1、 Shell脚本编程_概述

Shell脚本就是由Shell命令组成的执行文件，将一些命令整合到一个文件中，进行处理业务逻辑，脚本不用编译即可运行，它通过解释器解释运行。

常见的解释器

```
#!/bin/sh #不推荐(了解)
#!/bin/bash
#!/usr/bin/python
#!/bin/awk
```

让#!后跟的字符表示要启动的程序，该程序读取该文件执行。

好处：命令行中的命令都可以放到一个文件中！省的每次都得重新写大量的shell命令。

案例实战

```
[root@node1 ~]# mkdir ch4
[root@node1 ~]# cd ch4
[root@node1 ch4]# vim my.sh
# !/bin/bash
ls   /
echo  "hello world!"
```

执行方式：

- bash/sh file

```
[root@node1 ch4]# type sh
sh 是 /usr/bin/sh
[root@node1 ch4]# type bash
bash 是 /usr/bin/bash
[root@node1 ch4]# bash my.sh
bin  boot  dev  etc  home  lib  lib64
media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
hello world!
[root@node1 ch4]# sh my.sh
bin  boot  dev  etc  home  lib  lib64
media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
hello world!
```

- 当前shell：source file

```
[root@node1 ch4]# source my.sh
bin  boot  dev  etc  home  lib  lib64
media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
hello world!
```

- ./fileName.sh （需要添加可执行的权限）

```
[root@node1 ch4]# ./my.sh
-bash: ./my.sh: 权限不够
[root@node1 ch4]# chmod +x my.sh
[root@node1 ch4]# ./my.sh
bin  boot  dev  etc  home  lib  lib64
media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
hello world!
```

实时效果反馈

1. 当前目录下有一个shell脚本文件`do.sh`，以下选项中不能正确执行该脚本的选项是

A `bash do.sh`

B `source do.sh`

C `./do.sh`，需要提前通过命令`chmod +x do.sh`添加执行权限

D `run do.sh`

答案：

1=>D

2、awk脚本编程实现消费统计

原来命令：

```
awk  '{
    split($3, date, "-")
    if (date[2] == "01") {
        map_name_sala[$1]+=$5
        if($2=="0"){
            map_name_role[$1]="Manager"
        }else{
            map_name_role[$1]="Worker"
        }
    }
}
END{
    for(name in map_name_sala){
        print
name"\t"map_name_sala[name]"\t"map_name_role[na
me]
    }
}'  awk.txt
```

升级为awk脚本：

```
[root@node1 ~]# vim awk.sh
#!/bin/awk -f
{
    split($3, date, "-")
    if (date[2] == "01") {
        map_name_sala[$1]+=$5
        if($2=="0"){
            map_name_role[$1]="Manager"
        }else{
            map_name_role[$1]="Worker"
```

```
        }
    }
}
END{
    for(name in map_name_sala){
        print
name"\t"map_name_sala[name]"\t"map_name_role[na
me]
    }
}
[root@node1 ~]# chmod  +x  awk.sh
[root@node1 ~]# ./awk.sh  awk.txt
```

3、 函数的定义与调用

java语言中的方法

```
public int methodName(int x,String name){
......
}
```

javascript函数:

```
function mN(x,name){
......
}
```

Shell脚本中的函数:

```
myShellName () {
   command1
   command2
   command3
   ……
}
```

函数调用：

```
myshellname
```

案例实战：

```
[root@node1 ch4]# vim fun.sh
#!/bin/bash
# 定义函数
myFunName(){
    echo "hello world $1"
}
# 函数调用
myFunName www.bjsxt.com
[root@node1 ch4]# chmod +x fun.sh
[root@node1 ch4]# ./fun.sh
hello world www.bjsxt.com
```

总结：

1. bash是一个程序，shell是一个bash进程
2. bash是一个解释器，启动器
3. 解释执行用户的输入指令，可以通过shell启动其他的进程
4. 将要执行的命令放到一个文件中，该文件也被称为一个shell脚本。
5. 在文件的开头写上如下内容，用于指定该脚本由哪个程序负责解释执行：

- #!/bin/bash
- #!/usr/bin/python
- #!/bin/awk -f

1. 当前bash进程中执行脚本：source fileName

```
[root@node1 ch4]# echo $$
14729
[root@node1 ch4]# cat my.sh
#!/bin/bash
ls -l /
echo "hello world"
echo $$
pstree -p
[root@node1 ch4]# source my.sh
总用量 24
lrwxrwxrwx.   1 root root    7 8月  17
01:28 bin -> usr/bin
dr-xr-xr-x.   5 root root 4096 8月  17
01:33 boot
....
hello world
14729
systemd(1)─┬─NetworkManager(677)─┬─{Networ
kManager}(690)
           |
└─{NetworkManager}(692)
           |-....

├─sshd(902)─┬─sshd(934)───bash(1033)
           |
├─sshd(1034)───sftp-server(1038)
           |
├─sshd(14726)───bash(14729)───pstree(15815
)
```

2. 子进程执行：bash mysh.sh或者./mysh.sh(需要该文件具有可执行权限)

```
[root@node1 ch4]# ./my.sh
总用量 24
lrwxrwxrwx.   1 root root    7 8月  17
01:28 bin -> usr/bin
......
hello world
15802
systemd(1)─┬─NetworkManager(677)─┬─{Networ
kManager}(690)
           |
└─{NetworkManager}(692)
           ├─......

├─sshd(902)─┬─sshd(934)───bash(1033)
           |
├─sshd(1034)───sftp-server(1038)
           |
├─sshd(14726)───bash(14729)───my.sh(15802)
───pstree(15804)
```

8. 定义函数：

```
funName() {
    各种命令
}
```

1. 直接输入funName就可以执行了

4、 Shell脚本中的变量

变量的类型：

1. 环境变量：比如PATH、LANG、JAVA_HOME
2. 本地变量：定义shell脚本中、函数外
3. 局部变量：定义在函数中的变量
4. 位置变量：$1 、$2
5. 特殊变量：$# 、$*、$@、$?

