第2天-Shell流程控制-函数编程

一、Shell 编程之条件结构

1、Shell 条件测试语法

test 测试表达式	利用test命令进行条件测试表达式，test命令与测试表达式之间至少有一个空格
[测试表达式]	通过[ ]中括号进行条件测试表达式，[]中括号边界与测试表达式之间至少有一个空格
[[测试表达式]]	通过[[ ]]双中括号进行条件测试表达式，[[ ]]双中括号与测试表达式之间至少有一个空格
((测试表达式))	通过(( ))双小括号进行条件测试表达式，( ))双小括号两端不需要空格，常用于整数对比

1、符号说明

```
(())    数值比较，运算  C语言
[[]]    条件测试，支持正则
$(())   整数运算

$[]     整数运算
$()     命令替换 shell会先执行括号的cmd，然后将结果作为变量进行替换，替换只能替换标准输出，错误输出不能替换。
${}     Shell中变量的原形，用于限定变量名称的范围，并且支持通配符

[]      条件测试
()      子shell中执行
{}      在当前 shell 执行


# ()是重新开一个子shell然后执行，而{}则是在当前shell里执行。
# ()最后一个命令可以不用分号，{}最后一个命令要用分号。
# ()里第一个命令和左边括号不必有空格，而{}第一个命令和左括号之间必须有一个空格。
# ()和{}里的某个命令的重定向只影响该命令， 而括号外的重定向则影响到括号里的所有命令。
[root@qfedu.com ~]# var=test
[root@qfedu.com ~]# echo var
var
[root@qfedu.com ~]# echo $var
test
[root@qfedu.com ~]# (var=notest;echo $var)
notest
[root@qfedu.com ~]# {var=notest;echo $var}
{var=notest: command not found
test}
[root@qfedu.com ~]# {var=notest;echo $var;}
```

```
-su: syntax error near unexpected token `}'
[root@qfedu.com ~]# { var=notest;echo $var;}
notest
[root@qfedu.com ~]# echo $var
notest
# {}修改了变量的值。表明在当前shell中运行的
[root@qfedu.com ~]# var=test
[root@qfedu.com ~]# echo $var
test
[root@qfedu.com ~]# (var=notest;echo $var)
notest
[root@qfedu.com ~]# echo $var
test
#  () 里的执行完毕后没有改变变量的值，说明在子shell中执行的

# $(( ))和$[ ]的用途一致,用来作整数运算。在 bash 中，$(( ))的整数运算符号大致有这些:
# + - * / 加、减、乘、除
# % 余数运算
# & | ^ ! AND、OR、XOR、NOT运算
```

举例:

```
[root@qfedu.com ~]# a=5; b=7; c=2
[root@qfedu.com ~]# echo $((a+b*c))
[root@qfedu.com ~]# echo $[a+b*c]
19

[root@qfedu.com ~]# echo $(((a+b)/c))
[root@qfedu.com ~]# echo $[(a+b)/c]
6

[root@qfedu.com ~]# echo $(((a*b)%c))
[root@qfedu.com ~]# echo $[(a*b)%c]
1

# $(( ))中的变量名称也可以在其前面加 $ 符号: $(($a+$b*$c))也可以得到 19 的结果。

# $(( ))还可以作不同进制(如二进制、八进位、十六进制)运算，只是输出结果皆为十进制而已。

[root@qfedu.com ~]# echo $((16#2a))  # 16进位转十进制
42

# 当前的 umask 是 022，新建文件的权限为:
[root@qfedu.com ~]# umask 022
[root@qfedu.com ~]# echo "obase=8; $(( 8#666 & (8#777 ^ 8#$(umask)) ))" | bc
[root@qfedu.com ~]# 644

# 单纯用(( ))也可以重定义变量值，或作testing:

[root@qfedu.com ~]# a=5
[root@qfedu.com ~]# echo $((a++))


# 将 a 重定义为 6
```

```
[root@qfedu.com ~]# echo $[a--]
[root@qfedu.com ~]# 5

[root@qfedu.com ~]# echo $((a--))
[root@qfedu.com ~]# 4

[root@qfedu.com ~]# a=5; b=7; ((a < b));echo $?
0
# 常见的用于(( ))的测试符号有以下这些：< 小于,> 大于,<= 小于或等于,>= 大于或等于,== 等于,!= 不等于
```

- 双中括号[[ ]]中可以使用通配符进行匹配，这是其区别于其它几种语法的地方
- &&，||，<，>等操作符可用于双中括号[[ ]]中，但不能应用于[ ]中，在[ ]中一般用-a，-o，-lt，-gt来代替

举例：

```
[root@qfedu.com ~]# test -f /tmp/test.txt && echo 1 || echo 0
[root@qfedu.com ~]# [ -f /tmp/test.txt ] && echo 1 || echo 0
[root@qfedu.com ~]# [[ -f /tmp/test.txt ]] && echo 1 || echo 0
[root@qfedu.com ~]# ((3>2)) && echo 1 || echo 0
```

