

08 SHELL编程


创建用户

useradd username



Users

Add User...

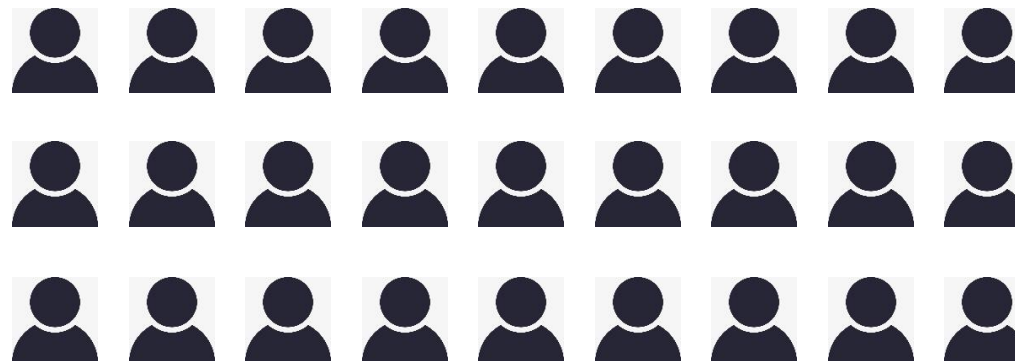


lei

Password

Automatic Login ☒

Last Login



.....

Shell程序

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，它以交互方式解释和执行用户输入的命令；作为程序设计语言，它有变量、关键字，有各种控制语句，支持函数模块，有自己的语法结构。

Shell程序是使用系统提供的命令编写的文本文件，可以帮助用户轻松地完成任务，提高使用和维护系统的效率。在Linux的发行版本中就包含了很多的shell程序，这些脚本有的是为了完成系统参数的设置，例如前面介绍的/etc/profile等文件；有的是为了完成某项系统服务的启动工作，例如/etc/rc.d/init.d目录下的所有脚本。另外，Linux的图形界面也是通过shell脚本解释启动的，很多应用程序本身就是一个shell程序。

创建和执行SHELL程序

Shell程序是通过shell命令解释器解释执行的，不生成二进制的可执行代码，这一点和python类似。

创建和执行一个shell程序非常简单，一般需要以下3个步骤：

- (1)利用文本编辑器创建脚本内容。
- (2)使用“chmod”命令设置脚本的可执行权限。
- (3)执行脚本。

```
lei@lei-VirtualBox:~$ vim hello.sh
lei@lei-VirtualBox:~$ cat hello.sh
#!/bin/bash

echo "hello world"
lei@lei-VirtualBox:~$ chmod +x hello.sh
lei@lei-VirtualBox:~$ ./hello.sh
hello world
```

创建和执行SHELL程序

通常shell脚本都是以解释器声明开始的，例如#!/bin/bash，其中“#!”后面的“/bin/bash”，表示实际使用的解释器。例如，以sh作为解释器，则可以该声明可以是“#!/bin/sh”。注意：与其它行不同，这里前面虽然以“#”开头，但不是注释行。

```
lei@lei-VirtualBox:~$ cat hello.sh
#!/bin/sh

echo "hello world!"
lei@lei-VirtualBox:~$ chmod +x hello.sh
lei@lei-VirtualBox:~$ ./hello.sh
hello world!
```

指定SHELL程序执行脚本

可以在指定的Shell下执行脚本，以脚本名作为参数。基本用法如下：

Shell名称 脚本名 [参数]

```
lei@lei-VirtualBox:~$ bash hello.sh  
hello world!  
lei@lei-VirtualBox:~$ sh hello.sh  
hello world!
```

这种方式运行的脚本不必在第一行指定Shell解释器，也不需要赋予可执行权限。

环境变量

Shell中使用的变量分三类：

环境变量：是Linux系统环境的一部分，通常不需要用户去定义。Shell使用环境变量来存储系统信息，这些变量可以提供给在shell中执行的程序使用，不同的shell会有不同的环境变量及其设置的方法。

内部变量：是由系统提供的，用户不能修改它们。

用户变量：是用户在编写shell脚本的时候定义的，可以在shell脚本中任意使用和修改。

如果希望一个用户定义的变量能够在定义它的shell脚本以外使用，就必须使用export命令。例如，“export var”命令就是将用户定义的变量var添加到系统变量列表中，这样就可以在定义var变量脚本以外的地方使用。

```
lei@lei-VirtualBox:~$ echo $test
lei@lei-VirtualBox:~$ export test="testEnvVar"
lei@lei-VirtualBox:~$ echo $test
testEnvVar
```

