令来产生结果。

$ rpm -qa | sort

abrt-1.1.14-1.fc14.i686

abrt-addon-ccpp-1.1.14-1.fc14.i686

除非你的眼神特别好，否则可能根本来不及看清楚命令的输出。由于管道操作是实时运行的，

所以只要rpm命令一输出数据，sort命令就会立即对其进行排序。等到rpm命令输出完数据，sort命令就已经将数据排好序并显示了在显示器上。

可以在一条命令中使用任意多条管道。可以持续地将命令的输出通过管道传给其他命令来细

化操作。

在这个例子中，sort命令的输出会一闪而过，所以可以用一条文本分页命令（例如less或

more）来强行将输出按屏显示。

$ rpm -qa | sort | more

这行命令序列会先执行rpm命令，将它的输出通过管道传给sort命令，然后再将sort的输

出通过管道传给more命令来显示，在显示完一屏信息后停下来。这样你就可以在继续处理前停下来阅读显示器上显示的信息，如图11-1所示。

如果想要更别致点，也可以搭配使用重定向和管道来将输出保存到文件中。

$ rpm -qa | sort > rpm.list

$ more rpm.list

abrt-1.1.14-1.fc14.i686

abrt-addon-ccpp-1.1.14-1.fc14.i686

不出所料，rpm.list文件中的数据现在已经排好序了。

到目前为止，管道最流行的用法之一是将命令产生的大量输出通过管道传送给more命令。

这对ls命令来说尤为常见，如图11-2所示。

图11-2和ls命令一起使用more命令

ls -l命令产生了目录中所有文件的长列表。对包含大量文件的目录来说，这个列表会相当

长。通过将输出管道连接到more命令，可以强制输出在一屏数据显示后停下来。

## 7 执行数学运算

另一个对任何编程语言都很重要的特性是操作数字的能力。遗憾的是，对shell脚本来说，这个处理过程会比较麻烦。在shell脚本中有两种途径来进行数学运算。

### 1 expr命令

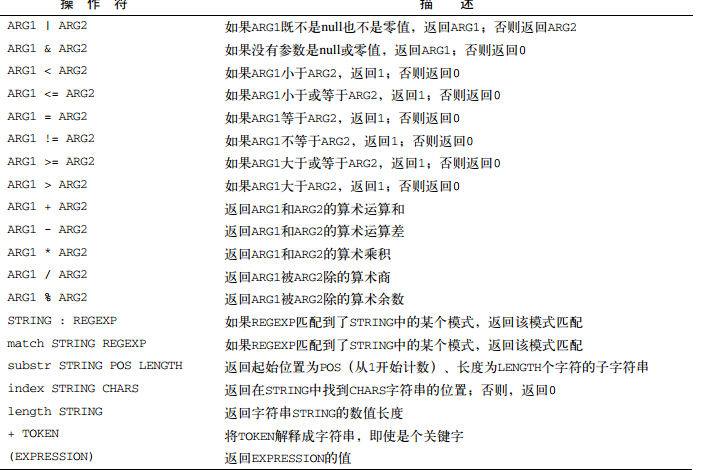
最开始，Bourne shell提供了一个特别的命令用来处理数学表达式。expr命令允许在命令行

上处理数学表达式，但是特别笨拙。

$ expr 1 + 5

6

expr命令能够识别少数的数学和字符串操作符.如下表所示



尽管标准操作符在expr命令中工作得很好，但在脚本或命令行上使用它们时仍有问题出现。

许多expr命令操作符在shell中另有含义（比如星号）。当它们出现在在expr命令中时，会得到一

些诡异的结果。

$ expr 5 \* 2

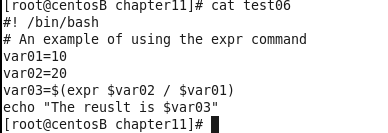
expr: syntax error

$

要解决这个问题，对于那些容易被shell错误解释的字符，在它们传入expr命令之前，需要使用shell的转义字符（反斜线）将其标出来。

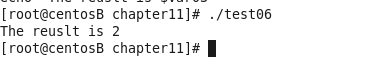
$ expr 5 \\* 2

现在，麻烦才刚刚开始！在shell脚本中使用expr命令也同样复杂：



要将一个数学算式的结果赋给一个变量，需要使用命令替换来获取expr命令的输出：

$ chmod u+x test06



幸好bash shell有一个针对处理数学运算符的改进。

### 2 使用方括号

bash shell为了保持跟Bourne shell的兼容而包含了expr命令，但它同样也提供了一种更简单的方法来执行数学表达式。在bash中，在将一个数学运算结果赋给某个变量时，可以用美元符和方括号（$[ operation ]）将数学表达式围起来。

$ var1=$[1 + 5]

$ echo $var1

6

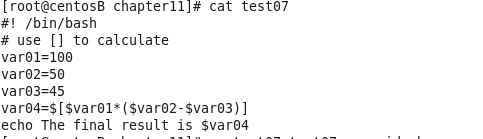
$ var2=$[$var1 \* 2]

