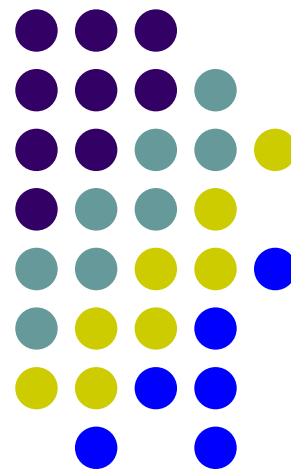
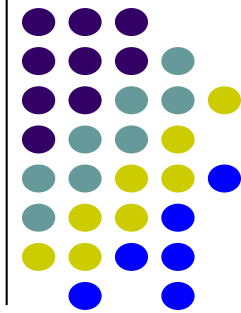


# 第11章

## Linux shell编程

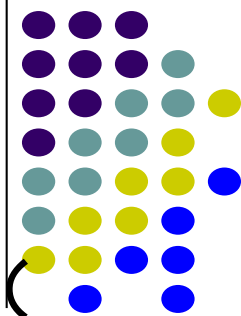


# 本章内容

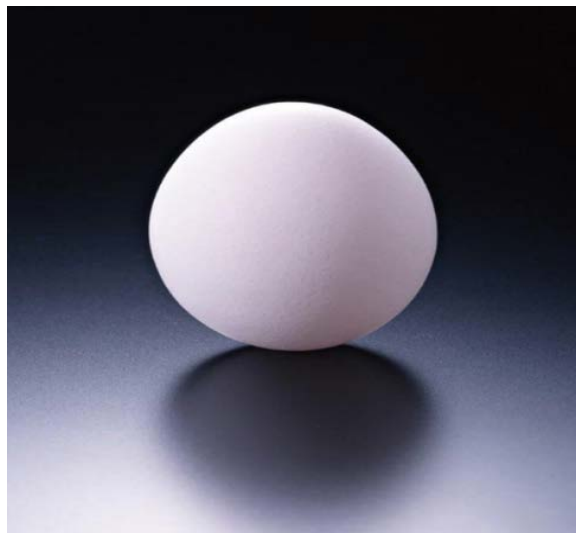


- **Shell 语言概述**
- **创建Shell程序**
- **Shell 变量**
- **Shell 表达式**
- **Shell的流程控制**
- **Shell函数**

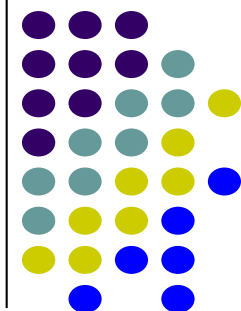
# 11.1 Shell 简介



- Shell是介于使用者和Linux 操作系统之核心程序（kernel）之间的一个接口。
- 为了对用户屏蔽内核的复杂性，也为了保护内核以免用户误操作造成损害，在内核的周围建了一个外壳(shell)。用户向shell提出请求，shell解释并将请求传给内核。
- 早期Shell主要用作命令解释器，经过不断扩充和发展，现在已是命令语言、命令解释器、程序设计语言的统称。

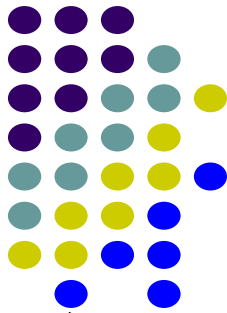


# Shell 简介



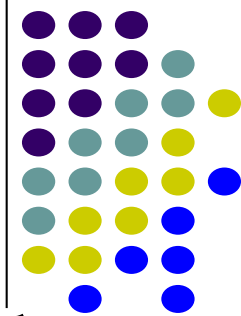
- Linux中的shell有多种类型，其中最常用的几种是 Bourne shell (sh)、C shell (csh) 和Korn shell (ksh)。三种shell各有优缺点。Bourne shell是UNIX最初使用的shell，并且在每种UNIX上都可以使用。Bourne shell在shell编程方面相当优秀，但在处理与用户的交互方面做得不如其他几种shell。Linux操作系统缺省的shell是Bourne Again shell，它是Bourne shell的扩展，简称**Bash**，与Bourne shell完全向后兼容，并且在Bourne shell的基础上增加、增强了很多特性。

# Shell的主要功能



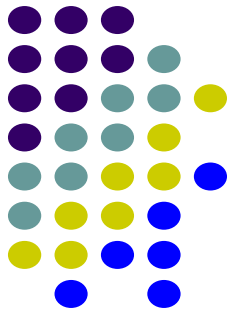
- 用户在与Shell交互时所输入的命令行必须符合Shell命令的语法和语义规范，才能被Shell理解并执行。前面章节介绍的都是基于单个命令行的交互执行方式。当用户登录到计算机系统时，会启动Shell程序，其基本功能是**解释并执行用户键入的各种命令**，把相应命令程序加载到主存，启动并运行。
- Shell也是一种**可编程的程序设计语言**。将若干个Shell命令行写入一个文件就构成了一个Shell程序，它可以被Shell逐条解释执行。
- Linux系统用Shell程序来实现系统的**初启、配置、管理和维护**等工作。

# Shell语言的特点



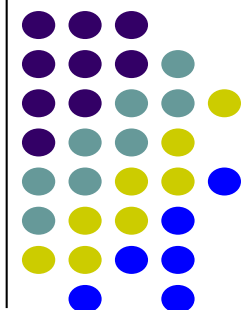
- 与其他编程语言相比，Shell语言具有如下特点：
  - (1) Shell是一种**解释性语言**。就是说，用Shell语言编写的程序不需要编译，可以直接由Shell进程解释执行。
  - (2) Shell是**基于字符串的语言**。Shell只是做字符串处理，不支持复杂的数据结构和运算。Shell的输出也全部是字符方式的。
  - (3) Shell是**命令级语言**。Shell程序全部由命令而不是语句组成，几乎所有的Shell命令和可执行程序都用来编写Shell程序。Shell命令丰富，命令的组合功能也十分强大。
- 另外需要**说明**的是，不同版本的Shell程序可能不完全兼容。

# Shell脚本



- 利用文本编辑器，事先把一系列Linux命令或可执行程序放到文件中，然后**修改文件的访问权限**，使之能够像系统命令或实用程序一样**执行**，这样的文本文件就是**Shell脚本**，或称Shell程序。简单地讲，Shell脚本就是一种包含若干Linux命令或可执行程序的文本文件。
- 当执行Shell脚本时，文件中的所有命令将从头到尾，一个一个地顺序执行（除非Shell脚本中含有控制结构语句）。就像用户在终端前，以命令行方式，每次输入一个命令，让系统依次执行一样。
- 通过使用shell**使大量的任务自动化**，shell脚本在处理自动循环或大的任务方面可节省大量的时间，且功能强大。

## 11.2 创建Shell程序



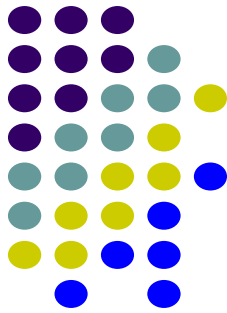
- 通常，Shell脚本的第一行均包含一个以“#!”为起始标志的文本行。指明系统用哪种Shell解释用户的Shell程序。“#!”特殊标志实际是一个文件类型标识码，表示当前的文件是一个可执行的Shell脚本文件。紧随“#!”标志的是一个路径文件名，指向用于执行当前Shell脚本文件的命令解释程序。

例如： `#!/bin/bash`

表示调用Linux系统默认的Shell，即Bash，来解释执行当前的Shell脚本。



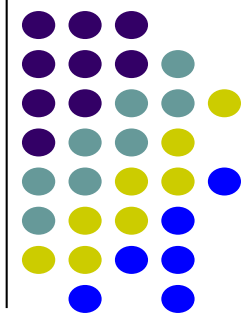
# Shell的基本元素



## Shell的基本元素：

- (1) `#!/bin/bash` 必须的，指出shell的类型
- (2) 注释行，使用“#”符号。
- (3) 变量
- (4) 控制

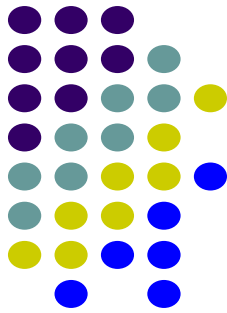
# 编写Shell脚本步骤



## 步骤：

- Shell脚本程序可以用vi等文字编辑器编写，将脚本命名为脚本功能.sh（一般以.sh为文件后缀，没有也能执行）；
- Shell脚本程序总是以#!/bin/bash开头，它通知使用系统上默认的shell解释器。
- 脚本程序编写完毕后，在运行前需要使用chmod命令，赋予该脚本文件可执行的权限。
- 运行脚本程序的命令./filename

