

# shell脚本编程

## 课程总目标

- 该课程主要包括以下内容：

① Shell的基本语法结构

如：变量定义、条件判断、循环语句(for、until、while)、分支语句、函数和数组等；

② 基本正则表达式的运用；

③ 文件处理三剑客：grep、sed、awk工具的使用；

④ 使用shell脚本完成一些较复杂的任务，如：服务搭建、批量处理等。

**说明：以上内容仅仅是基本要求，还有很多更深更难的语法需要扩充学习。**

- 今日目标

- 熟悉grep、cut、sort等小工具和shell中的通配符的使用
- 熟练掌握shell变量的定义和获取（重点）
- 能够进行shell简单的四则运算
- 熟悉条件判断语句,如判断整数，判断字符串等
- 熟悉流程控制语句基本语法

## 学习前奏

### 1. 文件处理工具

#### 1.1 grep工具

```
grep用于根据关键字进行行过滤
grep options 'keys' filename
OPTIONS:
  -i: 不区分大小写
  -v: 查找不包含指定内容的行,反向选择
  -w: 按单词搜索
  -o: 打印匹配关键字
  -c: 统计匹配到的次数
  -n: 显示行号
  -r: 逐层遍历目录查找
  -A: 显示匹配行及前面多少行
  -B: 显示匹配行及后面多少行
  -C: 显示匹配行前后多少行
  -l: 只列出匹配的文件名
  -L: 列出不匹配的文件名
  -e: 使用正则匹配
  -E: 使用扩展正则匹配
  ^key: 以关键字开头
  key$: 以关键字结尾
  ^$: 匹配空行
```

--color=auto : 可以将找到的关键词部分加上颜色的显示

临时设置:

```
# alias grep='grep --color=auto' //只针对当前终端和当前用户生效
```

永久设置:

1) 全局 (针对所有用户生效)

```
vim /etc/bashrc
alias grep='grep --color=auto'
source /etc/bashrc
```

2) 局部 (针对具体的某个用户)

```
vim ~/.bashrc
alias grep='grep --color=auto'
source ~/.bashrc
```

示例:

```
# grep -i root passwd      忽略大小写匹配包含root的行
# grep -w ftp passwd       精确匹配ftp单词
# grep -w hello passwd     精确匹配hello单词;自己添加包含hello的行到文件中
[root@server shell01]# grep -w hello passwd
ftpabc:helloworld:hello:world
[root@server shell01]# grep hello passwd
ftpabc:helloworld:hello:world

# grep -wo ftp passwd      打印匹配到的关键字ftp
# grep -n root passwd     打印匹配到root关键字的行号
# grep -ni root passwd    忽略大小写匹配统计包含关键字root的行
# grep -nic root passwd   忽略大小写匹配统计包含关键字root的行数
# grep -i ^root passwd    忽略大小写匹配以root开头的行
# grep bash$ passwd       匹配以bash结尾的行
# grep -n ^$ passwd       匹配空行并打印行号
# grep ^# /etc/vsftpd/vsftpd.conf 匹配以#号开头的行
# grep -v ^# /etc/vsftpd/vsftpd.conf 匹配不以#号开头的行
# grep -A 5 mail passwd   匹配包含mail关键字及其后5行
# grep -B 5 mail passwd   匹配包含mail关键字及其前5行
# grep -C 5 mail passwd   匹配包含mail关键字及其前后5行
```

## 1.2 cut工具

cut用于列截取

-c: 以字符为单位进行分割。  
-d: 自定义分隔符, 默认为制表符。  
-f: 与-d一起使用, 指定显示哪个区域。

```
# cut -d: -f1 1.txt      以:冒号分割, 截取第1列内容
# cut -d: -f1,6,7 1.txt  以:冒号分割, 截取第1,6,7列内容
# cut -c4 1.txt          截取文件中每行第4个字符
# cut -c1-4 1.txt        截取文件中每行的1-4个字符
# cut -c4-10 1.txt       从第4个字符开始截取后面所有字符
# cut -c5- 1.txt         从第5个字符开始截取后面所有字符
```

课堂练习：

用小工具列出你当系统的运行级别。5/3

```
[root@server shell01]# tail -1 /etc/inittab |cut -d: -f2
3
[root@server shell01]# runlevel
N 3
[root@server shell01]# runlevel |cut -d' ' -f2
3
```

## 1.3 sort工具

**sort**：将文件的每一行作为一个单位，从首字符向后，依次按ASCII码值进行比较，最后将他们按升序输出。

- u : 去除重复行
- r : 降序排列，默认是升序
- o : 将排序结果输出到文件中 类似 重定向符号>
- n : 以数字排序，默认是按字符排序
- t : 分隔符
- k : 第N列
- b : 忽略前导空格。
- R : 随机排序，每次运行的结果均不同。

示例：

```
# sort -n -t: -k3 1.txt      按照用户的uid进行升序排列
# sort -nr -t: -k3 1.txt    按照用户的uid进行降序排列
# sort -n 2.txt             按照数字排序
# sort -nu 2.txt            按照数字排序并且去重
# sort -nr 2.txt
# sort -nru 2.txt
# sort -nru 2.txt
# sort -n 2.txt -o 3.txt     按照数字排序并将结果重定向到文件
# sort -R 2.txt
# sort -u 2.txt
```

## 1.4 uniq工具

**uniq**：去除连续重复行

- i: 忽略大小写
- c: 统计重复行次数
- d: 只显示重复行

```
# uniq 2.txt
# uniq -d 2.txt
# uniq -dc 2.txt
```

## 1.5 tee工具

**tee**工具从标准输入读取并写入标准输出和文件，即：双向覆盖重定向<屏幕输出|文本输入>  
**-a** 双向追加重定向

```
# echo hello world
# echo hello world|tee file1
# cat file1
# echo 999|tee -a file1
# cat file1
```

## 1.6 paste工具

**paste**工具用于合并文件行

**-d**: 自定义间隔符，默认是tab  
**-s**: 串行处理，非并行

```
[root@server shell01]# cat file1
hello world
999
[root@server shell01]# cat file2
777 888
hi heimeimei
# paste file1 file2
# paste file2 file1
# cat file1
# paste -d: file1 file2
# cat file1
# cat file2
# paste -s file1 file2
```

## 1.7 tr工具

**字符转换：替换，删除**

**tr**用来从标准输入中通过替换或删除操作进行字符转换；主要用于删除文件中控制字符或进行字符转换。  
使用**tr**时要转换两个字符串：字符串1用于查询，字符串2用于处理各种转换。

```
commands|tr 'string1' 'string2'
tr 'string1' 'string2' < filename
```

```
tr options 'string1' < filename
```

**-d** 删除字符串1中所有输入字符。  
**-s** 删除所有重复出现字符序列，只保留第一个；即将重复出现字符串压缩为一个字符串。

**a-z** 任意小写

**A-Z** 任意大写

**0-9** 任意数字

**[:alnum:]** all letters and digits//所有字母和数字

```

[:alpha:]      all letters//所有字母
[:blank:]      all horizontal whitespace//所有水平空白
[:cntrl:]      all control characters//所有控制字符
\b Ctrl-H      退格符
\f Ctrl-L      走行换页
\n Ctrl-J      新行
\r Ctrl-M      回车
\t Ctrl-I      tab键

[:digit:]      all digits//所有数字
[:graph:]      all printable characters, not including space//所有可打印的字符, 不包含空格
[:lower:]      all lower case letters//所有小写字母
[:print:]      all printable characters, including space//所有可打印的字符, 包含空格
[:punct:]      all punctuation characters//所有的标点符号
[:space:]      all horizontal or vertical whitespace//所有水平或垂直的空格
[:upper:]      all upper case letters//所有大写字母
[:xdigit:]     all hexadecimal digits//所有十六进制数字
[=CHAR=]       all characters which are equivalent to CHAR//所有字符

```

[root@server shell01]# cat 3.txt 自己创建该文件用于测试

```

ROOT:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
boss02:x:516:511:./home/boss02:/bin/bash
vip:x:517:517:./home/vip:/bin/bash
stu1:x:518:518:./home/stu1:/bin/bash
mailnull:x:47:47:./var/spool/mqueue:/sbin/nologin
smmsp:x:51:51:./var/spool/mqueue:/sbin/nologin
aaaaaaaaaaaaaaaaaaaaa
bbbbbb11111122222222222233333333cccccccc
hello world 888
666
777
999

```

```

1116 tr -d '[:/]' < 3.txt      删除文件中的:/
1117 cat 3.txt |tr -d '[:/]'    删除文件中的:/
1118 cat 3.txt
1119 tr '[0-9]' '@' < 3.txt      将文件中的数字替换为@符号
1120 tr '[a-z]' '[A-Z]' < 3.txt  将文件中的小写字母替换成大写字母
1124 tr -s '[a-z]' < 3.txt      匹配小写字母并将重复的压缩为一个
1125 tr -s '[a-z0-9]' < 3.txt    匹配小写字母和数字并将重复的压缩为一个
1126 tr -d '[:digit:]' < 3.txt   删除文件中的数字
1128 tr -d '[:blank:]' < 3.txt   删除水平空白