$ echo $var2

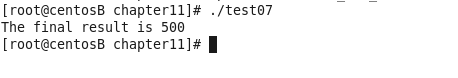
12

$

用方括号执行shell数学运算比用expr命令方便很多。这种技术也适用于shell脚本。



运行这个脚本会得到如下输出。

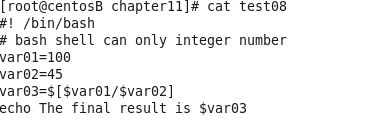


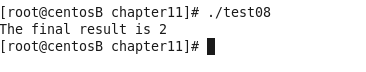
同样，注意在使用方括号来计算公式时，不用担心shell会误解乘号或其他符号。shell知道它不是通配符，因为它在方括号内。

在bash shell脚本中进行算术运算会有一个主要的限制。请看下例：

现在，运行一下，看看会发生什么：

$ chmod u+x test8





$

bash shell数学运算符只支持整数运算。若要进行任何实际的数学计算，这是一个巨大的限制。

说明z shell（zsh）提供了完整的浮点数算术操作。如果需要在shell脚本中进行浮点数运算，可以考虑看看z shell。

### 3 浮点解决方案

有几种解决方案能够克服bash中数学运算的整数限制。最常见的方案是用内建的bash计算器，叫作bc。

#### 1. bc的基本用法

bash计算器实际上是一种编程语言，它允许在命令行中输入浮点表达式，然后解释并计算该表达式，最后返回结果。bash计算器能够识别：

 数字（整数和浮点数）

 变量（简单变量和数组）

 注释（以#或C语言中的/\* \*/开始的行）

 表达式

 编程语句（例如if-then语句）

 函数

可以在shell提示符下通过bc命令访问bash计算器：

$ bc

bc 1.06.95

Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.

This is free software with ABSOLUTELY NO WARRANTY.

For details type 'warranty'.

12 \* 5.4

64.8

3.156 \* (3 + 5)

25.248

quit

$

这个例子一开始输入了表达式12 \* 5.4。bash计算器返回了计算结果。随后每个输入到计

算器的表达式都会被求值并显示出结果。要退出bash计算器，你必须输入quit。

浮点运算是由内建变量scale控制的。必须将这个值设置为你希望在计算结果中保留的小数

位数，否则无法得到期望的结果。

$ bc -q

3.44 / 5

0

scale=4

3.44 / 5

.6880

quit

$

scale变量的默认值是0。在scale值被设置前，bash计算器的计算结果不包含小数位。在将

其值设置成4后，bash计算器显示的结果包含四位小数。-q命令行选项可以不显示bash计算器冗长的欢迎信息。

除了普通数字，bash计算器还能支持变量。

$ bc -q

var1=10

var1 \* 4

40

var2 = var1 / 5

print var2

2

quit

$

变量一旦被定义，你就可以在整个bash计算器会话中使用该变量了。print语句允许你打印

变量和数字。

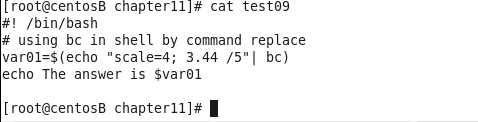
#### 2. 在脚本中使用bc

现在你可能想问bash计算器是如何在shell脚本中帮助处理浮点运算的。还记得命令替换吗？是的，可以用命令替换运行bc命令，并将输出赋给一个变量。基本格式如下：

variable=$(echo "options; expression" | bc)

第一部分options允许你设置变量。如果你需要不止一个变量，可以用分号将其分开。

expression参数定义了通过bc执行的数学表达式。这里有个在脚本中这么做的例子。



这个例子将scale变量设置成了四位小数，并在expression部分指定了特定的运算。运行

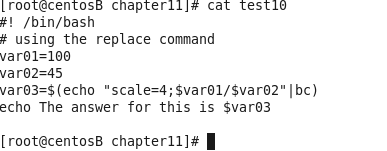
这个脚本会产生如下输出。

$ chmod u+x test9



$

太好了！现在你不会再只能用数字作为表达式值了。也可以用shell脚本中定义好的变量。

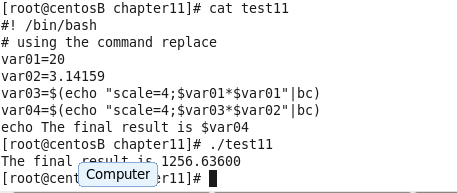


脚本定义了两个变量，它们都可以用在expression部分，然后发送给bc命令。别忘了用美

元符表示的是变量的值而不是变量自身。这个脚本的输出如下。



当然，一旦变量被赋值，那个变量也可以用于其他运算。



这个方法适用于较短的运算，但有时你会涉及更多的数字。如果需要进行大量运算，在一个

命令行中列出多个表达式就会有点麻烦。

有一个方法可以解决这个问题。bc命令能识别输入重定向，允许你将一个文件重定向到bc

命令来处理。但这同样会叫人头疼，因为你还得将表达式存放到文件中。

最好的办法是使用内联输入重定向，它允许你直接在命令行中重定向数据。在shell脚本中，

你可以将输出赋给一个变量。

variable=$(bc << EOF

options

statements

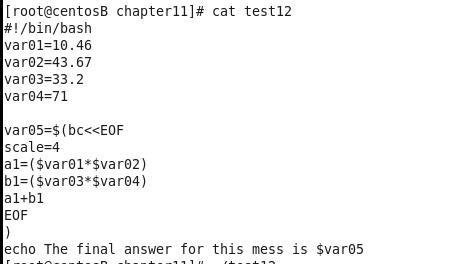
expressions

EOF

)