查看环境变量

在Ubuntu中可以使用env命令来查看系统当前的环境变量及其取值。env命令是environment的缩写，用于列出所有的环境变量。

```
lei@lei-VirtualBox:~$ env | tail -10
LOGNAME=lei
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
XDG_RUNTIME_DIR=/run/user/1000
XAUTHORITY=/run/user/1000/gdm/Xauthority
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
SESSION_MANAGER=local/lei-VirtualBox:@/tmp/.ICE-unix/1240,unix/lei-VirtualBox:/tmp/.ICE-unix/1240
LESSOPEN=| /usr/bin/lesspipe %s
GTK_IM_MODULE=ibus
_=/usr/bin/env
```

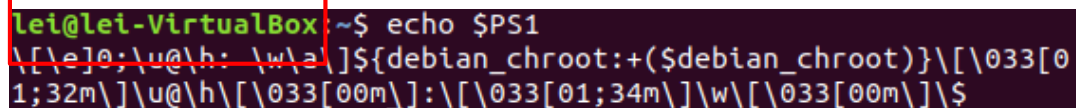
利用管道符和tail命令与env命令结合只显示系统环境变量的最后10个。

查看环境变量

若要查看当前某个环境变量的值，可以使用echo命令，并在环境变量的前面加上“\$”即可。

例如,查看当前的命令主提示符，可以输入如下命令：

```
echo $PS1
```



```
lei@lei-VirtualBox:~$ echo $PS1
\[\e]0;\u@\h: \w\a\]${debian_chroot:+($debian_chroot)}\[\033[0
1;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$
```

命令主提示符是Linux的shell程序为用户输入命令而设置的提示符。

查看环境变量

环境变量PATH记录了命令执行时的默认搜索路径，即当用户在命令提示符后输入命令时，Linux系统会按照PATH设置的路径搜索该命令，然后再执行该命令。PATH变量的值由多个路径组成，各路径之间使用 “:” 隔开。

```
echo $PATH
```

```
lei@lei-VirtualBox:~$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

SHELL配置文件

用户可以通过export命令来查看和设置常用的环境变量，基本用法：

```
export 变量名=变量值
```

只在当前的shell或其子shell下有效，一旦shell关闭，变量就失效了。

如果要使环境变量永久生效，则需要编辑配置文件。配置文件有全局的配置文件/etc/profile，也有用户个人的配置文件 ~/.bashrc。

设置JAVA环境变量

如果添加的环境变量只对当前用户有效，可以写入用户主目录下的.bashrc文件：

```
vim ~/.bashrc
```

添加语句：

```
export CLASS_PATH=.:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
```

注销或者重启可以使修改生效，如果要使添加的环境变量马上生效：

```
source ~/.bashrc
```

设置JAVA环境变量

要使环境变量对所有用户有效，可以修改profile文件：

```
sudo vim /etc/profile
```

添加语句：

```
export CLASS_PATH=.:$JAVA_HOME/lib:$JAVA_HOME/jre/lib
```

注销或者重启可以使修改生效，如果要使添加的环境变量马上生效：

```
source /etc/profile
```

注释

注释符 “#” 通常使用在shell脚本程序或应用程序的配置文件中，使用 “#” 开头的行为注释行，shell在解释该脚本程序的时候不会执行该行。对于有经验的程序员来说，注释行的使用可以增加程序的可读性，也可以使日后的维护更加简单。

SHELL变量

shell脚本是一种弱类型的脚本语言。所谓弱类型脚本语言，就是在shell脚本中，对类型的要求不严格，同一个变量可以随着使用场合的不同，存储不同类型的数据。弱类型语言变量使用灵活，但是编程者需要注意对变量当前存储的数据类型的检查。

1) 变量的声明

变量的使用不需要显式的声明，或者说赋值就可以认为是变量的声明。通常，给一个变量赋值应采用如下的格式：

变量名=值

注意：等号两边不允许有分隔符（包括空格，制表位和回车符）。

例如：

```
a1="hello"
```

```
b1=90
```

变量命名规则

变量名的命名应当遵循以下规则：

- 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Shell中的关键字。

SHELL变量

2) 变量的访问

通常，要访问一个变量，可以采取在变量名前加一个\$的方法，即“\$变量名”。

例如，要访问上面定义的变量a1可以采用如下的方法：

