Shell 编程教程

为什么要进行 shell 编程

在 Linux 系统中，虽然有各种各样的图形化接口工具，但是 sell 仍然是一个非常灵活的工具。Shell 不仅仅是命令的收集，而且是一门非常棒的编程语言。您可以通过使用 shell 使大量的任务自动化，shell 特别擅长系统管理任务，尤其适合那些易用性、可维护性和便携性比效率更重要的任务。

总的来说 shell 有两种含义：

1 命令：是用户与系统之间的桥梁

2 语言：可以使用 shell 语法以及命令来进行各种小程序的编写，类似于 windows 下的.bat(批处理文件)

下面，让我们一起来看看 shell 是如何工作的：

编写一个 shell 需要三个步骤

A 建立 shell 程序

B 改变 shell 程序属性 chmod +x shell pro

C 运行 shell 程序 在当前路径下 ./shell pro

a 建立 shell 脚本

Linux 中有好多中不同的 shell，但是通常我们使用 bash (bourne again shell) 进行 shell 编程，因为 bash 是免费的并且

很容易使用。所以在本文中笔者所提供的脚本都是使用 bash（但是在大多数情况下，这些脚本同样可以在 bash 的大姐，bourne shell 中运行）。

如同其他语言一样，通过我们使用任意一种文字编辑器，比如 nedit、kedit、emacs、vi 等来编写我们的 shell 程序。

程序必须以下面的行开始（必须放在文件的第一行）：

```
# !/bin/bash********
# This is a sample programe
# Author:esnow
# Date:2002-03-10
```

符号#!用来告诉系统它后面的参数是用来执行该文件的程序。在这个例子中我们使用/bin/bash 来执行程序。

b 改变 shell 脚本属性

当编辑好脚本时，如果要执行该脚本，还必须使其可执行。

要使脚本可执行：

chmod +x filename

c 运行 shell 脚本

然后，您可以通过输入： ./filename 来执行您的脚本。

注释

程序的规范性

在进行 shell 编程时，以#开头的句子表示注释，直到这一行的结束。我们真诚地建议您在程序中使用注释。如果您使用了注释，那么即使相当长的时间内没有使用该脚本，您也能在很短的时间内明白该脚本的作用及工作原理。

变量

有三种使用方法

A.直接在程序中写入

B.根据执行 shell 脚本的时候传递变量

C.运行脚本，等待用户输入

A

1 赋值

在其他编程语言中您必须使用变量。在 shell 编程中，所有的变量都由字符串组成，并且您不需要对变量进行声明。要赋值给一个变量，您可以这样写：

变量名=值

2 调用

取出变量值可以加一个美元符号（$）在变量前面：

```
#!/bin/bash
#对变量赋值：
a="hello world"
#  现在打印变量 a 的内容：
echo "A is:"
echo $a
```

在您的编辑器中输入以上内容，然后将其保存为一个文件 first。之后执行 chmod +x first

使其可执行，最后输入./first 执行该脚本。

这个脚本将会输出：

```
A is:
hello world
```

有时候变量名很容易与其他文字混淆，比如：

```
num=2
echo "this is the $numnd"
```

这并不会打印出"this is the 2nd"，而仅仅打印"this is the "，因为 shell 会去搜索变量 numnd 的值，但是这个变量时没有值的。可以使用花括号来告诉 shell 我们要打印的是 num 变量：

```
num=2
```

echo "this is the ${num}nd"

这将打印： this is the 2nd

有许多变量是系统自动设定的，这将在后面使用这些变量时进行讨论。

B ./shellpro　A B C

这个时候想显示 A B C 的内容可以使用

$1 $2 $3 ($1=A,$2=B,$3=C)

$* 表示传递的所有参数

$# 表示传递的参数数量

$0 表示 shell 脚本的名

编辑

[root@www home]# vi par

echo "echo par"

echo $1

echo $2

echo $3

echo "\$0 is $0"

echo "par num is $#"

更改属性

chmod +x par

运行

./par a b c

[root@www home]# ./par a b c

echo par

a

b

c

$0 is ./par

par num is 3

另外：shift 在后面介绍，用来移除一个参数，使用$#变少

如果您需要处理数学表达式，那么您需要使用诸如 expr 等程序（见下面）。

除了一般的仅在程序内有效的 shell 变量以外，还有环境变量。由 export 关键字处理过的变量叫做环境变量。我们不对环境变量进行讨论，因为通常情况下仅仅在登录脚本中使用环境变量。