2、获取帮助

```
[root@qfedu.com ~]# man test
```

2、Shell 测试表达式用法

1、文件测试表达式

-d 文件	**文件存在且为目录则为真**
-f 文件	文件存在且为普通文件则为真
-e 文件	文件存在则为真，不辩别是目录还是文件
-s 文件	文件存在且文件大小不为0则为真
-r 文件	文件存在且可读则为真，与执行脚本的用户权限也有关
-w 文件	文件存在且可写则为真，与执行脚本的用户权限也有关
-x 文件	文件存在且可执行则为真，与执行脚本的用户权限也有关
-L 文件	文件存在且为链接文件则为真
f1 -nt f2	文件f1比文件f2新则为真，根据文件的修改时间计算
f1 -ot f2	文件f1比文件f2旧则为真，根据文件的修改时间计算

- 文件测试 [操作符 文件或目录]

```
[root@qfedu.com  ~]# test -d /home
[root@qfedu.com  ~]# echo $?
0
[root@qfedu.com  ~]# test -d /home11111
[root@qfedu.com  ~]# echo $?
1
[root@qfedu.com  ~]# [ -d /home ]
[root@qfedu.com  ~]# [ ! -d /ccc ]  && mkdir /ccc
[root@qfedu.com  ~]# [ -d /ccc ]  || mkdir /ccc
```

2、字符串测试表达式

	参数	功能
-z	s1	如果字符串s1的长度为0，则测试条件为真
-n	s1	如果字符串s1的长度大于0，则测试条件为真
sl		如果字符串s1不是空字符串，则测试条件为真
=或==	s1=s2	如果s1等于s2，则测试条件为真,“=”前后应有空格
!=	s1!=s2	如果s1不等于s2，则测试条件为真
<	s1	如果按字典顺序s1在s2之前，则测试条件为真
>	s1>s2	如果按自定顺序s1在s2之后，则测试条件为真

1、注意

- 对于字符串的比较，一定要将字符串加比引号后再比较。如[-n "$string"]
- =与!=可用于判断两个字符串是否相同

2、字符串比较

```
# 提示：字符串必须使用双引号
[root@qfedu.com  ~]# [ "$USER" = "root" ];echo $?
0
[root@qfedu.com  ~]# [ "$USER" == "root" ];echo $?
0

[root@qfedu.com  ~]# BBB=""
[root@qfedu.com  ~]# echo ${#BBB}
0
[root@qfedu.com  ~]# [ -z "$BBB" ]              # 字符长度是为0
[root@qfedu.com  ~]# echo $?
0
[root@qfedu.com  ~]# [ -n "$BBB" ]              # 字符长度不为0
[root@qfedu.com  ~]# echo $?
1


[root@qfedu.com  ~]# var1=111
```

```
[root@qfedu.com  ~]# var2=
[root@qfedu.com  ~]#                                  # var3变量没有定义
[root@qfedu.com  ~]# echo ${#var1}
3
[root@qfedu.com  ~]# echo ${#var2}
0
[root@qfedu.com  ~]# echo ${#var3}
0
[root@qfedu.com  ~]# [ -z "$var1" ];echo $?
1
[root@qfedu.com  ~]# [ -z "$var2" ];echo $?
0
[root@qfedu.com  ~]# [ -z "$var3" ];echo $?
0
[root@qfedu.com  ~]# [ -n "$var1" ];echo $?
0
[root@qfedu.com  ~]# [ -n "$var2" ];echo $?
1
[root@qfedu.com  ~]# [ -n "$var3" ];echo $?
1
```

3、整数操作符

在[]和test中使用	在[[ ]]和(( ))中使用	说明
-eq	==或=	等于，全拼为equal
-nq	!=	不等于，全拼为not equal
-gt	>	大于，全拼为greater than
-ge	>=	大于等于，全拼为greater equal
-lt	<	小于，全拼为less than
-le	<=	小于等于，全拼为less equal

1、判断变量是不是数字

```
[root@qfedu.com  ~]# num10=123
[root@qfedu.com  ~]# num20=ssss1114ss
[root@qfedu.com  ~]# [[ "$num10" =~ ^[0-9]+$ ]];echo $?
0
[root@qfedu.com  ~]# [[ "$num20" =~ ^[0-9]+$ ]];echo $?
1
```

2、数值比较 [整数1 操作符 整数2]

```
[root@qfedu.com  ~]# disk_use=$(df -P |grep '/$' |awk '{print $5}' |awk -F% '{print $1}')
[root@qfedu.com  ~]# [ $disk_use -gt 90 ] && echo "war......"
[root@qfedu.com  ~]# [ $disk_use -gt 60 ] && echo "war......"
war......

[root@qfedu.com  ~]# id -u
0
[root@qfedu.com  ~]# [ $(id -u) -eq 0 ] && echo "当前是超级用户"
当前是超级用户
[alice@qfedu.com  ~]$ [ $UID -eq 0 ] && echo "当前是超级用户" || echo "you不是超级用户"
you不是超级用户
```