```

```
1131 tr -d '[:space:]' < 3.txt 删除所有水平和垂直空白
```

## 小试牛刀

1. 使用小工具分别截取当前主机IP；截取NETMASK；截取广播地址；截取MAC地址

```
[root@server shell01]# ifconfig eth1|grep Bcast|cut -d: -f2|cut -d' ' -f1
10.1.1.250
[root@server shell01]# ifconfig eth1|grep Bcast|cut -d: -f2|tr -d '[ a-zA-Z]'
10.1.1.250
[root@server shell01]# ifconfig eth1|grep Bcast|tr -d '[:a-zA-Z]'|tr ' ' '@'|tr -s '@'|tr '@'
'\n'|grep -v ^$
10.1.1.250
10.1.1.255
255.255.255.0

[root@server shell01]# ifconfig eth0|grep 'Bcast'|tr -d [:alpha:]|tr '[:]' '\n'|grep -v ^$
10.1.1.2
10.1.1.255
255.255.255.0

[root@server shell01]# ifconfig eth1|grep HWaddr|cut -d' ' -f11
00:0C:29:23:5E:72

[root@server shell01]# ifconfig eth0|grep HWaddr|tr -s ' '|cut -d' ' -f5
00:0C:29:41:A2:FC

[root@server shell01]# ifconfig eth1|grep HWaddr|tr -s ' '|cut -d' ' -f5
00:0C:29:23:5E:72
```

2. 将系统中所有普通用户的用户名、密码和默认shell保存到一个文件中，要求用户名密码和默认shell之间用tab键分割

```
[root@server shell01]# grep bash$ passwd |grep -viE 'root|mysql'
pos:x:500:500:./home/pos:/bin/bash
stu1:x:501:501:./home/stu1:/bin/bash
[root@server shell01]# grep bash$ passwd |grep -viE 'root|mysql'|cut -d: -f1,2,7
pos:x:/bin/bash
stu1:x:/bin/bash
[root@server shell01]# grep bash$ passwd |grep -viE 'root|mysql'|cut -d: -f1,2,7|tr ':' '\t'
pos  x  /bin/bash
stu1 x  /bin/bash
[root@server shell01]# grep bash$ passwd |grep -viE 'root|mysql'|cut -d: -f1,2,7|tr ':' '\t'
|tee a.txt
pos  x  /bin/bash
stu1 x  /bin/bash
[root@server shell01]# cat a.txt
pos  x  /bin/bash
stu1 x  /bin/bash
```

注释:

-E 匹配扩展正则表达式, |代表或者, 是一个扩展正则

```
[root@server shell01]# grep bash$ passwd|grep -vi ^root|cut -d: -f1,2,7|tr ':' '\t'|tee 4.txt
```

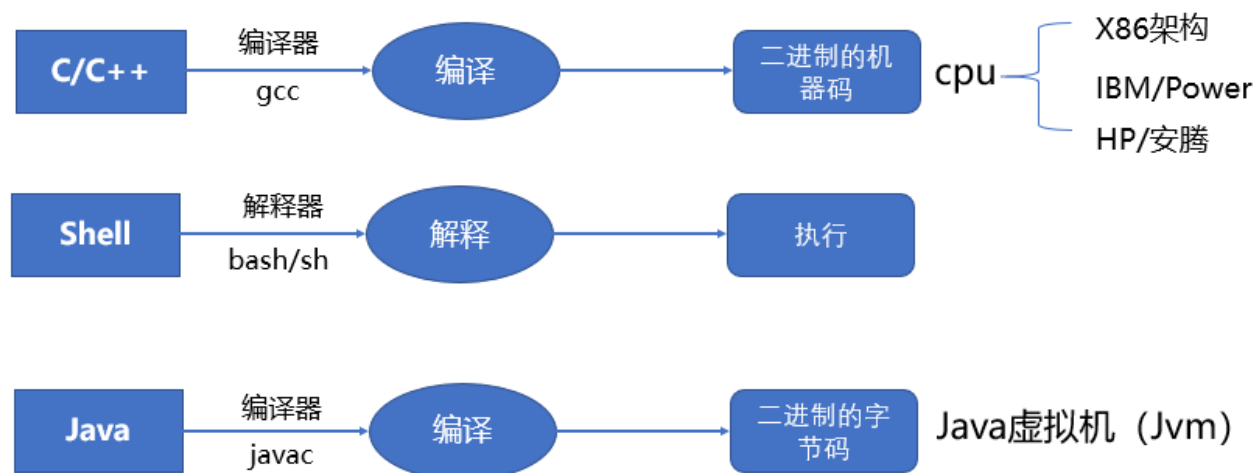
## 2. 编程语言分类

- 编译型语言:

程序在执行之前需要一个专门的编译过程, 把程序编译成为机器语言文件, 运行时不需要重新翻译, 直接使用编译的结果就行了。程序执行效率高, 依赖编译器, 跨平台性差些。如C、C++

- 解释型语言:

程序不需要编译, 程序在运行时由解释器翻译成机器语言, 每执行一次都要翻译一次。因此效率比较低。比如Python/JavaScript/ Perl /ruby/Shell等都是解释型语言。

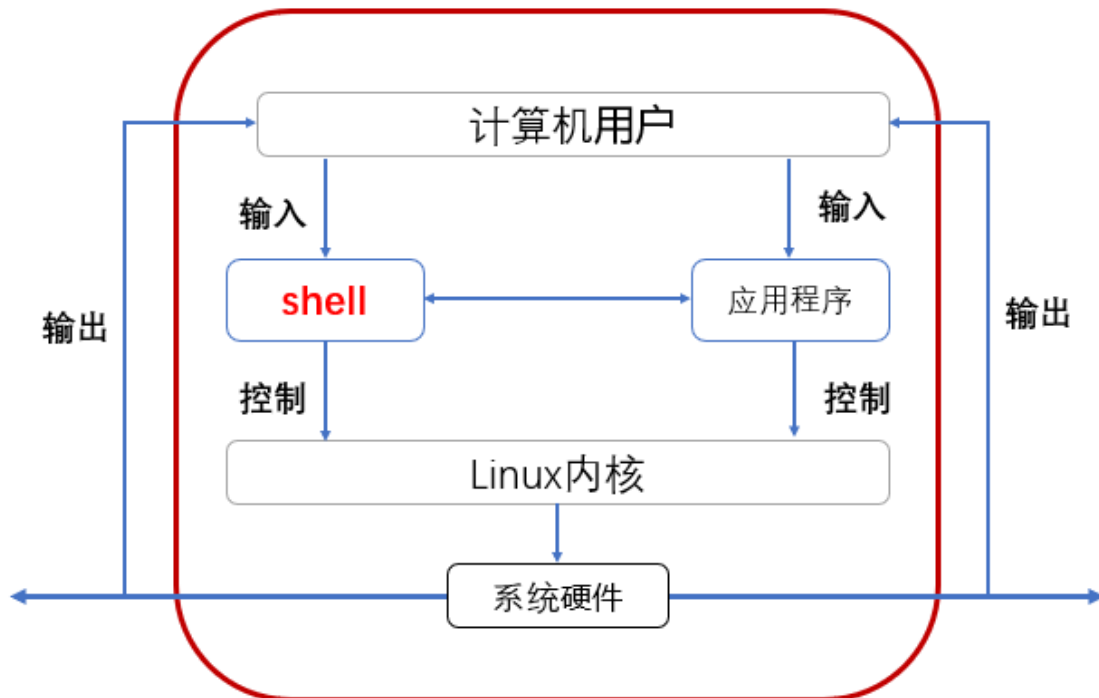


- 总结:

编译型语言比解释型语言速度较快, 但是不如解释型语言跨平台性好。如果做底层开发或者大型应用程序或者操作系统开发一般都用编译型语言; 如果是一些服务器脚本及一些辅助的接口, 对速度要求不高、对各个平台的兼容性有要求的话则一般都用解释型语言。

## 3. shell介绍

shell介于内核与用户之间，负责命令的解释



总结:

- **shell: 人机交互的一个桥梁**
- **shell种类:**

```
[root@Misshou ~]# cat /etc/shells
/bin/sh      #是bash shell的一个快捷方式
/bin/bash    #bash shell是大多数Linux默认的shell，包含的功能几乎可以涵盖shell所有的功能
/sbin/nologin #表示非交互，不能登录操作系统
/bin/dash    #小巧，高效，功能相比少一些
/bin/tcsh    #是csh的增强版，完全兼容csh
/bin/csh     #具有C语言风格的一种shell，具有许多特性，但也有一些缺陷
```

- 用户在终端（终端就是bash的接口）输入命令  
| bash //bash就是shell的一种类型（bash shell） | kernel | 物理硬件等

## 4. shell脚本

- **什么是shell脚本?**

- 一句话概括

简单来说就是将需要执行的命令保存到文本中，按照顺序执行。它是解释型的，意味着不需要编译。

- 准确叙述

**若干命令 + 脚本的基本格式 + 脚本特定语法 + 思想 = shell脚本**

- **什么时候用到脚本?**



重复化、复杂化的工作，通过把工作的命令写成脚本，以后仅仅需要执行脚本就能完成这些工作。

- ①自动化分析处理
- ②自动化备份
- ③自动化批量部署安装
- ④等等...

