shell编程

Shell本身是一个用C语言编写的程序，它是用户使用Unix/Linux的桥梁，用户的大部分工作都是通过Shell完成的。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Unix/Linux系统内核的一部分，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的特性及其使用方法，是用好Unix/Linux系统的关键。

Shell有两种执行命令的方式：

交互式（**Interactive**）：解释执行用户的命令，用户输入一条命令，Shell就解释执行一条。

批处理（**Batch**）：用户事先写一个Shell脚本(Script)，其中有很多条命令，让Shell一次把这些命令执行完，而不必一条一条地敲命令。

Shell脚本和编程语言很相似，也有变量和流程控制语句，但Shell脚本是解释执行的，不需要编译，Shell程序从脚本中一行一行读取并执行这些命令，相当于一个用户把脚本中的命令一行一行敲到Shell提示符下执行。

Shell初学者请注意，在平常应用中，建议不要用 root 帐号运行 Shell 。作为普通用户，不管您有意还是无意，都无法破坏系统；但如果是 root，那就不同了，只要敲几个字母，就可能导致灾难性后果。

Shell需要依赖其他程序才能完成大部分的工作，这或许是它的缺陷，但它不容置疑的长处是：简洁的脚本语言标记方式，而且比C语言编写的程序开发周期要短。

Unix/Linux上常见的Shell脚本解释器有bash、sh、csh、ksh等，习惯上把它们称作一种Shell。我们常说有多少种Shell，其实说的是Shell脚本解释器。bash是Linux标准默认的shell，本教程也基于bash讲解。

bash由Brian Fox和Chet Ramey共同完成，是BourneAgain Shell的缩写。

sh 由Steve Bourne开发，是Bourne Shell的缩写，sh 是Unix 标准默认的shell。

1 Shell脚本语言与编译型语言的差异

大体上，可以将程序设计语言可以分为两类：编译型语言和解释型语言。

编译型语言

很多传统的程序设计语言，例如C、C++和Java，都是编译型语言。这类语言需要预先将我们写好的源代码(source code)转换成目标代码(object code)，这个过程被称作“编译”。

运行程序时，直接读取目标代码(object code)。由于编译后的目标代码(object code)非常接近计算机底层，因此执行效率很高，这是编译型语言的优点。

但是，由于编译型语言多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象，往往实现一个简单的功能需要大量复杂的代码。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。

解释型语言

解释型语言也被称作“脚本语言”。

执行这类程序时，解释器(interpreter)需要读取我们编写的源代码(source code)，并将其转换成目标代码(object code)，再由计算机运行。

因为每次执行程序都多了解释的过程，因此效率有所下降。

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。

2 什么时候用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好用的脚本。

但是，考虑到Shell脚本的命令限制和效率问题，下列情况一般不使用Shell：

1. 资源密集型的任务，尤其在需要考虑效率时（比如，排序，hash等等）。
2. 需要处理大任务的数学操作，尤其是浮点运算，精确运算，或者复杂的算术运算（这种情况一般使用C/C++）。
3. 有跨平台（跨操作系统）移植需求（一般使用C 或Java）。
4. 复杂的应用，在必须使用结构化编程的时候（需要变量的类型检查，函数原型，等等）。
5. 对于影响系统全局性的关键任务应用。
6. 对于安全有很高要求的任务，比如你需要一个健壮的系统来防止入侵、破解、恶意破坏等等。
7. 项目由连串的依赖的各个部分组成。

8. 需要大规模的文件操作。
9. 需要多维数组的支持。
10. 需要数据结构的支持，比如链表或树等数据结构。
11. 需要产生或操作图形化界面 GUI。
12. 需要直接操作系统硬件。
13. 需要 I/O 或socket 接口。
14. 需要使用库或者遗留下来的老代码的接口。
15. 私人的、闭源的应用（shell 脚本把代码就放在文本文件中，全世界都能看到）。

如果你的应用符合上边的任意一条，那么就考虑一下更强大的语言吧——比如C/C++，或者是Java。即使如此，你会发现，使用shell来原型开发你的应用，在开发步骤中也是非常有用的。

3 第一个Shell脚本

打开文本编辑器，新建一个文件，扩展名为sh（sh代表shell），扩展名并不影响脚本执行，见名知意就好。

输入一些代码：

```
#!/bin/bash
echo "Hello World !"
```

“#!” 是一个约定的标记，它告诉系统这个脚本需要什么解释器来执行，即使用哪一种Shell。echo命令用于向窗口输出文本。

3.1 作为可执行程序

将上面的代码保存为test.sh，并 cd 到相应目录：

```
chmod +x ./test.sh  #使脚本具有执行权限
./test.sh  #执行脚本
```

运行其它二进制的程序也一样，直接写test.sh，linux系统会去PATH里寻找有没有叫test.sh的，而只有/bin, /sbin, /usr/bin，/usr/sbin等在PATH里，你的当前目录通常不在PATH里，所以写成test.sh是会找不到命令的，要用./test.sh告诉系统说，就在当前目录找。

