link: null
title: 珠峰架构师成长计划
description: 查看shell的帮助信息
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=448 sentences=435, words=3554

1. shell基础

- shell是一个命令行解释器，它为用户提供了一个向Linux内核发送请求以便运行程序的界面系统级程序
- 用户可以用Shell来启动、挂起、停止或者编写一些程序
- Shell还是一个功能相当强大的编程语言，易编写，易调试，灵活性较强。
- Shell是解释执行的脚本语言，在Shell中可以直接调用Linux系统命令。

1.1 查看支持的shell

- `cat /etc/shells`

查看shell的帮助信息

命令 作用 man bash 查看bash的命令帮助 info bash 查看bash的文档 help 命令显示bash支持的命令 help cd 如果想看某个命令的帮助可以 `help`

命令

1.2 echo

- 输出命令
- -e 支持反斜线控制的字符转换

控制字符 作用 \n 换行符 \r 回车键 \t 制表符，也就是Tab键 \v 垂直制表符 \onnn 按照八进制ASCII码表输出字符，其中0为数字零，nnn是三位八进制数 \xhh 按照十六进制ASCII码表输出字符，其中hh是两位十六进制数 符号 颜色 #30m 黑色 #31 红色 #32 绿色 #33 黄色 #34 蓝色 #35 洋红 #36 青涩 #37 白色

```
echo -e "12\r\n34\t56\v78"
echo -e "\0141"
echo -e "\x61"
echo -e "\e[1;31m warning \e[0m"
```

1.3 编写执行shell

```
echo -e "\e[1;34m hello world \e[0m"
```

赋予执行权限，直接运行

```
chmod 755 hello.sh
./hello.sh
```

通过Bash调用执行脚本

```
bash hello.sh
```

1.4 别名

- 命令别名 == 小名
- 临时生效
- alias
- alias rm="rm -i"
- 写入环境变量配置文件 vi ~/.bashrc
- source ~/.bashrc
- unalias 别名 删除别名

1.5 命令的生效顺序

- 绝对路径或者相对路径
- 别名
- bash内部命令
- 按照$PATH环境变量定义的目录查找顺序找到的第一个命令

1.6 命令快捷键

命令 含义 ctrl+c 强制终止当前命令 ctrl+l 清屏 ctrl+a 光标移动到命令行首 ctrl+e 光标移动到命令行尾 ctrl+u 从光标所在的位置删除到行首 ctrl+z 把命令放入后台 fg 把任务带回前台 ctrl+r 在历史命令中搜索

1.7 历史命令

- history [选项] [历史命令保存文件]
- 选项
    - -c 清空历史命令
    - -w 把缓存中的历史命令写入历史命令保存文件 ~/.bash_history
- 默认保存1000条 /etc/profile `HISSIZE=10000`

1.8 调用

- 使用上下箭头调用以前的历史命令
- 使用 !n 重复执行第n条历史命令
- 使用 !! 重复执行上一条命令
- 使用 !字符 重复执行最后一条以该字符串开头的命令

```
history -c
1  echo 1
2  echo 2
3  echo 3
!2
!!

!echo
```

1.9 输出重定向

1.9.1 标准输入输出

设备 设备文件名 文件描述符 类型 键盘 /dev/stdin 0 标准输入 显示器 /dev/stdout 1 标准输出 显示器 /dev/stderr 2 标准错误输出 类型 符号 作用 标准输出重定向 命令 > 文件 以覆盖的方式，把命令的正确输出输出到指定的文件或设备当中 标准输出重定向 命令 >> 文件 以追加的方式，把命令的正确输出输出到指定的文件或设备当中 错误输出重定向 命令>文件 以覆盖的方式，把命令的错误输出输出到指定的文件或设备当中 错误输出重定向 命令>>文件 以追加的方式，把命令的错误输出输出到指定的文件或设备当中 正确输出和错误输出同时保存 命令>文件 2>&1 以覆盖的方式，把正确输出和错误输出都保存到同一个文件当中 正确输出和错误输出同时保存 命令>>文件 2>>&1 以追加的方式，把正确输出和错误输出都保存到同一个文件当中 正确输出和错误输出同时保存 命令&>文件 以覆盖的方式，把正确输出和错误输出都保存到同一个文件当中 正确输出和错误输出同时保存 命令&>>文件 以追加的方式，把正确输出和错误输出都保存到同一个文件当中 正确输出和错误输出同时保存 命令>>文件1 2>文件2 以覆盖的方式，正确的输出追加到文件1中，把错误输出追加到文件2中