Shell 命令和流程控制

在 shell 脚本中可以使用三类命令：

1)Unix 命令:

虽然在 shell 脚本中可以使用任意的 unix 命令，但是还是由一些相对更常用的命令。这些命令通常是用来进行文件和文字操作的。

常用命令语法及功能

echo "some text": 将文字内容打印在屏幕上

ls: 文件列表

wc –l file；wc -w file ；wc -c file: 计算文件行数计算文件中的单词数计算文件中的字符数

cp sourcefile destfile: 文件拷贝

mv oldname newname : 重命名文件或移动文件

rm file: 删除文件

grep 'pattern' file: 在文件内搜索字符串比如：grep 'searchstring' file.txt

cut -b colnum file: 指定欲显示的文件内容范围，并将它们输出到标准输出设备比如：输出每行第 5 个到第 9 个字符 cut -b5-9 file.txt 千万不要和 cat 命令混淆，这是两个完全不同的命令

cat file.txt: 输出文件内容到标准输出设备（屏幕）上

file somefile: 得到文件类型

read var: 提示用户输入，并将输入赋值给变量

sort file.txt: 对 file.txt 文件中的行进行排序

uniq: 删除文件中出现的重复行比如： sort file.txt | uniq

expr: 进行数学运算 Example: add 2 and 3expr 2 "+" 3

[root@sunyore shell_esnow]# expr 2"+"7

2+7

[root@sunyore shell_esnow]# expr 2 "+" 7

9

find: 搜索文件比如：根据文件名搜索 find . -name filename -print

tee: 将数据输出到标准输出设备(屏幕) 或文件比较，如：somecommand | tee outfile

basename file: 返回不包含路径的文件名比如：basename /bin/tux 将返回 tux

dirname file: 返回文件所在路径比如：dirname /bin/tux 将返回 /bin

head file: 打印文本文件开头几行

head -n num filename

tail file : 打印文本文件末尾几行

tail –n num filename

sed: Sed 是一个基本的查找替换程序。可以从标准输入（比如命令管道）读入文本，并将结果输出到标准输出（屏幕）。该命令采用正则表达式（见参考）进行搜索。不要和 shell 中的通配符相混淆。比如：将 linuxfocus 替

换为 LinuxFocus ：

cat text.file | sed 's/linuxfocus/LinuxFocus/g' > newtext.file

awk: awk 用来从文本文件中提取字段。缺省地，字段分割符是空格，可以使用-F 指定其他分割符。cat file.txt | awk -F, '{print $1 "," $3 }'这里我们使用”，”作为字段分割符，同时打印第一个和第三个字段的内容。如果该文件内容如下： Adam Bor, 34, IndiaKerry Miller, 22, USA 命令输出结果为：Adam Bor, IndiaKerry Miller

cat temp | awk '{print $1 "?" $3 "!" $4 }'

Default:space is value

2) 概念: 管道, 重定向和 backtick

这些不是系统命令，但是他们真的很重要。

管道 (|) 将一个命令的输出作为另外一个命令的输入。

grep "hello" file.txt | wc -l

在 file.txt 中搜索包含有”hello”的行并计算其行数。

在这里 grep 命令的输出作为 wc 命令的输入。当然您可以

使用多个命令。

重定向：将命令的结果输出到文件，而不是标准输出（屏幕）。

> 写入文件并覆盖旧文件

>> 加到文件的尾部，保留旧文件内容。

反短斜线

使用反短斜线可以将一个命令的输出作为另外一个命令的一个命令行参数。

命令：

find . -mtime -1 -type f -print

用来查找过去 24 小时（-mtime –2 则表示过去 48 小时）内修改过的文件。如果您想将所有查找到的文件打一个包，则可以使用以下脚本：

```
#!/bin/bash
# The ticks are backticks (`) not normal quotes ('):
tar -zcvf lastmod.tar.gz `find . -mtime -1 -type f -print`
```

3) 流程控制

"if" 表达式 如果条件为真则执行 then 后面的部分：

if ....; then

....

elif ....; then

....

else

....