3.2 作为解释器参数

这种运行方式是，直接运行解释器，其参数就是shell脚本的文件名，如：

```
/bin/bash test.sh
```

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。

4 shell变量

Shell支持自定义变量。

4.1 定义变量

1. 定义变量时，变量名不加美元符号（$），如：

```
variableName="value"
```

注意，变量名和等号之间不能有空格，这可能和你熟悉的所有编程语言都不一样。

1. 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）。

变量定义举例：

```
myName="wenong"
myNum=100
```

4.2 使用变量

- 使用一个定义过的变量，只要在变量名前面加美元符号$即可，如：

```
myName="wenong"
echo $myName
echo ${myName}
```

- 变量名外面的大括号是可选的，加不加都行，加大括号是为了帮助解释器识别变量的边界，比如下面这种情况：

```
skill="Shell"
echo "I am good at ${skill}Script"
```

如果不给skill变量加花括号，写成`echo "I am good at $skillScript"`当成一个变量（其值为空），代码执行结果就不是我们期望的样子了。

推荐给所有变量加上花括号，这是个好的编程习惯。

4.3 重新定义变量

已定义的变量，可以被重新定义，如：

```
myName="wenong"
echo ${myName}
myName="huang"
echo ${myName}
```

这样写是合法的，但注意，第二次赋值的时候不能写`$myName="huang"`使用变量值的时候才加美元符（$），负责直接使用变量名。

4.4 只读变量

使用 readonly 命令可以将变量定义为只读变量，只读变量的值不能被改变。下面的例子尝试更改只读变量，结果报错：

```
#!/bin/bash
myName="wenong"
readonly myName
myName="huang"
```

运行脚本，结果如下：

```
./test.sh: 行 10: myName: 只读变量
```

4.5 删除变量

使用 unset 命令可以删除变量。语法：

```
unset variable_name
```

变量被删除后不能再次使用；unset 命令不能删除只读变量。举个例子：

```
#!/bin/bash
myName="wenong"
unset myName
echo $myName
```

上面的脚本没有任何输出。

4.6 变量类型

运行shell时，会同时存在两种变量：

1) 局部变量

局部变量在脚本或命令中定义，仅在当前shell实例中有效，其他shell启动的程序不能访问局部变量。

局部变量只在创建它们的shell中可用。

```
where@ubuntu:~$ myName="wenong"
where@ubuntu:~$ echo myName
myName
where@ubuntu:~$
```

2) 环境变量

所有的程序，包括shell启动的程序，都能访问环境变量，有些程序需要环境变量来保证其正常运行。必要的时候shell脚本也可以定义环境变量。可以在创建它们的shell及其派生出来的任意子进程中使用。

```
where@ubuntu:~$ export myName_env="wenong_env"
where@ubuntu:~$ echo $myName_env
wenong_env
where@ubuntu:~$
```

环境变量从父进程复制给子进程。

shell变量中有一部分是环境变量，有一部分是局部变量，这些变量保证了shell的正常运行

test.sh

```
#!/bin/sh
myName_sh="wenong_sh"
export myName_env_sh="wenong_env_sh"
./test2.sh
```

test2.sh

```
echo $myName_sh
echo $myName_env_sh
```

运行结果：

```
where@ubuntu:~$ ./test.sh  #直接在test.sh中执行test2.sh能打印出myName_env_sh变量,不能打印出
myName_sh变量。

wenong_env_sh
```

5 Shell特殊变量

前面已经讲到，变量名只能包含数字、字母和下划线，因为某些包含其他字符的变量有特殊含义，这样的变量被称为特殊变量。

例如，$ 表示当前Shell进程的ID，即pid，看下面的代码：

```
$echo $$
```

运行结果

```
29949
```

变量	含义
$0	当前脚本的文件名
$n	传递给脚本或函数的参数。n 是一个数字，表示第几个参数。例如，第一个参数是$1，第二个参数是$2。
$#	传递给脚本或函数的参数个数。
$*	传递给脚本或函数的所有参数。
$@	传递给脚本或函数的所有参数。被双引号" "包含时，与 $* 稍有不同，下面将会讲到。
$?	上个命令的退出状态，或函数的返回值。
$$	当前Shell进程ID。对于 Shell 脚本，就是这些脚本所在的进程ID。

5.1 命令行参数

运行脚本时传递给脚本的参数称为命令行参数。命令行参数用 $n 表示，例如，$1 表示第一个参数，$2 表示第二个参数，依次类推。

请看下面的脚本：

```
#!/bin/bash
echo "File Name: $0"
echo "Param1: $1"
echo "Param2: $2"
echo "All Params: $@"
echo "All Params: $*"
echo "Param count : $#"
```

运行结果：

```
where@ubuntu:~$ ./test.sh 1 2 3
File Name: ./test.sh
Param1: 1
Param2: 2
All Params: 1 2 3
All Params: 1 2 3
Param count : 3
```

5.2 退出状态

`$?` 可以获取上一个命令的退出状态。所谓退出状态，就是上一个命令执行后的返回结果。退出状态是一个数字，一般情况下，大部分命令执行成功会返回 0，失败返回 1。不过，也有一些命令返回其他值，表示不同类型的错误。下面例子中，命令成功执行：

```
$ls -l
$echo $?
0
$
```

$? 也可以表示函数的返回值，后续将会讲解。

6 shell数组

bash支持一维数组（不支持多维数组），并且没有限定数组的大小。类似与C语言，数组元素的下标由0开始编号。获取数组中的元素要利用下标，下标可以是整数或算术表达式，其值应大于或等于0。