3、C语言风格的数值比较

```
[root@qfedu.com  ~]# ((1<2));echo $?
0
[root@qfedu.com  ~]# ((1==2));echo $?
1
[root@qfedu.com  ~]# ((1>2));echo $?
1
[root@qfedu.com  ~]# ((1>=2));echo $?
1
[root@qfedu.com  ~]# ((1<=2));echo $?
0
[root@qfedu.com  ~]# ((1!=2));echo $?
0
[root@qfedu.com  ~]# ((`id -u`>0));echo $?
1
[root@qfedu.com  ~]# (($UID==0));echo $?
0
```

4、实例

```
# 案例1:
[root@qfedu.com  ~]# cat test02.sh
#!/bin/bash
# 判断用户输入的是否是数字
read -p "请输入一个数值: " num

if [[ ! "$num" =~ ^[0-9]+$ ]];then
    echo "你输入的不是数字，程序退出!!!"
    exit
fi

echo ccc

# 案例2:
[root@qfedu.com  ~]# cat test03.sh
# !/bin/bash
# 判断用户输入的是否是数字
read -p "请输入一个数值: " num
```

```
while :
do
        if [[ $num =~ ^[0-9]+$ ]];then
                break
        else
                read -p "不是数字，请重新输入数值： " num
        fi
done

echo "你输入的数字是：$num"
```

4、逻辑操作符

在[]和test中使用	在[[ ]]和(( ))中使用	说明
-a	&&	and，与，两端都为真，则结果为真
-o	||	or，或，两端有一个为真，则结果为真
!	!	not，非，两端相反，则结果为真

```
[root@qfedu.com  ~]# [ 1 -lt 2 -a 5 -gt 10 ];echo $?
1
[root@qfedu.com  ~]# [ 1 -lt 2  -o 5 -gt 10 ];echo $?
0

[root@qfedu.com  ~]# [[ 1 -lt 2 && 5 -gt 10 ]];echo $?
1
[root@qfedu.com  ~]# [[ 1 -lt 2 || 5 -gt 10 ]];echo $?
0
```

5、测试表达式的区别总结

测试表达式符号	test	[ ]	[[ ]]	(( ))
边界是否需要空格	需要	需要	需要	不需要
逻辑操作符	!、-a、 -o	!、-a、 -o	!、&&、||	!、&&、||
整数比较操作符	-eq、-ne、-lt、-gt、-ge、-le	-eq、-ne、-lt、-gt、-ge、-le	-eq、-ne、-lt、-gt、-ge、-le或=、!=、<、>、>=、<=	=、!=、<、>、>=、<=
字符串比较操作符	=、==、!=	=、==、!=	=、==、!=	=、==、!=
是否支持通配符	不支持	不支持	支持	不支持

1、变量为空或未定义长度都为0

```
[root@qfedu.com  ~]# [ "$USER" = "root" ];echo $?
0
[root@qfedu.com  ~]# [ "$USER" = "alice" ];echo $?
1
[root@qfedu.com  ~]# [ "$USER" != "alice" ];echo $?
0

[root@qfedu.com  ~]# [ "$USER" = "root" ];echo $?
0
[root@qfedu.com  ~]# [ "$USER" =~ ^r ];echo $?
bash: [: =~: binary operator expected
2
[root@qfedu.com  ~]# [[ "$USER" =~ ^r ]];echo $?         # 使用正则
0
```

2、Shell 脚本执行测试

```
# 执行脚本:
[root@qfedu.com  ~]# ./01.sh                          # 需要执行权限     在子shell中执行
[root@qfedu.com  ~]# bash 01.sh                       # 不需要执行权限    在子shell中执行

[root@qfedu.com  ~]# . 01.sh                          # 不需要执行权限    在当前shell中执行
[root@qfedu.com  ~]# source 01.sh                     # 不需要执行权限    在当前shell中执行
# 提示:通常修改系统配置文件中如 /etc/profile 的PATH等变量后,使之在当前shell中生效

# 调试脚本:
[root@qfedu.com  ~]# sh -n 02.sh                      # 仅调试 syntax error
[root@qfedu.com  ~]# sh -vx 02.sh                     # 以调试的方式执行,查询整个执行过程
```

3、Shell 分支if语句

1、单分支 IF 条件语句

1、语法格式

```
if [ 条件判断式 ];then
    条件成立时,执行的程序
fi

# if语句使用fi结尾和一般语言使用大括号结尾不同
# [条件判断式] 就是使用test命令判断,所以中括号和条件判断式之间必须有空格
# then 后面跟符号条件之后执行的程序,可以放在[]之后,用";"分割。也可以换行写入,就不需要";"了
```

2、实例

```
# 判断登录的用户是否为root
#!/bin/bash
# 把当前用户名赋值给变量test
test=$(env | grep "USER" | cut -d "=" -f 2)
```

```
if [ "$test"==root ];then
  echo "current user is root"
fi

# 判断分区使用率
#!/bin/bash
test=$(df -h | grep sda5 | awk '{print $5}' | cut -d "%" -f 1)
# 把分区使用率作为变量值赋予变量 test
if [ -ge 90 ];then
    echo "文件满了"
fi
```