fi

大多数情况下，可以使用测试命令来对条件进行测试。比如可以比较字符串、判断文件是否存在及是否可读等等…

通常用" [ ] "来表示条件测试。注意这里的空格很重要。要确保方括号的空格。 下面是 test 命令，非常重要

A 字符串比较

= 两个字符串相等

！= 两个字符串不等

-z 空串

-n 非空串

B 数值比较

-eq 数值相等。

-ne 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

C 文件、变量以及目录

[-f "somefile"] ：判断是否是一个文件

[-x "/bin/ls"] ：判断/bin/ls 是否存在并有可执行权限

[-n "$var"] ：判断$var 变量是否有值

["$a" = "$b"] ：判断$a 和$b 是否相等

[-d “$a”] ：是否是目录

[-s “$a”] ：文件长度大于 0，非空

[-L “$a”] ：是否是个符号链接文件

[-r “$a”] ：是否可读

[-z “$a”] ：是否为空

执行 man test 可以查看所有测试表达式可以比较和判断的类型。

直接执行以下脚本：

```
#!/bin/bash
if [ "$SHELL" = "/bin/bash" ]; then
    echo "your login shell is the bash (bourne again shell)"
else
    echo "your login shell is not bash but $SHELL"
fi
```

变量$SHELL 包含了登录 shell 的名称，我们和/bin/bash 进行了比较。

快捷操作符

熟悉 C 语言的朋友可能会很喜欢下面的表达式：

[-f "/etc/shadow"] && echo "This computer uses shadow passwors"

这里 && 就是一个快捷操作符，如果左边的表达式为真则执行右边的语句。您也可以认为是逻辑运算中的与操作。上例中表示如果/etc/shadow 文件存在则打印” This computer uses shadow passwors”。同样或操作(||)在 shell 编程中也是可用的。这里有个例子：

```
#!/bin/bash
mailfolder="/var/spool/mail/esnow"
if [ -r "$mailfolder" ];then
        echo "Can read $mailfolder"
else
        echo "Can not read $mailfolder";
          exit 0;
fi
```

```
echo "$mailfolder has mail from:"
grep "^From " $mailfolder
#Notice ^ means :From head &
```

该脚本首先判断 mailfolder 是否可读。如果可读则打印该文件中的"From" 一行。如果不可读则条件生效，打印错误信息后脚本退出。这里有个问题，那就是我们必须有两个命令：

-打印错误信息

-退出程序

我们使用花括号以匿名函数的形式将两个命令放到一起作为一个命令使用。一般函数将在下文提及。

不用与和或操作符，我们也可以用 if 表达式作任何事情，但是使用与或操作符会更便利很多。

case 表达式可以用来匹配一个给定的字符串，而不是数字。

```
case ... in
...) do something here ;;
esac
```

file 命令可以辨别出一个给定文件的文件类型，比如：

```
file lf.gz
```

这将返回：

```
lf.gz: gzip compressed data, deflated, original filename,
last modified: Mon Aug 27 23:09:18 2001, os: Unix
```

我们利用这一点写了一个叫做 smartzip 的脚本，该脚本可以自动解压 bzip2, gzip 和 zip 类型的压缩文件：

```
#!/bin/bash
ftype=`file "$1"`
case "$ftype" in
"$1: Zip archive"*)
    unzip "$1" ;;
"$1: gzip compressed"*)
    gunzip "$1" ;;
"$1: bzip2 compressed"*)
    bunzip2 "$1" ;;
*) error "File $1 can not be uncompressed with smartzip";;
esac
```

您可能注意到我们在这里使用了一个特殊的变量$1。该变量包含了传递给该程序的第一个参数值。也就是说，当我们运行：

```
smartzip articles.zip
```

$1 就是字符串 articles.zip

select 表达式是一种 bash 的扩展应用，尤其擅长于交互式使用。用户可以从一组不同的值中进行选择。

```
select var in ... ; do
   break
done
.... now $var can be used ....
```

下面是一个例子：

```
#!/bin/bash
echo "What is your favourite OS?"
select var in "Linux" "Gnu Hurd" "Free BSD" "Other"; do
            break
done
echo "You have selected $var"
```

下面是该脚本运行的结果：

```
What is your favourite OS?
1) Linux
2) Gnu Hurd
3) Free BSD
4) Other
#? 1
You have selected Linux
```

您也可以在 shell 中使用如下的 loop 表达式：

```
while ...; do
....
done
```

while-loop 将运行直到表达式测试为真。will run while the expression that we test for is true. 关键字"break" 用来跳出循环。而关键字”continue”用来不执行余下的部分而直接跳到下一个循环。

for-loop 表达式查看一个字符串列表 (字符串用空格分隔) 然后将其赋给一个变量：

```
for var in ....; do
   ....
done
```