# 一个简单的Shell程序



简单脚本示例：

```
vi ShowHello.sh
```

以下为脚本内容：

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

```
#ShowHello.sh
```

```
#Show hello
```

```
echo Hello World !
```

```
echo -n "Today is"
```

```
date "+%A, %B, %d, %Y."
```

保存，退出vi编辑器。

为脚本添加可执行权限：

```
[root@Linux root] # chmod +x ShowHello.sh
```

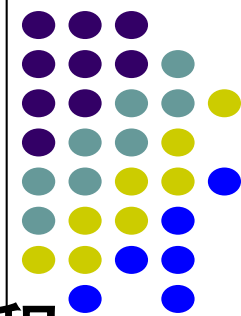
运行脚本

```
[root@Linux root] # ./ShowHello.sh
```

```
Hello World!
```

```
Today is 星期一 , 六月 11, 2018.
```

# echo命令



- linux的echo命令是一个标准输出命令, 在shell编程中极为常用;

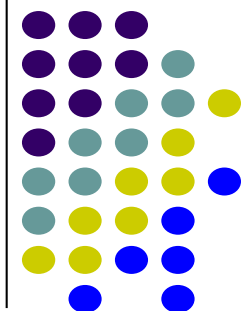
## echo命令

功能：在显示器上输出指定的字符串。

格式： **echo [-n] 字符串**

- 选项-n表示输出文字后不换行;
- 字符串可加引号, 也可不加引号。用echo命令输出加引号的字符串时, 将字符串原样输出; 用echo命令输出不加引号的字符串时, 将字符串中的各个单词作为字符串输出, 各字符串之间用一个空格分割。

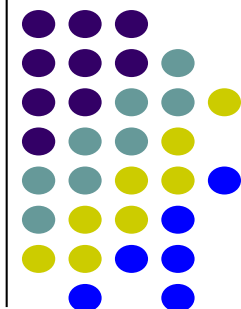
# 输入/输出



- **echo的命令选项**
  - **-n**            输出文字后不换行；
  - **-e**            解释转义字符；

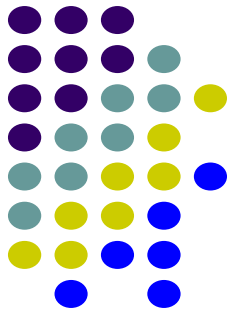
转义字符	含义
<b>\a</b>	<b>响铃</b>
<b>\b</b>	<b>删除前一个字符</b>
<b>\f</b>	<b>换行但光标仍旧停留在原来的位置</b>
<b>\n</b>	<b>换行且光标移至行首</b>
<b>\nnn</b>	<b>插入nnn（八进制）所代表的ASCII字符</b>
<b>\t</b>	<b>\插入tab;</b>

# echo -e 实例



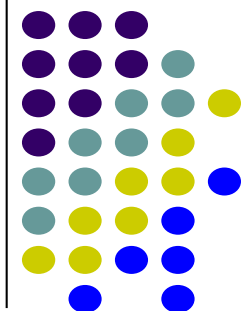
```
root@Linux2018:~  
文件(F) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)  
[root@Linux2018 root]# echo -e "A\bDDDD"  
DDDD  
[root@Linux2018 root]# echo -e "A\nDDDD"  
A  
DDDD  
[root@Linux2018 root]# echo -e "A\tDDDD"  
A      DDDD  
[root@Linux2018 root]# echo -e "A\101DDDD"  
AADDDD  
[root@Linux2018 root]#
```

# Shell特殊字符



- Shell定义了一些**特殊字符**，称为**元字符**，它们对Shell有特殊的含义。Shell在读入命令行后要先对命令行进行扫描，找出元字符进行相应的替换或处理，以确定要执行的程序和它的参数及执行方式等。
- Shell的元字符包括：文件通配符、输入/输出重定向及管道符、注释符、命令执行控制符、命令组合与替换符、转义符等。

# 命令执行控制符



命令执行控制符用于控制命令的执行方式，指示Shell何时该执行这个命令以及在何处（前台、后台）执行这个命令。

## 1. 顺序执行（；）

格式：命令1 ； 命令2

含义：顺序执行命令1和命令2。

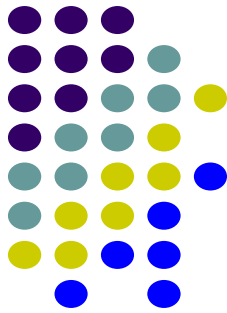
## 2. 逻辑与（&&）

格式：命令1 && 命令2

含义：“逻辑与”执行，若命令1执行成功则执行命令2；  
否则不执行命令2。



# 命令执行控制符



## 3. 逻辑或（ || ）

格式：命令1 || 命令2

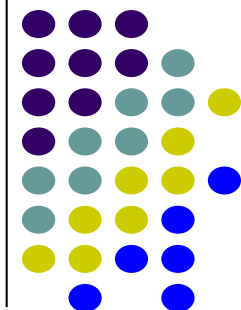
含义：“逻辑或”执行，若命令1执行成功则不执行命令2；  
否则继续执行命令2。

## 4. 后台命令（ & ）

格式：命令 &

含义：后台执行命令。

# 命令组合符



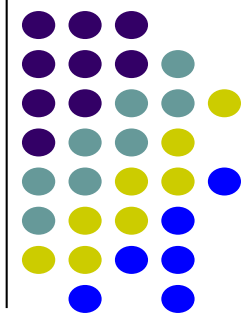
- **命令组合符**的作用是指示Shell将多个命令组合在一起执行。组合的目的是对这些命令统一进行某种操作。
- 命令的组合形式有两种：

**{命令； 命令； .....}**

**(命令； 命令； .....)**

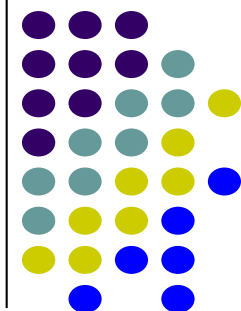
两种组合形式的区别在于前者只在本Shell中执行命令列表，不派生新的Shell子进程，命令执行的结果会影响当前的Shell环境；后者是派生一个新的子Shell进程来执行命令列表。命令在子Shell环境中执行，结果不会影响当前环境。

# 11.3 shell变量



- Shell提供了定义和使用变量的功能。用户可以将一些常用的数据存放在Shell变量中，并在命令行中引用这些变量。
- 变量分类
  - **环境变量**是为系统内核、系统命令和应用程序提供运行环境而设定的变量，由系统提供，不用定义，可以修改。常用的如PATH、HOME等。
  - **系统变量**是系统提供，不用定义，不能修改。
  - **用户变量**是在编写shell过程中定义的，可以在shell中任意修改。

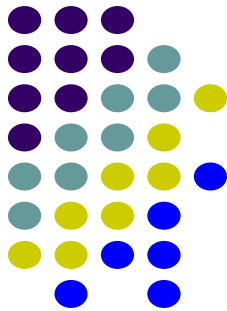
# 环境变量



- 当Shell程序启动时，会自动设置的一组变量叫环境变量，环境变量不随Shell程序的结束而消失。下表列举了常用的环境变量：

变量名	说明
\$HOME	设置用户默认的目录
\$PATH	设置命令搜索路径，以冒号分隔
\$LOGNAME	用户登录名
\$TERM	设置终端类型
\$LD_LIBRARY_PATH	寻找库的路径,以冒号分隔
\$SHELL	当前使用的Shell,也指出Shell的路径名
\$PS1	系统提示符，特权用户是“#”，普通用户是“\$”
\$PS2	设置系统的次提示符，通常是‘>’字符
\$IFS	输入区的分隔符