2、双分支语句

1、语法格式

```
if [ 条件判断式 ]; then
   条件成立时,执行的程序
else
   条件不成立时，执行的另一个程序
fi
```

2、实例

```
# 判断输入的是不是目录
#!/bin/bash
read -t 30 -p "please input a dir :" dir
if[ -d "$dir" ];then # 注意前后的空格
    echo "输入的是目录"
else
    echo "输入的不是目录"
fi

# 判断 apache 是否启动
#!/bin/bash
test = $(ps aux | grep httpd | grep -v grep)
# 截取httpd进程，并把结果赋予变量test
if [ -n test ];then
# 如果test不为空
  echo "the apache is on running!" >> /~/running.log
else
  /etc/rc.d/init.d/httpd start &> dev/null
  echo "the apache is restart!" >> /~/restart.log
fi
```

3、多分支语句

1、语法格式

```
if [ 条件判断式1 ]
 then
  当条件判断式1成立时，执行程序1
elif [ 条件判断式2 ]
 then
  当条件判断式2成立时，执行程序2
...省略更多条件....
else
  当所有条件都不成立，最后执行此程序
fi
```

2、实例

```
#!/bin/bash
# 从键盘输入获取数字赋值给变量age
read age
if (( $age <= 2 )); then
    echo "婴儿"
elif (( $age >= 3 && $age <= 8 )); then
    echo "幼儿"
elif (( $age >= 9 && $age <= 17 )); then
    echo "少年"
elif (( $age >= 18 && $age <=25 )); then
    echo "成年"
elif (( $age >= 26 && $age <= 40 )); then
    echo "青年"
elif (( $age >= 41 && $age <= 60 )); then
    echo "中年"
else
    echo "老年"
fi
```

4、Shell 分支case语句

case 语句和 if...elif...else 语句一样都是多分支条件语句，不过和多分支 if 条件语句不同的是，case 语句只能判断一种条件关系，而 if 语句可以判断多种条件关系。

1、case 语法格式

```
case 变量名 in
    值1)
        如果变量的值等于值1则执行指令1
        ;;
    值2)
        如果变量的值等于值2则执行指令2
        ;;
    值3)
        如果变量的值等于值3则执行指令3
        ;;
    *)
        如果变量的值不等于以上列出的任何值则执行默认指令
esac
```

2、case 语句的使用总结

- case 语句比较适合变量值较少且为固定的数字或字符串集合情况(非不确定的内容，例如范围)，如果变量的值是已知固定的start/stop/restart等元素，那么采用case语实现就比较适合
- case主要是写服务的启动脚本，一般情况下，传参不同且具有少量的字符串，其适用范围窄
- if就是取值判断、比较、应用比case更广。几乎所有的case语句都可以用if条件语句实现
- case语句就相当于多分支的if/elif/else语句，但case语句的优势是更规范、易

3、case 语句案例

1、判断输入内容

```
1.apple
2.pear
3.banana
4.cherry
# 当用户输入对应的数字选择水果的时候，告诉他选择的水果是什么，并给水果单词加上一种颜色（随意），要求用case语句实现。
[root@qfedu.com ~]# cat fruit.sh
#!/bin/bash
#################################################################
# File Name: fruit.sh
# Version: V1.0
# Author: qfedu
# Organization:
# Created Time :
# Description:
#################################################################
cat <<EOF
    1.apple
    2.pear
    3.banana
    4.cherry
EOF
read -p "请输入您的选择:" num
red="\033[31m"
green="\033[32m"
yewllo="\033[33m"
```

```
blue="\033[34m"
tailer="\033[0m"
case $num in
    1)
        echo -e "$red apple $tailer"
        ;;
    2)
        echo -e "$green pear $tailer"
        ;;
    3)
        echo -e "$yewllo banana $tailer"
        ;;
    4)
        echo -e "$blue cherry $tailer"
        ;;
    *)
        echo "Usage:$0{1|2|3|4}"
        exit 1
esac
```

2、判断输入执行输入指令

```
[root@qfedu.com ~]# cat rsync.sh
#!/bin/bash
#################################################################
# File Name: rsync.sh
# Version: V1.0
# Author: qfedu
# Organization:
# Created Time :
# Description:
#################################################################
. /etc/init.d/functions
# rsyncd进程号路径
rsyncd_pid_path=/var/run/rsyncd.pid
# 创建锁文件
lockfile=/var/lock/subsys/rsyncd
start() {
    if [ ! -f $rsyncd_pid_path ]
    then
        rsync --daemon
        retval=$?
        if [ $retval -eq 0 ]
        then
            action "rsync is start ok" /bin/true
            touch $lockfile
            return $retval
        else
            action "rsync is start fail" /bin/false
            return $retval
        fi
    else
        echo "rsync in runing.."
```

```
    fi
}
stop() {
    if [ -f $rsyncd_pid_path ]
    then
        rsyncd_pid=`cat $rsyncd_pid_path`
        #判断进程是否存在
        if (kill -0 $rsyncd_pid &>/dev/null)
        then
            kill $rsyncd_pid
            retval=$?
            if [ $retval -eq 0 ]
            then
                action "rsync is stop ok" /bin/true
                rm -f $lockfile
                return $retval
            else
                action "rsync stop fail" /bin/false
                return $retval
            fi
        fi
    else
        echo "$rsyncd_pid_path is not exist or rsyncd does not startup"
    fi
}

case $1 in
    start)
        start
        retval=$?
        ;;
    stop)
        stop
        retval=$?
        ;;
    restart)
        stop
        retval=$?
        sleep 1
        start
        retval=$?
        ;;
    *)
        echo "Usage:$0{start|stop|restart}"
        exit 1
esac
exit $retval
```