6.1定义数组

在Shell中，用括号来表示数组，数组元素用“空格”符号分割开。定义数组的一般形式为：

array_name=(value1 ... valueN)

例如：

```
array_name=(value0 value1 value2 value3)
```

还可以单独定义数组的各个分量：

```
array_name[0]=value0
array_name[1]=value1
array_name[2]=value2
```

可以不使用连续的下标，而且下标的范围没有限制。

6.2 读取数组

读取数组元素值的一般格式是：

${array_name[index]}

例如：

```
value=${array_name[2]}
```

举个例子：

```
#!/bin/sh
num[0]="0"
num[1]="1"
num[2]="2"
num[3]="3"
num[4]="4"
echo "${num[0]}"
echo "${num[1]}"
```

运行脚本，输出：

```
$./test.sh
0
1
```

使用@ 或 * 可以获取数组中的所有元素，例如：

```
${array_name[*]}
${array_name[@]}
```

举个例子：

```
#!/bin/sh
num[0]="0"
num[1]="1"
num[2]="2"
num[3]="3"
num[4]="4"
echo "First Method: ${num[*]}"
echo "Second Method: ${num[@]}"
```

运行脚本，输出：

```
$./test.sh
First Method: 0 1 2 3 4
Second Method: 0 1 2 3 4
```

6.3 获取数组的长度

获取数组长度的方法与获取字符串长度的方法相同，例如：

```
# 取得数组元素的个数
length=${#array_name[@]}
# 或者
length=${#array_name[*]}
# 取得数组单个元素的长度
lengthn=${#array_name[n]} #n表示数组的具体某个下标
```

7 shell替换

7.1 shell变量替换

如果表达式中包含特殊字符，Shell 将会进行替换。例如，在双引号中使用变量就是一种替换，举个例子：

```
#!/bin/bash
a=10
echo  "Value of a is $a"
```

运行结果：

```
Value of a is 10
```

7.2 命令替换

命令替换是指Shell可以先执行命令，将输出结果暂时保存，在适当的地方输出。命令替换的语法：

```
`command`
```

注意：是反引号，不是单引号，这个键位于 ***Esc*** 键下方。

下面的例子中，将命令执行结果保存在变量中：

```
#!/bin/bash
DATE=`date`
echo "Date is $DATE"
USERS=`who | wc -l`
echo "Number of users are $USERS"
```

运行结果：

```
Date is 2016年 08月 15日 星期一 12:45:09 CST
Number in user are 2
```

7.3 变量替换

变量替换可以根据变量的状态（是否为空、是否定义等）来改变它的值

可以使用的变量替换形式：

形式	说明
${var}	变量本来的值
${var:-word}	如果变量 var 为空或已被删除(unset)，那么返回 word，但不改变 var 的值。
${var:=word}	如果变量 var 为空或已被删除(unset)，那么返回 word，并将 var 的值设置为 word。
${var:+word}	如果变量 var 被定义，那么返回 word，但不改变 var 的值。
${var:?message}	如果变量 var 为空或已被删除(unset)，那么将消息 message 送到标准错误输出，可以用来检测变量 var 是否可以被正常赋值。若此替换出现在Shell脚本中，那么脚本将停止运行。

请看下面的例子：

```
#!/bin/bash

echo "1 ${var:-"hello"}"
echo "1 $var"

echo ${var:="hello"}
echo "2  $var"

echo ${var:+"world"}
echo "3 $var"

unset var
echo ${var:?"error"}
echo "4 ${var}"
```

运行结果：

```
where@ubuntu:~$ ./test.sh
1 hello
1
2 hello
2 hello
3 world
3 hello
./test.sh: 行 19: var: error
```

8 与用户交互

8.1 echo

echo命令的功能是在显示器上显示一段文字，一般起到一个提示的作用。

```
echo [-options] [string]
```

```
-n #不要在最后自动换行
-e #处理转义字符
```

例如：

```
where@ubuntu:~$ echo -n "helloworld"
helloworldwhere@ubuntu:~$
```

再举个例子：

```
where@ubuntu:~$ echo -e "\f\x30"

0
where@ubuntu:~$
```

下面的转义字符都可以用在 echo 中：

转义字符	含义
\	反斜杠
\b	退格（删除键）
\f	换页(FF)，将当前位置移到下页开头
\n	换行
\r	回车
\t	水平制表符（tab键）
\v	垂直制表符

8.2 read

read命令是用于从终端或者文件中读取输入的内部命令，read命令读取整行输入，每行末尾的换行符不被读入。在read命令后面，如果没有指定变量名，读取的数据将被自动赋值给特定的变量REPLY。

1. 从标准输入读取输入并赋值给变量。

```
read [var]
```

例如:

```
where@ubuntu:~$ read var
wenong
where@ubuntu:~$ echo $var
wenong
```

1. 从标准输入读取输入到第一个空格或者回车，将输入的第一个单词放到变量中，第二个单词放第二个变量中，以此类推，剩下的字符留给最后一个变量。

```
read [var1] [var2] ...
```

例如：

```
where@ubuntu:~$ read var1 var2 var3
1 2 3 4 5 6
where@ubuntu:~$ echo $var1
1
where@ubuntu:~$ echo $var2
2
where@ubuntu:~$ echo $var3
3 4 5 6
```

1. 从标准输入读取一行并赋值给特定变量REPLY。

例如：

```
readwhere@ubuntu:~$ read
hello
where@ubuntu:~$ echo $REPLY
hello
where@ubuntu:~$
```