# 系统变量

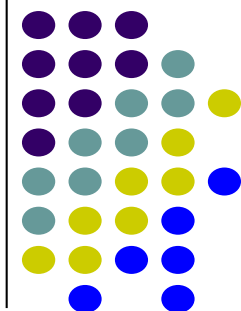


Shell常用的系统变量如下：

变量名	说明
\$n	\$1 表示第一个参数，\$2 表示第二个参数 ...
\$#	能提供给脚本的位置参数的个数
\$0	当前程序的名称
\$?	前一个命令或函数的返回码
\$*	以"参数1 参数2 ..."形式保存所有参数
\$@	以"参数1" "参数2" ... 形式保存所有参数
!	上一个后台命令的进程号
\$\$	当前Shell脚本的进程号

程序举例：sysvar.sh

# 系统变量



- **shift命令用于移动位置参数，比如将\$2中的内容移动到\$1，\$3中的内容移至\$2，\$4中的内容移至\$3，以此类推，但位置参数\$0不参与移位。**
- **shift命令的格式：shift [ n ]**  
选项n表示向左移动参数的个数，默认为1。

**vi movelab1**

**#!/bin/bash**

**# this program show shift**

**echo \$0**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift**

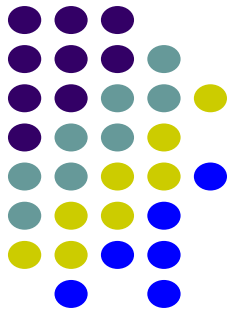
**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift**

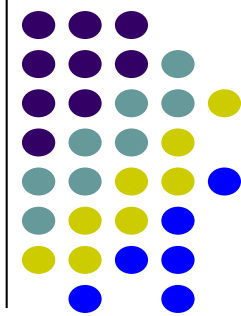
**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9**

**执行此程序，其输出如下：**



```
root@Linux2018:~
文件(F)  编辑(E)  查看(V)  终端(T)  转到(G)  帮助(H)
[root@Linux2018 root]# vi movelab1
[root@Linux2018 root]# chmod +x movelab1
[root@Linux2018 root]# ./movelab1 11 22 33 44 55 66 77 88 99
./movelab1
11,22,33,44,55,66,77,88,99
22,33,44,55,66,77,88,99,
33,44,55,66,77,88,99,,
44,55,66,77,88,99,,,
55,66,77,88,99,,,,
66,77,88,99,,,,,
77,88,99,,,,,,
88,99,,,,,,,
99,,,,,,,,
[root@Linux2018 root]#
```





**vi movelab2**

**#!/bin/bash**

**# this program show shift**

**echo \$0**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift 2**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift 2**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift 2**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift 2**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift 2**

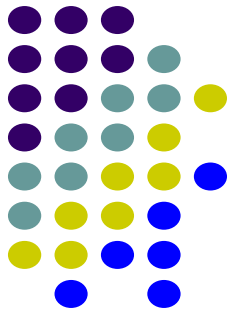
**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift 2**

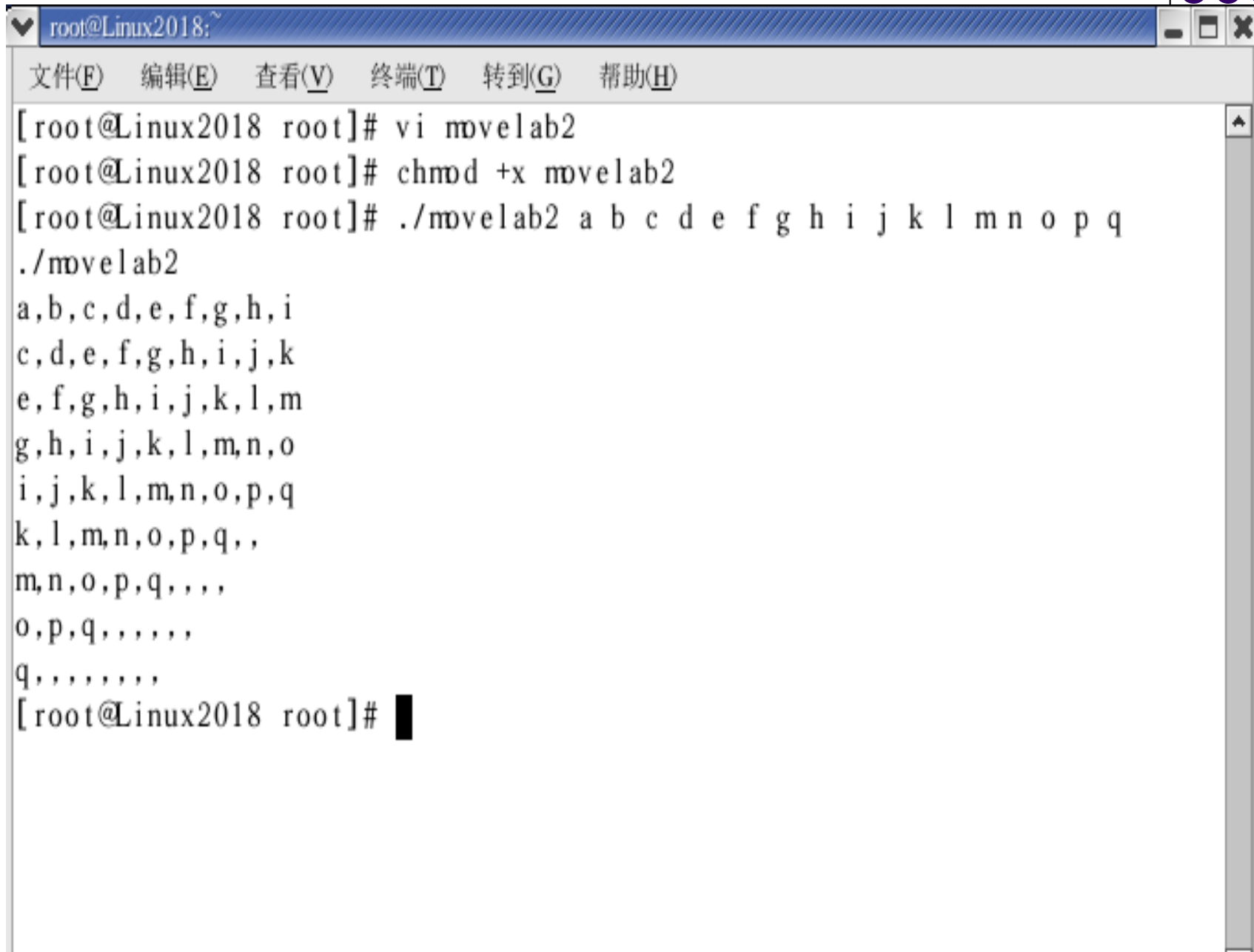
**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9;shift 2**

**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9**

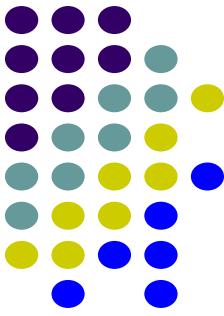
**echo \$1,\$2,\$3,\$4,\$5,\$6,\$6,\$7,\$8,\$9**

**执行此程序，其输出如下：**





```
root@Linux2018:~
文件(F) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)
[root@Linux2018 root]# vi movelab2
[root@Linux2018 root]# chmod +x movelab2
[root@Linux2018 root]# ./movelab2 a b c d e f g h i j k l m n o p q
./movelab2
a,b,c,d,e,f,g,h,i
c,d,e,f,g,h,i,j,k
e,f,g,h,i,j,k,l,m
g,h,i,j,k,l,m,n,o
i,j,k,l,m,n,o,p,q
k,l,m,n,o,p,q,,
m,n,o,p,q,,,
o,p,q,,,,,
q,,,,,,,
[root@Linux2018 root]#
```



```
vi sysvar.sh
```

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

```
# this script explains how the system variable works
```

```
echo "The name of this program is $0"
```

```
echo "You've input $# parameters. They are $*"
```

```
echo "And the first one them is $1"
```

