# Shell编程入门

## 什么是shell

Shell本身是一个用C语言编写的程序，它是用户使用Linux的桥梁。Shell既是一种命令语言，又是一种程序设计语言。作为命令语言，它交互式地解释和执行用户输入的命令；作为程序设计语言，它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支。它虽然不是Linux系统核心的一部分，但它调用了系统核心的大部分功能来执行程序、建立文件并以并行的方式协调各个程序的运行。因此，对于用户来说，shell是最重要的实用程序，深入了解和熟练掌握shell的特性极其使用方法，是用好Linux系统的关键。可以说，shell使用的熟练程度反映了用户对Linux使用的熟练程度。

那么，什么是shell呢？

当一个用户登录Linux系统之后，系统初始化程序init（Linux系统启动的第一个进程）就为每一个用户运行一个称为shell（外壳）的程序。那么，shell是什么呢？确切一点说，shell就是一个命令行解释器，它为用户提供了一个向Linux内核发送请求以便运行程序的界面系统级程序，用户可以用shell来启动、挂起、停止甚至是编写一些程序。

当用户使用Linux时是通过命令来完成所需工作的。一个命令就是用户和shell之间对话的一个基本单位，它是由多个字符组成并以换行结束的字符串。shell解释用户输入的命令，就象DOS里的command.com所做的一样，所不同的是，在DOS中，command.com只有一个，而在Linux下比较流行的shell有好几个，每个shell都各有千秋。一般的Linux系统都将bash作为默认的shell。

## 几种流行的shell对比

目前流行的shell有ash、bash、ksh、csh、zsh等，你可以用下面的命令来查看你自己的shell类型：

　　#echo $SHELL

$SHELL是一个环境变量，它记录用户所使用的shell类型。你可以用命令：

　　#shell-name

来转换到别的shell，这里shell-name是你想要尝试使用的shell的名称，如ash等。这个命令为用户又启动了一个shell，这个shell在最初登录的那个shell之后，称为下级的shell或子shell。使用命令：

　　$exit

可以退出这个子shell。

使用不同的shell的原因在于它们各自都有自己的特点，下面作一个简单的介绍：

### ash

ash shell是由Kenneth Almquist编写的，Linux中占用系统资源最少的一个小shell，它只包含24个内部命令，因而使用起来很不方便。

### bash

bash是Linux系统默认使用的shell，它由Brian Fox和Chet Ramey共同完成，是BourneAgain Shell的缩写，内部命令一共有40个。Linux使用它作为默认的shell是因为它有诸如以下的特色：

可以使用类似DOS下面的doskey的功能，用方向键查阅和快速输入并修改命令。

自动通过查找匹配的方式给出以某字符串开头的命令。

包含了自身的帮助功能，你只要在提示符下面键入help就可以得到相关的帮助。

### ksh

ksh是Korn shell的缩写，由Eric Gisin编写，共有42条内部命令。该shell最大的优点是几乎和商业发行版的ksh完全兼容，这样就可以在不用花钱购买商业版本的情况下尝试商业版本的性能了。

### csh

csh是Linux比较大的内核，它由以William Joy为代表的共计47位作者编成，共有52个内部命令。该shell其实是指向/bin/tcsh这样的一个shell，也就是说，csh其实就是tcsh。

### zch

zch是Linux最大的shell之一，由Paul Falstad完成，共有84个内部命令。如果只是一般的用途，是没有必要安装这样的shell的。

## Shell脚本编程语言与编译型语言的差异

许多中型、大型的程序都是用编译型语言写成，例如Fortran、Ada、Pascal、C、C++或Java。这类程序只要从源代码(source code)转换成目标代码(object code)，便能直接通过计算机来执行（注1）。

编译型语言的好处是高效，缺点则是：它们多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象。例如，在C++里，就很难进行“将一个目录里所有的文件复制到另一个目录中”之类的简单操作。

脚本编程语言通常是解释型(interpreted)的。这类程序的执行，是由解释器(interpreter)读入程序代码，并将其转换成内部的形式，再执行（注2）。请注意，解释器本身是一般的编译型程序。

## 为什么要使用Shell脚本

使用脚本编程语言的好处是，它们多半运行在比编译型语言还高的层级，能够轻易处理文件与目录之类的对象。缺点是：它们的效率通常不如编译型语言。不过权衡之下，通常使用脚本编程还是值得的：花一个小时写成的简单脚本，同样的功能用C或C++来编写实现，可能需要两天，而且一般来说，脚本执行的速度已经够快了，快到足以让人忽略它性能上的问题。脚本编程语言的例子有awk、Perl、Python、Ruby与Shell。