4.1 本地变量

当前shell所有

生命周期跟当前shell一样

```
[root@node1 ch4]# a=88
[root@node1 ch4]# echo $a
88
[root@node1 ch4]# myfuna(){
> myvar=66
> echo $myvar
> }
#访问不到该值
[root@node1 ch4]# echo $myvar

[root@node1 ch4]# myfuna
66
[root@node1 ch4]# echo $myvar
66
[root@node1 ch4]# name=bjsxt
[root@node1 ch4]# echo $name
bjsxt
#会将nameisverygood当成一个变量名称
[root@node1 ch4]# echo $nameisverygood
```

```
#如果变量的结果和其它字符串连接在一起时，为了便于区分需要将${变量名}xxx
[root@node1 ch4]# echo ${name}isverygood
bjsxtisverygood
```

严格意义上的shell中的本地变量：

```
[root@node1 ch4]# vim bendi.sh
#!/bin/bash
a=88
echo "a = $a"
myfuna(){
   myvar=66
   echo "inner myfuna  myvar=$myvar"
}
echo "before myfuna   myvar=$myvar"
myfuna
echo "after myfuna   myvar=$myvar"
[root@node1 ch4]# chmod +x bendi.sh
[root@node1 ch4]# ./bendi.sh
a = 88
before myfuna   myvar=
inner myfuna  myvar=66
after myfuna   myvar=66
```

4.2 局部变量：

只能用于函数 local var=100

注意：局部变量只能在函数内部使用。

```
[root@node1 ch4]# myfunb(){
> local myvar=1001
> echo "inner myfunb  myvar=$myvar"
> }
[root@node1 ch4]# echo $myvar

[root@node1 ch4]# myfunb
inner myfunb  myvar=1001
[root@node1 ch4]# echo $myvar

[root@node1 ch4]#
```

shell脚本的方式：

```
[root@node1 ch4]# vim localVar.sh
myfunb(){
    local myvar=1001
    echo "inner myfunb myvar=$myvar"
}
echo "before myfunb myvar=$myvar"
#调用函数
myfunb
echo "after myfunb myvar=$myvar"
[root@node1 ch4]# chmod +x localVar.sh
[root@node1 ch4]# ./localVar.sh
before myfunb myvar=
inner myfunb myvar=1001
after myfunb myvar=
#局部变量只能在函数内部使用
```

4.3 位置变量

格式：$正整数。比如$1，$2，${11}

```
[root@node1 ch4]# myfunc(){
> echo $1
> }
[root@node1 ch4]# myfunc

[root@node1 ch4]# myfunc hello
hello
[root@node1 ch4]# myfunc1(){
> echo "第一个参数：$1"
> echo "第四个参数：$4"
> }
[root@node1 ch4]# myfunc1 a b c
第一个参数：a
第四个参数：
[root@node1 ch4]# myfunc1 a b c d
第一个参数：a
第四个参数：d
[root@node1 ch4]# myfunc2(){
> echo "第一个参数：$1"
> echo "第四个参数：$4"
> echo "第十三个参数：$13"
> }
[root@node1 ch4]# myfunc2 1 2 3 4 5 6 7 8 9 10
11 12 13
第一个参数：1
第四个参数：4
第十三个参数：13
[root@node1 ch4]# myfunc2 0 2 3 4 5 6 7 8 9 10
11 12 13
第一个参数：0
```

```
第四个参数：4
第十三个参数：03
[root@node1 ch4]# myfunc2 8 2 11 4 5 6 7 8 9 10
11 12 13
第一个参数：8
第四个参数：4
第十三个参数：83
[root@node1 ch4]# myfunc3(){
> echo "第一个参数：$1"
> echo "第四个参数：$4"
> echo "第十三个参数：${13}"
> }
[root@node1 ch4]# myfunc2 8 2 11 4 5 6 7 8 9 10
11 12 13
第一个参数：8
第四个参数：4
第十三个参数：83
[root@node1 ch4]# myfunc3 8 2 11 4 5 6 7 8 9 10
11 12 13
第一个参数：8
第四个参数：4
第十三个参数：13
```

shell脚本的位置变量：

```
[root@node1 ch4]# vim locationVar.sh
#!/bin/bash
# shell脚本中获取该脚本调用时，脚本后面跟着的参数
echo "第一个参数:$1"
echo "第三个参数:$3"
echo "第十个个参数:${10}"
myfunci(){
    #接收的函数调用时传递的参数
```

```
    echo "inner myfunci:$1"
}
myfunci a b
[root@node1 ch4]# chmod +x locationVar.sh
[root@node1 ch4]# ./locationVar.sh w b c d e 1
2 3 4 5 6
第一个参数:w
第三个参数:c
第十个个参数:5
inner myfunci:a
```