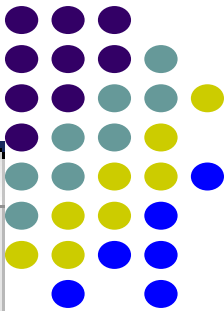
```
echo "And the second one them is $2"
```

```
echo "The PID of this program is $$"
```

```
echo "..."
```

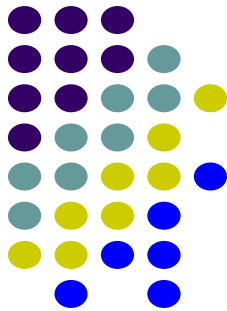
```
echo "You've excuted correctly,and return code is $?"
```

**其输出如下：**



```
root@Linux2018:~  
文件(F) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)  
[root@Linux2018 root]# ./sysvar.sh hello world  
The name of this program is ./sysvar.sh  
You've input 2 parameters. They are hello world  
And the first one them is hello  
And the second one them is world  
The PID of this program is 4051  
...  
You've excuted correctly,and the return code is 0  
[root@Linux2018 root]#
```

# 用户变量



## 1. 变量名

- 命令规则：以字母或下划线开头，由字母、数字、下划线组成，**第一个字符不能为数字，变量名区分大小写。**

## 2. 变量赋值

Shell下的变量无需声明，赋值的同时即声明了变量。

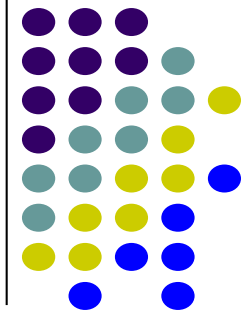
- 格式：  
变量名=字符串

例如：x=6

a="hello world"

- 注意：**赋值号**两边不能有空格**，如果字符串中含有空格，应用引号将字符串括起来。

# 用户变量



## 3. 引用变量值

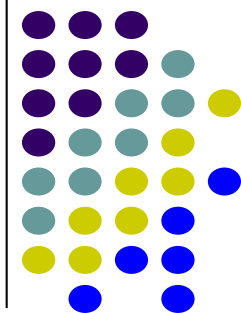
- 访问shell变量的值，必须在变量名前面加\$符号： **\$变量名**
- 如果变量名包含在其他字符串中，这时就需要用{ }将变量名括起来。
- 例如：

```
[root@ Linux root] # a=good
```

```
[root@ Linux root] # echo "${a}morning"  
goodmorning
```

- 为了避免变量名混淆，建议总是使用{ }将变量名括起来。

# 用户变量



## 4. 变量清除

如果所设置的变量不需要时可以清除。

- 格式：

**unset 变量名**

例如：

```
[root@ Linux root] # a=good
```

```
[root@ Linux root] # echo "${a}morning"
```

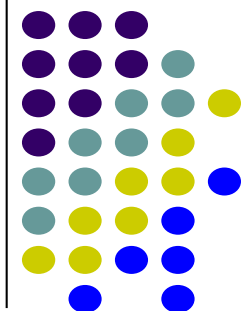
```
goodmorning
```

```
[root@ Linux root] # unset a
```

```
[root@ Linux root] # echo "${a}morning"
```

```
morning
```

# 用户变量



- 如果赋值后不希望改变变量，使其类似常量，可以使用 `readonly` 命令将其设为只读。例如：  

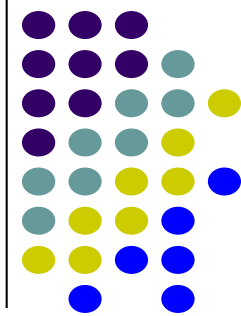
```
[root@ Linux root] # a=good
```

```
[root@ Linux root] # readonly a
```

 //或直接在赋值时设为只读  
此时，若要用 `unset` 重置变量 `a` 或者对 `a` 重新赋值，则会产生错误。
- 另外，Shell 的变量默认为全局作用的，如果需要在一定范围内生效，则需要加上 `local` 限制，使用命令“`local a`”就将设置 `a` 为局部变量。



# read命令

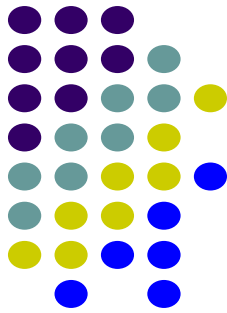


## read 命令

在shell程序设计中，变量的值可以作为字符串从键盘读入，然后赋给指定的变量。

### 格式：read 变量

- 通常用在shell脚本中与用户进行交互的场合。该命令可以一次读取多个变量的值，变量和输入的值都需要使用空格隔开。
- 如果指定的变量名少于字段数量，则多出的字段数量分配给最后一个变量，如果指定的变量命令多于字段数量，则多出的变量赋值为空。



例如：用read命令读入变量的值。

```
[root@ Linux root] # read var1 var2 var3 var4  
1 22 333 4444 55555
```

```
[root@ Linux root] # echo $var1 $var2 $var3  
1 22 333
```

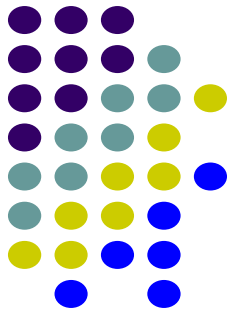
```
[root@ Linux root] # echo $var4  
4444 55555
```

```
[root@ Linux root] # read var1 var2 var3 var4  
1 22 333
```

```
[root@ Linux root] # echo $var1 $var2 $var3  
1 22 333
```

```
[root@ Linux root] # echo $var4
```

```
[root@ Linux root] #
```



例如，向某人问好：

```
vi Hello.sh
```

以下为脚本内容：

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

```
#Hello.sh
```

```
#To show hello to somebody
```

```
echo -n "Enter Your Name:"
```

```
read NAME
```

```
echo "Hello, $NAME!"
```

保存，退出vi编辑器。

为脚本添加可执行权限：

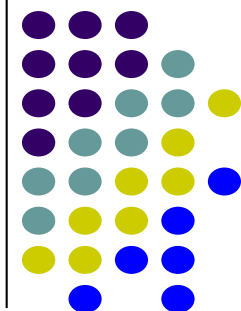
```
[root@Linux root] # chmod +x Hello.sh
```

```
[root@Linux root] # ./Hello.sh
```

```
Enter Your Name:lily
```

```
Hello,lily!
```

# 11.4 表达式的比较



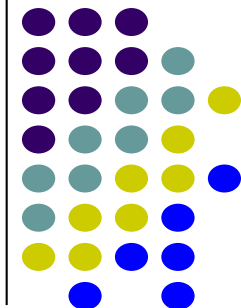
## 1.字符串比较

作用：测试字符串是否相等、长度是否为零，字符串是否为NULL。为真返回值为0，为假返回值为1。

常用的字符串操作符：

字符串操作符	含义及返回值
=	比较两个字符串是否相同，相同则为"真"
!=	比较两个字符串是否相同，不同则为"真"
-n	比较字符串长度是否大于零，如果大于零则为"真"
-z	比较字符串的长度是否等于零，如果等于则为"真"

# 表达式的比较



例如，从键盘读入两个字符串，判断这两个字符串是否相等。

(1)用vi编辑程序

```
[root@ Linux root] # vi test1
```

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

```
read ar1
```

```
read ar2
```

```
[ "$ar1" = "$ar2" ]
```

```
echo $?
```