因为Shell似乎是各UNIX系统之间通用的功能，并且经过了POSIX的标准化。因此，Shell脚本只要“用心写”一次，即可应用到很多系统上。因此，之所以要使用Shell脚本是基于：

* 简单性：Shell是一个高级语言；通过它，你可以简洁地表达复杂的操作。
* 可移植性：使用POSIX所定义的功能，可以做到脚本无须修改就可在不同的系统上执行。
* 开发容易：可以在短时间内完成一个功能强大又妤用的脚本。

## 如何运行shell命令？第一行为什么是# !/bin/sh？

我们可以使用任意一种文字编辑器，比如gedit、kedit、emacs、vi(这个最常用)等来编写shell脚本。因为shell程序是解释执行的，所以不需要编译装配成目标程序，按照shell编程的惯例，以bash为例，程序的第一行一般为

“# !/bin/bash”，其中#

表示该行是注释，叹号“！”告诉shell运行叹号之后的命令并用文件的其余部分作为输入，也就是运行/bin/bash并让/bin/bash去执行shell程序的内容。

执行shell程序的方法有三种：

1、sh shell程序文件名

这种方法的命令格式为：

bash shell程序文件名

这实际上是调用一个新的bash命令解释程序，而把shell程序文件名作为参数传递给它。新启动的shell将去读指定的文件，执行文件中列出的命令，当所有的命令都执行完结束。该方法的优点是可以利用shell调试功能。

2、sh<shell程序文件名

格式为：

bash<shell 程序文件名

这种方式就是利用输入重定向，使shell命令解释程序的输入取自指定的程序文件。

3、用chmod命令使shell程序成为可执行的

一个文件能否运行取决于该文件的内容本身可执行且该文件具有执行权。对于shell程序，当用编辑器生成一个文件时，系统赋予的许可权限都是644(rw-r-r--)，因此，当用户需要运行这个文件时，只需要直接键入文件名即可。

在这三种运行shell程序的方法中，最好按下面的方式选择：当刚建立一个shell程序，对它的正确性还没有把握时，应当使用第一种方式进行调试。当一个shell程序已经调试好时，应使用第三种方式把它固定下来，以后只要键入相应的文件名即可，并可被另一个程序所调用。

## shell变量基础—shell自定义变量

### Shell定义变量需要遵循的规则

Shell编程中，使用变量无需事先声明，同时变量名的命名须遵循如下规则：

* 首个字符必须为字母（a-z，A-Z）
* 中间不能有空格，可以使用下划线（\_）
* 不能使用标点符号
* 不能使用bash里的关键字（可用help命令查看保留关键字）

### shell变量赋值

需要给变量赋值时，可以这么写：

变量名=值

### shell变量的引用

要取用一个变量的值，只需在变量名前面加一个$ （ 注意: 给变量赋值的时候，不能在"="两边留空格 ）。例如：

#!/bin/sh

# 对变量赋值：

a="hello world" #等号两边均不能有空格存在

# 打印变量a的值：

echo "A is:" $a

注：上边的#号为注释，将在后边讲到。

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混淆，比如：

num=2

echo "this is the $numnd"

上述脚本并不会输出"this is the 2nd"而是"this is the "；这是由于shell会去搜索变量numnd的值，而实际上这个变量此时并没有值。这时，我们可以用花括号来告诉shell要打印的是num变量：

num=2

echo "this is the ${num}nd"

其输出结果为：this is the 2nd

注意：shell的默认赋值是字符串赋值。比如：

var=1

var=$var+1

echo $var

打印出来的不是2而是1＋1。为了达到我们想要的效果有以下几种表达方式：

let "var+=1"

var=$[$var + 1]

var=`expr $var + 1`#注意加号两边的空格，否则还是按照字符串的方式赋值。

注意：前两种方式在bash下有效，在sh下会出错。

let表示数学运算，expr用于整数值运算，每一项用空格隔开，$[]将中括号内的表达式作为数学运算先计算结果再输出。

Shell脚本中有许多变量是系统自动设定的，我们将在用到这些变量时再作说明。除了只在脚本内有效的普通shell变量外，还有环境变量，即那些由export关键字处理过的变量。本文不讨论环境变量，因为它们一般只在登录脚本中用到。