二、Shell 编程之循环结构

1、Shell 循环 for 语句

for循环的运作方式，是讲串行的元素意义取出，依序放入指定的变量中，然后重复执行含括的命令区域（在do和done 之间），直到所有元素取尽为止。

其中，串行是一些字符串的组合，彼此用$IFS所定义的分隔符（如空格符）隔开，这些字符串称为字段。

1、for 循环的语法结构

```
for 变量 in 值集合
do
   执行命令
done
```

2、for 语法说明

- for 每次从值集合中取一个值赋值给变量
- do - done 将赋值后的变量带入执行的命令得到执行结果，
- 重复以上两个步骤，直到值集合中的值被一一获取赋值给变量的到所有结果，循环结束

3、实例

1、用 for 循环创建 demo1-demo10，然后在 demo1-demo10 创建 test1-test10 的目录

```
#!/bin/bash
for a in {1..10}
do
        mkdir /datas/demo$a
        cd /datas/demo$a
        for b in {1..10}
        do
                mkdir test$b
        done
done

#!/bin/bash
for k in $( seq 1 10 )     # seq  a b   用于产生从 a 到 b 之间的所有整数
do
   mkdir /root/demo${k}
   cd  /root/demo${k}
   for l in $( seq 1 10 )
   do
      mkdir test${l}
      cd  /root/demo${k}
   done
   cd ..
done
```

2、列出 var 目录下各子目录占用磁盘空间的大小

```
#!/bin/bash
DIR="/var"
cd $DIR
for k in $(ls $DIR)        # 对/var目录中每一个文件，进行for循环处理
do
  [ -d $k ] && du -sh $k   # 如果/var下的文件是目录，则使用du -sh计算该目录占用磁盘空间的大小
done
```

2、Shell 循环 while 语句

1、while 循环语法结构

```
while 条件测试
do
  执行命令
done
```

2、while 语法说明

- while 首先进行条件测试，如果传回值为0（条件测试为真），则进入循环，执行命令区域，否则不进入循环
- 满足 while 测试条件，执行命令区域，直到 while 的测试条件不满足结束执行while循环（如果条件一直满足执行无穷循环）。

3、实例1 while 循环读取文件的内容

```
#!/bin/bash
while read a                    # 使用read有标准输入读取数据，赋值变量 demo 中，如果读到的数据非空，就进入循环，显示读取到的内容
do
     echo $a
done < /datas/6files

#!/bin/bash
while read demo
do
  echo ${demo}
done < /home/scripts/testfile  # 将 /home/scripts/testfile 的内容按行输入给 read 读取
```

4、实例2 while 条件测试

```
#!/bin/bash
declare -i i=1          # 声明设置 i 和 sum为整数型
declare -i sum=0
while ((i<=10))         # while 条件测试：只要i值小于或者等于10，就执行循环
do
    let sum+=i          # sum+=i 和 sum=sum+i 是一样的，sum累加上i
    let i++             # let i++，i 的值递增 1，此行是改变条件测试的命令，一旦 i 大于10，可终止循环
done                    # 遇到 done，回到 while 条件测试
echo $sum               # 直到 while 条件不满足， 显示 sum 的值
```

4、实例3 while 99 乘法表

```
#!/bin/bash
a=1
b=1
while ((a <=9))
do
    while ((b<=a))
    do
        let "c=a*b"            # 声明变量c
        echo -n  "$a*$b=$c "   # echo 输出显示的格式，-n不换行输出
        let b++
     done
     let a++
     let b=1  # 因为每个乘法表都是1开始乘，所以b要重置
    echo ""   # 显示到屏幕换行
done
```

3、Shell 循环 until 语句

- while循环的条件测试是测真值，until循环则是测假值。

1、until 循环的语法结构

```
until 条件测试
do
 执行命令
done
```

2、until 语法说明

- until 条件测试结果为假（传回值不为0），就进入循环。
- 条件测试不满足，执行命令区域。直到 until 条件满足，结束执行until 循环（如果条件一直不满足则执行无穷循环）。

3、实例1 until 单层条件测试

```
#!/bin/bash
declare -i i=10      # 声明i和sum为整数型
declare -i sum=0
until ((i>10))       # 条件测试：只要i值未超过10，就进入循环
do
  let sum+=i         # sum+=i和sum=sum+i是一样的，sum累加上i
  let ++i             # i的值递增1，此行是改变条件测试的命令，一旦i大于10，可终止循环
done                  # 遇到done，回到 until条件测试
echo $sum             # 直到 until 的条件满足显示sum的值
```