在下面的例子中，将分别打印 ABC 到屏幕上：

```
#!/bin/bash
for var in A B C ; do
   echo "var is $var"
done
```

下面是一个更为有用的脚本 showrpm，其功能是打印一些 RPM 包的统计信息：

```
#!/bin/bash
# list a content summary of a number of RPM packages
# USAGE: showrpm rpmfile1 rpmfile2 ...
# EXAMPLE: showrpm /cdrom/Thizlinux/RPMS/*.rpm
for rpmpackage in $*; do
   if [ -r "$rpmpackage" ];then
      echo "======== $rpmpackage =========="
      rpm -qi -p $rpmpackage
   else
      echo "ERROR: cannot read file $rpmpackage"
   fi
done
```

这里出现了第二个特殊的变量$*，该变量包含了所有输入的命令行参数值。如果您运行 showrpm openssh.rpm w3m.rpm webgrep.rpm

此时 $* 包含了 3 个字符串，即 openssh.rpm, w3m.rpm and webgrep.rpm.

引号

在向程序传递任何参数之前，程序会扩展通配符和变量。这里所谓扩展的意思是程序会把通配符（比如*）替换成合适的文件名，它变量替换成变量值。为了防止程序作这种替换，您可以使用引号：让我们来看一个例子，假设在当前目录下有一些文件，两个 txt 文件， mail.txdt 和 tux.txt。

#!/bin/bash

echo *.txt

这将打印出" mail.txdt tux.txt "的结果。

引号 (单引号和双引号) 将防止这种通配符扩展：

#!/bin/bash

echo "*.txt"

echo '*.txt'

这将打印"*.txt" 两次。

单引号更严格一些。它可以防止任何变量扩展。双引号可以防止通配符扩展但允许变量扩展。

#!/bin/bash

echo $SHELL

echo "$SHELL"

```
echo '$SHELL'
```

运行结果为：

```
/bin/bash
/bin/bash
$SHELL
```

最后，还有一种防止这种扩展的方法，那就是使用转义字符——反斜杆：

```
echo *.jpg
echo \$SHELL
```

这将输出：

```
*.jpg
$SHELL
```

Here documents

当要将几行文字传递给一个命令时，here documents（目前还没有见到过对该词适合的翻译）一种不错的方法。对每个脚本写一段帮助性的文字是很有用的，此时如果我们四有那个 here documents 就不必用 echo 函数一行行输出。一个 "Here document" 以 << 开头，后面接上一个字符串，这个字符串还必须出现在 here document 的末尾。下面是一个例子，在该例

子中，我们对多个文件进行重命名，并且使用 here documents 打印帮助：

```
vi here
#!/bin/bash
# we have less than 3 arguments. Print the help text:
if [ $# -lt 3 ] ; then
cat <<HELP
ren -- renames a number of files using sed regular expressions

USAGE: ren 'regexp' 'replacement' files...

EXAMPLE: rename all *.HTM files in *.html:
   ren 'HTM$' 'html' *.HTM

HELP
   exit 0
fi
OLD="$1"
NEW="$2"
# The shift command removes one argument from the list of
# command line arguments.
```

```
shift
shift
# $* contains now all the files:
for file in $*; do
    if [ -f "$file" ] ; then
        newfile=`echo "$file" | sed "s/${OLD}/${NEW}/g"`
        if [ -f "$newfile" ]; then
            echo "ERROR: $newfile exists already"
        else
            echo "renaming $file to $newfile ..."
            mv "$file" "$newfile"
        fi
    fi
done
```

这是一个复杂一些的例子。让我们详细讨论一下。第一个 if 表达式判断输入命令行参数是否小于 3 个 (特殊变量$# 表示包含参数的个数) 。如果输入参数小于 3 个，则将帮助文字传递给 cat 命令，然后由 cat 命令将其打印在屏幕上。打印帮助文字后程序退出。 如果输入参数等于或大于 3 个，我们就将第一个参数赋值给变量 OLD，第二个参数赋值给变量 NEW。下一步，我们使用两个 shift 命令将第一个和第二个参数从参数列

表中删除，这样原来的第三个参数就成为参数列表$*的第一个参数。然后我们开始循环，命令行参数列表被一个接一个地被赋值给变量$file。接着我们判断该文件是否存在，如果存在则通过 sed 命令搜索和替换来产生新的文件名。然后将反短斜线内命令结果赋值给 newfile。这样我们就达到了我们的目的：得到了旧文件名和新文件名。然后使用 mv 命令进行重命名。