注意：位置变量在shell脚本内部(在函数外部)，接收是./locationVar.sh w b c d e 1 2 3 4 5 6 传递的参数。在shell脚本内的函数内，接收的是函数调用时的传递的参数。

4.4 特殊变量

- $#：位置参数个数
- $*：参数列表，双引号引用为一个字符串 ./otherVar.sh a b c d e #"a b c d e"
    - 所有的参数作为一个字符串 5个参数作为一个字符串
- $@：参数列表，双引号引用为单独的字符串 "a" "b" "c" "d" "e"
    - 所有的参数作为单个的字符串 5个参数作为五个字符串
- $?：上一个命令的退出状态
    - 0：成功
    - 非0：失败

    案例实战：

```
[root@node1 ch4]# vim otherVar.sh
#!/bin/bash
echo "位置参数的个数:$#"
echo "位置参数*的参数列表:$*"
```

```
echo "位置参数@的参数列表:$@"
echo "上一个命令执行的结果状态:$?"
[root@node1 ch4]# chmod +x otherVar.sh
[root@node1 ch4]# ./otherVar.sh 1 2 3 a b c d e
位置参数的个数:8
位置参数*的参数列表:1 2 3 a b c d e
位置参数@的参数列表:1 2 3 a b c d e
上一个命令执行的结果状态:0
[root@node1 ch4]# mkdir adir/bdir
mkdir: 无法创建目录"adir/bdir": 没有那个文件或目录
[root@node1 ch4]# echo $?
1
[root@node1 ch4]# [ -d mydir ]
[root@node1 ch4]# echo $?
1
[root@node1 ch4]# [ -d /root ]
[root@node1 ch4]# echo $?
0
[root@node1 ch4]# [ -d mydir ] && echo "mydir is dir"
[root@node1 ch4]# [ -d /root ] && echo "/root is dir"
/root is dir
```

5、 数组

数组定义：

数组赋值可以使用复合赋值的方式，形式是`name=(value1 ... valuen)`，这里每个值的形式都是[subscript]=value。vlaue必须出现。如果出现了可选的括号和下标，将为这个下标赋值，下标从 0 开始。单独的数组元素通过name[subscript]=value 来赋值或修改。

数组的任何元素都可以用 ${name[subscript]} 来引用。花括号是必须的，以避免和路径扩展冲突。

unset销毁数组：

- unset name[subscript] 将销毁下标是 subscript 的 元 素 。
- unset name, 这里 name 是一个数组，或者 unset name[subscript], 这里 subscript 是 * 或者是 @，将销毁整个数组。

案例实战：

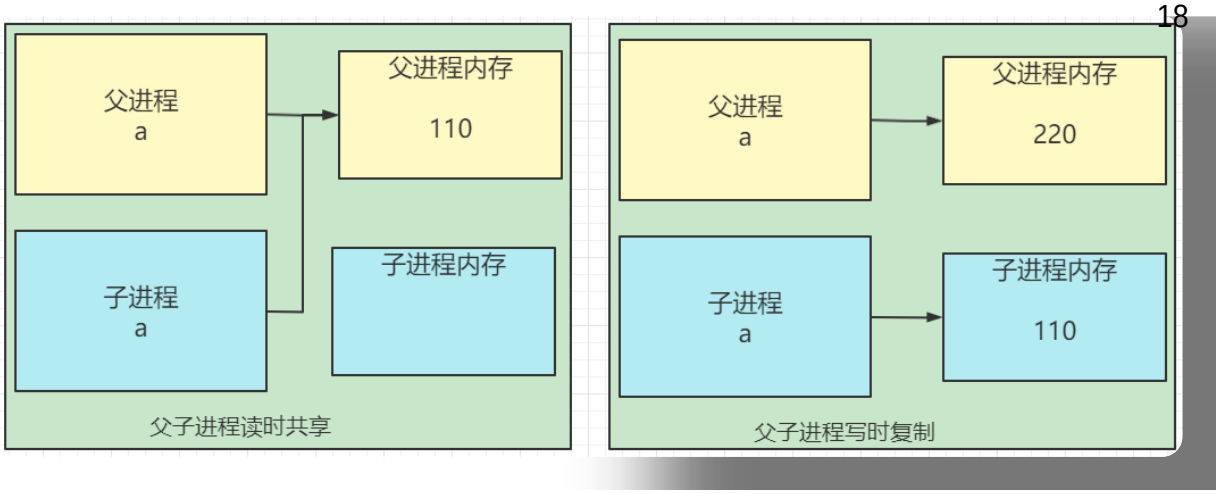
```
#定义数组，并赋初始化的值
[root@node1 ch4]# sxt=(a b c)
# $name 获取第一个元素
[root@node1 ch4]# echo $sxt
a
# ${name[subscript]} 获取指定下标处的元素值
[root@node1 ch4]# echo ${sxt[1]}
b
[root@node1 ch4]# echo ${sxt[2]}
c
# ${sxt[*]} 获取数组中全部的元素
[root@node1 ch4]# echo ${sxt[*]}
a b c
# ${sxt[@]} 获取数组中全部的元素
[root@node1 ch4]# echo ${sxt[@]}
a b c
# $sxt[1] 错误的写法
[root@node1 ch4]# echo $sxt[1]
```

```
a[1]
# name[subscript]=value 修改值
[root@node1 ch4]# sxt[1]=B
[root@node1 ch4]# echo ${sxt[@]}
a B c
# 销毁 下标为2对应的元素c
[root@node1 ch4]# unset sxt[2]
[root@node1 ch4]# echo ${sxt[@]}
a B
[root@node1 ch4]# unset sxt[*]
[root@node1 ch4]# sxt=(a b c)
# 销毁整个数组 sxt
[root@node1 ch4]# unset sxt
[root@node1 ch4]# echo $sxt

[root@node1 ch4]# sxt=(a b c)
# 销毁整个数组 sxt
[root@node1 ch4]# unset sxt[*]
[root@node1 ch4]# echo $sxt

[root@node1 ch4]# sxt=(a b c)
# name[subscript]=value 赋值
[root@node1 ch4]# sxt[3]=d
[root@node1 ch4]# echo ${sxt[*]}
a b c d
```