1.9.2 输入重定向

- wc命令的功能为统计指定文件中的行数、字数、字节数, 并将统计结果显示输出
- 命令 < 文件把文件做为命令的输入
- 命令 << 标识符 标识符把标识符之间内容作为命令的输入

```
- wc < access.log
[root@localhost ~]# wc < hello world
> !
 1  2 12
```

1.10 管道符号

1.10.1 多命令顺序执行

多命令执行符 格式 作用 案例 ; 命令1;命令2 多个命令执行，命令之间没有任何逻辑联系 echo 1;echo 2; && 命令1&&命令2 逻辑与 当命令1正确执行，则命令2才会执行 当命令1执行不正确，则命令2不会执行 echo 1&&echo 2; \ 命令1\ 命令2 逻辑或 当命令1执行不正确，则命令2才会执行 当命令1正确执行，则命令2不会执行 echo 1\echo 2;

```
- date;ls;date;ls
- ls && echo yes || echo no
```

1.10.2 管道符号

- 命令1的正确输出会作为命令2的操作对象
- 命令1|命令2

```
ls /etc/ | more
netstat -an | grep ESTABLISHED | wc -l
```

1.10.3 通配符

匹配文件名和目录名

通配符 作用 ? 匹配一个任意字符 * 匹配0个或任意字符，也就是可以匹配任意内容 [] 匹配中括号中任意一个字符 [-] 匹配中括号中任意一个字符,-代表范围 [^] 匹配不是中括号中的一个字符

1.10.4 其它符号

符号 作用 ' 单引号。在单引号中所有的特殊符号，如$和都没有特殊含义 "" 双引号，在双引号里特殊符号都没有特殊含义，但是 $ \ 例外，拥有调用变量值，引用命令和转义的含义

反引号，扩起来的是系统命令 $() 和反引号一样 # 在shell脚本中，#开头的行代表注释 $ 用于调用变量的值 \ 转义符号

```
-  a=`ls`
-  b=$(ls)
```

2. 变量

2.1 什么是变量

- 可以变化的量
- 变量必须以字母或下划线开头，名字中间只能由字母，数字和下划线组成
- 变量名的长度不得超过255个字符
- 变量名在有效范围内必须唯一
- 变量默认类型都是字符串

2.2 变量的分类

- 字符串
- 整型
- 浮点型
- 日期型

2.3 用户自定义变量

- 这些变量的值是自己定义的
- 变量名不能为数字开头
- 等号左右两边不能有空格

2.3.1 定义变量

```
name="zhufeng"
age=10
```

2.3.2 输出变量值

```
echo $变量名
```

2.3.3 值默认都是字符串

```
$ x=1
$ y=2
$ z=3
$ z=$x+$y+$z
$ echo $z
```

2.3.4 在赋值的时候引用变量

```
y="$y"2
y=${y}2
```

2.3.5 set

- 查询系统中默认所有已经生效的变量，包括系统变量,也包括自定义变量

```
set | grep zhufeng
```

```
$ set -u
$ echo $a
bash: a: unbound variable
`
```

2.3.6 unset

- 删除变量

```
unset a
```

2.4 环境变量

- 环境变量是全局变量，而自定义变量是局部变量
- 自定义变量会在当前的shell中生效，而环境变量会在当前shell以及其子shell中生效
- 这种变量主要保存的是和系统操作环境相关的数据
- 变量可以自定义，但是对系统生效的环境变量名和变量作用是固定的

```
bash
pstree
```

2.4.1 自定义环境变量

```
export 变量名=变量值
export envname=prod
```

2.4.2 env

- 仅仅用来查看环境变量，而不看到本地变量

```
env
```

2.4.3 常用环境变量

变量名 含义 示例 HOSTNAME 主机名 HOSTNAME=localhost SHELL 当前的shell SHELL=/bin/bash TERM 终端环境 TERM=xterm HISTSIZE 历史命令条数 HISTSIZE=1000 SSH_CLIENT 当前操作环境如果是用SSH连接的话，这里会记录客户端IP SSH_CLIENT=192.168.1.100 57596 22 SSH_TTY SSH连接的终端 SSH_TTY=/dev/pts/1 USER 当前登录的用户 USER=root