- **如何学习shell脚本？**

1. 尽可能记忆更多的命令
2. 掌握脚本的标准的格式（指定魔法字节、使用标准的执行方式运行脚本）
3. 必须**熟悉掌握**脚本的基本语法（重点）

- **学习脚本的秘诀：**

多看（看懂）——>多模仿（多练）——>多思考

- **脚本的基本写法：**

```
#!/bin/bash
//脚本第一行， #! 魔法字符，指定脚本代码执行的程序。即它告诉系统这个脚本需要什么解释器来执行，也就是使用
哪一种Shell

//以下内容是对脚本的基本信息的描述
# Name: 名字
# Desc:描述describe
# Path:存放路径
# Usage:用法
# Update:更新时间

//下面就是脚本的具体内容
commands
...
```

- **脚本执行方法：**

- 标准脚本执行方法（建议）：（魔法字节指定的程序会生效）

```
[root@MissHou shell01]# cat 1.sh
#!/bin/bash
#xxxx
#xxx
#xxx
hostname
date
[root@MissHou shell01]# chmod +x 1.sh
[root@MissHou shell01]# ll
total 4
-rwxr-xr-x 1 root root 42 Jul 22 14:40 1.sh
[root@MissHou shell01]# ./1.sh
MissHou.itcast.cc
Sun Jul 22 14:41:00 CST 2018
[root@MissHou shell01]# ./1.sh
```

```
MissHou.itcast.cc
Sun Jul 22 14:41:30 CST 2018
```

- 非标准的执行方法（不建议）：（魔法字节指定的程序不会运作）

```
[root@MissHou shell01]# bash 1.sh
MissHou.itcast.cc
Sun Jul 22 14:42:51 CST 2018
[root@MissHou shell01]# sh 1.sh
MissHou.itcast.cc
Sun Jul 22 14:43:01 CST 2018
[root@MissHou shell01]#
[root@MissHou shell01]# bash -x 1.sh
+ hostname
MissHou.itcast.cc
+ date
Sun Jul 22 14:43:20 CST 2018
```

-x:一般用于排错，查看脚本的执行过程

-n:用来查看脚本的语法是否有问题

注意：如果脚本没有加可执行权限，不能使用标准的执行方法执行，bash 1.sh

其他：

```
[root@server shell01]# source 2.sh
server
Thu Nov 22 15:45:50 CST 2018
[root@server shell01]# . 2.sh
server
Thu Nov 22 15:46:07 CST 2018
```

source 和 . 表示读取文件，执行文件里的命令

## 5. bash基本特性

### 5.1 命令和文件自动补全

Tab只能补全命令和文件（RHEL6/Centos6）

### 5.2 常见的快捷键

<b>^c</b>	终止前台运行的程序	
<b>^z</b>	将前台运行的程序挂起到后台	
<b>^d</b>	退出 等价exit	
<b>^l</b>	清屏	
<b>^a</b>	home 光标移到命令行的最前端	//编辑命令
<b>^e</b>	end 光标移到命令行的后端	//编辑命令
<b>^u</b>	删除光标前所有字符	//编辑命令
<b>^k</b>	删除光标后所有字符	//编辑命令
<b>^r</b>	搜索历史命令	//命令需连续

## 5.3 常用的通配符（重点）

```
*: 匹配0或多个任意字符
?: 匹配任意单个字符
[list]: 匹配[list]中的任意单个字符    //[abc]、[1-4]
[!list]: 匹配除list中的任意单个字符
{string1,string2,...}: 匹配string1,string2或更多字符串
```

## 5.4 bash中的引号（重点）

- 双引号"`"` :会把引号的内容当成整体来看待，允许通过`$`符号引用其他变量值
- 单引号"`'`" :会把引号的内容当成整体来看待，禁止引用其他变量值，shell中特殊符号都被视为普通字符
- 反撇号"```" :反撇号和`$()`一样，引号或括号里的命令会优先执行，如果存在嵌套，反撇号不能用

```
[root@server dir1]# echo "${hostname}"
server
[root@server dir1]# echo '${hostname}'
$(hostname)
[root@server dir1]# echo "hello world"
hello world
[root@server dir1]# echo 'hello world'
hello world

[root@server dir1]# echo $(date +%F)
2018-11-22
[root@server dir1]# echo `echo $(date +%F)`
2018-11-22
[root@server dir1]# echo `date +%F`
2018-11-22
[root@server dir1]# echo `echo `date +%F``
date +%F
[root@server dir1]# echo $(echo `date +%F`)
2018-11-22
```

# 变量的定义

## 1. 变量的分类

- **本地变量**：当前用户自定义的变量。当前进程中有效，其他进程及当前进程的子进程无效。
- **环境变量**：当前进程有效，并且能够被子进程调用。
  - 查看当前用户的环境变量 `//env`
  - 查询当前用户的所有变量(临时变量与环境变量) `//set`
  - **export** `//将当前变量变成环境变量`

```
[root@MissHou tmp]# export A=hello      //临时将一个本地变量（临时变量）变成环境变量
[root@MissHou tmp]# env|grep ^A
A=hello
```

永久生效:

```
vim /etc/profile 或者 ~/.bashrc
export A=hello
或者
A=hello
export A
```

- **全局变量**: 全局所有的用户和程序都能调用, 且继承, 新建的用户也默认能调用.

```
$HOME/.bashrc          当前用户的bash信息 (alias、umask等)
$HOME/.bash_profile    当前用户的环境变量
$HOME/.bash_logout     每个用户退出当前shell时最后读取的文件

/etc/bashrc            使用bash shell用户全局变量
grep --color=auto
umask

/etc/profile           系统和每个用户的环境变量信息

mycat_home=/usr/local/mycat/bin
export mycat_home
执行mycat命令
# mycat
$ mycat

/etc/profile→$HOME/.bash_profile→$HOME/.bashrc→/etc/bashrc→$HOME/.bash_logout
```

- **系统变量(内置bash中变量)**: shell本身已经固定好了它的名字和作用.

**\$?**: 上一条命令执行后返回的状态, 当返回状态值为0时表示执行正常, 非0值表示执行异常或出错  
若退出状态值为0, 表示命令运行成功  
若退出状态值为127, 表示command not found  
若退出状态值为126, 表示找到了该命令但无法执行 (权限不够)  
若退出状态值为1&2, 表示没有那个文件或目录

**\$\$**: 当前所在进程的进程号      echo \$\$    eg: kill -9 `echo \$\$` = exit    退出当前会话  
**#!**: 后台运行的最后一个进程号    (当前终端)    # gedit &  
**!\$**: 调用最后一条命令历史中的参数  
**!!**: 调用最后一条命令历史

**\$#**: 脚本后面接的参数的个数  
 **\$\***: 脚本后面所有参数, 参数当成一个整体输出, 每一个变量参数之间以空格隔开  
 **@\$**: 脚本后面所有参数, 参数是独立的, 也是全部输出  
 **\$0**: 当前执行的进程/程序名    echo \$0  
 **\$1~\$9**: 位置参数变量  
 **\${10}~\${n}**: 扩展位置参数变量    第10个位置变量必须用{}大括号括起来

```
[root@MissHou shell01]# cat 2.sh
#!/bin/bash
#xxxx
echo "\$0 = $0"
echo "\$# = $# "
echo "\$* = $*"
echo "\$@ = @$"
echo "\$1 = $1"
echo "\$2 = $2"
echo "\$3 = $3"
echo "\$11 = ${11}"
echo "\$12 = ${12}"
```

了解\$\*和\$@的区别:

\$\* :表示将变量看成一个整体

\$@ :表示变量是独立的

```
#!/bin/bash
for i in "$@"
do
echo $i
done

echo "=====我是分割线======"

for i in "$*"
do
echo $i
done
```

```
[root@MissHou shell01]# bash 3.sh a b c
a
b
c
=====我是分割线=====
a b c
```

## 2. 什么时候需要定义变量?

- 如果某个内容需要多次使用, 并且在代码中**重复出现**, 那么可以用变量代表该内容。这样在修改内容的时候, 仅仅需要修改变量的值。
- 在代码运作的过程中, 可能会把某些命令的执行结果保存起来, 后续代码需要使用这些结果, 就可以直接使用这个变量。