6、 横跨父子进程的变量



案例实战

```
[root@node1 ~]# vim son_proc.sh
#!/bin/bash
echo  "inner son_prc.sh a="$a
echo  "---------"
[root@node1 ~]# chmod +x son_proc.sh
# 父进程中定义a=110
[root@node1 ~]# a=110
# ./son_proc.sh执行时启动一个子进程，此时变量a还未被共享，所以在子进程中获取不到变量a的值
[root@node1 ~]# ./son_proc.sh
inner son_prc.sh a=
---------
# 将a导出为环境变量，父进程中定义的变量a被共享了，子进程就可以访问了
[root@node1 ~]# export a
[root@node1 ~]# ./son_proc.sh
inner son_prc.sh a=110
---------
```

子进程睡20s，在此期间修改环境变量的值，查看export是导出还是共享

```
[root@node1 ~]# vim son_proc.sh
#!/bin/bash
```

```
echo  "before sleep inner son_prc.sh a="$a
echo  "---------"
sleep 20
echo  "after sleep inner son_prc.sh a="$a
[root@node1 ~]# echo $a
110
# 启动子进程时 a=110 ,并且之前export a
[root@node1 ~]# ./son_proc.sh &
[1] 28687
[root@node1 ~]# before sleep inner son_prc.sh
a=110
---------
# 再次修改父进程中的变量a
[root@node1 ~]# a=220
# 并将修改后的变量a导出为环境变量
[root@node1 ~]# export a
[root@node1 ~]#
[root@node1 ~]# after sleep inner son_prc.sh
a=110
[1]+  完成                    ./son_proc.sh
```

sleep后，子进程不能打印220，而是打印110 。说明：父进程修改值不会影响子进程的变化，子进程值的修改也不会影响父进程的值。fork子进程后，共享变量的值在内存中是读时共享，写时复制。

扩展： 管道两边的命令在当前shell的两个子进程中执行。

```
[root@node1 ch4]# b=22&&echo $b
22
[root@node1 ch4]# echo $b
22
[root@node1 ch4]# unset b
[root@node1 ch4]# b=22|echo $b

[root@node1 ch4]# echo $b

[root@node1 ch4]#
```

7、 单双引号和反引号

- 双引号：弱引用，参数扩展
- 单引号：强引用，不可嵌套
- 花括号扩展不能被引用

```
a=99
echo "$a"    # 99双引号引用，弱引用
echo "\"$a\""    #"99"
echo '$a'     #$a  单引号引用，强引用
#花括号扩展，创建adir，bdir，cdir三个目录
mkdir  ./{a,b,c}dir
#花括号扩展，拷贝/etc/profile以
及/etc/init.d/network到当前目录
cp  /etc/{profile,init.d/network}  ./
```

- 反引号实现命令替换：

命令替换允许我们将shell命令的输出赋值给变量。它是脚本编程中的一个主要部分。

命令替换会创建子shell进程来运行相应的命令。子shell是由运行该脚本的shell所创建出来的一个独立的子进程。由该子进程执行的命令无法使用脚本中所创建的变量。

```
#反引号：`ls -l`
#当前工作目录/opt/apps，将xx.gz远程拷贝到node2节点的/opt/apps目录中
scp xx.gz node2:/opt/apps
#还可以简化为如下行代码（尤其在目录层级比较的多的场景下）
scp xx.gz node2:`pwd`
#另外一种实现方式，可以嵌套
$(ls  -l  /)
```

反引号提升扩展优先级，先执行反引号的内容，再执行其他的。

```
#错误
myvar=echo "hello"
#正确的：
myvar=`echo "hello"`
myvar=$(echo "hello")
#命令替换的嵌套
myvar=$(echo $(echo "hello world"))
myvar=$(echo "hello world")
myvar="hello world"
```

8、 表达式

8.1 逻辑表达式

- command1 && command2 逻辑与：只有两个表达式都为真时，结果才为真。
  - 如果command1退出状态是0，则执行command2
  - 如果command1退出状态不是0，则不再执行command2
- command1 || command2 逻辑或：只少有一个为真，结果就为真。
  - 如果command1的退出状态不是0，则执行command2
  - 如果command1的退出状态是0，则不再执行command2

```
[ -d /hello ]
test -d /hello || echo "文件夹/hello不存在"
test -d /bin && echo "文件夹/bin存在"
test -f profile && rm -f profile && touch
profile

ls / && echo ok
ls / || echo ok
```

