40. shell script 程序设计

先前谈到shell（bash）的时候，我们曾说过它内置了一套程序语言。事实上，你可写出“程序”（或是“shell script”）来完成单一命令所不能完成的工作。如同任何好的程序语言，shell也有“变量”、“条件判断”（if-then-else）、循环、输入、输出等元素。单单就shell script这个主题，就足够写出一本书了，所以这里只提供刚好足以入门的基本参考信息。至于完整的说明信息，请参阅`info bash`或“bash shell入门”（*http://www.oreilly.com.tw/chinese/linux/learn_bashshell.html*）。

40.1 空格与换行

`bash` shell script对于空格与换行非常敏感，因为这套程序语言中的“关键字”（keyword）其实是由shell估算的命令，而命令的参数之间必须以空格隔开。类似地，出现于命令中间的换行符（`\n`）会使得shell误认为命令不完整。所以，学习shell script的第一课就是养成正确的语法习惯，以免造成日后不必要的调试麻烦。

40.2 变量

变量（variable）让你用一个名称来代表某种意义的数值或字符串：

```
$ MYNAME="smith "
$ MYAGE=35
$ echo $MYNAME $MYAGE
smith 35
```

存储于变量中的值，其实本质上都是字符串，即使它们全部都是数字。不过，在必要时，shell也能将这种“纯数字字符串”当成数值来处理：

```
$ NUMBER="10"
$ expr $NUMBER + 5 （+符号的左右两侧至少要有一个空格）
15
```

在shell script里表示变量值时，最好以双引号界定，以免造成运行时错误。如果没有双引号，当shell遇到没定义的变量（通常是因为输错变量名称造成的）或是变量值里含有空格时，就可能引发意想不到的后果，造成script的行为错乱。

```
$ FILENAME="My Document"                    含有空格的文件名
$ ls $FILENAME                              列出来试试
ls: My: No such file or directory           糟了！ls 见到两个参数
ls: Document: No such file or directory
$ ls -l "$FILENAME"                         这样才对
My Document                                 ls 只见到一个参数
```

如果变量名称与另一个字符串紧接在一起，则必须以一对花括号界定，以免发生意料外的情况：

```
$ HAT="fedora"
$ echo "The plural of $HAT is $HATs"
The plural of fedora is                 糟了！没“ HATs ”这个变量
$ echo "The plural of $HAT is ${HAT}s"
The plural of fedora is fedoras         这才是我们要的结果
```

40.3 输入与输出

Script的输出主要是由 echo 与 printf 命令提供。161 页的“ 34. 屏幕输出 ”已介绍过这两个命令。

```
$ echo "Hello world"
Hello world
$ printf "I am %d years old\n" `expr 20 + 20`
I am 40 years old
```

shell script 的输入主要是靠 read 命令来取得，它每次从 stdin 读入一行数据，并将其存入一个变量中：

```
$ read name
Sandy Smith
$ echo "I read the name $name"
I read the name Sandy Smith
```

40.4 逻辑值与返回值

程序的精华在于“ 条件判断 ”与“ 循环 ”，而这其中的关键就在于“ 逻辑测试 ”（Boolean test），也就是分辨“ 真 ”（true）与“ 假 ”（false）。对于shell，数值 0 代表“ 真 ”或“ 成功 ”，除此之外的其他数值一律视为“ 假 ”或“ 失败 ”。

此外，任何Linux命令结束时，都会返回一个代表运行结果的整数值给shell，

此值称为“返回值”（return value）或“结束状态”（exit status）。你可用特殊变量 $? 来表示返回值：

```
$ cat myfile
My name is Sandy Smith and
I really like Fedora Linux
$ grep Smith myfile
My name is Sandy Smith and          找到一处相同
$ echo $?
0                                   所以结束状态为“成功”
$ grep aardvark myfile
$ echo $?                           没找到
1                                   所以结束状态为“失败”。
```

许多Linux命令的返回值具有特殊含义，各命令的manpage里通常会说明返回值的真正意义。唯一可以确定的是，返回值0一定是代表成功，因为这是所有 Linux 命令的共识，同时也是 POSIX 标准的规定。

test 和“[”

对于只涉及数值和字符串的逻辑表达式，可用 bash shell 内置的 `test` 命令来计算其逻辑值。如果计算结果为“真”，则 `test` 返回 0；否则返回 1：

```
$ test 10 -lt 5          10 < 5 ？
$ echo $?
1                        当然不
$ test -n "hello"        “hello”字符串的长度不为0吗？
$ echo $?
0                        没错，长度不是0
```