## 3. 变量的定义规则

1. 默认情况下, shell里定义的变量是不分类型的, 可以给变量赋与任何类型的值; 等号两边不能有空格, 对于有空格的字符串做为赋值时, 要用引号引起来

变量名=变量值

2. 变量的获取方式: `$变量名`      `${变量名}`

```
[root@MissHou shell01]# a=12345678
[root@MissHou shell01]# echo $a
12345678
[root@MissHou shell01]# echo ${a}
12345678
[root@MissHou shell01]# echo ${a:2:3}    a表示变量名; 2表示从第3个字符开始; 3表示后面3个字符
345
```

如果获取变量的全部两个都可以; 如果获取变量的某一部分, 用`${}`

3. 取消变量:      `unset 变量名`

4. 变量名区分大小写, 同名称但大小写不同的变量名是不同的变量

5. 变量名可以是字母或数字或下划线, 但是不能以数字开头或者特殊字符

```
[root@MissHou shell01]# 1a=hello
-bash: 1a=hello: command not found
[root@MissHou shell01]# ?a=hello
-bash: ?a=hello: command not found
[root@MissHou shell01]# _a=hello
[root@MissHou shell01]# echo $_a
hello
```

6. 命令的执行结果可以保存到变量

```
[root@server shell01]# kernel=`uname -r`
[root@server shell01]# echo $kernel
2.6.32-431.el6.x86_64
[root@server shell01]# name=$(uname -n)
[root@server shell01]# echo $name
server.itcast.cc
```

7. 有类型变量 `declare`

- i 将变量看成整数
- r 使变量只读 `readonly`
- x 标记变量通过环境导出 `export`
- a 指定为索引数组 (普通数组) ; 查看普通数组
- A 指定为关联数组; 查看关联数组

```
[root@server shell01]# a=10
[root@server shell01]# b=20
[root@server shell01]# echo $a+$b
10+20
```

```
[root@server shell01]# declare -i a=2
[root@server shell01]# declare -i b=4
[root@server shell01]# declare -i c=$a+$b
[root@server shell01]# echo $c
6
```

```
[root@server shell01]# AAAA=hello
[root@server shell01]# export AAAA
[root@server shell01]# env|grep AAAA
AAAA=hello
[root@server shell01]# declare -x BBBB=hello
[root@server shell01]# env|grep BBBB
BBBB=hello
```

## 8. 数组

普通数组：只能使用整数作为数组索引(元素的下标)

关联数组：可以使用字符串作为数组索引(元素的下标)

普通数组定义：用括号来表示数组，数组元素(变量)用“空格”符号分割开。定义数组的一般形式为：

一次赋一个值：

变量名=变量值

```
array[0]=v1
```

```
array[1]=v2
```

```
array[3]=v3
```

一次赋多个值：

```
array=(var1 var2 var3 var4)
```

```
array1=(`cat /etc/passwd`) //将文件中每一行赋值给array1数组
```

```
array2=(`ls /root`)
```

```
array3=(harry amy jack "Miss Hou")
```

```
array4=(1 2 3 4 "hello world" [10]=linux)
```

读取数组：

`${array[i]}` i表示元素的下标

使用@ 或 \* 可以获取数组中的所有元素：

获取第一个元素

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

```
echo ${array[*]} 获取数组里的所有元素
```

```
echo ${#array[*]} 获取数组里所有元素个数
```

```
echo ${!array[@]} 获取数组元素的索引下标
```

```
echo ${array[@]:1:2} 访问指定的元素；1代表从下标为1的元素开始获取；2代表获取后面几个元素
```

查看普通数组信息：

```
[root@server ~]# declare -a
```

```
declare -a array3=('[0]="harry" [1]="amy" [2]="jack" [3]="Miss Hou")'
```

```
declare -a array4=('[0]="1" [1]="2" [2]="3" [3]="4" [4]="hello world" [10]="linux")'
```

//以下为课堂视频内容：

```
[root@server shell01]# array[0]=var1
```

```
[root@server shell01]# array[1]=var2
```

```
[root@server shell01]# array[2]=var3
```

```
[root@server shell01]# array1=(uu1 uu2 uu3 uu4)
```

```
[root@server shell01]# ls
```

```
1.sh 2.sh 3.sh 4.sh passwd
```

```
[root@server shell01]# array2=(`ls .`)
```

```
[root@server shell01]# array3=(jack harry "Miss Hou" [5]=tom)
```

```
[root@server shell01]# declare -a
```

```
declare -a array=('[0]="var1" [1]="var2" [2]="var3")'
```

```
declare -a array1=('[0]="uu1" [1]="uu2" [2]="uu3" [3]="uu4")'
```

```

declare -a array2='([0]="1.sh" [1]="2.sh" [2]="3.sh" [3]="4.sh" [4]="passwd")'
declare -a array3='([0]="jack" [1]="harry" [2]="Miss Hou" [5]="tom")'
[root@server shell01]#
[root@server shell01]#
[root@server shell01]# echo ${array[*]}
var1 var2 var3
[root@server shell01]# echo ${array[@]}
var1 var2 var3
[root@server shell01]# echo ${array[2]}
var3
[root@server shell01]# echo ${array2[@]}
1.sh 2.sh 3.sh 4.sh passwd
[root@server shell01]# echo ${array2[3]}
4.sh
[root@server shell01]#
[root@server shell01]# echo ${array2[*]:2:2}
3.sh 4.sh
[root@server shell01]# echo ${#array2[*]}
5
[root@server shell01]# echo ${!array2[*]}
0 1 2 3 4
[root@server shell01]# echo ${!array3[*]}
0 1 2 5

```

#### 关联数组定义:

##### 首先声明关联数组

```

declare -A asso_array1
declare -A asso_array2
declare -A asso_array3

```

#### 数组赋值:

##### 一次赋一个值:

##### 数组名[索引|下标]=变量值

```

[root@server ~]# asso_array1[linux]=one
[root@server ~]# asso_array1[java]=two
[root@server ~]# asso_array1[php]=three

```

##### 一次赋多个值:

```

[root@server ~]# asso_array2=([name1]=harry [name2]=jack [name3]=amy [name4]="Miss Hou")

```

##### 查看关联数组:

```

[root@server ~]# declare -A
declare -A asso_array1='([php]="three" [java]="two" [linux]="one" )'
declare -A asso_array2='([name3]="amy" [name2]="jack" [name1]="harry" [name4]="Miss Hou" )'

[root@server ~]# echo ${asso_array1[linux]}
one
[root@server ~]# echo ${asso_array1[php]}
three
[root@server ~]# echo ${asso_array1[*]}
three two one
[root@server ~]# echo ${!asso_array1[*]}
php java linux
[root@server ~]# echo ${#asso_array1[*]}

```



```
3
[root@server ~]# echo ${#asso_array2[*]}
4
[root@server ~]# echo ${!asso_array2[*]}
name3 name2 name1 name4
```

#### 9. 交互式定义变量的值 read 主要用于让用户去定义变量值

- p 提示信息
- n 字符数 (限制变量值的字符数)
- s 不显示
- t 超时 (默认单位秒) (限制用户输入变量值的超时时间)

```
[root@MissHou shell01]# cat 1.txt
10.1.1.1 255.255.255.0
```

```
[root@MissHou shell01]# read -p "Input your IP and Netmask:" ip mask < 1.txt
[root@MissHou shell01]# echo $ip
10.1.1.1
[root@MissHou shell01]# echo $mask
255.255.255.0
```

#### 10. 其他变量 (扩展)

1) 取出一个目录下的目录和文件: dirname和 basename

2) 变量"内容"的删除和替换

一个 "%"代表从右往左去掉一个/key/

两个 "%"代表从右往左最大去掉/key/

一个 "#"代表从左往右去掉一个/key/

两个 "##"代表从左往右最大去掉/key/

```
# A=/root/Desktop/shell/mem.txt
# echo $A
/root/Desktop/shell/mem.txt
# dirname $A 取出目录
/root/Desktop/shell
# basename $A 取出文件
mem.txt
# echo ${A%/*} 从右往左去掉 "/"内容
/root/Desktop/shell
# echo ${A%.*} 从右往左最大长度去掉.后的内容
/root/Desktop/shell/mem
# echo ${A%.txt} 从右往左最大长度去掉.txt内容
/root/Desktop/shell/mem
# echo ${A##*/} 从左往右最大去掉所有 "/"
mem.txt
# echo ${A#*/}
Desktop/shell/mem.txt
```

//以下扩展内容自己练习完成

```
1000 url=www.taobao.com
```

```
1001 echo ${#url}      获取变量的长度
1008 echo ${url#*.*}
1009 echo ${url##*.*}
1011 echo ${url%.*}
1012 echo ${url%%.*}
```