8.2 算术表达式

算术表达式：

- let 算数运算表达式
  - `let C=$A+$B`
- $[算术表达式]
  - `C=$[$A+$B]`
- $((算术表达式))!!!
  - `C=$((A+B))`
- expr算术表达式
  - 表达式中各操作数及运算符之间要有空格，同时要使用命令引用

- `C=`expr $A + $B``

案例实战

```
[root@node1 ~]# a=1
[root@node1 ~]# b=2
[root@node1 ~]# let c=$a+$b
[root@node1 ~]# echo $c
[root@node1 ~]# d=$((a+b))
[root@node1 ~]# echo $d
[root@node1 ~]# ((a++))
[root@node1 ~]# echo  $a
#前置++表示先自身++之后再参与计算
#后置++表示先计算，再自身++
[root@node1 ~]# c=$((a+b)) && echo $c
4
[root@node1 ~]# c=$((a--+b)) && echo $c
4
[root@node1 ~]# c=$((a--+b)) && echo $c
3
[root@node1 ~]# echo $a
0
[root@node1 ~]# echo $b
2
[root@node1 ~]# c=$((--a+b)) && echo $c
1
[root@node1 ~]# c=$((a--+b)) && echo $c
1
[root@node1 ~]# echo $a $b
-2 2
[root@node1 ~]# e=$((a*b))
[root@node1 ~]# echo $e
```

8.3 条件表达式

条件表达式

- [表达式]
- test 表达式
  - -d 是否为存在的目录
  - -f 是否为存在的文件
  - -e 是否存在
- [[表达式]]

计算3 > 2的结果，打印返回值，计算3 < 2的结果，打印返回值。

echo $? 打印上一行命令的执行结果

```
# -gt 大于  -lt 小于 -eq  等于  -ge 大于等于  -le 小于等于
[root@node1 ~]# test 3 -gt 2
[root@node1 ~]# echo $?
0
[root@node1 ~]# test 3 -lt 2
[root@node1 ~]# echo $?
1
[root@node1 ~]# test 5 -lt 2
[root@node1 ~]# echo $?
1
成立返回0（true），不成立返回1（false）
[root@node1 ~]# [ 3 -gt 2 ]
[root@node1 ~]# echo $?
0
[root@node1 ~]# [ 5 -lt 2 ]
[root@node1 ~]# echo $?
```

```
1
[root@node1 ~]# [ 5 -lt 2]
-bash: [: missing `]'
[root@node1 ~]# [5 -lt 2 ]
-bash: [5: command not found
#错误的
[root@node1 ~]# test 3 > 2
[root@node1 ~]# echo $?
0
#错误的
[root@node1 ~]# test 3 < 2
[root@node1 ~]# echo $?
0

[root@node1 ~]# test 3 -ge 2
[root@node1 ~]# echo $?
0
[root@node1 ~]# test 3 -ge 3
[root@node1 ~]# echo $?
0
[root@node1 ~]# test 3 -le 3
[root@node1 ~]# echo $?
0
[root@node1 ~]# test 3 -eq 3
[root@node1 ~]# echo $?
0
[root@node1 ~]# test 3 -eq 4
[root@node1 ~]# echo $?
1
[root@node1 ~]# test 3 -gt 2 && echo ok
ok
[root@node1 ~]# [ 3 -gt 2 ] && echo ok
```

```
ok
[root@node1 ~]# test 3 -gt 8 && echo error
[root@node1 ~]# [ 3 -gt 8 ] && echo error
```

[]和内容之间一定要有空格，否则抛错。

9、 分支语句

- if
- case

9.1 if

- **单分支结构**

```
if  [  条件判断  ]
 then
    //命令
fi
if  [  条件判断  ];  then
 条件成立执行，命令;
fi  将if反过来写,就成为fi  结束if语句
```

- **双分支结构**

```
if  [  条件1  ];then
  条件1成立执行，指令集1
else
  条件1不成执行指令集2;
 fi
```

- **多分支结构**

```
if  [  条件1  ];then
  条件1成立，执行指令集1
elif  [  条件2  ];then
 条件2成立，执行指令集2
else
  条件都不成立，执行指令集3
 fi
```

案例实战

```
#使用[]命令判断
[root@node1 ch4]# vim if1.sh
#!/bin/bash
if [ 3 -gt 2 ];then
   echo "3 > 2 is  true"
fi

if [ $1 -gt 3 ];then
   echo "$1 > 3 is true"
else
   echo "$1 <= 3 is true"
fi
[root@node1 ch4]# chmod +x if1.sh
[root@node1 ch4]# ./if1.sh 8
3 > 2 is  true
8 > 3 is true
```

猜数字案例

```
[root@node1 ch4]# vim guessNum.sh
#!/bin/bash
answer=20
if [ $1 -gt $answer ];then
```

```
    echo "猜大了!!"
elif [ $1 -eq $answer ];then
    echo "恭喜您,猜对了!"
else
    echo "猜小了!!"
fi
[root@node1 ch4]# chmod +x guessNum.sh
[root@node1 ch4]# ./guessNum.sh 30
猜大了!!
[root@node1 ch4]# ./guessNum.sh 20
恭喜您,猜对了!
[root@node1 ch4]# ./guessNum.sh 10
猜小了!!
```

9.2 case

基本格式:

```
case  $变量名称  in  “值1")
 程序段1
 ;;
“值2")
 程序段2
 ;;
*)
 exit  1
 ;;
esac
```

案例实战:

判断用户输入的是哪个数，1-7显示输入的数字，1显示 Mon,2:Tue,3:Wed,4:Thu,5:Fir,6-7:weekend,其它值的时候，提示：please input [1,7]，该如何实现?

```
[root@node1 ch4]# vim case1.sh
#!/bin/bash
read -p "please input a number[1,7]:" num
case $num in
1)
   echo "Mon"
;;
2)
   echo "Tue"
;;
3)
   echo "Wed"
;;
4)
   echo "Thu"
;;
5)
   echo "Fir"
;;
[6-7])
   echo "Weekend"
;;
*)
   echo "please input [1,7]"
;;
esac
[root@node1 ch4]# chmod +x case1.sh
#各种情况都要测试一遍，防止脚本写错。
```

```
[root@node1 ch4]# ./case1.sh
please input a number[1,7]:1
Mon
[root@node1 ch4]# ./case1.sh
please input a number[1,7]:2
Tue
[root@node1 ch4]# ./case1.sh
please input a number[1,7]:3
Wed
[root@node1 ch4]# ./case1.sh
please input a number[1,7]:4
Thu
[root@node1 ch4]# ./case1.sh
please input a number[1,7]:5
Fir
[root@node1 ch4]# ./case1.sh
please input a number[1,7]:6
Weekend
[root@node1 ch4]# ./case1.sh
please input a number[1,7]:7
Weekend
[root@node1 ch4]# ./case1.sh
please input a number[1,7]:8
please input [1,7]
[root@node1 ch4]# ./case1.sh
please input a number[1,7]:0
please input [1,7]
```

或者

```
[root@node1 ch4]# cp case1.sh case2.sh
[root@node1 ch4]# vim case2.sh
#!/bin/bash
```

```
read -p "please input a number[1,7]:" num
case $num in
1)
   echo "Mon"
   exit 0
;;
2)
   echo "Tue"
   exit 0
;;
3)
   echo "Wed"
   exit 0
;;
4)
   echo "Thu"
   exit 0
;;
5)
   echo "Fir"
   exit 0
;;
[6-7])
   echo "Weekend"
   exit 0
;;
*)
   echo "please input [1,7]"
   exit 1
;;
esac
[root@node1 ch4]# ./case2.sh
```

```
please input a number[1,7]:3
Wed
[root@node1 ch4]# echo $?
0
[root@node1 ch4]# ./case2.sh
please input a number[1,7]:9
please input [1,7]
[root@node1 ch4]# echo $?
1
```

10、 循环语句

10.1 while

- while循环语句

```
while  [  condition  ] ; do
 命令
done
#或者

while  [  condition  ]
 do
 命令
done
```

案例实战

案例一：每隔两秒打印系统负载情况，如何实现?

```
[root@node1 ch4]# vim while1.sh
#!/bin/bash
while true
do
```

```
    # 打印系统的负载日志
    uptime
    # 休眠2秒
    sleep 2
done
# 添加执行权限，不加执行权限的话，无法通过./while1.sh
去执行
[root@node1 ch4]# chmod +x while1.sh
[root@node1 ch4]# ./while1.sh
 12:35:27 up  3:20,  2 users,  load average:
0.02, 0.02, 0.05
 12:35:29 up  3:20,  2 users,  load average:
0.02, 0.02, 0.05
 12:35:31 up  3:20,  2 users,  load average:
0.01, 0.02, 0.05
 12:35:33 up  3:20,  2 users,  load average:
0.01, 0.02, 0.05
 12:35:35 up  3:20,  2 users,  load average:
0.01, 0.02, 0.05
 12:35:37 up  3:20,  2 users,  load average:
0.01, 0.02, 0.05
 12:35:39 up  3:20,  2 users,  load average:
0.01, 0.02, 0.05
 #结束使用Ctrl+C
```

案例二：使用while循环，编写shell脚本，计算1+2+3+...+100的和并输出，如何实现?

```
vim while2.sh

#!/bin/bash
sum=0
i=1
while [ $i -le 100 ]  #while和[之间要加一个空格
true则执行
do
 sum=$((sum+i))
 i=$((i+1)) #运算结果为变量赋值可以使用$(( … ))
done
echo "the result of '1+2+3+...+100' is $sum"
```

或者:

```
#!/bin/bash
sum=0
i=1
while [ $i -le 100 ]  #while和[之间要加一个空格
true则执行
do
 let sum=$sum+$i
 let i=$i+1 #运算结果为变量赋值可以使用$(( … ))
done
echo "the result of '1+2+3+...+100' is $sum"
```

10.2 for

基本格式

for...do...done循环

语法

```
for  变量名 in 变量取值列表
do
命令
done
```

提示：在此结构中“in变量取值列表”可省略，省略时相当于使用for i in “$@”.

```
for 男人  in  世界的男人
do
  if [ 有房 ]&&[ 有车 ] &&[ 存款 ] &&[ 会做家务 ] &&[ 帅气 ];then
       echo “女生喜欢”
  else
       rm -f 男人
  fi
done
```

直接列出元素的方法，比如：1 2 3 4 #=>需要空格隔开

```
[root@node1 ch4]#  vim for1.sh
#!/bin/bash
for num in 1 2 3 4
do
    echo $num
done
[root@node1 ch4]# chmod +x for1.sh
[root@node1 ch4]# ./for1.sh
1
2
3
4
```

目前是1到4，还可以这样写，如果到10万、1000W,还可以这样写吗?