4、实例2 until 双层条件测试

```
#!/bin/bash
a=1
```

```
b=1
until ((a>9))             # 条件测试：只要a值未超过9，就进入循环，一旦超过9就不执行，until和while条件相反，条件真就done结束
do
     until ((b>a))        # b>a,一旦b大于a就不执行
     do
          let "c=a*b"
           echo -n "$a*$b=$c "
           let b++
     done
     let a++
     let b=1
     echo ""
done
```

4、Shell 循环控制

1、Shell 循环控制说明

- break，continue，exit 一般用于循环结构中控制循环的走向。

命令	说明
break n	n 表示跳出循环的次数，如果省略 n 表示跳出整个循环
continue n	n 表示退到第n层继续循环，如果省略n表示跳过本次循环进入下一次循环
exit n	退出当前的shell程序，并返回 n，n 也可以省略
return	用于返回一个退出值给调用的函数
shift	用于将参数列表list左移指定次数，最左端的那个参数就从列表中删除，其后边的参数继续进入循环

2、break 指令

- break[N]：提前结束第N层循环，最内层为第1层

```
while CONDITION1; do
      CMD1
      ...
      if CONDITION2; then
                break
      fi
      CMDn
      ...
done
```

- continue：提前结束本次循环，提前进入下一轮循环，continue 2 跳出本次内层循环，进入外层循环
- break：结束本次循环（整个），退出脚本

- 实例

```
[root@qfedu.com  ~]#vim test.sh
#!/bin/bash
for i in {1..10}
do
        [ $i -eq 5 ] && break
        echo i=$i
        sleep 0.5
done
echo test is finished

[root@qfedu.com  ~]#chmod +x test.sh
[root@qfedu.com  ~]#./test.sh
i=1
i=2
i=3
i=4
test is finished


[root@localhost ~]# cat break1.sh
#!/bin/bash
for((i=0;i<=5;i++))
  do
    if [ $i -eq 3 ];then
         break;
    fi
       echo $i
   done
echo "ok"

# 运行结果为:
[root@localhost ~]# bash break1.sh
0
1
2
ok
```


3、continue 指令

- continue [N]：提前结束第N层的本轮循环，而直接进入下一轮判断；最内层为第1层

```
while CONDITION1;do
      CMD1
      ...
      if CONDITION2; then
                continue
      fi
      CMDn
      ...

done
```

- 实例

```
[root@localhost ~]# cat continue.sh
#!/bin/bash
for((i=0;i<=5;i++))
  do
    if [ $i -eq 3 ];then
         continue;
    fi
       echo $i
   done
echo "ok"

#运行结果为:
[root@localhost ~]# bash continue.sh
0
1
2
4
5
ok


[root@qfedu.com  ~]#vim test.sh
#!/bin/bash
for i in {1..10}
do
        [ $i -eq 5 ] && continue
        echo i=$i
        sleep 0.5
done
echo test is finished

[root@qfedu.com  ~]#./test.sh
i=1
i=2
i=3
i=4
i=6
i=7
i=8
i=9
i=10
test is finished
```

4、exit 指令

- 实例

```
[root@localhost ~]# cat exit.sh

#!/bin/bash
```

```
for((i=0;i<=5;i++))
  do
    if [ $i -eq 3 ];then
         exit
    fi
         echo $i
   done
echo "ok"

#运行结果为:
[root@localhost ~]# bash exit.sh
0
1
2
```

5、shift 指令

- shift 命令用于将参数列表 list 左移指定次数，最左端的那个参数就从列表中删除，其后边的参数继续进入循环。
- shift[N]：用于将参量列表 list 左移指定次数，缺省为左移一次。
- 参量列表 list 一旦被移动，最左端的那个参数就从列表中删 除。while 循环遍历位置参量列表时，常用到 shift
- 实例

```
[root@qfedu.com  ~]# vim demo.sh
#!/bin/bash
while [ $# -gt 0 ]
do
        echo $*
        shift
done

[root@qfedu.com  ~]# ./demo.sh a b c d e f g h
a b c d e f g h
b c d e f g h
c d e f g h
d e f g h
e f g h
f g h
g h
h
[root@qfedu.com  ~]# vim shift.sh
#!/bin/bash
until [ -z "$1" ]
do
        echo "$1"
        shift
done
echo

[root@qfedu.com  ~]# ./shfit.sh a b c d e f g h

-bash: ./shfit.sh: No such file or directory
```

```
[root@qfedu.com  ~]# ./shift.sh a b c d e f g h
a
b
c
d
e
f
g
```

1、shift 指令实例：创建指定的多个用户

- 实例

```
#!/bin/bash
if [ $# -eq 0 ];then
   echo "请在脚本后输入参数（例如：$0 arg1）"
   exit 1
else
   while [ -n "$1" ];do
      useradd $1
      echo 123|passwd --stdin $1
      shift
   done
fi
```