1. 把单词清单读入数组里

```
read -a [arrayname]
```

例如：

```
where@ubuntu:~$ read -a array
1 2 3 4 5
where@ubuntu:~$ echo ${array[2]}
3
```

1. 打印提示，等待输入

```
read -p [info] [var]
```

例如：

```
where@ubuntu:~$ read -p "what is your name?" name
what is your name?wenong
where@ubuntu:~$ echo $name
wenong
```

1. 读超时

```
read -t [timeout] [var]
```

例如：

```
where@ubuntu:~$ read -t 3 var
where@ubuntu:~$                    #3秒后退出read命令
```

1. 读取指定个数字符

```
read -n [size] [var]
```

例如：

```
where@ubuntu:~$ read -n 2 var
dkwhere@ubuntu:~$ echo $var      #输入2个字符后，read命令自动退出。
dk
where@ubuntu:~$
```

1. 自定义结束输入行

```
read -d [char] [var]
```

例如：

```
where@ubuntu:~$ read -d ':' var
huang:where@ubuntu:~$ echo $var  #输入：后read自动退出。
huang
where@ubuntu:~$
```

1. 隐藏输入字符

```
read -s [var]
```

例如：

```
where@ubuntu:~$ read -s var
where@ubuntu:~$ echo $var
wenong
where@ubuntu:~$
```

9 Shell运算符

Bash 支持很多运算符，包括算数运算符、关系运算符、布尔运算符、字符串运算符和文件测试运算符。

9.1 算术运算符

- 也可以使用表达式$(( )) 或 $[]

运算符	说明	举例
+	加法	`$(($a + $b))`
-	减法	`$(($a - $b))`
*	乘法	`$(($a * $b))`
/	除法	`$(($a / $b))`
%	取余	`$(($a % $b))`
=	赋值	a=$b 将把变量 b 的值赋给 a。

```
#!/bin/sh
a=10
b=20
val=$(($a + $b))
echo "a + b : $val"
val=$(($a - $b))
echo "a - b : $val"
val=$(($a * $b))
echo "a * b : $val"
val=$(($a / $b))
echo "a / b : $val"
val=$[$a % $b]
echo "a % b : $val"
```

9.2 关系运算符

关系运算符只支持数字，不支持字符串，除非字符串的值是数字。

运算符	说明	举例
-eq	检测两个数是否相等，相等返回 true。	[$a -eq $b]
-ne	检测两个数是否相等，不相等返回 true。	[$a -ne $b]
-gt	检测左边的数是否大于右边的，如果是，则返回 true。	[$a -gt $b]
-lt	检测左边的数是否小于右边的，如果是，则返回 true。	[$a -lt $b]
-ge	检测左边的数是否大等于右边的，如果是，则返回 true。	[$a -ge $b]
-le	检测左边的数是否小于等于右边的，如果是，则返回 true。	[$a -le $b]
==	用于比较两个数字，相同则返回 true。	[a == b]
!=	用于比较两个数字，不相同则返回 true。	[a != b]

注意：条件表达式要放在方括号之间，并且要有空格，例如 **[expression]** 是错误的，必须写成 **[expression]**。

来看一个关系运算符的例子：

```
#!/bin/sh
a=10
b=20
if [ $a -eq $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi

if [ $a -ne $b ]
then
   echo "a is not equal to b"
else
   echo "a is equal to b"
fi

if [ $a -gt $b ]
then
   echo "a is greater than b"
else
   echo "a is not greater than b"
fi

if [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "a is not less than b"
fi

if [ $a -ge $b ]
then
    echo "a is greater than or  equal to b"
else
    echo "a is not greater than or equal to b"
fi

if [ $a -le $b ]
then
   echo "$a -le $b: a is less than or  equal to b"
else
   echo "$a -le $b: a is not less than or equal to b"
fi
```

运行结果：

```
a is not equal to b
a is not equal to b
a is not greater than b
a is less than b
a is not greater than or equal to b
a is less than or  equal to b
```

9.3 逻辑运算符

运算符	说明	举例
!	非运算，表达式为 true 则返回 false，否则返回 true。	[! false]
-o	逻辑或运算，有一个表达式为 true 则返回 true。	[$a -lt 20 -o $b -gt 100]
-a	逻辑与运算，两个表达式都为 true 才返回 true。	[$a -lt 20 -a $b -gt 100]

```
#!/bin/bash
a=10
b=20
if [ ! $a -lt 100 ]
then
    echo "! $a -lt 100 : return true"
else
    echo  "! $a -lt 100 : return false"
fi
if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : return true"
else
   echo "$a -lt 100 -a $b -gt 15 : return false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : return true"
else
   echo "$a -lt 100 -o $b -gt 100 : return false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : return true"
else
   echo "$a -lt 100 -o $b -gt 100 : return false"
fi
```

运行结果：

```
! 10 -lt 100 : return false
10 -lt 100 -a 20 -gt 15 : return true
10 -lt 100 -o 20 -gt 100 : return true
10 -lt 5 -o 20 -gt 100 : return false
```

9.4 字符串运算符

运算符	说明	举例
=	检测两个字符串是否相等，相等返回 true。	[$a = $b]
!=	检测两个字符串是否相等，不相等返回 true。	[$a != $b]
-z	检测字符串长度是否为0，为0返回 true。	[-z $a]
str	检测字符串是否为空，不为空返回 true。	[$a]