## shell环境变量详解：shell如何设置环境变量

shell在开始执行时就已经定义了一些和系统的工作环境有关的变量，这些变量用户还可以重新定义，常用的shell环境变量有：

* HOME：用于保存注册目录的完全路径名。
* PATH：用于保存用冒号分隔的目录路径名，shell将按PATH变量中给出的顺序搜索这些目录，找到的第一个与命令名称一致的可执行文件将被执行。
* TERM：终端的类型。
* UID：当前用户的标识符，取值是由数字构成的字符串。
* PWD：当前工作目录的绝对路径名，该变量的取值随cd命令的使用而变化。
* PS1：主提示符，在特权用户下，缺省的主提示符是“#”，在普通用户下，缺省的主提示符是“$”。
* PS2：在shell接收用户输入命令的过程中，如果用户在输入行的末尾输入“\”然后回车，或者当用户按回车键时shell判断出用户输入的命令没有结束时，显示这个辅助提示符，提示用户继续输入命令的其余部分，缺省的辅助提示符是“>”。

## shell位置参数应用详解|shell内部参数详解

### 一、位置参数

位置参数是一种在调用shell程序的命令行中按照各自的位置决定的变量，是在程序名之后输入的参数。位置参数之间用空格分隔，shell取第一个位置参数替换程序文件中的，第二个替换，依次类推。是一个特殊的变量，它的内容是当前这个shell程序的文件名，所以，不是一个位置参数，在显示当前所有的位置参数时是不包括的。

位置参数的值可以用$N得到，N是一个数字，如果为1，即$1.类似C语言中的数组，Linux会把输入的命令字符串分 段并给每段进行标号，标号从0开始。第0号为程序名字，从1开始就表示传递给程序的参数。如$0表示程序的名字，$1表示传递给程序的第一个参数，以此类 推。 After $9, 在位置参数$9之后的参数必须用括号括起来，例如：${10}, ${11}, ${12}。

特殊变量$\*和$@ 表示所有的位置参数。

## 二、内部参数

上述过程中的$0是一个内部变量，它是必须的，而$1则可有可无。和$0一样的内部变量还有以下几个：

* $# ----传递给程序的总的参数数目
* $? ----上一个代码或者shell程序在shell中退出的情况，如果正常退出则返回0，反之为非0值。
* $\* ----传递给程序的所有参数组成的字符串。
* $n ----表示第几个参数，$1 表示第一个参数，$2 表示第二个参数 ...
* $0 ----当前程序的名称
* $@----以"参数1" "参数2" ... 形式保存所有参数
* $$ ----本程序的(进程ID号)PID
* $! ----上一个命令的PID

列子1：

if [ -n "$1" ] #如果$1即参数1不为空，又即存在参数1 # 被测试的变量被双引号引起

then

echo "Parameter #1 is $1" # 使用引号来使#被转义

fi

if [ -z "$1" ] #如果$1即参数1为空（零），又即不存在参数1

then

fi

## shell参数置换的变量

shell提供了参数置换能力以便用户可以根据不同的条件来给变量赋不同的值。参数置换的变量有四种，这些变量通常与某一个位置参数相联系，根据指定的位置参数是否已经设置类决定变量的取值，它们的语法和功能分别如下：

* 缺省值置换

变量=${参数-word}：如果设置了参数，则用参数的值置换变量的值，否则用word置换。即这种变量的值等于某一个参数的值，如果该参数没有设置，则变量就等于word的值

* 缺省值赋值

变量=${参数=word}：如果设置了参数，则用参数的值置换变量的值，否则把变量设置成word，然后再用word替换参数的值。注意，位置参数不能用于这种方式，因为在Shell程序中不能为位置参数赋值

* 空值错误

变量=${参数？word}：如果设置了参数，则用参数的值置换变量的值，否则就显示word并从Shell中退出，如果省略了word，则显示标准信 息。这种变量要求一定等于某一个参数的值。如果该参数没有设置，就显示一个信息，然后退出，因此这种方式常用于出错指示

* 有值置换

变量=${参数+word}：如果设置了参数，则用word置换变量，否则不进行置换

## Echo命令用法：

bash命令解释程序包含了一些内部命令。内部命令在目录列表时是看不见的，它们由shell本身提供。echo是其中之一。

命令格式：echo arg

功能：在屏幕上打印出由arg指定的字符串。

另外，还可以用echo实现更复杂的输出格式控制：

1、显示转义字符

echo "\"It is a test\""

结果将是: "It is a test"

同样，双引号也可以省略

2、显示变量

read name (输入OK)

echo "$name It is a test"

结果将是: OK It is a test

同样双引号也可以省略

如果变量与其它字符相连的话，需做如下处理:

read mouth (输入8)

echo "${mouth}-1-2009"