2、运行结果

```
[root@qfedu.com  ~]# bash test.sh a1 a2 a3
更改用户 a1 的密码 。
passwd：所有的身份验证令牌已经成功更新。
更改用户 a2 的密码 。
passwd：所有的身份验证令牌已经成功更新。
更改用户 a3 的密码 。
passwd：所有的身份验证令牌已经成功更新。
```

分析：如果没有输入参数（参数的总数为0），提示错误并退出；反之，进入循环；若第一个参数不为空字符，则创建以第一个参数为名的用户，并移除第一个参数，将紧跟的参数左移作为第一个参数，直到没有第一个参数，退出。

3、打印直角三角形的字符

- 实例

```
#!/bin/bash
while (($# > 0));do
   echo "$*"
   shift
done
```

4、运行结果

```
[root@qfedu.com  ~]# bash test1.sh 1 2 3 4 5 6 7
1 2 3 4 5 6 7
2 3 4 5 6 7
3 4 5 6 7
4 5 6 7
5 6 7
6 7
7
```

三、Shell 编程之函数

Shell 函数的本质是一段可以重复使用的脚本代码，这段代码被提前编写好了，放在了指定的位置，使用时直接调取即可

1、定义函数

- 可以带function fun() 定义，也可以直接fun() 定义,不带任何参数。

```
# 方法一
function name {
  commands
  [return value]
}
# 方法二
name() {
  commands
  [return value]
}
```

- function 是 Shell 中的关键字，专门用来定义函数；
- name 是函数名；
- commands 是函数要执行的代码，也就是一组语句；
- return value 表示函数的返回值，其中 return 是 Shell 关键字，专门用在函数中返回一个值；这一部分可以写也可以不写。
- 由 { } 包围的部分称为函数体，调用一个函数，实际上就是执行函数体中的代码。
- 函数的优势
    - 方便n次使用，减少代码量，使之方便，整洁。
    - 当需要修改里面的重复代码时，只需要修改一次函数即可实现需求；
    - 将函数写进文件，需要时直接通过文件调用

2、调用函数

1、执行不带参数的函数

- 直接输入函数名即可，不需要带括号，

```
functionName
```

- 执行函数时，函数名前的关键字function和函数名后面的()均不需要带
- 函数的定义必须要在执行的程序前定义或加载

2、执行带参数的函数

```
functionName arg1 arg2
```

- Shell中的位置参数(1/2.../You can't use 'macro parameter character #' in math mode?/$@)均可以做为函数的参数进行传递
- $0比较特殊，仍然是父脚本的名称
- 此时父脚本的参数会临时被函数的参数所掩盖或隐藏
- 函数的参数变量是在函数体内里面进行定义

3、函数的执行总结

- Shell各种程序的执行顺序为：系统别名->函数->系统命令->可执行文件等
- 函数执行时，会和调用它的脚本共享变量，也可以为函数设定局部变量及特殊位置参数
- 在Shell函数里面，return和exit功能类似，区别是return是退出函数，exit则是退出脚本
- return语句会返回一个值给调用函数的程序，exit则会返回一个值给执行当前脚本的Shell
- 如果将函数单独存放为一个文件，在加载时需要使用source或 . 进行加载
- 在函数内部一般使用local定义局部变量，仅在函数体内有效

4、调用函数

```
[root@qfedu.com  ~]# cat testfunction.sh
#!/bin/bash
# first function
function HelloWorld() {
  echo "Hello world"
}
# second function
Welcome() {
  echo "Welcome to qfedu"
}
# third function
function HelloShell {
 echo "Hello Shell"
}
# file functions
HelloWorld              # 调用函数
Welcome
HelloShell
[root@qfedu.com  ~]# bash testfunction.sh
Hello world
Welcome to qfedu
Hello Shell
```

5、从文件中调用函数

```
[root@qfedu.com  ~]# cat filefunction.sh
function Sum () {
 for((i=1;i<=100;i++))
  do
    ((sum=sum+i))
  done
  echo '{1..100} sum is :' $sum
}
[root@qfedu.com  ~]# cat filefunctionfromfile.sh
#!/bin/bash
path="/root/Test/filefunction.sh"
if [ -f ${path} ]
   then
    source $path # 加载函数
    Sum          # 调用函数
else
   echo "file not exist or error"
fi
[root@qfedu.com  ~]# bash filefunctionfromfile.sh
{1..100} sum is : 5050
```

3、函数参数传递

```
[root@qfedu.com  ~]# cat functionwithargs.sh
#!/bin/bash
function Add () {    # 定义函数
  ((sum=$1+$2))
  echo "$1 + $2 sum is" ${sum}
}
Add $1 $2           # 调用函数并传递参数

[root@qfedu.com  ~]# bash functionwithargs.sh 100 150
100 + 150 sum is 250
[root@qfedu.com  ~]# bash functionwithargs.sh 509 150
509 + 150 sum is 659
```