看一个例子：

```
#!/bin/bash

a=$1
b=$2

if [ $a = $b ]
then
   echo "a = b"
else
   echo "a != b"
fi
if [ $a != $b ]
then
   echo "a != b"
else
   echo "a = b"
fi
if [ -z $a ]
then
   echo "a length is zero"
else
   echo "a length is not zero"
fi
if [ $b ]
then
   echo "b is not empty"
else
   echo "b is empty"
fi
```

运行结果：

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
abc : string is not empty
```

9.5 文件测试运算符

文件测试运算符用于检测 linux文件的各种属性。

操作符	说明	举例
-b file	检测文件是否是块设备文件，如果是，则返回 true。	[-b $file]
-c file	检测文件是否是字符设备文件，如果是，则返回 true。	[-b $file]
-d file	检测文件是否是目录，如果是，则返回 true。	[-d $file]
-f file	检测文件是否是普通文件（既不是目录，也不是设备文件），如果是，则返回 true。	[-f $file]
-p file	检测文件是否是有名管道，如果是，则返回 true。	[-p $file]
-r file	检测文件是否可读，如果是，则返回 true。	[-r $file]
-w file	检测文件是否可写，如果是，则返回 true。	[-w $file]
-x file	检测文件是否可执行，如果是，则返回 true。	[-x $file]
-s file	检测文件是否为空（文件大小是否大于0），不为空返回 true。	[-s $file]
-e file	检测文件（包括目录）是否存在，如果是，则返回 true。	[-e $file]

例如，下面的代码，将检测该文件的各种属性：

```
#!/bin/bash
File=$1
if [ -r $File ]
then
   echo "$File has read permission"
else
   echo "$File not have read permission"
fi

if [ -w $File ]
then
   echo "$File has write permission"
else
   echo "$File not have write permission"
fi

if [ -x $File ]
then
   echo "$File has execute permission"
else
   echo "$File not have execute permission"
fi

if [ -f $File ]
then
   echo "$File is regular file"
else
   echo "$File is special file"
fi

if [ -d $File ]
then
   echo "$File is directory"
else
   echo "$File is not directory"
fi

if [ -s $File ]
then
   echo "$File size is not zero"
else
   echo "$File size is zero"
fi

if [ -e $File ]
then
   echo "$File exist"
else
   echo "$File not exist"
fi
```

运行结果：

```
File has read access
File has write permission
File has execute permission
File is  ordinary file
This is not  directory
File size is zero
File exists
```

10 shell注释

以“#”开头的行就是注释，会被解释器忽略。

sh里没有多行注释，只能每一行加一个#号。只能像这样：

```
    #--------------------------------------------
    # 这是一个自动打ipa的脚本，基于webfrogs的ipa-build书写：
    # https://github.com/webfrogs/xcode_shell/blob/master/ipa-build
    # 功能：自动为etao ios app打包，产出物为14个渠道的ipa包
    # 特色：全自动打包，不需要输入任何参数
    #--------------------------------------------
    ##### 用户配置区 开始 #####
    #
    #
    # 项目根目录，推荐将此脚本放在项目的根目录，这里就不用改了
    # 应用名，确保和Xcode里Product下的target_name.app名字一致
    #
    ##### 用户配置区 结束  #####
```

如果在开发过程中，遇到大段的代码需要临时注释起来，过一会儿又取消注释，怎么办呢？每一行加个#符号太费力了，可以把这一段要注释的代码用一对花括号括起来，定义成一个函数，没有地方调用这个函数，这块代码就不会执行，达到了和注释一样的效果。

11 shell字符串

字符串是shell编程中最常用最有用的数据类型（除了数字和字符串，也没啥其它类型好用了），字符串可以用单引号，也可以用双引号，也可以不用引号。单双引号的区别跟PHP类似。

11.1 单引号

```
str='this is a string'
```

单引号字符串的限制：

- 单引号里的任何字符都会原样输出，单引号字符串中的变量是无效的；
- 单引号字串中不能出现单引号（对单引号使用转义符后也不行）。

11.2 双引号

```
name='wenong'
str="your name are \"$name\"! \n"
```

双引号的优点：

- 双引号里可以有变量
- 双引号里可以出现转义字符

11.3 拼接字符串

```
name="wenong"
greeting="hello, "$name" !"
greeting_1="hello, ${name} !"
echo $greeting $greeting_1
```

注意：拼接字符串的时候，字符串之间不能有空格。

11.4获取字符串长度

#号可用来统计字符串的长度

```
string="abcd"
echo ${#string} #输出 4
```

11.5 提取子字符串

```
string="alibaba is a great company"
echo ${string:1:4}  #输出liba
echo ${string:1}    #输出libaba is a great company
```

12 printf

printf 命令用于格式化输出， 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：printf 由 POSIX 标准所定义，移植性要比 echo 好。

如同 echo 命令，printf 命令也可以输出简单的字符串：

```
$printf "Hello, Shell\n"
Hello, Shell
$
```

printf 不像 echo 那样会自动换行，必须显式添加换行符(\n)。

printf 命令的语法：