EOF文本字符串标识了内联重定向数据的起止。记住，仍然需要命令替换符号将bc命令的输出赋给变量。

现在可以将所有bash计算器涉及的部分都放到同一个脚本文件的不同行。下面是在脚本中使用这种技术的例子。



将选项和表达式放在脚本的不同行中可以让处理过程变得更清晰，提高易读性。EOF字符串

标识了重定向给bc命令的数据的起止。当然，必须用命令替换符号标识出用来给变量赋值的命令。你还会注意到，在这个例子中，你可以在bash计算器中赋值给变量。这一点很重要：在bash计算器中创建的变量只在bash计算器中有效，不能在shell脚本中使用。

## 8退出脚本

迄今为止所有的示例脚本中，我们都是突然停下来的。运行完最后一条命令时，脚本就结束

了。其实还有另外一种更优雅的方法可以为脚本划上一个句号。

shell中运行的每个命令都使用退出状态码（exit status）告诉shell它已经运行完毕。退出状态码是一个0～255的整数值，在命令结束运行时由命令传给shell。可以捕获这个值并在脚本中使用。

### 8.1 查看退出状态码

Linux提供了一个专门的变量$?来保存上个已执行命令的退出状态码。对于需要进行检查的

命令，必须在其运行完毕后立刻查看或使用$?变量。它的值会变成由shell所执行的最后一条命令

的退出状态码。

$ date

Sat Jan 15 10:01:30 EDT 2014

$ echo $?

0

$

按照惯例，一个成功结束的命令的退出状态码是0。如果一个命令结束时有错误，退出状态

码就是一个正数值。

$ asdfg

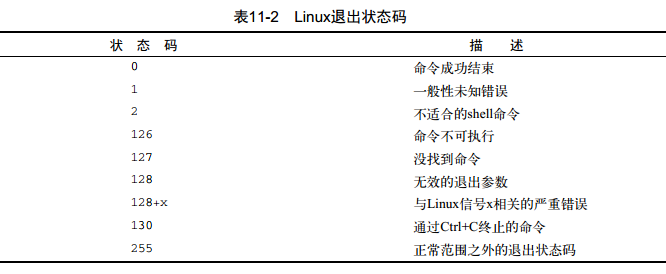
-bash: asdfg: command not found

$ echo $?

127

$

无效命令会返回一个退出状态码127。Linux错误退出状态码没有什么标准可循，但有一些可用的参考，如表所示。



退出状态码126表明用户没有执行命令的正确权限。

$ ./myprog.c

-bash: ./myprog.c: Permission denied

$ echo $?

126

$

另一个会碰到的常见错误是给某个命令提供了无效参数。

$ date %t

date: invalid date '%t'

$ echo $?

1

$

这会产生一般性的退出状态码1，表明在命令中发生了未知错误。

### 8.2 exit命令

默认情况下，shell脚本会以脚本中的最后一个命令的退出状态码退出。

$ ./test6

The result is 2

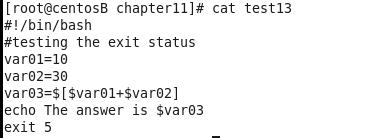
$ echo $?

0

$

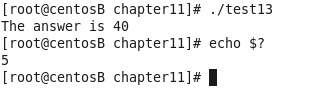
你可以改变这种默认行为，返回自己的退出状态码。exit命令允许你在脚本结束时指定一

个退出状态码。

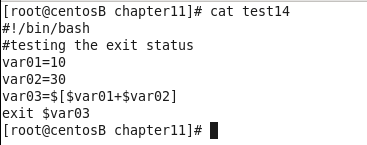


当查看脚本的退出码时，你会得到作为参数传给exit命令的值。

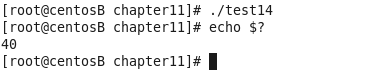
$ chmod u+x test13



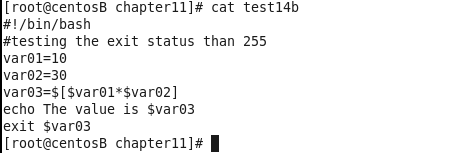
也可以在exit命令的参数中使用变量。



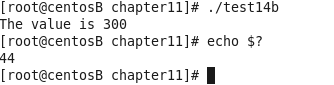
当你运行这个命令时，它会产生如下退出状态。



你要注意这个功能，因为退出状态码最大只能是255。看下面例子中会怎样。



现在运行它的话，会得到如下输出。



退出状态码被缩减到了0～255的区间。shell通过模运算得到这个结果。一个值的模就是被除后的余数。最终的结果是指定的数值除以256后得到的余数。在这个例子中，指定的值是300（返回值），余数是44，因此这个余数就成了最后的状态退出码。