使用大括号的方法

```
[root@node1 ch4]# echo {1..8}
1 2 3 4 5 6 7 8
[root@node1 ch4]# echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x
y z
[root@node1 ch4]# echo 10.13.20.{1..4}
10.13.20.1 10.13.20.2 10.13.20.3 10.13.20.4
```

使用{}升级脚本:

```
[root@node1 ch4]# cp for1.sh for2.sh
[root@node1 ch4]# vim for2.sh
#!/bin/bash
for num in {1..4}
do
    echo $num
done
[root@node1 ch4]# ./for2.sh
1
2
3
4
```

{1..4}产生的结果是连续的数字，如果需求是1到100的偶数或奇数或3的倍数等需求时，是无法满足。

使用seq –s 分隔符 起始 步长 终点 seq -s " " 1 1 4

```
[root@node1 ch4]# seq -s " " 1 1 4
1 2 3 4
# [1,50]之间的奇数
[root@node1 ch4]# seq -s " " 1 2 50
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33
35 37 39 41 43 45 47 49
# [1,50]之间的偶数
[root@node1 ch4]# seq -s " " 2 2 50
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34
36 38 40 42 44 46 48 50
# [1,50]之间的3的倍数
[root@node1 ch4]# seq -s " " 3 3 50
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
# [1,50]之间的5的倍数
[root@node1 ch4]# seq -s " " 5 5 50
```

```
5 10 15 20 25 30 35 40 45 50
[root@node1 ch4]# cp for1.sh for3.sh
[root@node1 ch4]# vim for3.sh
#!/bin/bash
for num in `seq -s " " 1 1 4`
do
    echo $num
done
[root@node1 ch4]# ./for3.sh
1
2
3
4
# 另外一种实现方式 $()
[root@node1 ch4]# cp for3.sh for4.sh
[root@node1 ch4]# vim for4.sh
#!/bin/bash
for num in $(seq -s " " 1 1 4)
do
    echo $num
done
[root@node1 ch4]# ./for4.sh
1
2
3
4
```

11、 正则表达式

正则表达式使用单个字符串来描述、匹配一系列符合某个语法规则的字符串。在文本编辑器里，正则表达式通常被用来检索、替换符合某个模式的文本。在Linux中sed，grep，awk，vim等命令都支持通过正则表达式进行模式匹配。

案例用到的数据文件hello.txt

```
hello world
are you ok?
areyou ok?
areyou are youok?
aaare you ok?
aare you ok
aaaare you ok
abcre you ok?
xxre you ok
are yyyou ok?
xk
zk
ok
yk
zzk
zxzxk
bxx
cxx
dxx
areyou are youok?
zk kz 1
kz zk 2
okk koo 3
zkkz
kzzk
```

11.1 基本匹配

1. "." 匹配任意单个字符

```
#匹配“f任意单个字符p”
[root@node1 ch4]# grep f.p /etc/passwd
ftp:x:14:50:FTP
User:/var/ftp:/sbin/nologin
#或者
[root@node1 ch4]# cat /etc/passwd | grep
f.p
ftp:x:14:50:FTP
User:/var/ftp:/sbin/nologin
#
[root@node1 ch4]# grep  "a.re"  hello.txt
[root@node1 ch4]# grep  "a..re"  hello.txt
```

2. ^ 匹配一行的开头

```
# 匹配文件/etc/passwd文件中以r开头的行
[root@node1 ch4]# cat /etc/passwd|grep ^r
root:x:0:0:root:/root:/bin/bash
[root@node1 ch4]# grep ^r /etc/passwd
root:x:0:0:root:/root:/bin/bash
#匹配行首，该行第二个字符一定得是k
[root@node1 ch4]# grep  "^.k"  hello.txt
xk
zk
ok
yk
zk kz 1
okk koo 3
zkkz
```

3. $ 匹配一行的结尾

```
[root@node1 ch4]# cat /etc/group|grep x$
mail:x:12:postfix
[root@node1 ch4]# grep x$ /etc/group
mail:x:12:postfix
#该行最少两个字符，最后一个是k
[root@node1 ch4]# grep  ".k$"  hello.txt
aare you ok
aaaare you ok
xxre you ok
xk
zk
ok
yk
zzk
zxzxk
kzzk
```

总结：^$ 匹配空行

```
[root@node1 ch4]# cat /etc/profile|grep ^$



[root@node1 ch4]#
```

4. * 不单独使用，它和上一个字符连用，表示匹配上一个字符0次或多次

```
#匹配rt rot root 等等
[root@node1 ch4]# cat /etc/group |grep
ro*t
root:x:0:gm
[root@node1 ch4]# grep ro*t /etc/group
root:x:0:gm
#匹配are aare xre，0到多个a字符
[root@node1 ch4]# grep "a*re" hello.txt
```

5. [ ] 表示匹配某个范围内的一个字符，例如

[1,3] 匹配1或者3

[1-5] 匹配1 2 3 4 5中的任意一个字符

[b-d] 匹配b c d中的任意一个字符

```
[root@node1 ch4]# grep "[b-d]" hello.txt
hello world
abcre you ok?
bxx
cxx
dxx
```

[xz] 匹配 x z中的任意一个字符

```
#匹配zk和xk
grep  "[xz]k"  hello.txt
```

```
[root@node1 ch4]# grep  "[xz]k"  hello.txt
xk
zk
zzk
zxzxk
zk kz 1
kz zk 2
zkkz
kzzk
```