```
printf  format-string  [arguments...]
```

format-string 为格式控制字符串，arguments 为参数列表。printf()在C语言入门教程中已经讲到，功能和用法与 printf 命令类似这里仅说明与C语言printf()函数的不同：

- printf 命令不用加括号
- format-string 可以没有引号，但最好加上，单引号双引号均可。
- 参数多于格式控制符(%)时，format-string 可以重用，可以将所有参数都转换。

- arguments 使用空格分隔，不用逗号。

请看下面的例子：

```
# format-string为双引号
$ printf "%d %s\n" 1 "abc"
1 abc

# 单引号与双引号效果一样
$ printf '%d %s\n' 1 "abc"
1 abc

# 没有引号也可以输出
$ printf %s abcdef
abcdef

# 格式只指定了一个参数，但多出的参数仍然会按照该格式输出，format-string 被重用
$ printf %s abc def
abcdef
$ printf "%s\n" abc def ghi
abc
def
ghi
$ printf "%s %s %s\n" a b c d e f g h i j
a b c
d e f
g h i
j

# 如果没有 arguments，那么 %s 用NULL代替，%d 用 0 代替
$ printf "%s and %d \n"
and 0
```

注意，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。

13 if语句

if 语句通过关系运算符判断表达式的真假来决定执行哪个分支。Shell 有三种 if ... else 语句：

- if ... fi 语句；
- if ... else ... fi 语句；
- if ... elif ... else ... fi 语句。

1) if ... else 语句

if ... else 语句的语法：

```
if [ expression ]
then
   ...
fi
```

如果 expression 返回 true，then 后边的语句将会被执行；如果返回 false，不会执行任何语句。

最后必须以 fi 来结尾闭合 if，fi 就是 if 倒过来拼写，后面也会遇见。

注意：**expression** 和方括号**[ ]**之间必须有空格，**if**与**[**之间也要有空格，否则会有语法错误。

举个例子：

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
    echo "a is equal to b"
fi

if [ $a != $b ]
then
    echo "a is not equal to b"
fi
```

运行结果：

```
a is not equal to b
```

2) if ... else ... fi 语句

if ... else ... fi 语句的语法：

```
if [ expression ]
then
   ...
else
   ...
fi
```

如果 expression 返回 true，那么 then 后边的语句将会被执行；否则，执行 else 后边的语句。

举个例子：

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
    echo "equal"
else
    echo "not equal"
fi
```

执行结果：

```
a is not equal to b
```

3) if ... elif ... fi 语句

if ... elif ... fi 语句可以对多个条件进行判断，语法为：

```
if [ expression 1 ]
then
   ...
elif [ expression 2 ]
then
   ...
elif [ expression 3 ]
then
   ...
else
   ...
fi
```

哪一个 expression 的值为 true，就执行哪个 expression 后面的语句；如果都为 false，那么不执行任何语句。

举个例子：

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

运行结果：

```
a is less than b
```

if ... else 语句也可以写成一行，以命令的方式来运行，像这样：

```
if [ 2 -eq 2 ]; then echo 'The two numbers are equal!'; fi;
```

14 case语句

case ... esac 与其他语言中的 switch ... case 语句类似，是一种多分枝选择结构。

case 语句匹配一个值或一个模式，如果匹配成功，执行相匹配的命令。case语句格式如下：

```
case 值 in
模式1)
    command1
    command2
    command3
;;
模式2）
    command1
    command2
    command3
;;
*)
    command1
    command2
    command3
;;
esac
```

case工作方式如上所示。取值后面必须为关键字 in，每一模式必须以右括号结束。取值可以为变量或常数。匹配发现取值符合某一模式后，其间所有命令开始执行直至 ;;。;; 与其他语言中的 break 类似，意思是跳到整个 case 语句的最后。

取值将检测匹配的每一个模式。一旦模式匹配，则执行完匹配模式相应命令后不再继续其他模式。如果无一匹配模式，使用星号 * 捕获该值，再执行后面的命令。

下面的脚本提示输入1到4，与每一种模式进行匹配：

```
echo 'Input a number:'
read Num
case $Num in
    1)  echo 'You select 1'
    ;;
    2)  echo 'You select 2'
    ;;
    3)  echo 'You select 3'
    ;;
    4|5)  echo 'You select 4 or 5'
    ;;
    *)  echo 'default'
    ;;
esac
```

输入不同的内容，会有不同的结果，例如：

```
Input a number:
3
You select 3
```

再举一个例子：

```
#!/bin/bash
option=$1
case ${option} in
   -f) echo "param is -f"
      ;;
   -d) echo "param is -d"
      ;;
   *)
      echo "$0:usage: [-f ] | [ -d ]"
      exit 1  #退出码
      ;;
esac
```

运行结果：

```
$./test.sh
test.sh: usage: [ -f filename ] | [ -d directory ]
$ ./test.sh -f
param is -f
$ ./test.sh -d
param is -d
$
```

15 for语句

与其他编程语言类似，Shell支持for循环。

for循环一般格式为：

```
for 变量 in 列表
do
    command1
    command2
    ...
    commandN
done
```

列表是一组值（数字、字符串等）组成的序列，每个值通过空格分隔。每循环一次，就将列表中的下一个值赋给变量。

in 列表是可选的，如果不用它，for 循环使用命令行的位置参数。

- 例如，顺序输出当前列表中的数字：