```
echo "a1 is $a1"
```

但是，有时候这种方法会产生混淆。

例如，希望使用变量a1来输入“hello Linux”字符串。如果使用echo “\$a1Linux” 就会得不到期待的字符。这是因为bash会把“a1Linux” 作为一个变量来处理。

变量替换

此时可以选择使用以下的几种用法（其中，value代表一个变量可能取的具体的值）：

- `${变量var}`：替换为变量本来的值。
- `${变量var:-value}`：如果指定的变量var存在，则返回var的值，否则返回value。
- `${变量var:=value}`：如果指定的变量var存在，则返回var的值，否则先将value赋给var，然后再返回value。
- `${变量var:+value}`：如果指定的变量var存在，则返回value，否则返回空值。
- `${变量var:?value}`：如果指定的变量var存在，则返回该var的值，否则将错误提示消息value送到标准错误输出并退出shell程序。
- `${变量var:offset[:length]}`：offset和length是整数，中括号表示可选部分。表示返回从变量var的第offset+1个字符开始长度为length的子串。如果中括号部分省略，则表示返回变量var第offset+1个字符后面的子串。

变量替换

```
var="hello"
```

定义变量var，并被赋值为“hello”

```
echo $var ${title:-"marry"}!
```

```
hello marry!
```

变量title在前面都没有被赋值，所以\${title:-"marry"}返回 “marry”。

```
echo $var ${title:+"tom"}!
```

```
hello !
```

变量title仍然没有被赋值，即不存在，所以\${title:+"tom"}返回空值。

```
echo $var ${title:? "title is null or empty"}!
```

```
bash: title: title is null or empty
```

变量title仍然没有被赋值，即不存在，所以\${title:? "title is null or empty"}返回了错误信息，即 “bash: title: title is null or empty”。

SHELL变量

```
echo $var ${title:="tom and marry"}!
```

```
hello tom and marry!
```

到此为止变量title仍然没有被定义，所以title被赋值为 “tom and marry”，并返回该值。

```
echo $var ${title:+"somebody"}!
```

```
hello somebody!
```

此时变量title已经存在，故返回 “somebody”。

```
echo $var ${title:8:5}!
```

```
hello marry!
```

此处变量title已经存在，且值为tom and marry，取其第9个字符，即 “m”开始后面5个字符，也就是 “marry”。

删除变量

使用unset命令可以删除变量，用法如下：

unset 变量名

```
lei@lei-VirtualBox:~$ a=10
lei@lei-VirtualBox:~$ echo $a
10
lei@lei-VirtualBox:~$ unset a
lei@lei-VirtualBox:~$ echo $a
```

SHELL内部变量

在shell程序中存在一些特殊变量，这些变量分别是\$0、\$1、...\$n，以及\$#、\$*、\$@、\$?和\$。

- \$0存放的是命令行的命令名，
- \$n存放的是传递给脚本或函数的第n个参数，
- \$#存放传递给脚本或函数的参数个数，
- \$*和\$@均用于存放传递给脚本或函数的所有参数，两者的区别在于被双引号包含时，\$*把所有的参数作为一个整体，而\$@则把所有的参数看作是类似于字符串数组一样，可以单独访问这些参数，
- \$?存放上个命令的退出状态，或函数的返回值，
- \$\$存放当前Shell的PID。

命令行参数

上述内部变量中\$0、\$1、...\$n，以及\$#、\$*、\$@表示运行shell程序时传递给脚本的参数，通常称为命令行参数或位置参数。

```
#!/bin/bash

echo "Script name: $0"
echo "First parameter: $1"
echo "Second parameter: $2"
echo "All parameters: $@"
echo "All parameters: $*"
echo "Number of Parameters: $#"
```

```
lei@lei-VirtualBox:~$ bash test_parameters.sh 111 222
Script name: test_parameters.sh
First parameter: 111
Second parameter: 222
All parameters: 111 222
All parameters: 111 222
Number of Parameters: 2
```

\$*与\$@

\$*与\$@不被双引号（ "" ）包含时，都以 "\$1" 、 "\$2" 、 ...、 "\$n" 的形式输出所有参数，但是被双引号包含时， "\$*" 会将所有参数作为一个整体，以 "\$1 \$2 ... \$n" 的形式输出所有参数， "\$@" 会将各个参数分开，以 "\$1" 、 "\$2" 、 ...、 "\$n" 的形式输出所有参数。


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

```
echo Using '$*'  
for param in $*; do  
    printf '==>%s<==\n' "$param"  
done;
```

```
echo  
echo Using '$*'  
for param in "$*"; do  
    printf '==>%s<==\n' "$param"  
done;
```

```
echo  
echo Using '$@'  
for param in $@; do  
    printf '==>%s<==\n' "$param"  
done;
```

```
echo  
echo Using '$@';  
for param in "$@"; do  
    printf '==>%s<==\n' "$param"
```

不加双引号

加双引号

```
lei@lei-VirtualBox:~$ bash test_parameters.sh 111 222  
Using $*  
==>111<==  
==>222<==  
Using "$*"  
==>111 222<==  
Using $@  
==>111<==  
==>222<==  
Using "$@"  
==>111<==  
==>222<==
```

变量值输入输出

read命令

用read命令由标准输入读取数据，然后赋给指定的变量。其一般格式如下

read 变量1 [变量2]

例如：