替换: / 和 //

```
1015 echo ${url/ao/AO}
1017 echo ${url//ao/AO}  贪婪替换
```

替代: - 和 :- +和:+

```
1019 echo ${abc-123}
1020 abc=hello
1021 echo ${abc-444}
1022 echo $abc
1024 abc=
1025 echo ${abc-222}
```

`${变量名-新的变量值}` 或者 `${变量名=新的变量值}`

变量没有被赋值: 会使用“新的变量值” 替代

变量有被赋值 (包括空值): 不会被替代

```
1062 echo ${ABC:-123}
1063 ABC=HELLO
1064 echo ${ABC:-123}
1065 ABC=
1066 echo ${ABC:-123}
```

`${变量名:-新的变量值}` 或者 `${变量名:=新的变量值}`

变量没有被赋值或者赋空值: 会使用“新的变量值” 替代

变量有被赋值: 不会被替代

```
1116 echo ${abc=123}
1118 echo ${abc:=123}
```

```
[root@server ~]# unset abc
[root@server ~]# echo ${abc:+123}
```

```
[root@server ~]# abc=hello
[root@server ~]# echo ${abc:+123}
123
[root@server ~]# abc=
[root@server ~]# echo ${abc:+123}
```

`${变量名+新的变量值}`

变量没有被赋值或者赋空值: 不会使用“新的变量值” 替代

变量有被赋值: 会被替代

```
[root@server ~]# unset abc
[root@server ~]# echo ${abc+123}
```

```
[root@server ~]# abc=hello
[root@server ~]# echo ${abc+123}
123
```

```
[root@server ~]# abc=
[root@server ~]# echo ${abc+123}
123
${变量名:+新的变量值}
变量没有被赋值：不会使用“新的变量值” 替代
变量有被赋值（包括空值）： 会被替代

[root@server ~]# unset abc
[root@server ~]# echo ${abc?123}
-bash: abc: 123

[root@server ~]# abc=hello
[root@server ~]# echo ${abc?123}
hello
[root@server ~]# abc=
[root@server ~]# echo ${abc?123}

${变量名?新的变量值}
变量没有被赋值:提示错误信息
变量被赋值（包括空值）： 不会使用“新的变量值” 替代

[root@server ~]# unset abc
[root@server ~]# echo ${abc:?123}
-bash: abc: 123
[root@server ~]# abc=hello
[root@server ~]# echo ${abc:?123}
hello
[root@server ~]# abc=
[root@server ~]# echo ${abc:?123}
-bash: abc: 123

${变量名:?新的变量值}
变量没有被赋值或者赋空值时:提示错误信息
变量被赋值： 不会使用“新的变量值” 替代

说明：?主要是当变量没有赋值提示错误信息的，没有赋值功能
```

## 简单的四则运算

算术运算：默认情况下，shell就只能支持简单的整数运算

+ - \* / % (取模，求余数)

Bash shell 的算术运算有四种方式：

- 1、使用 `$(( ))`
- 2、使用 `$[ ]`
- 3、使用 `expr` 外部程式
- 4、使用 `let` 命令

注意:

```
n=1
```

```
let n+=1 等价于 let n=n+1
```

```
[root@MissHou ~]# expr 1 - 1
```

```
0
```

```
[root@MissHou ~]# expr 1 * 8
```

//乘法需要对\*进行转意

```
expr: syntax error
```

```
[root@MissHou ~]# expr 1 \* 8
```

```
8
```

```
[root@MissHou ~]# expr 1 / 8
```

```
0
```

```
[root@MissHou ~]# expr 1 % 8
```

```
1
```

```
[root@MissHou ~]# n=1
```

```
[root@MissHou ~]# let n+=1
```

```
[root@MissHou ~]# echo $n
```

```
2
```

```
[root@MissHou ~]# let n=n+2
```

```
[root@MissHou ~]# echo $n
```

```
4
```

```
[root@MissHou ~]# let n-=1
```

```
[root@MissHou ~]# echo $n
```

```
3
```

```
[root@MissHou ~]# let n=n-1
```

```
[root@MissHou ~]# echo $n
```

```
2
```

```
[root@MissHou ~]#
```

```
[root@MissHou ~]# let n*=8
```

```
[root@MissHou ~]# echo $n
```

```
16
```

```
[root@MissHou ~]# let n=n*2
```

```
[root@MissHou ~]# echo $n
```

```
32
```

```
[root@MissHou ~]# let n/=4
```

```
[root@MissHou ~]# echo $n
```

```
8
```

```
[root@MissHou ~]# let n=n/4
```

```
[root@MissHou ~]# echo $n
```

```
2
```

```
[root@MissHou ~]# let n**2
```

```
[root@MissHou ~]# echo $n
```

```
2
```

```
[root@MissHou ~]# let n**=2 //错误写法
```

```
-bash: let: n**=2: syntax error: operand expected (error token is "=2")
```

```
[root@MissHou ~]# let n=n**2
```

```
[root@MissHou ~]# echo $n
```

```
4
```

```
[root@MissHou ~]# let n=n**2
```

```
[root@MissHou ~]# echo $n
```

```
16
```

```
[root@MissHou ~]# n=2
[root@MissHou ~]# echo $n
2
[root@MissHou ~]# let n=n**8
[root@MissHou ~]# echo $n
256
```

思考：能不能用shell做小数运算？

```
[root@server shell01]# echo 1+1.5|bc
2.5
```

i++ 和 ++i （了解）

对变量的值的影响：

```
[root@node1 ~]# i=1
[root@node1 ~]# let i++
[root@node1 ~]# echo $i
2
[root@node1 ~]# j=1
[root@node1 ~]# let ++j
[root@node1 ~]# echo $j
2
```

对表达式的值的影响：

```
[root@node1 ~]# unset i j
[root@node1 ~]# i=1;j=1
[root@node1 ~]# let x=i++      先赋值，再运算
[root@node1 ~]# let y=++j      先运算，再赋值
[root@node1 ~]# echo $i
2
[root@node1 ~]# echo $j
2
[root@node1 ~]# echo $x
1
[root@node1 ~]# echo $y
2
```

总结：

```
$(( )) $[]
expr 注意空格, *要进行转义 \

let n+=1 等价 let n=n+1
let n=n**5    n有初值, 然后求次幂

i++ ++i
对变量本身没有影响 (自己+自己);
表达式中有影响; i++ 先赋值再运算 ++i先运算再赋值
let x=i++ let x=++i
```

## 条件判断

### 1. 语法格式

- 格式1: **test** 条件表达式
- 格式2: [ 条件表达式 ]
- 格式3: [[ 条件表达式 ]] 支持正则 =~

说明:

*man test* 去查看, 很多的参数都用来进行条件判断

### 2. 条件判断相关参数

- 与文件存在与否的判断

```
-e 是否存在 不管是文件还是目录, 只要存在, 条件就成立
-f 是否为普通文件
-d 是否为目录
-S socket
-p pipe
-c character
-b block
-L 软link

三种语法格式:
test -e file          只要文件存在条件为真
[ -d /shell01/dir1 ]  判断目录是否存在, 存在条件为真
[ ! -d /shell01/dir1 ] 判断目录是否存在, 不存在条件为真
[[ -f /shell01/1.sh ]] 判断文件是否存在, 并且是一个普通的文件

-s 判断文件是否有内容 (大小), 非空文件条件满足
说明: -s表示非空, ! -s 表示空文件
说明: 1.sh文件里有内容的。
[root@server shell01]# test -s 1.sh
[root@server shell01]# echo $?
0
[root@server shell01]# touch aaa
[root@server shell01]# cat aaa
```

```
[root@server shell01]# test -s aaa
[root@server shell01]# echo $?
1
[root@server shell01]# test ! -s aaa
[root@server shell01]# echo $?
0
[root@server shell01]# test ! -s 1.sh
[root@server shell01]# echo $?
1
```

- 文件权限相关的判断

- r 当前用户对其是否可读
- w 当前用户对其是否可写
- x 当前用户对其是否可执行
- u 是否有suid
- g 是否sgid
- k 是否有t位

- 两个文件的比较判断

```
file1 -nt file2 比较file1是否比file2新
file1 -ot file2 比较file1是否比file2旧
file1 -ef file2 比较是否为同一个文件，或者用于判断硬连接，是否指向同一个inode

test file1 -nt file2
[ file1 -ot file2 ]
```

- 整数之间的判断

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

- 字符串之间的判断

- z 是否为空字符串 字符串长度为0，就成立
- n 是否为非空字符串 只要字符串非空，就是成立
- string1 = string2 是否相等
- string1 != string2 不等
- ! 结果取反

```
[root@server shell01]# AAA=hello
[root@server shell01]# BBB=world
```

```
[root@server shell01]# test -z $AAA
[root@server shell01]# echo $?
1
[root@server shell01]# test -n $AAA
[root@server shell01]# echo $?
0

[root@server shell01]# [ $AAA = $BBB ]
[root@server shell01]# echo $?
1
[root@server shell01]# [ $AAA != $BBB ]
[root@server shell01]# echo $?
0
```

