Shell编程基础

从第一行开始

我们可以使用任意一种文字编辑器，比如gedit、kedit、emacs、vi等来编写shell脚本，它必须以如下行开始（必须放在文件的第一行）：

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

此行称为 shebang（就是 sharp (#) + bang (!) 的意思），会指引操作系统使用接下来指定的程序运行此文件。此处 /bin/bash 执行我们的文件。

一些人使用 #!/bin/sh 来让 sh 执行文件，按照习惯这表示任何支持 POSIX shell 命令语言的 sh程序。为了用上我们所喜欢的 bash 拓展语法功能，我们就不这么用了。如果你使用别的脚本，例如 /bin/tcsh，照着样子加个 #! 就行。

编辑结束并保存后，如果直接要执行该脚本，必须先使其具有可执行属性：

```
chmod +x filename
```

此后在该脚本所在目录下，输入 ./filename 即可执行该脚本。

变量

Shell 变量默认全都是字符串。

变量赋值和引用

Shell 编程中，使用变量无需事先声明。变量名的命名遵守正则表达式 [a-zA-Z_][a-zA-Z0-9_]+，也就是由大小写字母数字和下划线组成，且不能以数字开头。请注意 shell 环境中的确有以数字和特殊符号开头的变量名，但是那些东西不可以用接下来的方式赋值。

需要给变量赋值时，可以这么写：

```
varname=值 [var2=val2 ....]
```

请注意这边等号左右不能有空格。 要取用一个变量的值，只需在变量名前面加一个 $：

```
# assign
a="hello world"  # 等号两边均不能有空格存在
# print
printf '%s\n' "A is: $a"
```

挑个自己喜欢的编辑器，输入上述内容，并保存为文件 first，然后执行 chmod +x first 使其可执行，最后输入 ./first 执行该脚本。其输出结果如下：

```
A is: hello world
```

有时候变量名可能会和其它文字（匹配最长的符合变量名或特殊变量名要求的内容）混淆，比如：

```
num=2
echo "this is the $numnd"   # 输出 this is the       — shell 尝试寻找 $numnd 的值
echo "this is the ${num}nd" # 输出 this is the 2nd   — 修好了！
# 花括号可以隔开变量名，但是放歪了的话…
echo "this is the {$num}nd" # 输出 this is the {2}nd — 切是切开了，但是…
```

变量算术

Shell 变量默认都是字符串。这也就是说，你尝试这么做，肯定没用：

```
var=1
var=$var+1
echo $var # 输出 1+1
```

我们可以用很多方法达成我们的目标。首先是好孩子的方法——C 风格表达式。

```
var=0
# bash 里面可以用 (( )) 执行 C 风格的算术表达式。
# 如果你接下来还会读 if 那一段的话，你还会知道这玩意的返回和 C 的非零真假判断一致。
(( var += 1 )) # 这是一种，现在 var 是 1
(( var++ ))    # 这也是一种自增，2
(( var = var * var )) # 怎么来乘法了！var 现在是 4。
let 'var = var / 3'   # 还是只有 bash 才能用的拓展。除法是整数除法，向 0 舍入，1。
# 来一点不一定要 bash 的方法吧，毕竟 sh 还有很多种，总不能全报错了吧。
# $(( )) 会展开成为这个 C 表达式求值的结果。以前 bash 有个 $[ ] 一样，但是别去用。
echo $((var += 2))    # echo 出 3，var 现在是 3。
var=$((var-1))        # 展开成 var=2，var 现在是……还用说吗，2。
```

以前还有人用 expr 之类的外部程序来，不过这属于杀鸡用牛刀。并且调用外部程序浪费时间性能差。

```
var=1
var=$(expr "$var" + 1) # expr 收到三个参数 '1' '+' '1',

                       # 按照 expr --help 里面写的方法运行
                       # 然后输出替换掉 $() 这里变成 var=2。

var=`expr "$var" + 1`  # 前面一行的老写法，千万千万不要学。
```

Shell里的流程控制

if 语句

if 表达式如果条件命令组为真，则执行 then 后的部分。标准形式：

```
if
   判断命令，可以有很多个，真假取最后的返回值
then
   如果前述为真做什么
[ # 方括号代表可选，别真打进去了！
elif
   可以再来个判断，如果签名为假继续尝试这里
then
   如果前述为真做什么 ]
else
   如果全都不行做什么
fi # 结束，就是倒写的 if 啦。
```

现实生活中一般判断只有一个命令，所以你看到的一般是：

```
if ....; then # 你也可以写成 if 之后换行，这样就不用分号了。
   ....
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……在 bash 中一般采用更好用的 `...` 语法进行条件测试，而通用方法是 `[ ... ]`（相当于 `test ...`）。两者都接纳的常用测试语句有：

- `-f "filename"`

  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

前者可以使用 help [[查询用法，后者使用 help [(bash) 或 man test 查询。下面的语句用到了这个内容：

```
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 包含有登录shell的名称，我们拿它和 /bin/bash 进行比较以判断当前使用的shell是否为bash。你可能会问了，要是 bash 路径不是这个呢?