```
#!/bin/bash

read -p "please input two numbers: " v1 v2
echo $v1
echo $v2

lei@lei-VirtualBox:~$ bash test_read.sh 111 222
please input two numbers: 111 222
111
222
```

echo 命令

echo命令是将其后的参数输出。最好用双引号把所有参数括起来，这样不仅易读并且能使shell对它们进行正确的解释。

数组

Shell支持一维数组，不支持多维数组，数组下标由0开始。

数组的定义方法是将数组元素用空格分开放到括号里

数组名=(值1 值2 ... 值n)

例如

myarray=(A B C D)

也可单独定义数组中的元素

myarray[0]=A
myarray[1]=B

数组

访问数组元素格式为：

`${数组名[下标]}`

例如：

```
echo ${myarray[0]}
```

使用@或*获取数组中所有元素：

```
echo ${myarray[*]}
```

```
echo ${myarray[@]}
```

获取数组元素个数：

`${#数组名[@]}`

数组

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

```
a=(1 2 3 4)
```

```
echo "Length of a: ${#a[@]}"
```

```
echo "First element: ${a[0]}"
```

```
lei@lei-VirtualBox:~$ bash test_array.sh  
Length of a: 4  
First element: 1
```

数组切片

所谓数组切片就是将数组中的一部分元素取出来形成一个新的数组。

格式：

`${数组名[@]:m:n}`

其中m为切片的起始位置，n为提取的元素个数。

省略n表示取从m一直到数组结束的所有元素。

```
lei@lei-VirtualBox:~$ array=(a b c d)
lei@lei-VirtualBox:~$ echo ${array[@]:1}
b c d
```

省略m表示从数组开头取n个元素。

```
lei@lei-VirtualBox:~$ echo ${array[@]::2}
a b
```

SHELL表达式

利用运算符将变量或常量连接起来就构成了表达式。但是由于在shell中变量和常量没有特定的数据类型，因此在shell中单纯使用一个表达式并不能实现数学运算，而必须使用expr或let命令来指明表达式是一个运算式。expr命令会先求出表达式的值，然后送到标准输出显示。let命令会先求出表达式的值，然后赋值给一个变量，而不显示在标准输出上。

expr和let命令的使用方法如下：

```
expr <表达式>
```

```
let <表达式1> [表达式2 ...]
```

SHELL表达式

在expr命令的表达式中，需要用空格将数字运算符与操作数分隔开。例如：

expr 3+2

操作数3、2和运算符+之间没有空格，此时bash不会报错，而是把3+2作为字符串来处理。

expr 3 + 2

操作数3、2和运算符+之间有空格，此时bash认为是数字运算，返回5送到标准输出设备。

```
lei@lei-VirtualBox:~$ expr 3+2
3+2
lei@lei-VirtualBox:~$ expr 3 + 2
5
```


SHELL表达式

如果表达式中的运算符是 “<” 、 “>” 、 “&” 、 “*” 及 “|” 等特殊符号，需要使用双引号、单引号括起来，或将反斜杠 (\) 放在这些符号的前面。

例如：

```
expr 3 "*" 2
```

使用双引号将操作符*括起，此时bash返回乘积6。

```
lei@lei-VirtualBox:~$ expr 3*"2
3*2
lei@lei-VirtualBox:~$ expr 3 "*" 2
6
lei@lei-VirtualBox:~$ expr 3 * 2
expr: syntax error
```

SHELL表达式

let命令中的多个表达式之间需要空格隔开，而表达式内部无需使用空格。

例如：