#保存前一个命令的返回码

(2)设置权限:

```
[root@localhost bin] # chmod +x test1
```

(3)执行:

```
[root@localhost root] # ./test1
```

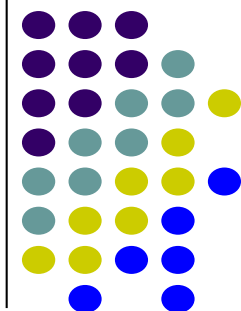
```
aaa
```

```
bbb
```

```
1
```

注意：“[”后面和“]”前面及等号“=”的前后都应有一空格；

# 表达式的比较

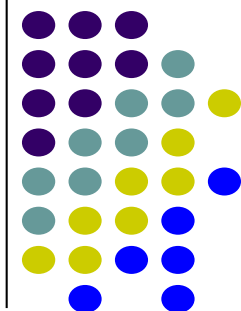


## 2.数字比较

在bash shell编程中的关系运算有别于其他编程语言。

运算符	含义
-eq	相等
-ge	大于等于
-le	小于等于
-ne	不等于
-gt	大于
-lt	小于

# 表达式的比较



例如，比较两个数字是否相等。

(1)用vi编辑程序

```
[root@ Linux root] # vi test2
```

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

```
read x y
```

```
if test $x -eq $y
```

```
then
```

```
    echo "$x= =$y"
```

```
else
```

```
    echo "$x!= $y"
```

```
fi
```

(2)设置权限:

```
[root@ Linux root] # chmod +x test2
```

(3)执行:

```
[root@ Linux root] # ./test2
```

```
50 100
```

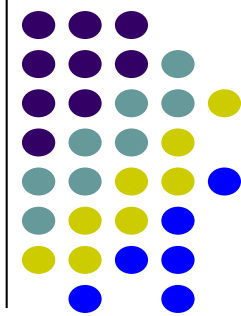
```
50!=100
```

```
[root@ Linux root] # ./test2
```

```
150 150
```

```
150= =150
```

# test命令



- 对条件进行判断，就需要使用test命令。test命令被用来判断表达式并且产生返回值。表达式为真，返回值为0，表达式为假，返回值为1。test命令可对整数、字符串、以及文件进行判断，其使用方法如下：

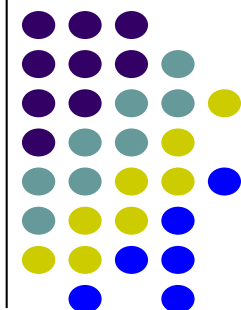
**test 表达式**

**或者**

**[表达式]**



# 表达式的比较



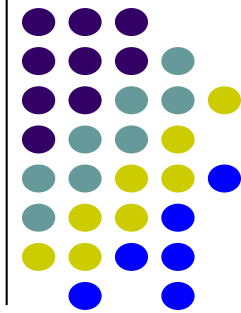
## 3.文件操作

文件测试表达式通常是为了测试文件的信息，一般由脚本来决定文件是否应该备份、复制或删除。

常用操作符：

运算符号	含义
-d	对象存在且为目录返回值为"是"
-f	对象存在且为文件返回值为"是"
-L	对象存在且为符号连接返回值为"是"
-r	对象存在且可读则返回值为"是"
-s	对象存在且长度非零则返回值为"是"
-w	对象存在且可写则返回值为"是"
-x	对象存在且可执行则返回值为"是"

# 表达式的比较



例如，判断zb目录是否存在/root下。

(1)用vi编辑程序

```
[root@ Linux root] # vi test3
#!/bin/bash
[ -d /root/zb ]
echo $?
```

#保存前一个命令的返回码

(2)设置权限：

```
[root@ Linux root] # chmod +x test3
```

(3)执行：

```
[root@ Linux root] # ./test3
1
```

(4) 在/root添加zb目录

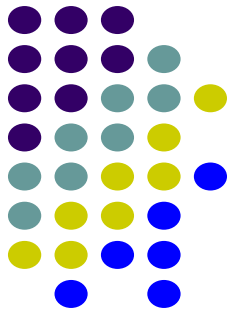
```
[root@ Linux root] # mkdir zb
```

(5)执行：

```
[root@ Linux root] # ./test3
0
```

**注意：**运行结果1表示判断的目录不存在，0表示判断的目录存在。

# 算术运算



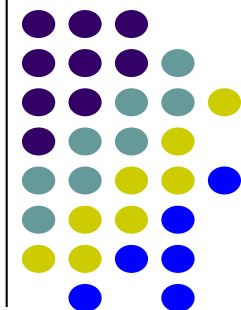
## 算术运算：**expr**命令

功能：该命令提供算术运算功能，并能对数字或非数字字符串进行计算。

说明：

- 字符\*（乘）和%（取余）在shell中有特殊含义，因此他们的前面必须有转义字符“\”  
如：`expr $a \* $b`
- 当有变量参与运算时，需要在变量名前面加“\$”

# 算术运算



- 说明：expr命令可以作关系运算
  - 当比较结果为真时，expr命令显示1；
  - 当比较结果为假时，expr命令显示0。

关系运算符	含义
=	等于
!=	不等于
<	小于
<=	小于等于
>	大于
>=	大于等于

例如，运行程序test8,从键盘读入x、y的值，然后做加法运算，最后输出结果。

(1)用vi编辑程序

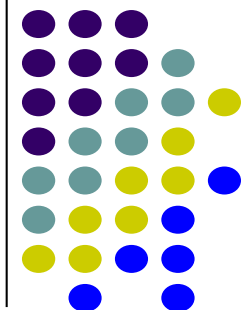
```
[root@ Linux root] # vi test8
#!/bin/sh
read x y
a=`expr $x + $y`
b=`expr $x \* $y`
echo "The sum is $a"
echo "The product is $b"
```

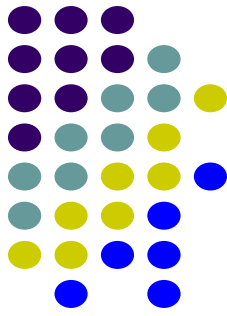
(2)设置权限：

```
[root@ Linux root] # chmod +x test8
```

(3)执行：

```
[root@ Linux root] # ./test8
45 78
The sum is 123
The product is 3510
```





例如，将一个Shell命令的输出作为另一个命令的参数。

（包含倒双引号，则要作命令替换）

```
[root@ Linux root] # echo “ Now my work directory is `pwd` ”
```

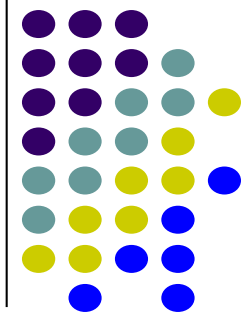
```
Now my work directory is /root
```

```
[root@ Linux root] # time=`date`
```

```
[root@ Linux root] # echo $time
```

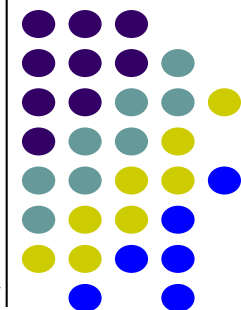
```
— 6月 11 15: 30: 37 CST 2018
```

# 11.5 Shell的流程控制



- 和其他编程语言相似，Shell编程也可以使用分支  
结循环结构的流程控制语句。
  - 分支结构：if、case
  - 循环结构：while、until、for
  - 循环控制：break、continue
  - 结束：return、exit

# 条件结构语句



**If 命令：**根据条件命令执行的结果决定后续命令的执行路径。

**语法：**

**if 条件命令**

**then 命令列表1**

**# 若\$? 为0，执行此分支**

**[ else 命令列表2 ]**

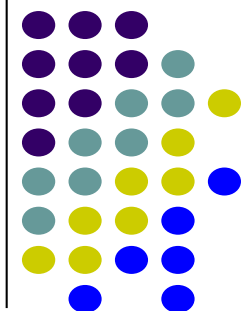
**# 否则，执行此分支**

**fi**

- Linux里的if的**结束标志**是将if反过来写成**fi**。
- If命令的执行过程是，首先执行条件命令，然后根据条件命令的退出状态作为条件决定后续的执行路径。若条件为真则执行命令列表1，否则执行命令列表2。以上是if语句最**基本的格式**。



# 条件结构语句



- If其它格式

If 条件命令

then

命令列表1

#可以是多条语句

[elif 条件2

then

命令列表2 #可以是多条语句]

...