2.4.4 path

- 系统搜索路径

```
# echo $PATH
/usr/lib/qt-3.3/bin:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin:/root/bin
```

如果想把一个自定义的脚本直接可以执行，或者把这个文件拷贝到目标目录下，或者把脚本所在目录添加到环境变量中的PATH路径中

hello.sh

```

echo hello
```

```
./hello.sh
/root/myshell/hello.sh
export PATH="$PATH":/root/myshell    临时生效
hello.sh
```

2.4.5 $PS1

```
# echo $PS1
[\u@\h \W]\$
root@localhost myshell]#
```

变量 提示符 \d 显示日期，格式为"星期 月 日" \H 完整的主机名 \t 24小时制时间,格式为 "HH:MM:SS" \A 24小时制时间，格式为 "HH:MM" \u 显示当前用户名 \w 显示所在目录完整名称 \W 显示所在目录的最后一个目录 $ 提示符，root为#,普通用户为$

```
export PS1='[\u@\h \w]\$ '
```

2.4.6 语系环境变量

- 查询当前系统语系
- 在Linux中通过locale来设置程序运行的不同语言环境，locale由ANSI C提供支持。locale的命名规则为
- LANG: 定义系统主语系的变量

```
locale
LANG=zh_CN.UTF-8

echo $LANG

下次开机之后我们系统环境
cat /etc/sysconfig/i18n
```

2.4.7 中文支持

- 图形界面可以支持中文
- 第三方工具比如xshell语系设置正确可以支持中文
- 虚拟机中的纯字符界面不支持中文

2.5 位置参数变量

- 这种变量主要是用来向脚本当中传递参数或数据的,变量名不能自定义,变量作用是固定的

位置参数变量 作用 $n n为数字，$0代表命令本身，$1-$9代表第1到第9个参数，10以上的参数需要用大括号包含,如${10} $* 这个变量代表命令中所有的参数，$*把所有的变看数看成一个整体 $@ 这个变量也代表命令行中所有的参数，不过$@把每个参数进行区分 $# 这个变量代表命令行中所有参数的个数

```

num1=$1
num2=$2
sum=$((num1+num2))
echo $sum
```

```

for i in "$@"
 do
   echo "i=$i"
 done
```

2.6 预定义变量

- 是脚本中已经定义好的变量，变量名不能自定义，变量作用也是固定的

位置参数变量 作用 $? 最后一次执行的命令的返回状态。0表示正确执行，非0表示不正确执行 $ 当前进程的进程号(PID) $! 后台运行的最后一个进程号(PID)

```
ls && echo yes
```

2.7 read

read [选项] [变量名]

选项 含义 -p 提示信息，在等待read输入时，输出提示信息 -t 秒数: read命令会一直等待用户输入，使用此选项可以指定等待时间 -n 字符数，read命令只接受指定的字符数，就会执行 -s 隐藏输入的数据，适用于机密信息的输入

```
read -p 'please input your name:' -t 5 name
echo -e "\n"
read -p 'please input you gender[m/f]:' -n 1 gender
echo -e "\n"
read -p 'please input your password:' -s password
echo -e "\n"
echo $name,$gender,$password
```

3. 运算符

- 弱类型并且默认是字符串类型

3.1 declare命令

3.1.1 declare命令

- 用来声明变量类型
- declare [+/-] [选项] 变量名

选项 含义 - 给变量设定类型属性 + 取消变量的类型属性 -a 将变量声明为数组类型 -i 将变量声明为整数型(integer) -x 将变量声明为环境变量 -r 将变量声明为只读变量 -p 显示指定变量的被声明的类型

```
//声明成整型
# a=1
# b=2
# c=$a+$b
# echo $c
1+2
# declare -i c=$a+$b
# echo $c
3
# declare +i c
# c=$a+$b
# echo $c
1+2
# declare -p c
declare -i c="3"

//声明环境变量
# declare -x kk=1
# bash
# set | grep kk

//只读
# declare -r x
# x=2
```

3.1.2 数组

```
names[0]=zhangsan
names[1]=lisi
# 声明为数组类型
declare -a names;
# 默认只打印第一个元素
echo ${names}
zhangsan
打印第2个元素
# echo ${names[1]}
lisi
# 打印全部
echo ${names[*]}
zhangsan lisi
```