```
let s=(2+3)*4
```

s结果为 $5*4=20$ 。

SHELL表达式

用于计算的数可以用变量表示：

`n=1`

`m=5`

`expr $n + $m`

也可以将expr的结果赋给变量：

`val=`expr 2 + 2``

注意表达式要使用反引号括起来。

一种简洁的方式是使用\$[]表达式进行数学计算：

`val=$((5+3))`

条件判断

在编写程序的时候，经常需要根据某个条件的测试进行程序执行分支的选择。这里的条件可能是某个表达式的值、文件的存取权限、某段代码的执行结果，或者是多个条件结果按照逻辑运算后的值。条件测试的结果只有真或假两种。需要注意的是，这里“真”的数值表示为0，“假”的数值表示为非0，与表达式的真值以及C语言的真值刚好相反。

逻辑表达式

在shell中条件判断的使用方法是，利用test命令或一对中括号[]包含条件测试表达式，这两种方法是等价的。它们的格式如下：

test cond_expr

或 [cond_expr]

注意：利用一对中括号时，**左右的中括号与表达式之间都必须存在空格。**

cond_expr是需要测试的条件表达式，可以是以下几种情况：

- (1) 文件存取属性测试：包括文件类型，文件的访问权限等。
- (2) 字符串属性测试，包括字符串长度，内容等。
- (3) 整数关系测试，包括大小比较，相等判断等。
- (4) 上述3种关系通过逻辑运算（与、或、非）的组合。

算术运算符

五种基本的算术运算：+（加）、-（减）、*（乘）、/（除）和%（取模）。

shell只提供整数的运算。

格式如下：

expr n1 运算符 n2

例：

expr 15 * 15

注意：在运算符的前后都留有空格，否则expr不对表达式进行计算，而直接输出它们。

整数关系运算符

参数	功能
$n1 \text{ -eq } n2$	如果整数 $n1$ 等于 $n2$ ($n1 = n2$) , 则测试条件为真
$n1 \text{ -ne } n2$	如果整数 $n1$ 不等于 $n2$ ($n1 \neq n2$) , 则测试条件为真
$n1 \text{ -lt } n2$	如果 $n1$ 小于 $n2$ ($n1 < n2$) , 则测试条件为真
$n1 \text{ -le } n2$	如果 $n1$ 小于等于 $n2$ ($n1 \leq n2$) , 则测试条件为真
$n1 \text{ -gt } n2$	如果 $n1$ 大于 $n2$ ($n1 > n2$) , 则测试条件为真
$n1 \text{ -ge } n2$	如果 $n1$ 大于等于 $n2$ ($n1 \geq n2$) , 则测试条件为真

字符串检测运算符

参数	功能
<code>str</code>	如果字符串 <code>str</code> 不是空字符串，则测试条件为真
<code>str1 = str2</code>	如果 <code>str1</code> 等于 <code>str 2</code> ，则测试条件为真(注意， “=”前后须有空格)
<code>str1 != str2</code>	如果 <code>str1</code> 不等于 <code>str2</code> ，则测试条件为真
<code>-n str</code>	如果字符串 <code>str</code> 的长度不为0，则测试条件为真
<code>-z str</code>	如果字符串 <code>str</code> 的长度为0，则测试条件为真

文件测试运算符

参数	功能
-r file	若文件存在并且是用户可读的，则测试条件为真
-w file	若文件存在并且是用户可写的，则测试条件为真
-x file	若文件存在并且是用户可执行的，则测试条件为真
-f file	若文件存在并且是普通文件，则测试条件为真
-d file	若文件存在并且是目录文件，则测试条件为真
-s file	若文件存在并且不是空文件，则测试条件为真

布尔运算符

参数	功能
!	逻辑非，放在任意逻辑表达式之前，原来真的表达式变为假，原来假的变为真
-a	逻辑与，放在两个逻辑表达式之间，仅当两个逻辑表达式都为真时，结果才为真
-o	逻辑或，放在两个逻辑表达式之间，其中只要有一个逻辑表达式为真时，结果就为真
()	圆括号，用于将表达式分组，优先得到结果。括号前后应有空格并用转义符 “\("和 “\)”

使用文件测试命令

首先使用test命令测试test.sh是否存在且可写，从ls -l命令返回的结果看，确实是test.sh文件存在且可写的，所以“echo \$?”命令返回0表示真。

然后又使用中括号测试testdir是不是目录以及是否可写，从ls -l命令的返回来看，testdir同样是目录且可写的，所以返回真。

```
lei@lei-VirtualBox:~$ ls -l test.sh
-rw-r--r-- 1 lei lei 330 Dec  2 13:29 test.sh
lei@lei-VirtualBox:~$ test -w test.sh
lei@lei-VirtualBox:~$ echo $?
0
```

```
lei@lei-VirtualBox:~$ ls -ld testdir
drwxr-xr-x 2 lei lei 4096 Dec  2 14:18 testdir
lei@lei-VirtualBox:~$ [ -d testdir -a -w testdir ]
lei@lei-VirtualBox:~$ echo $?
0
```

字符串测试命令

首先定义root_home变量，值为/root，变量lei_home，值为/home/lei，然后测试这两个字符串变量的值是否相等，结果为1表示不相等。

```
root_home="/root"
lei_home="/home/lei"
[ $root_home = $lei_home ]
echo $?
```

```
lei@lei-VirtualBox:~$ root_home="/root"
lei@lei-VirtualBox:~$ lei_home="/home/lei"
lei@lei-VirtualBox:~$ [ $root_home = $lei_home ]
lei@lei-VirtualBox:~$ echo $?
1
```

数值关系测试命令

首先定义变量var1，值为200，变量var2，值为300，接着测试var1的值是否等于var2的值。返回值为1，表示这两个变量不等。然后又测试var1是否小于var2，返回值为0，表示var1的值小于var2。