[else

命令列表3

#可以是多条语句 ]

fi

注意: elif是else if的缩写, 并且省略了fi

例如，输入一个字符串，如果是目录，则显示目录下的信息，如果为文件则显示文件的内容。

```
[root@ Linux root] # vi test9
```

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

```
echo "Please enter the directory name or file name"
```

```
read DORF
```

```
if [ -d $DORF ]
```

```
then
```

```
    ls $DORF
```

```
elif [ -f $DORF]
```

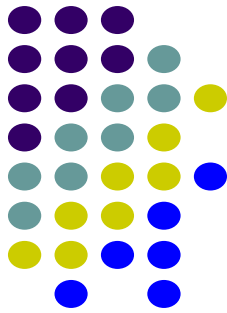
```
then
```

```
cat $DORF
```

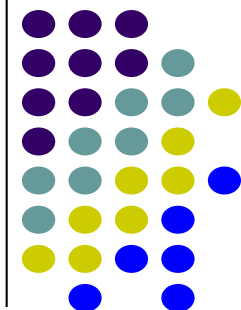
```
else
```

```
    echo "input error!"
```

```
fi
```



# 条件结构语句



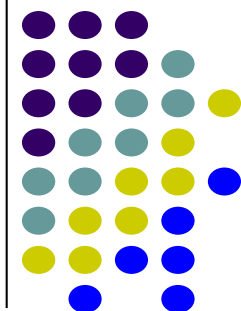
**case命令**是一个多选择语句，根据变量与哪种模式匹配即执行相应的语句序列。

**语法：**

```
case 测试字符串 in
模式1) 命令列表1; ;
模式2) 命令列表2; ;
...
模式n) 命令列表n; ;
esac
```

- 每个分支条件必须以两个分号 “; ;” 结尾。
- case命令的执行过程，先将测试字符串与各个模式字符串逐一比较，若发现了一个匹配的模式则执行该模式对应的命令列表。如果都不匹配，可用 “\*” 代替，相当于 default。

**注意：**若有多个匹配的模式时，只执行最前面的那个分支。



例如：Linux是一个多用户操作系统，编写一程序根据不同的用户登录输出不同的反馈结果。

(1) 用vi编辑脚本程序test\_b

```
[root@ Linux root] # vi test_b
```

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

```
case $USER in
```

```
zhangsan)
```

```
    echo "You are zhangsan!";;
```

```
lisi)
```

```
    echo "You are lisi";    //注意这里只有一个分号
```

```
    echo "Welcome!";;      //这里才是两个分号
```

```
root)
```

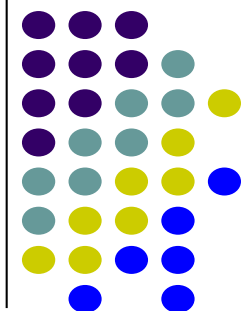
```
    echo "You are root! "; echo "Welcome!";;
```

```
    //将两命令写在一行，用一个分号作为分隔符
```

```
*)
```

```
    echo "Who are you?$USER?";;
```

```
esac
```



**(2)设置权限:**

```
[root@ Linux root] # chmod +x test_b
```

**(3)执行:**

```
[root@ Linux root] # ./test_b
```

**You are root**

**Welcome!**

# 循环结构语句



for 循环

语法:

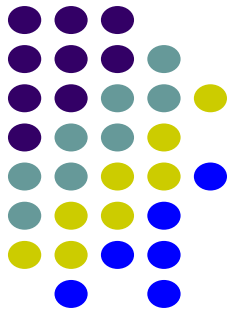
```
for 变量 [in 字符串列表]
```

```
do
```

```
    命令列表
```

```
done
```

- for命令执行的过程是，定义一个变量，它依次取字符串列表中的各个字符串的值。对每次取值都依次执行命令列表，直到所有的字符串都处理完。当没有指定字符串列表时，默认指脚本的参数列表，即for i 等同于 for i in “\$@”。



例如，在列表中的值：a,b,c,e,i,2,4,6,8用循环的方式把字母与数字分成两行输出。

(1)编辑脚本程序test\_c

```
[root@ Linux root] # vi test_c
#!/bin/bash
for i in a,b,c,e,i 2,4,6,8
do
echo $i
done
```

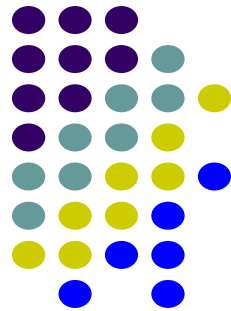
注意：在循环列表中的空格可表示换行。

(2)设置权限：

```
[root@ Linux root] # chmod +x test_c
```

(3)执行：

```
[root@ Linux root] # ./test_c
a,b,c,e,i
2,4,6,8
```



例如：求从1到100的和。

(1) 编辑脚本程序test5

```
[root@ Linux root] # vi test5
```

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

```
total=0
```

```
for ((j=1;j<=100;j++));
```

```
do
```

```
total=`expr $total + $j`
```

```
done
```

```
echo "The result is $total"
```

注意：for语句中的双括号不能省，最后的分号可有可无，表达式total=`expr \$total + \$j`的加号两边的空格不能省，否则会成字符串的连接。

(2) 设置权限：

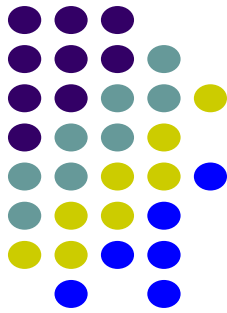
```
[root@ Linux root] # chmod +x test5
```

(3) 执行：

```
[root@ Linux root] # ./test5
```

```
The result is 5050
```





**例如：用for循环输出1到10间的奇数。**

**(1) 编辑脚本程序test\_a**

```
[root@ Linux root] # vi test_a  
#!/bin/bash  
for((j=0;j<=10;j++))  
do  
if(($j%2==1))  
then  
echo "$j"  
fi  
done
```

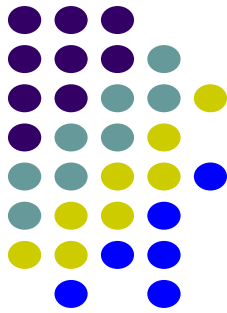
**(2)设置权限：**

```
[root@ Linux root] # chmod +x test_a
```

**(3)执行：**

```
[root@ Linux root] # test_a  
1  
3  
5  
7  
9
```

# 循环结构语句



## while循环

### 语法：

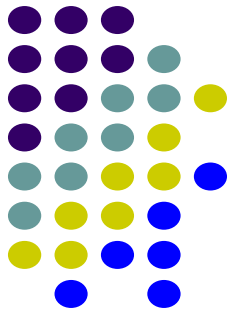
**while** 条件命令

**do**

命令列表

**done**

- **while**命令的作用是进行有条件的循环控制，只要**while**表达式为真，**do**和**done**之间的操作就一直在进行。



例如：用while循环求1到100的和。

(1)编辑脚本程序test\_e

```
[root@ Linux root] # vi test_e
total=0
num=0
while((num<=100));
do
total=`expr $total + $num`
((num+=1))
done
echo "The result is $total"
```

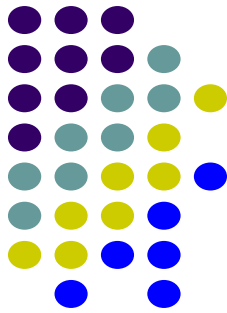
(2)设置权限：

```
[root@ Linux root] # chmod +x test_e
```

(3)执行：

```
[root@ Linux root] # ./test_e
The result is 5050
```

# 循环结构语句



## until循环

语法：

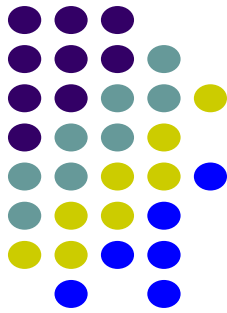
until 条件命令

do

命令列表

done

- until命令与while类似，区别在于while是当判断条件为真时执行循环，而until循环在判断条件为假时才停止循环。



例如:用until循环求1到100的和。

(1)编辑脚本程序test\_f

```
[root@ Linux root] # vi test_f
```

```
total=0
```

```
num=0
```

```
until [ $num -gt 100 ]
```

```
do
```

```
total=`expr $total + $num`
```

```
num=`expr $num + 1`
```

```
done
```

```
echo "The result is $total"
```

(2)设置权限:

```
[root@ Linux root] # chmod +x test_f
```

(3)执行:

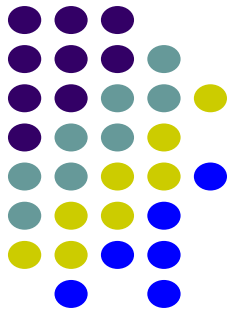
```
[root@ Linux root] # ./test_f
```

```
The result is 5050
```

# 退出循环语句



- **break**和**continue**命令的作用与C语言中的**break**和**continue**语句相同，用于在必要时跳出循环。
- **break**用于终止整个循环，**continue**用于终止本轮循环，直接进入下一轮循环。
- 另外，**break**和**continue**命令只能应用在**for**、**while**和**until**命令的循环体内。
- **exit**命令用于终止Shell脚本程序并返回值。该值可用“\$?”在下一条命令中获取。通常情况下正常执行的程序将返回0，而未正常结束的程序则返回1～255间的错误代码。



```
vi test_break
```

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

```
# this is break test script
```

```
i=1
```

```
for day in Mon Tue Wed Thu Fri
```

```
do
```

```
    echo "Weekday $((i++)) : $day"
```

```
    if [ $i -eq 3 ]; then
```

```
        break
```

```
    fi
```

```
done
```

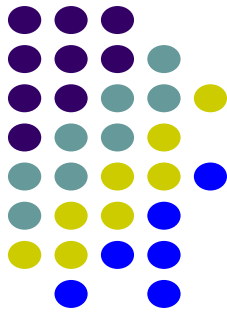
```
设置权限: [root@ Linux root] # chmod +x test_break
```

```
执行: [root@ Linux root] # ./test_break
```

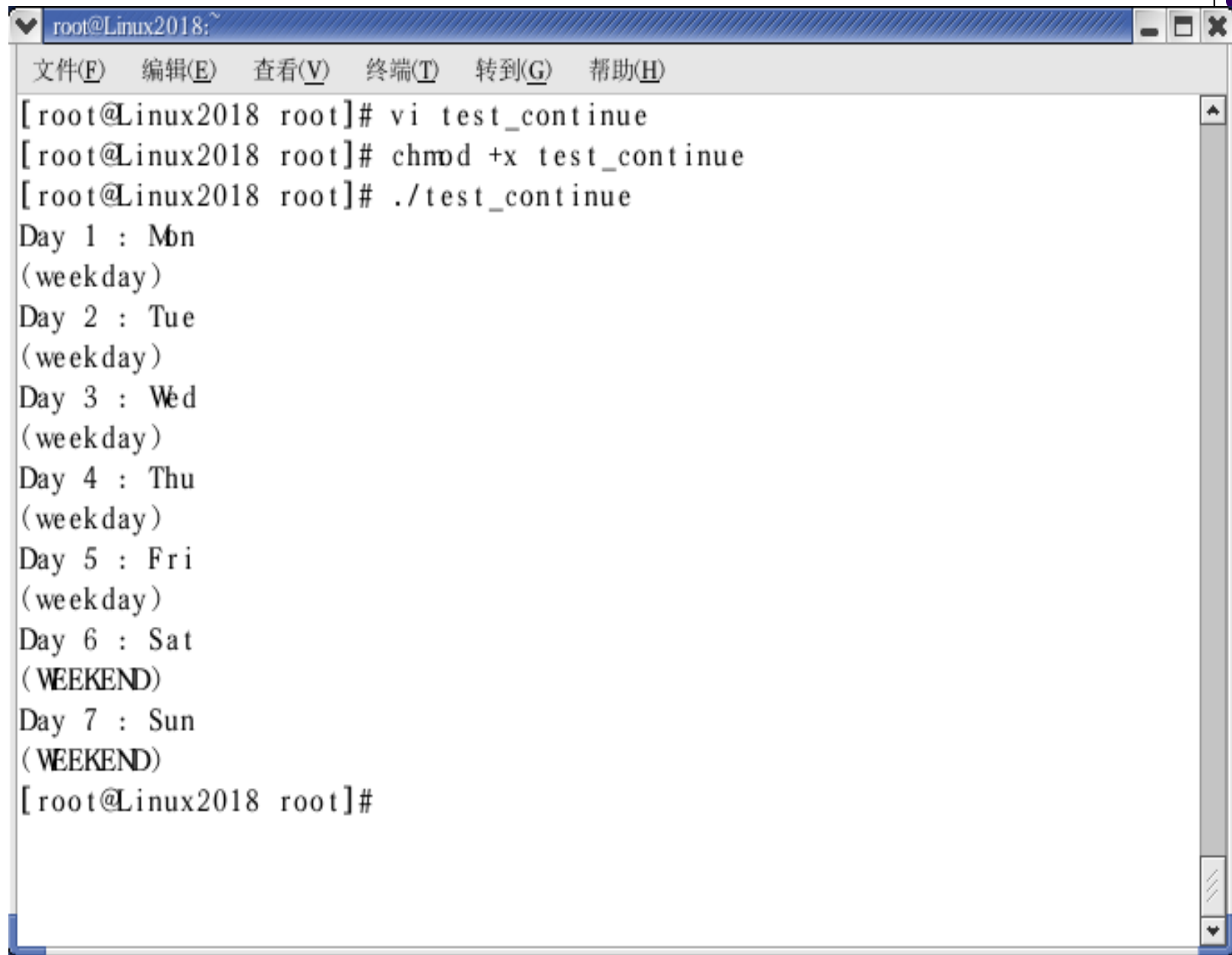
```
Weekday 1: Mon
```

```
Weekday 2: Tue
```

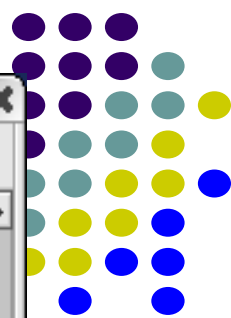
```
vi test_continue
#!/bin/bash
# this is continue test script
i=1
for day in Mon Tue Wed Thu Fri Sat Sun
do
    echo -n "Day $((i++)) : $day"
    if [ $i -eq 7 -o $i -eq 8 ]; then
        echo " (WEEKEND)"
        continue
    fi
    echo " (weekday)"
done
```





A terminal window titled 'root@Linux2018:~' with a menu bar containing '文件(F)', '编辑(E)', '查看(V)', '终端(T)', '转到(G)', and '帮助(H)'. The terminal displays the execution of a script named 'test\_continue'. The script prints the days of the week from Monday to Sunday, with weekdays labeled as '(weekday)' and weekends labeled as '(WEEKEND)'.

```
root@Linux2018:~  
文件(F) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)  
[root@Linux2018 root]# vi test_continue  
[root@Linux2018 root]# chmod +x test_continue  
[root@Linux2018 root]# ./test_continue  
Day 1 : Mbn  
(weekday)  
Day 2 : Tue  
(weekday)  
Day 3 : Wed  
(weekday)  
Day 4 : Thu  
(weekday)  
Day 5 : Fri  
(weekday)  
Day 6 : Sat  
(WEEKEND)  
Day 7 : Sun  
(WEEKEND)  
[root@Linux2018 root]#
```



例如：

```
vi test_d
```

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

```
for age in 58 14 -25 26
```

```
do
```

```
if [ $age -lt 0 ];
```

```
then
```

```
echo "$age is not a valid age. Exit ..."
```