```
for var in 1 2 3 4 5
do
    echo "The value is: $var"
done
```

运行结果：

```
The value is: 1
The value is: 2
The value is: 3
The value is: 4
The value is: 5
```

- 顺序输出字符串中的字符：

```
for str in 'This is a string'
do
    echo $str
done
```

运行结果：

```
This is a string
```

- 显示主目录下以 .bash 开头的文件：

```
#!/bin/bash
for FILE in $HOME/.bash*
do
    echo $FILE
done
```

运行结果：

```
/home/where/.bash_history
/home/where/.bash_logout
/home/where/.bashrc
```

16 while语句

while循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。其格式为：

```
while expression
do
   ...
done
```

- 命令执行完毕，控制返回循环顶部，从头开始直至测试条件为假。

```
COUNT=0
while [ $COUNT -lt 5 ]
do
    COUNT=$(($COUNT + 1))
    echo $COUNT
done
```

运行脚本，输出：

```
1
2
3
4
5
```

- while循环可用于读取键盘信息

下面的例子中，输入信息被设置为变量FILM，按结束循环。

```
echo -n 'input film: '
while read FILM
do
    echo "great film the $FILM"
done
```

运行脚本，输出类似下面：

```
input film:gongfu
great film gongfu
```

17 until循环

until 循环执行一系列命令直至条件为 true 时停止。until 循环与 while 循环在处理方式上刚好相反。一般while循环优于until循环，但在某些时候，也只是极少数情况下，until 循环更加有用。

until 循环格式为：

```
until [ expression ]
do
   ...
done
```

expression如果返回值为 false，则继续执行循环体内的语句，否则跳出循环。

例如，使用 until 命令输出 0 ~ 9 的数字：

```
#!/bin/bash
a=0
until [ ! $a -lt 10 ]
do
    echo $a
    a=$(($a + 1))
done
```

运行结果：

```
0
1
2
3
4
5
6
7
8
9
```

18 跳出循环

在循环过程中，有时候需要在未达到循环结束条件时强制跳出循环，像大多数编程语言一样，Shell也使用break 和 continue 来跳出循环。

18.1 break命令

break命令允许跳出所有循环（终止执行后面的所有循环）。

下面的例子中，脚本进入死循环直至用户输入数字大于5。要跳出这个循环，返回到shell提示符下，就要使用break命令。

```
#!/bin/bash
while :
do
    echo -n "Input a number between 1 to 5: "
    read Num
    case $Num in
        1|2|3|4|5) echo "Your number is $Num"
        ;;
        *) echo "break"
            break
        ;;
    esac
done
```

在嵌套循环中，break 命令后面还可以跟一个整数，表示跳出第几层循环。例如：

```
break n
```

表示跳出第 n 层循环。

下面是一个嵌套循环的例子，如果 var1 等于 2，并且 var2 等于 0，就跳出循环：

```
#!/bin/bash
for var1 in 1 2 3
do
   for var2 in 1 2 3
   do
      if [ $var1 -eq 1 -a $var2 -eq 2 ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

如上，break 2 表示直接跳出外层循环。运行结果：

```
1 0
1 5
```

18.2 continue命令

- continue命令，不执行后面的命令，继续循环。

```
#!/bin/bash
for var in 1 2 3 4 5
do
    if [ $var -eq 2 ]
    then
        continue
    fi
    echo "$var"
done
```

结果：

```
where@ubuntu:~$ ./test.sh
1
3
4
5
```

- 同样，continue 后面也可以跟一个数字，表示跳出第几层循环。

```
#!/bin/bash
for var1 in 1 2 3
do
   for var2 in 1 2 3
   do
      if [ $var1 -eq 1 -a $var2 -eq 2 ]
      then
          continue 2    #注意1跟2时的区别
      fi
       echo "$var1 $var2"
   done
done
```

运行结果：

```
where@ubuntu:~$ ./test.sh
1 1
2 1
2 2
2 3
3 1
3 2
3 3
```

19 shell函数

函数可以让我们将一个复杂功能划分成若干模块，让程序结构更加清晰，代码重复利用率更高。像其他编程语言一样，Shell 也支持函数。Shell 函数必须先定义后使用。

Shell 函数的定义格式如下：

```
function_name () {
    [commands]
    ...
    [ return value ]
}
```

如果你愿意，也可以在函数名前加上关键字 function：

```
function function_name () {
    [commands]
    ...
    [ return value ]
}
```

函数返回值，可以显式增加return语句；如果不加，会将最后一条命令运行结果作为返回值。

Shell 函数返回值只能是整数，一般用来表示函数执行成功与否，0表示成功，其他值表示失败。如果 return 其他数据，比如一个字符串，往往会得到错误提示：“numeric argument required”。

如果一定要让函数返回字符串，那么可以先定义一个变量，用来接收函数的计算结果，脚本在需要的时候访问这个变量来获得函数返回值。

- 先来看一个例子

```
#!/bin/bash
myfunc () {
   echo "hello world"
}
# Invoke your function
myfunc
```

运行结果：

```
$./test.sh
hello world
$
```

调用函数只需要给出函数名，不需要加括号。

19.1 函数返回值

```
#!/bin/bash
myfunc(){
   echo "hello world"
   return 5
}
myfunc
echo "myfunc return $?"
```