```
var1=100
var2=200
[ $var1 -eq $var2 ]
echo $?
[ $var1 -lt $var2 ]
echo $?
```

```
lei@lei-VirtualBox:~$ var1=100
lei@lei-VirtualBox:~$ var2=200
lei@lei-VirtualBox:~$ [ $var1 -eq $var2 ]
lei@lei-VirtualBox:~$ echo $?
1
lei@lei-VirtualBox:~$ [ $var1 -lt $var2 ]
lei@lei-VirtualBox:~$ echo $?
0
```

SHELL控制结构

Shell程序的控制结构是用于改变shell程序执行流程的结构。在shell程序的执行过程中可以根据某个条件的测试值，来选择程序执行的路径。在shell程序中，控制结构可以简单地分为分支和循环结构两类。Bash支持的分支结构有if结构和case结构，支持的循环结构有for结构、while结构和until结构。使用方法与C语言等程序设计语言中相应的结构类似。

IF条件语句

if结构是最常用的分支结构，其格式如下：

```
if 条件测试1
then
    command_list_1
[elif 条件测试2
then
    command_list_2 ]
[else
    command_list_3 ]
fi
```

其中，中括号部分为可选部分。当“条件测试1”为真时，执行command_list_1，否则如果存在elif语句，则测试“条件测试2”，如果为真，执行command_list_2。如果elif语句不存在或“条件测试2为假，则执行command_list_3。条件测试部分一般可以是test或[]修饰的条件表达式。

IF条件语句

例 根据用户输入的目录名称判断该目录是否存在，如果存在则进入该目录，否则测试同名文件是否存在，如果存在，则退出shell程序，否则新建同名目录，并进入该目录。

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

```
echo "input a directory name, please! "
```

```
read dir_name
```

```
#测试$dir_name目录是否存在
```

```
if [ -d $dir_name ]
```

```
then
```

```
    cd $dir_name
```

```
    echo "$dir_name has already existed,enter directory succeed"
```

```
#测试是否存在与$dir_name同名的文件
```

```
elif [ -f $dir_name ] ;
```

```
then
```

```
    echo "file: $dir_name has already existed,create directory failed"
```

```
    exit
```

```
else
```

```
    mkdir $dir_name
```

```
    cd $dir_name
```

```
    echo "$dir_name has not existed,create and enter directory succeed"
```

```
fi
```


IF条件语句

由于Linux不允许在同一目录下存在同名的文件和目录，所以如果\$dir_name不存在时，还要测试是否有同名的文件存在，然后才能新建该目录。

注意：then命令可以和if结构写在同一行，但是如果then命令和if结构在同一行时，then命令的前面一定要有一个分号，且分号与条件测试表达式之间用空格隔开。

CASE语句

If 结构用于分支选择较少的情况，当程序存在多个分支选择时，如果使用 if 结构，就必须使用多个 elif 结构，从而使得程序的结构冗余，此时可以选择使用case结构。case结构可以帮助程序灵活地完成多路分支的选择，而且程序结构直观、简洁。

CASE语句

case分支结构的格式如下：

```
case expr in
  模式1 )
    command_list_1
    ;;
  [ 模式2 )
    command_list_2
    ;;
  .....
  * )
    command_list_n
    ;; ]
esac
```

其中，expr可以是变量、表达式或shell命令等，模式为expr的取值。通常一个模式可以是expr的多种取值，使用|连接。模式中还可以使用通配符，星号（*）表示匹配任意字符值，问号（?）表示匹配任意一个字符，[..]可以匹配某个范围内的字符。

在case分支结构中，首先计算expr的值，然后根据求得的值查找匹配的模式，接着执行对应模式后面的命令序列，执行完成后，退出case结构。需要注意的是，在case结构的命令序列后面需要使用双分号（;;）分隔下一个模式。

CASE语句

```
#!/bin/bash

SYSTEM=`uname -s`
case "$SYSTEM" in

"Linux")
    echo "Linux"
    ;;

"FreeBSD")
    echo "FreeBSD"
    ;;

"Solaris")
    echo "Solaris"
    ;;

*)
    echo "Unknow"
    ;;
esac
```

```
lei@lei-VirtualBox:~$ bash demo_case.sh
Linux
```

FOR循环语句

for循环用于预先知道循环执行次数的程序段中，它是最常用的循环结构之一。for的格式如下：