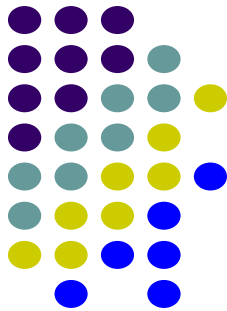
```
exit 100
```

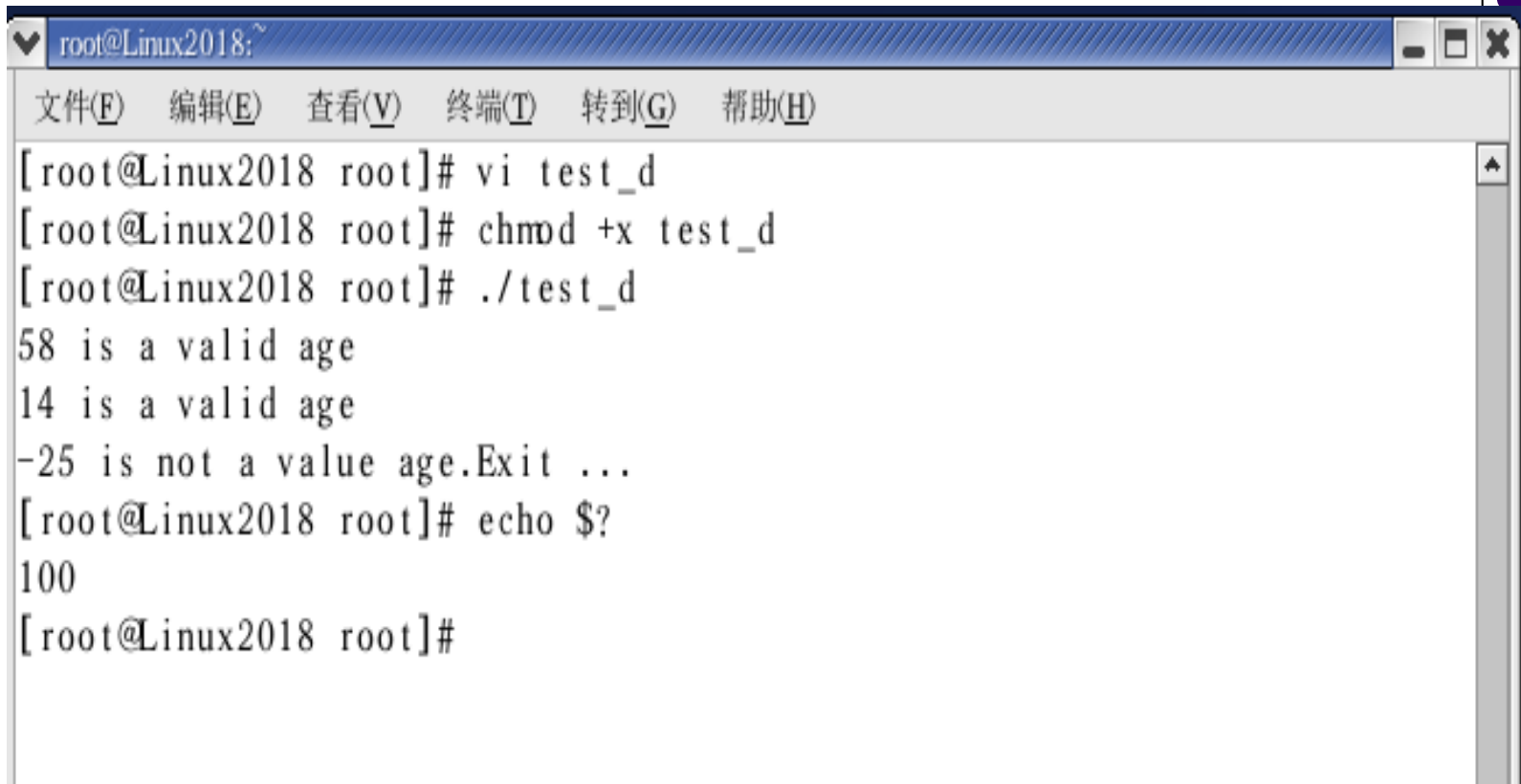
```
else
```

```
echo "$age is a valid age"
```

```
fi
```

```
done
```



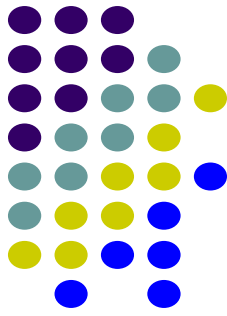
A terminal window titled 'root@Linux2018:~' with a menu bar containing '文件(F)', '编辑(E)', '查看(V)', '终端(T)', '转到(G)', and '帮助(H)'. The terminal shows the execution of a script 'test\_d' using 'vi', followed by 'chmod +x test\_d' and './test\_d'. The script outputs three lines: '58 is a valid age', '14 is a valid age', and '-25 is not a value age.Exit ...'. Then, the command 'echo \$?' is executed, resulting in the output '100'.

```
root@Linux2018:~  
文件(F) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)  
[root@Linux2018 root]# vi test_d  
[root@Linux2018 root]# chmod +x test_d  
[root@Linux2018 root]# ./test_d  
58 is a valid age  
14 is a valid age  
-25 is not a value age.Exit ...  
[root@Linux2018 root]# echo $?  
100  
[root@Linux2018 root]#
```

执行该脚本，到-25处程序就停止并跳出运行。执行“echo \$?”命令就会显示“100”。

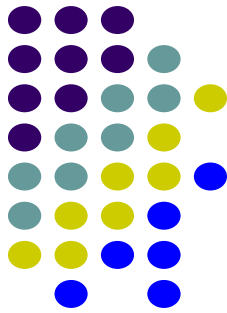
通常情况下，正常执行的程序将返回0，而未正常结束的程序则返回1~255间的错误代码。

# 11.6 shell函数



- shell程序也支持函数。函数能完成一个特定的功能，可以重复调用这个函数。Shell函数的名字必须唯一，且符合变量命名规则。函数必须声明：
- 函数格式如下：  
函数名 ( )  
{  
    函数体  
}
- 函数调用方式为：  
函数名 参数列表

# shell函数



例如，编写一函数add求两个数的和,这两个数用位置参数传入，最后输出结果。

## (1)编辑代码

```
[root@ Linux root] # vi test_g
#!/bin/bash
add( )
{
  a=$1
  b=$2
  z=`expr $a + $b`
  echo "The sum is $z"
}
add $1 $2
```

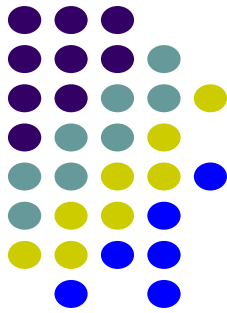
注意：函数定义完成后必须同时写出函数的调用，然后对此文件进行权限设定，再执行此文件。

## (2)修改权限

```
[root@ Linux root] # chmod +x test_g
```

## (3)程序运行结果

```
[root@ Linux root] # ./test_g 10 20
The sum is 30
```



例如，定义一个简单的函数

(1)编辑代码

```
[root@ Linux root] # vi test_h
```

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

```
sayhello ( ) {
```

```
    for name in $@; do
```

```
        echo "Hello ${name} !"
```

```
done
```

```
}
```

```
sayhello Lily Tom Candy
```

(2)修改权限

```
[root@ Linux root] # chmod +x test_h
```

(3)程序运行结果

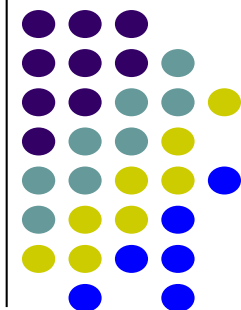
```
[root@ Linux root] # ./test_h
```

```
Hello Lily !
```

```
Hello Tom!
```

```
Hello Candy!
```

# 在shell脚本中调用其他shell脚本



在shell脚本的执行过程中，shell脚本支持调用另一个shell脚本，调用的格式为：

**程序名**

例：在shell脚本test\_1中调用test\_2

(1)编辑代码

```
#test_1脚本
```

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

```
echo "The main name is $0"
```

```
./test_2
```

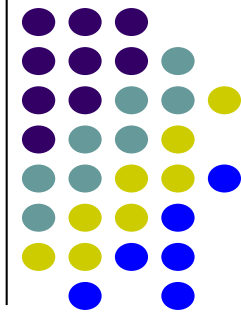
```
echo "The first string is $1"
```

```
#test_2脚本
```

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

```
echo "How are you $USER?"
```

# 在shell脚本中调用其他shell脚本



## (2)修改权限

```
[root@ Linux root] # chmod +x test_1
```

```
[root@ Linux root] # chmod +x test_2
```

## (3)程序运行结果

```
[root@ Linux root] # ./test_1 abc123
```

```
The main name is ./test_1
```

```
How are you root?
```

```
the first string is abc123
```