函数

如果您写了一些稍微复杂一些的程序，您就会发现在程序中可能在几个地方使用了相同的代码，并且您也会发现，如果我们使用了函数，会方便很多。一个函数是这个样子的：

```
functionname()
{
# inside the body $1 is the first argument given to the function
# $2 the second ...
body
}
```

您需要在每个程序的开始对函数进行声明。

下面是一个叫做 xtitlebar 的脚本，使用这个脚本您可以改变终端窗口的名称。这里使用了一个叫做 help 的函数。正如您可以

看到的那样，这个定义的函数被使用了两次。

```
#!/bin/bash
# vim: set sw=4 ts=4 et:

help()
{
    cat <<HELP
xtitlebar -- change the name of an xterm, gnome-terminal or
kde konsole

USAGE: xtitlebar [-h] "string_for_titelbar"

OPTIONS: -h help text

EXAMPLE: xtitlebar "cvs"

HELP
    exit 0
}

# in case of error or if -h is given we call the function help:
[ -z "$1" ] && help
```

```
[ "$1" = "-h" ] && help

# send the escape sequence to change the xterm titelbar:
echo -e "33]0;$107"
#
```

在脚本中提供帮助是一种很好的编程习惯，这样方便其他用户（和您）使用和理解脚本。

命令行参数

我们已经见过$* 和 $1, $2 ... $9 等特殊变量，这些特殊变量包含了用户从命令行输入的参数。迄今为止，我们仅仅了解了一些简单的命令行语法（比如一些强制性的参数和查看帮助的-h 选项）。但是在编写更复杂的程序时，您可能会发现您需要更多的自定义的选项。通常的惯例是在所有可选的参数之前加一个减号，后面再加上参数值 (比如文件名)。

有好多方法可以实现对输入参数的分析，但是下面的使用 case 表达式的例子无遗是一个不错的方法。

```
#!/bin/bash
help()
```

```
{
   cat <<HELP
This is a generic command line parser demo.
USAGE EXAMPLE: cmdparser -l hello -f -- -somefile1
somefile2
HELP
   exit 0
}
while [ -n "$1" ]; do
case $1 in
      -h) help;shift 1;; # function help is called
      -f) opt_f=1;shift 1;; # variable opt_f is set
      -l) opt_l=$2;shift 2;; # -l takes an argument -> shift by 2
      --) shift;break;; # end of options
      -) echo "error: no such option $1. -h for help";exit 1;;
      *) break;;
esac
done

echo "opt_f is $opt_f"
echo "opt_l is $opt_l"
echo "first arg is $1"
```

```
echo "2nd arg is $2"
```

您可以这样运行该脚本：

```
cmdparser -l hello -f -- -somefile1 somefile2
```

返回的结果是：

```
opt_f is 1
opt_l is hello
first arg is -somefile1
2nd arg is somefile2
```

这个脚本是如何工作的呢？脚本首先在所有输入命令行参数中进行循环，将输入参数与 case 表达式进行比较，如果匹配则设置一个变量并且移除该参数。根据 unix 系统的惯例，首先输入的应该是包含减号的参数。

实例

一般编程步骤

现在我们来讨论编写一个脚本的一般步骤。任何优秀的脚

本都应该具有帮助和输入参数。并且写一个伪脚本（framework.sh），该脚本包含了大多数脚本都需要的框架结构，是一个非常不错的主意。这时候，在写一个新的脚本时我们只需要执行一下 copy 命令：

cp framework.sh myscript

然后再插入自己的函数。

让我们再看两个例子：

文件循环程序

如果一个文件中的数据在不断增加，那么一段时间内他的体积会变大，在这样的情况下你也许想依次来改变文件，当然你可以使用 mv 命令改名字，然后 touch 出新文件来，如果使用程序来实现改怎么办呢？

```
#!/bin/bash
# vim: set sw=4 ts=4 et:
ver="0.1"
help()
{
    cat <<HELP
rotatefile -- rotate the file name
USAGE: rotatefile [-h] filename
```

```
OPTIONS: -h help text
EXAMPLE: rotatefile out
This will e.g rename out.2 to out.3, out.1 to out.2, out to out.1
and create an empty out-file

The max number is 10
version $ver
HELP
        exit 0
}
error()
{
        echo "$1"
        exit 1
}
while [ -n "$1" ]; do
case $1 in
        -h) help;shift 1;;
        --) break;;
        -*) echo "error: no such option $1. -h for help";exit 1;;
        *) break;;
esac
```

```
done
# input check:
if [ -z "$1" ] ; then
error "ERROR: you must specify a file, use -h for help"
fi
filen="$1"
# rename any .1 , .2 etc file:
for n in 9 8 7 6 5 4 3 2 1; do
        if [ -f "$filen.$n" ]; then
                p=`expr $n + 1`
                echo "mv $filen.$n $filen.$p"
                mv $filen.$n $filen.$p
        fi
done
# rename the original file:
if [ -f "$filen" ]; then
        echo "mv $filen $filen.1"
        mv $filen $filen.1
fi
echo touch $filen
touch $filen
```