## • 多重条件判断

逻辑判断符号：

-a	和 && (and 逻辑与)	两个条件同时满足，整个大条件为真
-o	和    (or 逻辑或)	两个条件满足任意一个，整个大条件为真
!	逻辑非	

```
[ 1 -eq 1 -a 1 -ne 0 ]    整个表达式为真
[ 1 -eq 1 ] && [ 1 -ne 0 ] 整个表达式为真
[ 1 -eq 1 -o 1 -ne 1 ]    整个表达式为真
[ 1 -eq 1 ] || [ 1 -ne 1 ]
```

```
[root@server shell01]# [ 1 -eq 0 ] && echo true || echo false
false
[root@server shell01]# [ 1 -eq 1 ] && echo true || echo false
true
```

总结：

- 1、 ; && ||都可以用来分割命令
- 2、 ; 完全不考虑前面的语句是否正确执行，都会执行；号后面的内容
- 3、&& 需要考虑&&前面的语句的正确性，前面语句正确执行才会执行&&后的内容；反之亦然  
make && make install
- 4、|| 需要考虑||前面的语句的非正确性，前面语句执行错误才会执行||后的内容；反之亦然
- 5、如果&&和||一起出现，从左往右依次看，按照以上原则

## 3. 示例

示例：

数值比较：

```
[root@server ~]# [ $(id -u) -eq 0 ] && echo "the user is admin"
[root@server ~]$ [ $(id -u) -ne 0 ] && echo "the user is not admin"
```



```
[root@server ~]$ [ $(id -u) -eq 0 ] && echo "the user is admin" || echo "the user is not admin"

[root@server ~]# uid=`id -u`
[root@server ~]# test $uid -eq 0 && echo this is admin
this is admin
[root@server ~]# [ $(id -u) -ne 0 ] || echo this is admin
this is admin
[root@server ~]# [ $(id -u) -eq 0 ] && echo this is admin || echo this is not admin
this is admin
[root@server ~]# su - stu1
[stu1@server ~]$ [ $(id -u) -eq 0 ] && echo this is admin || echo this is not admin
this is not admin
[stu1@server ~]$
```

类C风格的数值比较:

注意: 在(( ))中, =表示赋值; ==表示判断

```
1159 ((1==2));echo $?
1160 ((1<2));echo $?
1161 ((2>=1));echo $?
1162 ((2!=1));echo $?
1163 ((`id -u`==0));echo $?
1209 ((a=123));echo $a
1210 unset a
1211 ((a==123));echo $?
```

字符串比较:

注意: 双引号引起来, 看作一个整体; = 和 == 在 [ 字符串 ] 比较中都表示判断

```
1196 a='hello world';b=world
1197 [ $a = $b ];echo $?
1198 [ "$a" = "$b" ];echo $?
1199 [ "$a" != "$b" ];echo $?
1200 [ "$a" != " $b" ];echo $?
1201 [ "$a" == "$b" ];echo $?
1202 test "$a" != "$b";echo $?
```

错误

思考: [ ] 和 [[ ]] 有什么区别?

```
1213 a=
1214 test -z $a;echo $?
1215 a=hello
1216 test -z $a;echo $?
1217 test -n $a;echo $?
1217 test -n "$a";echo $?
```

```
# [ ' ' = $a ];echo $?
-bash: [: : unary operator expected
2
# [[ ' ' = $a ]];echo $?
```

0

```
1278 [ 1 -eq 0 -a 1 -ne 0 ];echo $?
1279 [ 1 -eq 0 && 1 -ne 0 ];echo $?
1280 [[ 1 -eq 0 && 1 -ne 0 ]];echo $?
```

## 总结

变量的定义规则：

1. 变量定义方法

变量名=变量值

2. 获取变量

\$变量名      \${变量名}

3. 交互式定义变量

read 变量1 变量2

-s

-p

-n

-t

4. 数组的定义

普通数组：索引下标是整数

关联数组：索引下标是字符串

array1=(var1 var2 var3)

array2[0]=var1

array2[2]=var2

declare -A books

books=( [linux]=10 [php]=14 [java]=20 )

获取数组：

\${array1[\*]}

\${array1[2]}

\${!books[\*]}

\${#books[\*]}

四则运算：

\$(( ))    \$[]    expr    let

条件判断：

test 表达式

[ 表达式 ]

[[ 表达式 ]]

判断文件存在与否：

-e

-f

-d

-L

...

判断整数：

-eq

```
-ne  
-gt  
-lt  
-ge  
-le
```

判断字符串:

```
-n    字符串长度非空, 条件成立    [ -n string ]  
-z    字符串长度空, 条件成立     [ -z string ]  
string1 = string2  
string1 != string2
```

判断文件是否为空文件:

```
-s    文件内容非空, 条件成立    [ -s /etc/passwd ]  条件为真  
! -s  文件内容为空, 条件成立
```

逻辑运算符:

```
&&  ||  !
```

## 流程控制语句

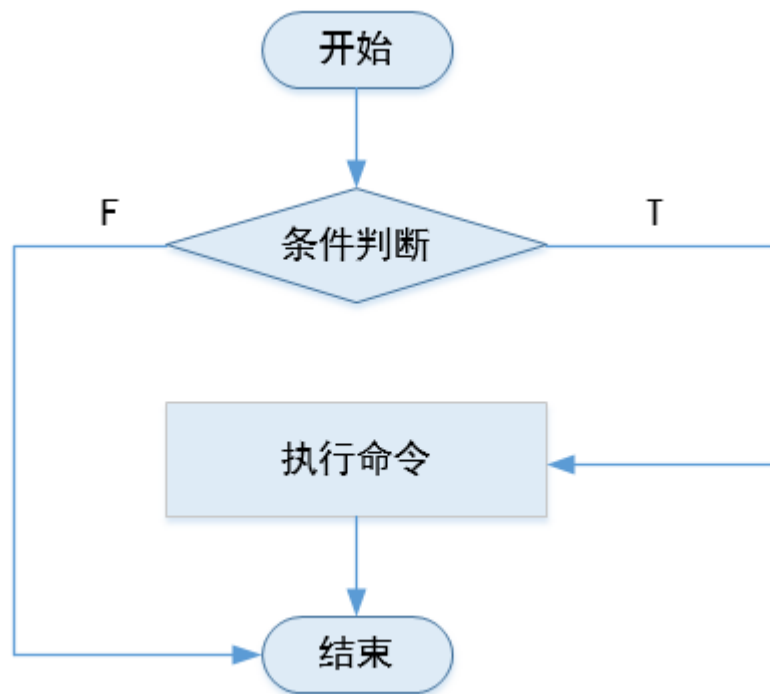
### 1. 基本语法结构

F:false 假

T:true 真

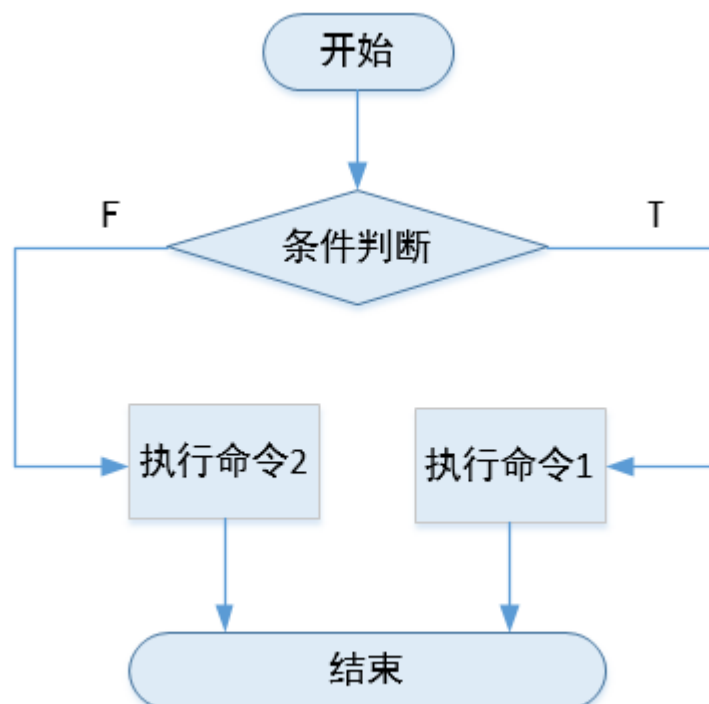
```
if [ condition ];then  
    command  
    command  
fi
```

```
[ 条件 ] && command
```



```
if [ condition ];then
    command1
else
    command2
fi

[ 条件 ] && command1 || command2
```