表 12 列出 `test` 常见的参数，它们可用于检查整数、字符串、文件的性质。

`test` 有一个不寻常的别名 `[`（左方括号），以便用于条件判断与循环中。当你使用这种写法时，必须在测试语句的末端补一个“]”符号（右方括号）。下列各项测试与前例是完全等效的：

```
$ [ 10 -lt 5 ]
$ echo $?
1
$ [ -n "hello" ]
$ echo $?
0
```

切记，“ [”是一个命令，它和任何其他命令一样，在命令名称与各个参数之间至少要保持一个空格的间隔，如果你疏忽了，将发生奇怪的事：

```
$ [ 5 -lt 4]                    在“ 4 ”和“ ] ”之间没有空格
bash: [: missing ']'
```

在此例中，`test` 认为它的最后一个参数是“ 4] ”字符串，所以向你抱怨它找不到结尾的“] ”符号。

表 12：test 命令的常用参数

文件测试	
`-d` *name*	测试 *name* 是否为一个目录
`-f` *name*	测试 *name* 是否为普通文件
`-L` *name*	测试 *name* 是否为符号链接
`-r` *name*	测试 *name* 文件是否存在且为可读
`-w` *name*	测试 *name* 文件是否存在且为可写
`-x` *name*	测试 *name* 文件是否存在且为可执行
`-s` *name*	测试 *name* 文件是否存在且其长度不为 0
f1 `-nt` *f2*	测试 *f1* 是否比 *f2* 更新
f1 `-ot` *f2*	测试 *f1* 是否比 *f2* 更旧
字符串测试	
s1 `=` *s2*	测试两个字符串的内容是否完全一样
s1 `!=` *s2*	测试两个字符串的内容是否有差异
`-z` *s1*	测试 *s1* 字符串的长度是否为 0
`-n` *s1*	测试 *s1* 字符串的长度是否不为 0
整数测试	
a `-eq` *b*	测试 *a* 与 *b* 是否相等
a `-ne` *b*	测试 *a* 与 *b* 是否不相等
a `-gt` *b*	测试 *a* 是否大于 *b*
a `-ge` *b*	测试 *a* 是否大于等于 *b*
a `-lt` *b*	测试 *a* 是否小于 *b*
a `-le` *b*	测试 *a* 是否小于等于 *b*

组合与否定测试	
`t1 -a t2`	AND（交集）：当 t1 与 t2 条件同时成立时，才算成立
`t1 -o t2`	OR（并集）：只要 t1 或 t2 任一条件成立，就算成立
`! your_test`	否定测试：当 your_test 失败时，则条件成立
`\( your_test \)`	改变运算顺序（与代数一样）

true 与 false

bash 内置两个与逻辑值有关的命令：true 与 false，它们唯一的作用是分别返回 0 与 1 结束状态：

```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

这两个命令主要是用于条件判断与循环中。

40.5 条件判断

if 语句依据条件判断的结果选择执行路径。条件可能是简单或复杂的命令。最简单的 if 语句形式是 if-then：

```
if command    若 command 的结束状态为 0
then
  body        条件成立时
fi
```

范例：

```
if [ `whoami` = "root" ]
then
  echo "You are the superuser"
fi
```

另一种形式是 if-then-else 语句：

```
if command
then
  body1       条件成立时
else
```

```
  body2       条件不成立时
fi
```

范例：

```
if [ `whoami` = "root" ]
then
  echo "You are the superuser"
else
  echo "You are an ordinary dude"
fi
```

最复杂的形式是 `if-then-elif-else`，这可让你判断许多条件：

```
if command1
then
  body1       当 command1 成立时
elif command2
then
  body2       当 command2 成立时
elif ...
...
else
  bodyN       当所有条件都不成立时
fi
```

范例：

```
if [ `whoami` = "root" ]
then
  echo "You are the superuser"
elif [ "$USER" = "root" ]
then
  echo "You might be the superuser"
elif [ "$bribe" -gt 10000 ]
then
  echo "You can pay to be the superuser"
else
  echo "You are still an ordinary dude"
fi
```

当需要判断的条件太多，但是受测对象都一样时，`if-then-elif-else` 就显得繁琐。这时候 `case` 语句是比较好的选择，它依据单一判断条件的结果，选择最适合执行的命令分支：