运行结果：

```
where@ubuntu:~$ ./test.sh
hello world
myfunc return 5
```

函数返回值在调用该函数后通过 $? 来获得。

19.2 嵌套调用

```
#!/bin/bash
myfunc1 () {
   echo "myfunc1"
   myfunc2
}
myfunc2 () {
   echo "myfunc2"
}
myfunc1
```

运行结果：

```
where@ubuntu:~$ ./test.sh
myfunc1
myfunc2
```

19.3 函数参数传递

```
#!/bin/bash
myfunc () {
   echo "myfunc $1 $2 $@"
}
myfunc
```

运行结果：

```
where@ubuntu:~$ ./test.sh
myfunc 1 2 1 2 3
```

19.4 函数取消

像删除变量一样，删除函数也可以使用 unset 命令，不过要加上 f 选项，如下所示：

```
unset f function_name
```

19.5 函数中的定义的变量

在函数中定义的变量在整个shell脚本中都能使用。

```
#!/bin/bash
myfunc () {
   echo "myfunc"
   var=888
}
myfunc
echo $var
```

运行结果：

```
where@ubuntu:~$ ./test.sh
myfunc
888
```

20 shell 输入输出重定向

Linux命令默认从标准输入设备(stdin)获取输入，将结果输出到标准输出设备(stdout)显示。一般情况下，标准输入设备就是键盘，标准输出设备就是终端，即显示器。

20.1 输出重定向

命令的输出不仅可以是显示器，还可以很容易的转移向到文件，这被称为输出重定向。

命令输出重定向的语法为：

```
$ command > file #这样，输出到显示器的内容就可以被重定向到文件。
```

例如，下面的命令在显示器上不会看到任何输出：

```
$ who > users
```

打开 users 文件，可以看到下面的内容：

```
$ cat users
oko          tty01   Sep 12 07:30
ai           tty15   Sep 12 13:32
ruth         tty21   Sep 12 10:10
pat          tty24   Sep 12 13:07
steve        tty25   Sep 12 13:03
$
```

输出重定向会覆盖文件内容，请看下面的例子：

```
$ echo "helloworld" > users
$ cat users
helloworld
```

如果不希望文件内容被覆盖，可以使用 `>>` 追加到文件末尾，例如：

```
$ echo "helloworld" >> users
$ cat users
helloworld
helloworld
$
```

20.2 输入重定向

和输出重定向一样，Linux命令也可以从文件获取输入，语法为：

```
command < file #这样，本来需要从键盘获取输入的命令会转移到文件读取内容。
```

例如：将输入重定向到 users 文件：

```
$ cat < users
helloworld
helloworld
$
```

20.3 重定向深入讲解

一般情况下，每个 Unix/Linux 命令运行时都会打开三个文件：

- 标准输入文件(stdin)：stdin的文件描述符为0，linux程序默认从stdin读取数据。
- 标准输出文件(stdout)：stdout 的文件描述符为1，linux程序默认向stdout输出数据。
- 标准错误文件(stderr)：stderr的文件描述符为2，linux程序会向stderr流中写入错误信息。

默认情况下，command > file 将 stdout 重定向到 file，command < file 将stdin 重定向到 file。

- 如果希望 stderr 重定向到 file，可以这样写：

```
$command 2> file
```

- 如果希望 stderr 追加到 file 文件末尾，可以这样写：

```
$command 2>> file
```

- 如果希望将 stdout 和 stderr 合并后重定向到 file，可以这样写：

```
$command > file 2>&1
```

或

```
$command >> file 2>&1
```

如果希望对 stdin 和 stdout 都重定向，可以这样写：

```
$command < file1 >file2
```

command 命令将 stdin 重定向到 file1，将 stdout 重定向到 file2。

命令	说明
command > file	将输出重定向到 file。
command < file	将输入重定向到 file。
command >> file	将输出以追加的方式重定向到 file。
n > file	将文件描述符为 n 的文件重定向到 file。
n >> file	将文件描述符为 n 的文件以追加的方式重定向到 file。
n>&m	将输出文件 m 和 n 合并。
n<&m	将输入文件 m 和 n 合并。
<<tag	将开始标记 tag 和结束标记 tag 之间的内容作为输入。

21.4 /dev/null 文件

如果希望执行某个命令，但又不希望在屏幕上显示输出结果，那么可以将输出重定向到 /dev/null：

```
$ command > /dev/null
```

/dev/null 是一个特殊的文件，写入到它的内容都会被丢弃；如果尝试从该文件读取内容，那么什么也读不到。但是 /dev/null 文件非常有用，将命令的输出重定向到它，会起到”禁止输出“的效果。

如果希望屏蔽 stdout 和 stderr，可以这样写：

```
$ command > /dev/null 2>&1
```

22 shell文件包含

像其他语言一样，Shell 也可以包含外部脚本，将外部脚本的内容合并到当前脚本。

Shell 中包含脚本可以使用：

```
. filename
```

或

```
source filename
```

两种方式的效果相同，简单起见，一般使用点号(.)，但是注意点号(.)和文件名中间有一空格。

例如，创建两个脚本，一个是被调用脚本 subscript.sh，内容如下：

```
url="http://wenong.so.c"
```

一个是主文件 main.sh，内容如下：

```
#!/bin/bash
. ./subscript.sh
echo $url
```

执行脚本：

```
$chomd +x main.sh
./main.sh
http://wenong.so.c
$
```

注意：被包含脚本不需要有执行权限。