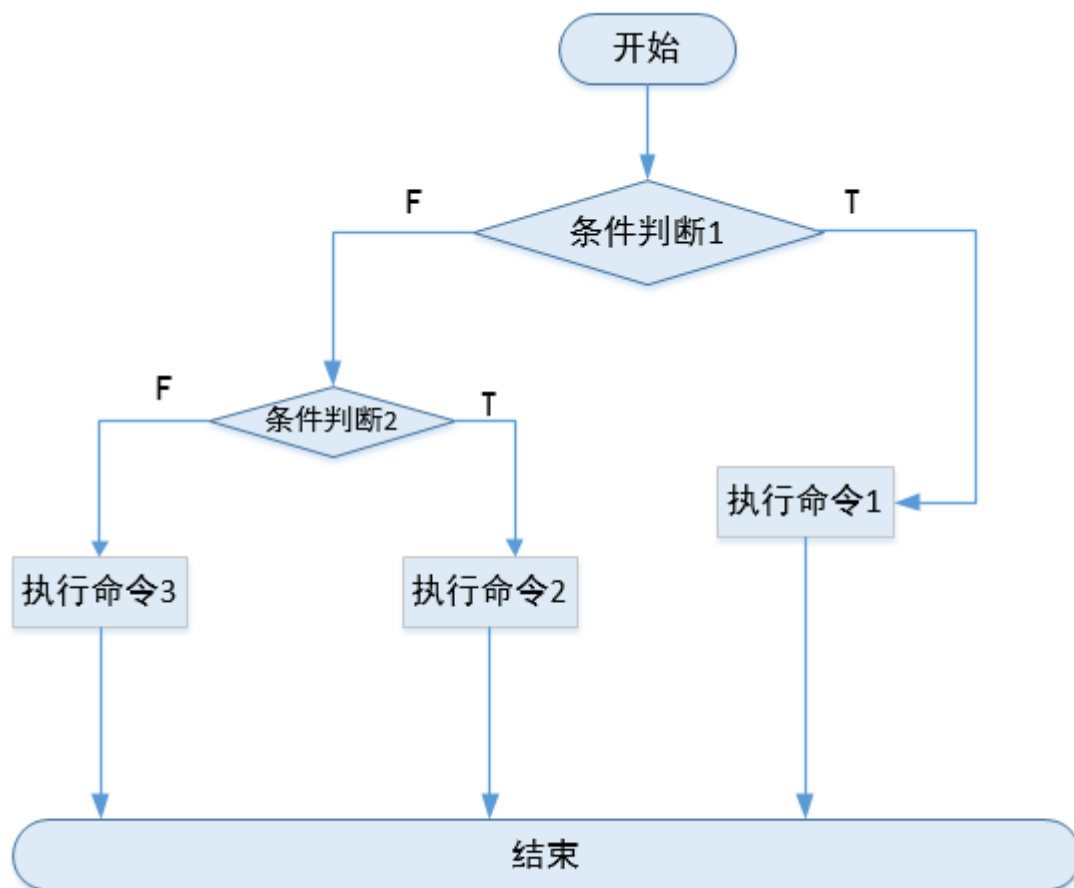
```

if [ condition1 ];then
    command1 结束
elif [ condition2 ];then
    command2 结束
else
    command3
fi

```

注释:

如果条件1满足, 执行命令1后结束; 如果条件1不满足, 再看条件2, 如果条件2满足执行命令2后结束; 如果条件1和条件2都不满足执行命令3结束.



```

if [ condition1 ];then
    command1
    if [ condition2 ];then
        command2
    fi
else
    if [ condition3 ];then
        command3
    elif [ condition4 ];then
        command4
    else

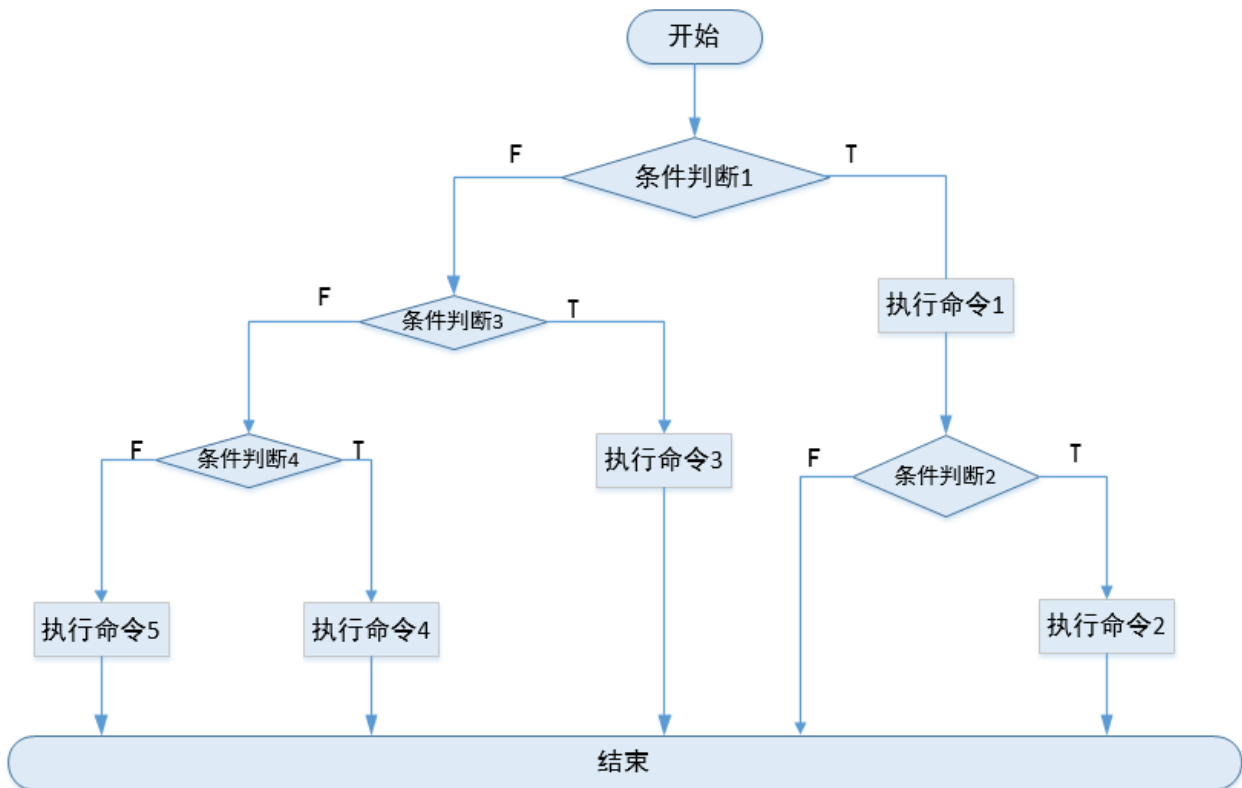
```

```
command5
fi
fi
```

注释:

如果条件1满足, 执行命令1; 如果条件2也满足执行命令2, 如果不满足就只执行命令1结束;

如果条件1不满足, 不看条件2; 直接看条件3, 如果条件3满足执行命令3; 如果不满足则看条件4, 如果条件4满足执行命令4; 否则执行命令5



## 2. 应用案例

需求1: 判断当前主机是否和远程主机是否ping通

思路:

1. 使用哪个命令实现 ping
2. 根据命令的执行结果状态来判断是否通
3. 根据逻辑和语法结构来编写脚本

步骤:

```
vim /shell02/ping.sh
#!/bin/bash
# Name:ping.sh
# Path:/shell02/
# Usage:/shell02/ping.sh
# ...

# 定义远程主机的IP地址
read -p "Input your remote ip:" remote_ip
```

```
# 通过ping程序来判断是否和远程主机ping通
ping -c1 $remote_ip &>/dev/null

# 根据上一个命令ping的执行状态来判断是否互通

if [ $? -eq 0 ];then
    echo "当前主机可以和远程主机:$remote_ip互通"
else
    echo "当前主机可以和远程主机:$remote_ip不通"
fi
```

#### 方法2:

```
vim ping.sh
#!/bin/bash
# 通过ping程序来判断是否和远程主机ping通
ping -c1 $1 &>/dev/null
# 根据上一个命令ping的执行状态来判断是否互通
test $? -eq 0 && echo "当前主机可以和远程主机:$1互通" || echo "当前主机和远程主机:$1不通"
```

#### 改进:

分析: 如果用户传参数的个数不等于1, 那么提示语法错误

```
#!/bin/bash
# 判断脚本后面参数的个数
[ $# -ne 1 ] && echo "Usage:$0 remote ip" && exit
# 通过ping程序来判断是否和远程主机ping通
ping -c1 $1 &>/dev/null
# 根据上一个命令ping的执行状态来判断是否互通
test $? -eq 0 && echo "当前主机可以和远程主机:$1互通" || echo "当前主机和远程主机:$1不通"
```