```
echo 'What would you like to do?'
read answer
```

```
case "$answer" in  受测对象为 answer 变量
eat)
  echo "OK, have a hamburger"
  ;;
sleep)
  echo "Good night then"
  ;;
*)
  echo "I'm not sure what you want to do"
  echo "I guess I'll see you tomorrow"
  ;;
esac
```

case 语句的标准语法是：

```
case string in
  expr1)
    body1
    ;;
  expr2)
    body2
    ;;
  ...
  exprN)
    bodyN
    ;;
  *)
    bodyelse
    ;;
esac
```

其中的*string*可以是任何值，但通常是类似$myvar这样的变量值；*expr1*、*expr2*… *exprN*是测试结果的模式（细节请参阅 info bash reserved case）；而最后的“ * ”代表前述模式都不匹配时，应选择的命令分支，它相当于 if 语句中最后的 else。每一组命令集合都必须以;;结束，就像下面这样：

```
case $letter in
  X)
    echo "$letter is an X"
    ;;
  [aeiou])
    echo "$letter is a vowel"
    ;;
  [0-9])
    echo "$letter is a digit, silly"
```

```
    ;;
  *)
  echo "I cannot handle  that"
  ;;
esac
```

40.6 循环

while循环由一个判断条件与一组命令构成, 只要判断条件持续成立, 就重复运行循环体内的命令。

```
while command        当command的结束状态为0时
do
   body
done
```

举例来说，若 myscript script 的内容是：

```
i=0
while [ $i -lt 3 ]
do
  echo "again"
  i=`expr $i + 1`
done
```

运行结果：

```
$ ./myscript
0
1
2
```

通常, while循环的主体应该包含能够改变判断条件的命令, 否则会造成死循环。

until循环也是由一个判断条件与一组命令构成，但是它与while循环相反：until循环会重复运行那一组命令，直到判断条件成立为止：

```
until command        当command的结束状态不是0
do
  body
done
```

范例：

```
i=0
until [ $i -gt 3 ]
do
  echo "again"
  i=`expr $i + 1`
done
```

运行结果：

```
$ ./myscript
0
1
2
```

for 循环由一个变量、一组数据（变量值）与一组命令构成，数据值会被依次代入变量，然后运行一次循环主体，直到所有数据值都被处理过为止。

```
for variable in list
do
  body
done
```

范例：

```
for name in Tom Jack Harry
do
  echo "$name is my friend"
done
```

运行结果：

```
$ ./myscript
Tom is my friend
Jack is my friend
Harry is my friend
```

for 循环特别适用于处理一系列文件。例如，当前工作目录下的特定类型文件：

```
for file in *.doc
do
  echo "$file is a stinky Microsoft Word file"
done
```

某些情况下，你或许会需要无穷循环。while 与 until 都有无穷循环的效果，你只要提供一个永远成立（或永远不成立）的条件即可：

```
while true
do
  echo "forever"
done
```

```
until false
do
  echo "forever again"
done
```

通常，你会想在无穷循环内放一个判断条件，并以 `break` 或 `exit` 来结束循环。真正的“无穷”循环其实很少见。

40.7 break 与 continue

`break` 命令可跳出它所在的最内层循环。假设有一个 `myscript`：

```
for name in Tom Jack Harry
do
  echo $name
  echo "again"
done
echo "all done"
```

它的运行结果原本是：

```
$ ./myscript
Tom
again
Jack
again
Harry
again
all done
```

现在加上 break：

```
for name in Tom Jack Harry
do
  echo $name
  if [ "$name" = "Jack" ]
  then
    break
  fi
  echo "again"
done
echo "all done"
```

看看会发生什么事：

```
$ ./myscript
Tom
again
Jack          发生了break
all done
```

continue迫使循环立刻跳过本回合未完成的部分，直接进入下一回合。同样以先前的myscript为例：

```
for name in Tom Jack Harry
do
  echo $name
  if [ "$name" = "Jack" ]
  then
    continue
  fi
  echo "again"
done
echo "all done"
```

看看会怎样：

```
$ ./myscript
Tom
again
Jack          发生continue
Harry
again
all done
```

break和continue都可以接受一个数值参数（break *N*、continue *N*）：对于break，*N*代表要跳出多少层循环；对于continue，则代表要略过多少回合。不过，实际中很少这样做，因为那会导致你的script混乱，所以我们也建议你最好尽量避免使用。

40.8 shell script的制作与运行

shell script本质上只是普通文本文件，凡是可以在bash提示符后输入的命令，都可以出现在script文件里。要运行script，你有三种选择：

标准方法

将下列文本加到script文件的顶端（第一行靠左对齐）：

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

然后改变文件访问模式，使其成为可执行文件：