&& 和 || 操作符

熟悉c语言的朋友可能会喜欢下面的表达式：

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

这里的 && 就是一个快捷操作符，如果左边的表达式为真（返回 0—“成功”）则执行右边的语句，你也可以把它看作逻辑运算里的与操作。上述脚本表示如果/etc/shadow文件存在，则打印“This computer uses shadow passwords”。

同样shell编程中还可以用或操作 (||)，例如：

```
#!/bin/bash

mailfolder=/var/spool/mail/james
[ -r "$mailfolder" ] || { echo "Can not read $mailfolder"; exit 1; }
echo "$mailfolder has mail from:"
grep "^From " $mailfolder
```

该脚本首先判断mailfolder是否可读，如果可读则打印该文件中以"From"开头的行。如果不可读则或操作生效，打印错误信息后脚本退出。需要注意的是，这里我们必须使用如下两个命令：

```
{
   echo "Can not read $mailfolder"; # 打印错误信息
   exit 1; # 退出程序
}
```

我们使用花括号以组合命令的形式将两个命令放到一起作为一个命令使用。即使不用与和或操作符，我们也可以用if表达式完成任何事情，但是使用与或操作符会更便利很多。

要注意 Shell 中的 && || 程序流操作符不表现任何优先级区别，完全是先看到谁就先处理谁的关系。

case 语句

case表达式可以用来匹配一个给定的字符串，而不是数字（可别和C语言里的switch...case混淆）。

```
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，如：file lf.gz，其输出结果为：

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

我们利用这点写了一个名为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" ;;
*)
    echo "File $1 can not be uncompressed with smartzip";;
esac
```

你可能注意到上面使用了一个特殊变量 $1，该变量包含有传递给该脚本的第一个参数值。也就是说，当我们运行：

```
smartzip articles.zip
```

$1 就是字符串 articles.zip。

select 循环语句

select 循环语句是bash的一种扩展应用，擅长于交互式场合。

用户可以从一组不同的值中进行选择：

```
pocket=()
select var in 跳跳糖 糖 很多糖 企鹅糖; do
  echo "除了 $var 还要什么吗？"
  if ((RANDOM%4 == 0)); then
    echo "呀！时间不够了，快上车！"
    break # break 还是那个 break
  fi
  pocket+=("$var")
done
echo "你最后说的那个 $var 弄丢了……"
IFS='、'
echo "现在口袋里只有：${pocket[*]}。"
IFS=$' \t\n'
```

下面是一个简单的示例：

```
#!/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"
```

该脚本的运行结果如下：

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

while/for 循环

在shell中，可以使用如下循环：

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

只要测试表达式条件为真，则while循环将一直运行。关键字"break"用来跳出循环，而关键字"continue"则可以跳过一个循环的余下部分，直接跳到下一次循环中。

for循环会查看一个字符串列表（字符串用空格分隔），并将其赋给一个变量：

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

下面的示例会把A B C分别打印到屏幕上：

```
#!/bin/bash

for var in A B C ; do
    echo "var is $var"
done
```

下面是一个实用的脚本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/RedHat/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变量@，该变量包含有输入的所有命令行参数值。如果你运行 *showrpmopenssh.rpmw3m.rpmwebgrep.rpm*，那么"@"(有引号) 就包含有 3 个字符串，即openssh.rpm, w3m.rpm和 webgrep.rpm。$*的意思是差不多的。但是只有一个字串。如果不加引号，带空格的参数会被截断。

Shell里的一些特殊符号

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。这里所谓的扩展是指程序会把通配符（比如*）替换成适当的文件名，把变量替换成变量值。我们可以使用引号来防止这种扩展，先来看一个例子，假设在当前目录下有两个jpg文件：mail.jpg和tux.jpg。

```
#!/bin/bash

echo *.jpg   # => mail.jpg tux.jpg
```

引号（单引号和双引号）可以防止通配符*的扩展：

```
#!/bin/bash

echo "*.jpg"  # => *.jpg
echo '*.jpg'  # => *.jpg
```

其中单引号更严格一些，它可以防止任何变量扩展；而双引号可以防止通配符扩展但允许变量扩展：

```
#!/bin/bash

echo $SHELL   # => /bin/bash
echo "$SHELL" # => /bin/bash
echo '$SHELL' # => $SHELL
```

此外还有一种防止这种扩展的方法，即使用转义字符—反斜杠 \ ：

```
echo \*.jpg   # => *.jpg
echo \$SHELL  # => $SHELL
```

Here Document

当要将几行文字传递给一个命令时，用here document是一种不错的方法。对每个脚本写一段帮助性的文字是很有用的，此时如果使用here document就不必用echo函数一行行输出。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 document打印帮助：

```
#!/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                    #这里HELP要顶格写，前面不能有空格或者TAB制表符。如果cat一行写成cat &lt;&lt; -HELP，前边可以带TAB.
    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
```