4、return 返回函数结果

```
[root@qfedu.com  ~]# cat functionwithreturn.sh
#!/bin/bash
function TestReturn() {
  if [ -d $1 ]
    then
      return "122"
   else
      return "222"
  fi
}

TestReturn $1

result=$? # 获取函数返回值
```

```
if [ ${result} == "122" ]
 then
   echo "$1 exist ,return value is:" ${result}
 else
   echo "$1 not exist ,return value is:" ${result}
fi

[root@qfedu.com  ~]# bash functionwithreturn.sh /etc/sysconfiggg
/etc/sysconfiggg not exist ,return value is: 222
[root@qfedu.com  ~]# bash functionwithreturn.sh /etc/sysconfig
/etc/sysconfig exist ,return value is: 122
```

- 在该示例中，主要通过 $? 获取返回值，但返回值的范围只能是 0~255

5、echo 返回函数结果

```
[root@qfedu.com  ~]# cat functionwithecho.sh
#!/bin/bash
function TestReturn() {
  if [ -d $1 ]
    then
      echo "122"
   else
      echo "222"
  fi
}

result=$(TestReturn $1) # 获取函数返回值
if [ ${result} == "122" ]
 then
   echo "$1 exist ,return value is:" ${result}
 else
   echo "$1 not exist ,return value is:" ${result}
fi

[root@qfedu.com  ~]# bash functionwithecho.sh /etc/sysconfig
/etc/sysconfig exist ,return value is: 122
[root@qfedu.com  ~]# bash functionwithecho.sh /etc/sysconfiggg
/etc/sysconfiggg not exist ,return value is: 222
```

- 在该示例中，主要使用 $() 获取返回值，在该方法中，没有范围限制，是一种比较安全的返回方式。

```
[root@qfedu.com  ~]# cat functionwithecho.sh
#!/bin/bash
function TestReturn() {
  if [ -d $1 ]
    then
      echo "$1 exist"
   else
      echo "$1 not exist"
  fi
}
```

```
result=$(TestReturn $1) # 获取返回值，返回的结果是字符串
if [ "${result}" == "$1 exist" ]
 then
   echo "$1 exist ,return value is:" ${result}
 else
   echo "$1 not exist ,return value is:" ${result}
fi

[root@qfedu.com  ~]# bash functionwithecho.sh /etc/sysconfiggg
/etc/sysconfiggg not exist ,return value is: /etc/sysconfiggg not exist
[root@qfedu.com  ~]# bash functionwithecho.sh /etc/sysconfig
/etc/sysconfig exist ,return value is: /etc/sysconfig exist
```

6、全局变量和局部变量

- 全局变量在shell 脚本中任何地方都能使用；局部变量在函数内部使用，声明前加一个 local 就好

```
[root@qfedu.com  ~]# cat test3.sh
function fun() {
    a=$[ $b + 5 ]
    c=$[ $a * 2 ]
}
a=4
b=6
fun

if [ $a -gt $b ]
then
    echo "$a is larger than $b"
else
    echo "$a is smaller than $b"
fi

function fun() {
    local  a=$[ $b + 5 ]
    c=$[ $a * 2 ]
}
a=4
b=6
fun

if [ $a -gt $b ]
then
    echo "$a is larger than $b"
else
    echo "$a is smaller than $b"
fi

[root@qfedu.com  ~]# bash test3.sh
11 is larger than 6
4 is smaller than 6
```

7、数组变量和函数

- $@ 变量会单独处理每个参数

```
[root@qfedu.com  ~]# cat test4.sh
function addarray() {
    local sum=0
    local newarray
    newarray=($(echo "$@"))
    for value in ${newarray[*]}
    do
        sum=$[ $sum + $value ]
    done
    echo $sum
}

myarray=(1 2 3 4 5)
# 这里 arg1=${myarray[*]} 也可以
arg1=$(echo ${myarray[*]})
result=$(addarray $arg1)
echo "The result is $result"

[root@qfedu.com  ~]# bash test4.sh
The result is 15

[root@qfedu.com  ~]# cat test5.sh
function arraydblr() {
    local origarray
    local newarray
    local elements
    local i
    origarray=($(echo "$@"))
    newarray=($(echo "$@"))
    elements=$[ $# -1 ]
    for (( i = 0; i <= $elements; i++))
    {
        newarray[$i]=$[ ${origarray[$i]} * 2 ]
    }
    echo ${newarray[*]}
}

myarray=(1 2 3 4 5)
arg1=$(echo ${myarray[*]})
result=($(arraydblr $arg1))
echo "The new array is: ${result[*]}"

[root@qfedu.com  ~]# cat test4.sh bash test5.sh
The new array is: 2 4 6 8 10
```

8、递归函数

```
[root@qfedu.com  ~]# cat test6.sh
```

```
function factorial() {
    if [ $1 -eq 1 ]
    then
        echo 1
    else
        local temp=$[ $1 - 1 ]
        local result=$(factorial $temp)
        echo $[ $result * $1 ]
    fi
}

read -p "Enter value: " value
result=$(factorial $value)
echo "The factorial of $value is: $result"

[root@qfedu.com  ~]# bash test6.sh
Enter value: 5
The factorial of 5 is: 120
```