结果将是: 8-1-2009

3、显示换行

echo "OK!\n"

echo "It it a test"

4、显示不换行

echo "OK!\c"

echo "It is a test"

5、显示结果定向至文件

echo "It is a test" > myfile

6、原样输出字符串，不进行转义或取变量(用单引号)

echo '$name\"'

7、显示命令执行结果

echo `date`

结果将显示当前日期

从上面可看出，双引号可有可无；单引号主要用在原样输出中。

## shell printf命令

printf命令：格式化输出命令， 是echo命令的增强版。它是C语言printf()库函数的一个有限的变形，并且在语法上有些不同。

注意：使用printf的脚本比使用echo移植性好。

如同echo命令，printf命令可以输出简单的字符串：

[root@master ~]#printf "Hello, Shell\n"

Hello, Shell

[root@master ~]#

应该可以马上发现，最大的不同在于：printf不像echo那样会自动提供一个换行符号。你必须显式地将换行符号指定成\n。

printf命令的完整语法有两个部分：

printf的语法格式：printf format-string [arguments...]

第一部分为描述格式规格的字符串，用来描述输出的排列方式，最好为此字符串加上引号。

此字符串包含按字面显示的字符以及格式声明，格式声明时特殊的占位符，用来描述如何显示相应的参数。

第二部分是与格式声明相对应的参数列表，例如一系列的字符串或变量值。格式声明由两部分组成：

百分比符号（%）和指示符。最常用的格式指示符有两个，%s用于字符串，而%d用于十进制整数。

格式字符串中，一般字符会按字面显示。转义序列则像echo那样，解释后再输出成相应的字符。格式声明以%符号开头，并以定义的字母集中的一个来结束，用来控制相应参数的输出。例如%s用于字符串的输出：

[root@master ~]# printf "The first program always prints'%s,%d\n'" Hello Shell

输出结果为：

-bash: printf: Shell: invalid number

The first program always prints 'Hello,0

'[root@master ~]#

当尝试以%d的格式来显示字符串Shell时，会发现有警告，提示Shell为无效的数字，此时会打印出默认值0;从这个试验中可以看出来：%s, %s两侧的单引号''并不是必须的。

printf的转义序列：

* \a 警告字符，通常为ASCII的BEL字符
* \b 后退
* \c 抑制（不显示）输出结果中任何结尾的换行字符（只在%b格式指示符控制下的参数字符串中有效），而且，任何留在参数里的字符、任何接下来的参数以及任何留在格式字符串中的字符，都被忽略
* \f 换页（formfeed）
* \n 换行
* \r 回车（Carriage return）
* \t 水平制表符
* \v 垂直制表符
* \\ 一个字面上的反斜杠字符
* \ddd 表示1到3位数八进制值的字符。仅在格式字符串中有效
* \0ddd 表示1到3位的八进制值字符

默认情况下，转义序列只在格式字符串中会被特别对待，也就是说，如果转义序列出现在参数列表的字符串中，将不会被解释：

[root@master ~]#printf "a string, no processing:<%s>\n" "A\nB"

当你使用%b格式指示符时，printf会解释参数字符串里的转义序列：

[root@master ~]#printf "a string, no processing:<%b>\n" "A\nB"

无论时在格式字符串内还是在使用%b所打印的参数字符串里，大部分的转义序列都是被相同对待。无论如何，\c与\0ddd只有搭配%b使用才有效，而\ddd只有在格式字符串里才会被解释。

printf格式指示符

* %b 相对应的参数被视为含有要被处理的转义序列之字符串。
* %c ASCII字符。显示相对应参数的第一个字符
* %d, %i 十进制整数
* %e 浮点格式
* %E 浮点格式
* %f 浮点格式
* %g %e或%f转换，看哪一个较短，则删除结尾的零
* %G %E或%f转换，看哪一个较短，则删除结尾的零
* %o 不带正负号的八进制值
* %s 字符串
* %u 不带正负号的十进制值
* %x 不带正负号的十六进制值，使用a至f表示10至15
* %X 不带正负号的十六进制值，使用A至F表示10至15
* %% 字面意义的%

根据POSIX标准：浮点格式%e、%E、%f、%g与%G是“不需要被支持”。这是因为awk支持浮点预算，且有它自己的printf语句。这样Shell程序中需要将浮点数值进行格式化的打印时，可使用小型的awk程序实现。然而，内建于bash、ksh93和zsh中的printf命令都支持浮点格式。