示例有点复杂，我们需要多花点时间来说明一番。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

You can't use 'macro parameter character #' in math mode

*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

Shell里的函数

如果你写过比较复杂的脚本，就会发现可能在几个地方使用了相同的代码，这时如果用上函数，会方便很多。函数的大致样子如下：

```
# 别笑，bash 里面函数名的确可以这样……
# (POSIX sh 函数名倒是和变量名要求差不多)
我是一个函数() {
    # 函数里面 $1 $2 对应函数所接受到的第一、第二……个参数。
    这里有很多命令
}
```

函数没有必要声明。只要在执行之前出现定义就行

下面是一个名为xtitlebar的脚本，它可以改变终端窗口的名称。这里使用了一个名为help的函数，该函数在脚本中使用了两次：

```
#!/bin/bash

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:
if [[ $1 == '' || $1 == '-h' ]]; then
  help
fi
# send the escape sequence to change the xterm titelbar:
echo -e "\033]0;$1\007"
#
```

在脚本中提供帮助是一种很好的编程习惯，可以方便其他用户（和自己）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表达式的例子无疑是一个不错的方法。

```
#!/bin/bash

help()
{
    cat &lt;&lt; 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 -&gt; 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"
```

你可以这样运行该脚本：

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

返回结果如下：

```
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。

Shell脚本示例

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。写一个框架脚本（framework.sh），该脚本包含了大多数脚本需要的框架结构，是一个非常不错的主意。这样一来，当我们开始编写新脚本时，可以先执行如下命令：

```
cp framework.sh myscript
```

然后再插入自己的函数。

让我们来看看如下两个示例。

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

b2d -- convert binary to decimal

USAGE: b2d [-h] binarynum

OPTIONS: -h help text

EXAMPLE: b2d 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" | sed 's/ //g' | wc -c `
    #sed 's/ //g'  所有空白去掉  sed 's/ /\t/g' 所有空白用t代替
    # now cut out the last char  抓取第numofchar个字节
    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 "$1" | cut -b -$numofcharminus1`
    #原来的 rval=`echo -n "$1" | cut -b 0-${numofcharminus1}`运行时出错.
    #原因是cut从1开始计数，应该是cut -b 1-${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 "$1" ] &amp;amp;&amp;amp; help
binnum="$1"
binnumorig="$1"

while [ -n "$binnum" ]; do
    lastchar "$binnum"
    if [ "$rval" = "1" ]; then
        sum=`expr "$weight" "+" "$sum"`
        # $expr 10 + 10    20 expr提示是计算操作
    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。

但是还记得前面怎么说的吗？进制转换哪需要这么麻烦：

```
#!/bin/bash
while read -p 'input a binary...' num; do
   if [[ $num == *[!01]* ]]; then
      echo "含有 0 1 之外的字符"
   fi
   echo "$((0x$num))"       # 在 num 头上糊一个 0x 然后跑数学计算——就完事了！
   printf "%d\n" "0x$num"    # printf 也可以凑热闹啊
done
```

如果你喜欢自己算的话，其实也可以从左到右来（反正数学计算不要有事没事玩 expr 啦）：

```
#!/bin/bash
# 人人皆知的 Horner 规则
value=0
echo "写一堆 1 0 完了回车"
while read -n 1 char; do
   case $char in
      (0|1) ;;       # 好
      ('')  break;; # 没了
      (*)   echo "你说啥? "; break;;
   esac
   ((value *= 2))
   ((value += char))
done
echo "$value"
```

文件循环移动

你可能有这样的需求并一直都这么做：将所有发出邮件保存到一个文件中。但是过了几个月之后，这个文件可能会变得很大以至于该文件的访问速度变慢；下面的脚本 rotatefile 可以解决这个问题。这个脚本可以重命名邮件保存文件（假设为outmail）为outmail.1，而原来的outmail.1就变成了 outmail.2 等等...

```
#!/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


    version $ver

HELP

    exit 0
}

if [[ $1 == '-h' || $1 == '' ]]; then
  help
fi

filename=$1

# 我们先找到最大的数字再说。
max=0
while [ -f "$filename.$((++max))" ]; do
  : # 什么都不用做，我们已经顺手用 ++max 自增了 max 了。
done

# 然后从最大的一路重命名下来。
for ((i=max; i>0; i--)); do
  # 数字加个 1，好给前一个让位子。
  mv "$filename.$i" "$filename.$((i+1))"
done

# 最后我们点名要重命名的：
if [ -f "$filename" ]; then
    mv "$filename" "$filename.1"
fi

# 重新创建一下。
: > "$filename"
```

。

脚本调试

最简单的调试方法当然是使用echo命令。你可以在任何怀疑出错的地方用echo打印变量值，这也是大部分shell程序员花费80%的时间用于调试的原因。Shell脚本的好处在于无需重新编译，而插入一个echo命令也不需要多少时间。

shell也有一个真正的调试模式，如果脚本"strangescript"出错，可以使用如下命令进行调试：

```
sh -x strangescript
```

上述命令会执行该脚本，同时显示所有变量的值。

shell还有一个不执行脚本只检查语法的模式，命令如下：

```
sh -n your_script
```

这个命令会返回所有语法错误。

我们希望你现在已经可以开始编写自己的shell脚本了，尽情享受这份乐趣吧！ :)