```
for var [ in value_list ]  
do  
    command_list  
done
```

其中，value_list是变量var需要取到的值，随着循环的执行，变量var需要依次从value_list中的第一个值，取到最后一个值。do和done结构之间的command_list是循环需要执行的命令序列，变量var每取一个值都会循环执行一次command_list中的命令。同样中括号部分为可选部分，如果省略了该部分，bash会从命令行参数中为var取值，即等同于“in \$@”。

FOR循环语句

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

```
array=(1 2 3 4 5 6 7 8 9)
```

```
for var in ${array[@]}  
do  
    echo $var  
done
```

```
lei@lei-VirtualBox:~$ bash demo_for.sh  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

FOR循环语句

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

```
array=(1 2 3 4 5 6 7 8 9)
```

```
for var in ${array[@]}  
do  
    echo $var  
done
```

```
lei@lei-VirtualBox:~$ bash demo_for.sh  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

FOR循环语句

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

```
for var in `seq 9`  
do  
    echo $var  
done
```

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

```
for var in `seq 1 9`  
do  
    echo $var  
done
```

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

```
for var in {1..9}  
do  
    echo $var  
done
```

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

```
for ((var=1;var<10;var++))  
do  
    echo $var  
done
```


WHILE和UNTIL循环结构

while和until循环结构的功能基本相同，主要用于循环次数不确定的场合。

while的格式如下：

```
while expr  
do  
    command_list  
done
```

until的格式如下：

```
until expr  
do  
    command_list  
done
```

从格式上看，二者的使用方法完全相同，但是二者对循环体执行的条件恰恰相反。在while循环中，只有expr的值为真时，才执行do和done之间的循环体，直到expr取值为假时退出循环。而在until循环中，只有expr的值为假时，才执行do和done之间的循环体，直到expr取值为真时退出循环。

WHILE和UNTIL循环结构

从上面的while和until循环的执行流程可以看出，expr的取值直接决定command_list的执行与否以及能否正常退出循环，因此通常在命令序列command_list中都存在修改expr取值的命令。否则while和until就无法退出command_list的执行循环，从而陷入死循环。通常，同一个问题如果可以使用while循环，就可以使用until循环。

WHILE和UNTIL循环结构

例 while和until循环结构示例。

while循环示例	until循环示例
<pre>#!/bin/bash clear loop=0</pre>	<pre>#!/bin/bash clear loop=0</pre>
<pre>while [\$loop -le 10]</pre>	<pre>until[\$loop -gt 10]</pre>
<pre>do let loop=\$loop+1 echo "the loop current value is: \$loop" done</pre>	<pre>do let loop=\$loop+1 echo "the loop current value is: \$loop" done</pre>

上面两段程序都是完成对循环变量loop加1的任务，两段程序的输出结果完全相同。对比两个程序可以发现，只有循环条件的设置不同。

其它循环语句

Continue语句跳过循环中位于它后面的语句，回到本层循环的开头，进行下一次循环。

语法格式如下：

`continue [n]`

其中n表示从包含continue语句的最内层循环体向外跳到第n层循环。默认值为1。

```
#!/bin/bash

for var in {1..5}
do
    if [ $var -le 3 ]
    then
        continue
    fi
    echo "var = $var"
done
```

```
lei@lei-VirtualBox:~$ bash demo_continue.sh
var = 4
var = 5
```

其它循环语句

Break用来终止一个重复执行的循环。循环可以是for、while或until语句。

语法格式如下：

```
break [n]
```

其中n表示要跳出几层循环。默认值为1。

```
#!/bin/bash

a=1
while [ $a -le 5 ]
do
    if [ $a -eq 3 ]
    then
        break
    fi
    echo "a = $a"
    a=$((a+1))
done
```

```
lei@lei-VirtualBox:~$ bash demo_break.sh
a = 1
a = 2
```

其它循环语句

Exit语句用来退出一个Shell程序，并设置退出值。

语法格式如下：

`exit [n]`

其中n是设定的退出值。如果未指定n，则退出值为最后一个命令的执行状态。

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

```
trap "echo 'exit...';exit 3" 2
```

```
for i in {1..5}; do
```

```
    echo $i
```

```
    sleep 1
```

```
done
```

trap命令实现信号的捕获及处理

```
lei@lei-VirtualBox:~$ bash demo_exit.sh
1
2
3
4
^Cexit...
lei@lei-VirtualBox:~$ echo $?
3
```

SHELL函数

和其他的高级程序设计语言一样，在bash中也可以定义使用函数。函数是一个语句块，它能够完成独立的功能，而且在需要的时候可以被多次使用。利用函数，shell程序将具有相同功能代码块提取出来，实现程序代码的模块化。在程序需要修改的时候，只需要修改被调用的函数，减少了程序调试和维护的强度。

在bash中，函数需要先定义后使用。函数定义的格式如下：