```
$ chmod +x myscript
```

为了方便，你可将写好的script放在搜索路径中（非必要步骤）。习惯上，个人写的script是放在*~/bin*目录下；若也要给其他用户使用，则是放在*/usr/local/bin*目录下。放在搜索路径中的script，可被当成普通命令来运行：

```
$ myscript
```

若script不是放在搜索路径中而是位于工作目录下，而且搜索路径中也没包含“.”（工作目录）（注20），则必须在script名称之前加上“./”，shell才能找到你的script：

```
$ ./myscript
```

以subshell运行

`bash`会将它的参数视为script文件的名称，并予以运行：

```
$ bash myscript
```

请注意，由于script是在subshell的环境里运行的，所以，script对于环境所做的任何改变（设定`shell`变量、改变工作目录等）仅止于subshell，而不影响login shell。

以login shell运行

对于会影响shell环境的script，应该交给当前的shell去运行：

```
$ . myscript
```

应该采用哪种方法，取决于script本身的性质。一般而言，工具性的script应该运行*#!/bin/bash*命令来保护好。至于为了应付临时工作而写的一次性script，那就要根据是否影响shell环境来决定了。

40.9 命令行参数

shell script也都能够接受命令行参数，就像其他Linux命令一样（事实上，

注20： 将工作目录纳入搜索路径确实很方便，但是基于安全考虑，Fedora和许多Linux distribution都没有这么做。

有许多Linux命令本身其实就是script。)。bash shell提供了一系列特殊变量，让你能够在 script 里处理参数。

首先，含有所有参数的特殊变量是`$@`，而`$1`、`$2`、`$3`等则代表个别参数：

```
$ cat myscript
#!/bin/bash
echo "My name is $1 and I come from $2"
echo "Your info : $@"
$ ./myscript Johnson Wisconsin
My name is Johnson and I come from Wisconsin
Your info : Johnson Wisconsin
```

很显然，script无法预先知道用户使用给几个参数。为此，bash 提供了另一个特殊变量 `$#` 来代表参数个数：

```
if [ $# -lt 2 ]
then
  echo "$0 error: you must supply two arguments"
else
  echo "My name is $1 and I come from $2"
fi
```

特殊变量`$0`代表script自己的名称。当script需要显示自己的用法或错误信息时，这个变量就可以派上用场：

```
$ ./myscript Bob
./myscript error: you must supply two args
```

用一个简单的`for`循环搭配`$@`特殊变量，就可以逐一处理每一个参数，不管实际有多少个：

```
for arg in $@
do
  echo "I found the argument $arg"
done
```

40.10 返回结束状态

`exit` 命令可用于结束 script，并返回指定的状态码给 shell。传统上，状态码0代表成功, 1（或任何非零值）代表失败。若script结束之前没调用`exit`，则 shell 会自动假设状态码为 0。

```
if [ $# -lt 2 ]
then
```

```
  echo "Error: you must supply two args"
  exit 1
else
  echo "My name is $1 and I come from $2"
fi
exit 0

$ ./myscript Bob
./myscript error: you must supply two args
$ echo $?
1
```

40.11 除了 shell Scripting 之外

shell script 的用途很广泛，但毕竟不是万能的。所以，Linux 中还有许多更强的脚本语言与编译语言。

语言	程序	信息来源
Perl	`perl`	*http://www.perl.com/*
Python	`python`	*http://www.python.org/*
C/C++	`gcc`	*http://www.gnu.org/software/gcc/*
Java	`javac`、`java`[注]	*http://java.sun.com/*
FORTRAN	`g77`	*http://www.gnu.org/software/fortran/fortran.html*
Ada	`gnat`	*http://www.gnu.org/software/gnat/gnat.html*

注： 必须另外安装，Fedora 与大多数 Linux distribution 都没随附。

后记

虽然本手册已经涵盖了 Linux 的许多命令与功能，但是这些都只是皮毛而已，Fedora与其他Linux包所提供的程序超过上千个！我们鼓励你自己去探索，持续学习 Linux 系统所带来的强大功能。祝好运！

致谢

衷心感谢我的编辑MikeLoukides，他是O'Reilly编辑群中的传奇人物。我也要感谢技术校阅人员，他们是 Ron Bellomo、Wesley Crossman、David

Debonnaire、Tim Greer、Jacob Heider和Eric van Oorschot, 以及VistaPrint的 Alex Schowtka与Robert Dulaney。最后，感激我的家庭成员Lisa和Sophie。