数值精度的定义精度的意义：

* %d,%i,%o,%u,%x,%X 要打印的最小位数。当值的位数小于此数字时，会在前面补零。默认精度（precision）为1
* %e，%E 要打印的最小位数。当值的位数小于此数字时，会在小数点后面补零，默认精度为6.精度为0时则表示不显示小数点
* %f 小数点右边的位数
* %g，%G 有效位数（significant digit）的最大数目
* %s 要打印字符的最大数目

## Shell test语句

test命令用于检查某个条件是否成立，它可以进行数值、字符和文件三个方面的测试，其测试符和相应的功能分别如下：

一、数值测试：

* -eq：等于则为真
* -ne：不等于则为真
* -gt：大于则为真
* -ge：大于等于则为真
* -lt：小于则为真
* -le：小于等于则为真

二、字符串测试：

* =：等于则为真
* !=：不相等则为真
* -z字符串：字符串长度伪则为真
* -n字符串：字符串长度不伪则为真

三、文件测试：

* -e文件名：如果文件存在则为真
* -r文件名：如果文件存在且可读则为真
* -w文件名：如果文件存在且可写则为真
* -x文件名：如果文件存在且可执行则为真
* -s文件名：如果文件存在且至少有一个字符则为真
* -d文件名：如果文件存在且为目录则为真
* -f文件名：如果文件存在且为普通文件则为真
* -c文件名：如果文件存在且为字符型特殊文件则为真
* -b文件名：如果文件存在且为块特殊文件则为真

另外，Linux还提供了与(“！”)、或(“-o)、非(“-a”)三个逻辑操作符用于将测试条件连接起来，其优先级为：“！”最高，“-a”次之，“-o”最低。

同时，bash也能完成简单的算术运算，格式如下：

$[expression]

例如：

var1=2

var2=$[var1\*10+1]

则：

var2的值为21。

## 条件语句

### if语句

shell程序中的条件分支是通过if条件语句来实现的，其一般格式为：

if 条件命令串

　then

　　条件为真时的命令串

　else

　　条件为假时的命令串

　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，要确保方括号前后的空格。

[ -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

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。下面是一个简单的if语句：

#!/bin/sh

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。

### case语句

if条件语句用于在两个选项中选定一项，而case条件选择为用户提供了根据字符串或变量的值从多个选项中选择一项的方法，其格式如下：

　　case string in

　　exp-1)

　　若干个命令行1

　　；；

　　exp-2)

　　若干个命令行2

　　；；

　　……

　　\*)

前面介绍过，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/sh

ftype=`file "$1"` # Note ' and ` is different

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语句

select表达式是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：

select var in ... ; do

break;

done

.... now $var can be used ....

下面是一个简单的示例：

#!/bin/sh

echo "What is your favourite OS?"

select var in "Linux" "Gnu Hurd" "Free BSD" "Other"; do

break;

done

echo "You have selected $var"

如果以上脚本运行出现 select ：NOT FOUND 将 #!/bin/sh 改为 #!/bin/bash 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

## 循环语句详解

### for语句

for循环对一个变量的可能的值都执行一个命令序列。赋给变量的几个数值既可以在程序内以数值列表的形式提供，也可以在程序以外以位置参数的形式提供。for循环的一般格式为：

　　for 变量名 [in 数值列表]

　　do

　　 若干个命令行

　　done

变量名可以是用户选择的任何字符串，如果变量名是var，则在in之后给出的数值将顺序替换循环命令列表中的$var。如果省略了in，则变量var的取值将是位置参数。对变量的每一个可能的赋值都将执行do和done之间的命令列表。

下面的示例会把A B C分别打印到屏幕上：

#!/bin/sh

for var in A B C ; do

echo "var is $var"

done

下面是一个实用的脚本showargs，其功能是打印一些命令行中所有参数的信息：

#!/bin/sh

for arg in $\*; do

echo $arg

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showargs arg1 arg2 arg3，那么 $\* 就包含有 3 个字符串，即arg1, arg2和 arg3。

### while和until

while 和 until命令都是用命令的返回状态值来控制循环的。

While 循环的一般格式为：

　　while

　　 若干个命令行1

　　do

　　 若干个命令行2

　　done

只要while的“若干个命令行1”中最后一个命令的返回状态为真，while循环就继续执行do...done之间的“若干个命令行2”。

until命令是另一种循环结构，它和while命令相似，其格式如下：

　　until

　　 若干个命令行1

　　do

　　 若干个命令行2

　　done

until循环和while循环的区别在于：while循环在条件为真时继续执行循环，而until则是在条件为假时继续执行循环。

Shell还提供了true和false两条命令用于建立无限循环结构的需要，它们的返回状态分别是总为0或总为非0