```
[function] fun_name ( )  
{  
    command_list  
    [return ret_value]  
}
```

SHELL函数

其中，function表示下面定义的是一个shell函数，可以省略。fun_name就是定义的函数名。
command_list就是实现函数功能的命令序列，称为函数体。函数一旦定义就可以被多次调用，而且函数调用的方法与shell命令的方法完全一致。函数调用的格式如下：

```
fun_name [param_1 param_2 ... param_n]
```

其中，fun_name是被调用的函数名，param_1、param_2、...param_n是调用时传递给函数的参数，各参数之间使用空格隔开。函数调用时是否需要传递参数，由函数的定义和功能决定。如果函数确实需要传递参数，此时可以使用\$0、\$1、...\$n，以及\$#、\$*和\$@这些特殊变量。

SHELL函数

例 向bash函数传递参数的示例。在bash脚本中定义函数，然后在该脚本时通过命令行传递参数。

```
#!/bin/bash

function fun1 ( )
{
    echo "Your command is:$0 $*"
    echo "Number of parameters \ $# is: $# "
    echo "Script file name \ $0 is: $0 "
    echo "Parameters \ $* is: $*"
    echo "Parameters \ $@ is: @$"
    count=1
}
```

SHELL函数

```
for param in $@
do
    echo "Parameters ( $count ) is: $param"
    let count=$count+1
done
}
fun1 $@
```

```
lei@lei-VirtualBox:~$ bash demo_func.sh 111 222 333
Your command is:demo_func.sh 111 222 333
Number of parameters $# is: 3
Script file name $0 is: demo_func.sh
Parameters $* is: 111 222 333
Parameters $@ is: 111 222 333
Parameters ( 1 ) is: 111
Parameters ( 2 ) is: 222
Parameters ( 3 ) is: 333
```

SHELL函数返回值

程序中的函数也可以有返回值，使用return命令可以从函数返回值。返回值只能是整数。一般函数正常结束时返回真，即0，否则返回假，即非0值。return使用的格式如下：

```
return [expr]
```

expr存在，0表示程序正常结束，非0值表示程序出错。如果expr省略，则以函数的最后一条命令的执行状态作为返回值。

另外，测试函数的返回值可以使用和shell命令的返回值相同的方法，即使用测试\$?值，也可以采用直接测试命令函数的返回值。

SHELL脚本书写的良好习惯

1. 开头指定脚本解释器

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

2. 开头加版本版权信息

```
#Date
```

```
#Author
```

```
#Mail
```

```
#Function
```

```
#Version
```

3. 脚本中不用中文注释

尽量用英文注释，防止本机或切换系统环境后出现中文乱码的困扰

4. 以.sh为扩展名

```
script-name.sh
```

SHELL脚本书写的良好习惯

5. 成对的符号应尽量一次性写出来，然后退格在符合里增加内容，以防止遗漏。

这些成对的符合包括：

{ } [] " " ' ' ` `

6. 中括号两端至少各一个空格，先退2格 然后进一格，双括号也是如此。

SHELL脚本书写的良好习惯

7. 对应流程控制语句，应一次将格式写完，再添加内容。

比如，一次性完成if语句的格式，应为

```
if 条件内容
then
    内容
fi
```

一次性完成for循环语句格式，应为

```
for
do
    内容
done
```

一次完成while语句格式，应为

```
while 条件
do
    内容
done
```

SHELL脚本书写的良好习惯

8. 通过缩进让代码更易读
9. 对应常规变量的字符串定义变量值应加双引号，并且等号前后不能有空格
10. 脚本中的单引号和双引号必须为英文状态下的符号
11. 变量名称应该具有相关意思，不能太随便。

SHELL脚本书写的良好习惯

脚本代码注释规范

良好的脚本代码注释习惯可以大大增加脚本的可读性。为读懂脚本，修改脚本提供了快捷的途径。

脚本代码注释可以使用#和分号两种方式进行注释，

- #一般用于描述性的注释，旨在说明代码的作用或怎么使用，
- ;分号通常用于示例性的注释，特别是在一些配置文件中常常会用到。

SHELL脚本书写的良好习惯

引用符号使用规范

Shell中有三种引用符号：双引号、单引号和反引号；

尽量少用单引号，对一个字符串多个特殊字符进行屏蔽特殊含义时才使用单引号；

对特殊字符进行屏蔽特殊含义时，使用反斜线进行屏蔽；

使用单引号屏蔽字符时，单引号内一般不使用其他引用符号，除非是打印特殊符号本身；

使用反引号进行执行一个shell命令时，反引号一般加其他引用符号，除非需要进行屏蔽特殊字符时才使用反斜线和单引号。