#### 参考:

```
#!/bin/bash
#xxx
read -p "请输入远程主机的IP地址:" ip
ping -c1 $ip &>/dev/null
if [ $? -eq 0 ];then
    echo "当前主机可以和远程主机$ip互通"
else
    echo "当前主机可以和远程主机$ip不能互通"
fi
```

```
[root@server shell02]# cat ping1.sh
#!/bin/bash
read -p "请输入要ping的主机IP:" ip
ping -c1 $ip &>/dev/null
test $? -eq 0 && echo "ip is ok" || echo "ip is not ok"
[root@server shell02]# cat ping2.sh
#!/bin/bash
#Uage:/shell02/ping2.sh ip
ping -c1 $1 &>/dev/null
test $? -eq 0 && echo "ip is ok" || echo "ip is not ok"
```

进一步改进:

```
#!/bin/bash
if [ $# -ne 1 ];then
    echo "语法错误, Usage:$0 远程IP地址" && exit
else
    ping -c1 $1 &>/dev/null
test $? -ne 0 && echo "当前主机可以和远程主机$1不能互通" || echo "当前主机可以和远程主机$1互通"
fi
```

或者

```
test $# -ne 1 && echo "语法错误, Usage:$0 远程IP地址" && exit
ping -c1 $1 &>/dev/null
test $? -ne 0 && echo "当前主机可以和远程主机$1不能互通" || echo "当前主机可以和远程主机$1互通"
```

要求:

判断脚本后面参数个数, 如果不是1则退出程序

```
[root@server shell02]# cat ping2.sh
#!/bin/bash
#Usage:/shell02/ping2.sh ip
test $# -ne 1 && echo " $0 Usage:$0 ip" && exit
ping -c1 $1 &>/dev/null
test $? -eq 0 && echo "ip is ok" || echo "ip is not ok"
```

需求2: 判断一个进程是否存在

思路:

1. 查看进程的相关命令 `ps -ef` `pgrep` `pidof`
2. 根据命令的返回状态值来判断进程是否存在 `$?`
3. 根据逻辑用脚本语言实现

```
#!/bin/bash
# xxxx
#定义变量
read -p "Input your process name:" name
#通过执行程序判断进程是否存在
pgrep $name &>/dev/null
[ $? -eq 0 ] && echo "$name进程存在" || echo "$name不存在"
```

```
#!/bin/bash
ps -ef httpd|grep -v "grep" &>/dev/null
if [ $? -eq 0 ];then
    echo "httpd进程存在"
else
    echo "httpd进程不存在"
fi
```



或者

```
#!/bin/bash
ps -ef httpd|grep -v "grep" &>/dev/null
[ $? -eq 0 ] && echo "httpd进程存在" || echo "httpd进程不存在"
```

```
#!/bin/bash
read -p "请输入需要判断的进程名(vsftpd):" pid
ps -ef|grep $pid|grep -v 'grep' &>/dev/null
或者
pgrep -l $pid &>/dev/null
[ $? -eq 0 ] && echo "该进程$pid存在" || echo "该进程不存在"
```

pgrep命令：以名称为依据从运行进程队列中查找进程，并显示查找到的进程id

选项

- o: 仅显示找到的最小（起始）进程号；
- n: 仅显示找到的最大（结束）进程号；
- l: 显示进程名称；
- P: 指定父进程号； pgrep -p 4764 查看父进程下的子进程id
- g: 指定进程组；
- t: 指定开启进程的终端；
- u: 指定进程的有效用户ID。

需求3: 判断一个服务是否正常（以httpd为例）:

思路:

1. 可以判断进程是否存在，用/etc/init.d/httpd status判断状态等方法
2. 最好的方法是直接去访问一下，通过访问成功和失败的返回值来判断

```
#!/bin/bash
wget http://10.1.1.2 &>/dev/null
[ $? -eq 0 ] && echo "该web服务是正常的" && rm -f /shell/shell01/index.* || echo "该web服务异常请检查"
```

### 3. 课堂练习

1、输入一个用户，用脚本判断该用户是否存在

```
#!/bin/bash
#定义变量，让用户自己输入需要判断的用户
read -p "请输入需要判断的用户:" user
#通过命令判断用户是否存在
id $user &>/dev/null
#通过命令的结果来判断是否存在
if [ $? -eq 0 ];then
    echo "$user用户存在"
else
    echo "$user用户不存在"
```

```
fi
```

或者

```
read -p "请输入需要判断的用户:" user
grep -w ^$user /etc/passwd &>/dev/null
[ $? -eq 0 ] && "$user用户存在" || echo "$user用户不存在"
```

## 2、判断vsftpd软件包是否安装，如果没有则自动安装（yum源已配好）

参考：

```
#!/bin/bash
#定义变量
software=vsftpd
#执行命令
rpm -q $software &>/dev/null
if [ $? -eq 0 ];then
    echo "此软件已安装"
else
    echo "此软件未安装，自动安装中"
    yum -y install $software &>/dev/null
    if [ $? -eq 0 ];then
        echo "安装成功"
    else
        echo "安装出错"
    fi
fi
```

```
#!/bin/bash
read -p "输入查询的程序：" name
#判断vsftpd 是否安装
rpm -q $name &> /dev/null
if [ $? -ne 0 ] ; then
    echo "程序未安装，正在安装 "
    yum -y install $name
else
    echo "程序已安装"
fi
```

## 3、判断当前内核主版本是否为2，且次版本是否大于等于6；如果都满足则输出当前内核版本

思路：

1. 先查看内核的版本号 `uname -r`
2. 先将内核的版本号保存到一个变量里，然后再根据需求截取出该变量的一部分：主版本和次版本
3. 根据需求进行判断

```
#!/bin/bash
kernel=`uname -r`
var1=`echo $kernel|cut -d. -f1`
var2=`echo $kernel|cut -d. -f2`
test $var1 -eq 2 -a $var2 -ge 6 && echo $kernel || echo "当前内核版本不符合要求"

或者

#!/bin/bash
kernel=`uname -r`
test ${kernel:0:1} -eq 2 -a ${kernel:2:1} -ge 6 && echo $kernel || echo '不符合要求'

其他命令参考：
uname -r|grep ^2.[6-9] || echo '不符合要求'
```

4、判断ftp服务是否已启动，如果已启动输出以下信息：

vsftpd服务器已启动... vsftpd监听的端口是: vsftpd的进程PID是:

```
参考1:
#!/bin/bash
service vsftpd status &>/dev/null
if [ $? -eq 0 ];then
    port=`netstat -tnltp|grep vsftpd|cut -d: -f2|cut -d' ' -f1`
    pid=`pgrep -l vsftpd|cut -d' ' -f1`
    echo -e "vsftpd服务器已启动...\nvsftpd监听的端口是:$port\nvsftpd的进程PID是:$pid"
else
    service vsftpd start &>/dev/null
    port=`netstat -tnltp|grep vsftpd|cut -d: -f2|cut -d' ' -f1`
    pid=`pgrep -l vsftpd|cut -d' ' -f1`
    echo -e "vsftpd服务器已启动...\nvsftpd监听的端口是:$port\nvsftpd的进程PID是:$pid"
fi

参考2:
vim /lt/2.sh
#!/bin/bash
duankou=`netstat -tnltp|grep vsftpd|cut -d: -f2|cut -d" " -f1`
pid=`pgrep -o vsftpd`

vim 1.sh
pgrep -l vsftpd >/dev/null
if [ $? -eq 0 ];then
    echo "vsftpd服务器已启动..."
    echo "vsftpd监听的端口是: $duankou"
    echo "vsftpd的进程PID是: $pid"
else
    echo "vsftpd服务器没启动"
    service vsftpd start
    source /lt/2.sh
fi
```

## 课后作业

1、判断/tmp/run目录是否存在，如果不存在就建立，如果存在就删除目录里所有文件 2、输入一个路径，判断路径是否存在，而且输出是文件还是目录，如果是链接文件，还得输出是有效的连接还是无效的连接 3、交互模式要求输入一个ip，然后脚本判断这个IP 对应的主机是否 能ping 通，输出结果类似于： Server 10.1.1.20 is Down! 最后要求把结果邮件到本地管理员root@localhost和mail01@localhost 4、写一个脚本/home/program，要求当给脚本输入参数hello时，脚本返回world,给脚本输入参数world时，脚本返回hello。而脚本没有参数或者参数错误时，屏幕上输出“usage:/home/program hello or world” 5、写一个脚本自动搭建nfs服务

```
#!/bin/bash
#xxx

#关闭防火墙和selinux
xxx

#配置本地yum源

#安装软件

#创建共享目录

#发布共享目录

#测试验证
```