这个脚本是如何工作的呢？在检测用户提供了一个文件

名以后，我们进行一个 9 到 1 的循环。文件 9 被命名为 10，文件 8 重命名为 9 等等。循环完成之后，我们将原始文件命名为文件 1 同时建立一个与原始文件同名的空文件。

二进制到十进制的转换

脚本 b2d 将二进制数 (比如 1101) 转换为相应的十进制数。这也是一个用 expr 命令进行数学运算的例子：

```
#!/bin/bash
# vim: set sw=4 ts=4 et:
help()
{
   cat <<HELP
b2h -- convert binary to decimal

USAGE: b2h [-h] binarynum

OPTIONS: -h help text

EXAMPLE: b2h 111010
will return 58
HELP
   exit 0
}
```

```
error()
{
        # print an error and exit
        echo "$1"
        exit 1
}

lastchar()
{
        # return the last character of a string in $rval
        if [ -z "$1" ]; then
                # empty string
                rval=""
                return
        fi
        # wc puts some space behind the output this is why we
need sed:
        numofchar=`echo -n "$1" | wc -c | sed 's/ //g' `
        # now cut out the last char
        rval=`echo -n "$1" | cut -b $numofchar`
}
```

```
chop()
{
    # remove the last character in string and return it in $rval
    if [ -z "$1" ]; then
        # empty string
        rval=""
        return
    fi
    # wc puts some space behind the output this is why we
need sed:
    numofchar=`echo -n "$1" | wc -c | sed 's/ //g' `
    if [ "$numofchar" = "1" ]; then
        # only one char in string
        rval=""
        return
    fi
    numofcharminus1=`expr $numofchar "-" 1`
    # now cut all but the last char:
    rval=`echo -n "$1" | cut -b 0-${numofcharminus1}`
}
```

```
while [ -n "$1" ]; do
case $1 in
        -h) help;shift 1;; # function help is called
        --) shift;break;; # end of options
        -*) error "error: no such option $1. -h for help";;
        *) break;;
esac
done
# The main program
sum=0
weight=1
# one arg must be given:-z means is empty string?
[ -z "$1" ] && help
binnum="$1"
binnumorig="$1"
'-n means the var have value
while [ -n "$binnum" ]; do
        lastchar "$binnum"
        if [ "$rval" = "1" ]; then
                sum=`expr "$weight" "+" "$sum"`
        fi
```

```
    # remove the last position in $binnum
    chop "$binnum"
    binnum="$rval"
    weight=`expr "$weight" "*" 2`
done
echo "binary $binnumorig is decimal $sum"
```

该脚本使用的算法是利用十进制和二进制数权值(1,2,4,8,16,..)，比如二进制"10"可以这样转换成十进制：

0 * 1 + 1 * 2 = 2

为了得到单个的二进制数我们是用了 lastchar 函数。该函数使用 wc –c 计算字符个数，然后使用 cut 命令取出末尾一个字符。Chop 函数的功能则是移除最后一个字符。

调试

最简单的调试命令当然是使用 echo 命令。您可以使用 echo 在任何怀疑出错的地方打印任何变量值。这也是绝大多数的 shell 程序员要花费 80%的时间来调试程序的原因。Shell 程序的好处在于不需要重新编译，插入一个 echo 命令也不需要多少时间。

shell 也有一个真实的调试模式。如果在脚本"strangescript" 中有错误，您可以这样来进行调试：

```
sh -x strangescript
```

这将执行该脚本并显示所有变量的值。

shell 还有一个不需要执行脚本只是检查语法的模式。可以这样使用：

sh -n your_script

这将返回所有语法错误。

我们希望您现在可以开始写您自己的 shell 脚本.

菜菜