[^zx]匹配除了z x之外的任意一个字符,比如：匹配不是zk和xk的

```
[root@node1 ch4]# grep  "[^zx]k"  hello.txt
are you ok?
areyou ok?
areyou are youok?
aaare you ok?
aare you ok
aaaare you ok
abcre you ok?
xxre you ok
are yyyou ok?
ok
yk
areyou are youok?
zk kz 1
okk koo 3
zkkz
```

[a-z] 匹配任意一个小写字母

[A-Z] 匹配任意一个大写字母

6. \ 表示转义，并不会单独使用。

由于所有特殊字符都有其特定匹配模式，当我们想匹配某一特殊字符本身时（例如，我想找出所有包含 '$' 的行），就会碰到困难。此时我们就要将转义字符和特殊字符连用，来表示特殊字符本身，例如

```
[root@node1 ch4]# grep \$PATH /etc/profile
                PATH=$PATH:$1
                PATH=$1:$PATH
export PATH=$PATH:$JAVA_HOME/bin
```

7. <,> 单词首尾边界

```
[root@node1 ch4]# grep  "\<are\>"
hello.txt #匹配单词边界
are you ok?
areyou are youok?
are yyyou ok?
```

```
areyou are youok?
[root@node1 ch4]#
[root@node1 ch4]# grep  "\<are"  hello.txt
#匹配单词开头
are you ok?
areyou ok?
areyou are youok?
are yyyou ok?
areyou are youok?
[root@node1 ch4]#
[root@node1 ch4]# grep  "re\>"  hello.txt
# 匹配单词尾
are you ok?
areyou are youok?
aaare you ok?
aare you ok
aaaare you ok
abcre you ok?
xxre you ok
are yyyou ok?
areyou are youok?
```

11.2 扩展匹配

grep默认工作于基本模式

-E选项让grep工作于扩展模式

基本正则表达式中元字符

- ?，+，{，|，(和)丢失了特殊意义，需要加\，反斜杠：
- ?，+，{，|，(和)，扩展模式不需要加反斜杠

重复匹配：

- ? 匹配0到1次

- +匹配1到多次
- {n} 匹配n次
- {n,} 匹配n到多次
- {m,n} 匹配m到n次

1. | 连接操作符，并集 (<are>)| (<you>)

2. +匹配一个到多个任意字符

```
#匹配a一个到多个任意字符re
[root@node1 ~]# grep "a+re" hello.txt
#发现查询不出来，为什么？
[root@node1 ~]# grep "a\+re" hello.txt
#或者
[root@node1 ~]# grep -E "a+re" hello.txt
```

3. 匹配次数

```
[root@node1 ch4]# grep -E  "a{3}"
hello.txt #匹配该行中3个a重复的
aaare you ok?
aaaare you ok
[root@node1 ch4]# grep  -E  "a{3,}"
hello.txt #匹配该行中>=3个a的
aaare you ok?
aaaare you ok
[root@node1 ch4]# grep  -E  "a{2,3}"
hello.txt #匹配该行中[2,3]个a的
aaare you ok?
aare you ok
aaaare you ok #bug并没有过滤掉>=4个a的
```

12、 shell 脚本检查

• sh [-nvx] scripts.sh

• 选项与参数:

• -n :不执行script,仅查询语法的问题; !!

• -v :在执行script前,先将scripts的内容输出到屏幕上;

• -x :将使用到的script内容显示到屏幕上,这是很有用的参数; ！ ！ ！

13、 企业面试真题

13.1 小米

问题1：Linux中如何将以a开头的文件找出来?

参考答案:

当前目录下查找：ls a* 或者 ll a*

整个文件系统中查找：find / -name a*

13.2 搜狐

问题1：Shell脚本里如何检查一个文件是否存在?

参考答案:

```
#!/bin/bash
if [ -f $1 ]; then
    echo "文件$1存在!"
else
    echo "文件$1不存在!"
fi
```

13.3 金山

问题1：根据用户给定的目录，输出该目录（包含其子级目录）下文件大小最大的文件。需要考虑递归子目录。

提示：$IFS默认是空格，制表符和换行符，IFS=$'\n' 使用$获取\n换行符的ascii码作为IFS的值，此时分隔符就是换行符了。

参考答案

思路：使用du命令加-a遍历用户指定目录（$1获取）的所有文件，使用管道将结果传递给sort，让sort使用数值序倒序排序，依次输出各个条目。

```
# !/bin/bash
oldIFS=$IFS
IFS=$'\n'
# 将for循环获取不同元素的标记修改为换行符
# for循环获取元素的时候使用空格区分各个不同的元素
for  item  in  `du  -a  $1 |  sort  -nr`
do
echo  $item
done
IFS=$oldIFS
#用完后重置IFS变量的值。
```

因为要获取最大的文件，需要改进

```
#!/bin/bash
oldIFS=$IFS
IFS=$'\n'
# 将for循环获取不同元素的标记修改为换行符
# for循环获取元素的时候使用空格区分各个不同的元素
for  item  in  `du -a  $1 |  sort -nr`
do
   fileName=`echo  $item | awk '{print  $2}'`
   if  [  -f  $fileName  ]; then
      echo  $fileName
      break
    fi
done
IFS=$oldIFS
#用完后重置IFS变量的值。
```