3.1.3 声明环境变量

- export最终执行的是 `declare -x`命令
- `declare -p` 可以查看所有的类型

```
export NAME=zhufeng
declare -x NAME=zhufeng
```

3.1.4 只读属性

```
#declare -r gender=m
#gender=f
-bash: gender: readonly variable
```

3.1.5 查询变量属性

- declare -p 查询所有变量的属性
- declare -p 变量名 查询指定变量的属性

3.2 数值运算的方法

- 只要用declare声明变量的时候指定类型就可以进行数值运算**3.2.1 expr或let #**
-
  - 号左右两侧必须有空格,否则还是整块输出

```
#num1=2
#num2=3
#s=$(expr $num1 + $num2)
#echo $s
5
```

```
#s=$(($num1+$num2))
#echo $s
5
#s=$[$num1+$num2]
#echo $s
5

d=$(date)
echo $d
```

** 3.2.2 优先级 #**

优先级	运算符	说明
13	-, +	单目负、单目正
12	!, ~	逻辑非、按位取反或补码
11	*, /, %	乘、除、取模
10	+, -	加、减
9	<<, >>	按位左移、按位右移
8	<=, >=, <, >	小于或等于、大于或等于、小于、大于
7	==, !=	等于、不等于
6	&	按位与
5	^	按位异或
4	|	按位或
3	&&	逻辑与
2	||	逻辑或
1	=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=	赋值、运算且赋值

```
result=$(((1+2)*4/2))
6
`
```

4. 环境变量配置文件

4.1 source 命令

- 修改完配置文件后，必须注销重新登录才能生效，使用source命令可以不用重新登录
- source 配置文件
- . 配置文件

4.2 环境变量配置文件简介

- PATH、HISTSIZE、PS1、HOSTNAME等环境变量写入对应的环境变量配置文件
- 环境变量配置文件中主要是定义地系统操作环境生效的系统默认环境变量，如PATH等 此下文件登录时起作用的环境变量

路径 说明 /etc/profile /etc/profile.d/*.sh /etc/bashrc ~/.bash_profile 只会对当前用户生效 ~/.bashrc 只会对当前用户生效

4.3 环境变量配置文件的功能

** 4.3.1 /etc/profile #**

- 在这里修改系统变量

```
cat  /etc/profile | grep USER
```

变量名 含义 USER 用户名 LOGNAME 登录名 MAIL 邮箱地址 PATH 查找路径 HOSTNAE 主机名 umask 权限掩码 /etc/profile.d/星.sh
** 4.3.2 ~/.bash_profile #**

- PATH 在这里修改PATH路径
- 调用 ~/.bashrc

** 4.3.3 ~/.bashrc #**

- 配置alias 改别名在这里
- 调用 /etc/bashrc

** 4.3.4 /etc/bashrc #**

- PS1 登录提示符在这里修改
- umask
- PATH变量
- 调用 /etc/profile.d/星.sh文件

** 4.3.5. 其它配置文件 #**

4.3.5.1 注销时生效的环境变量配置文件

~/.bash_logout

4.3.5.2 脚本历史

- 当正确退出计算机的时候会历史记录会写入文件 ~/.bash_history

4.3.5.3 Shell登录信息

本地终端欢迎信息 /etc/issue 远程终端欢迎信息 /etc/issue.net 不管远程还是本地都可以生效 /etc/motd

参数 含义 \d 当前系统日期 \s 显示操作系统名称 \l 显示登录的终端号 \m 显示硬件体系结构，如i386等 \n 显示主机名 \o 显示域名 \r 显示内核版本 \t 显示当前系统时间 \u 显示当前登录用户的序列号

```
vi /etc/ssh/sshd_config
Banner /etc/issue.net
service sshd restart
```

4.正则表达式

- regexper (https://regexper.com)

4.1 概念

- 正则表达式是用于描述字符排列和匹配模式的一种语法规则
- 它主要用于字符串的模式分割、匹配查找及替换操作

4.2 通配符

- 通配符用来匹配符合条件的文件名，通配符是完全匹配。
- ls find 这些命令不支持正则，只能使用通配符

符号 含义 ? 匹配任意一个字符 * 匹配任意多个字符 [] 匹配中括号中范围内的一个字符

4.3 正则表达式

- 正则表达式是用来在文件中匹配符合条件的字符串，是包含匹配。
- grep awk sed 等都可以支持正则表达式

4.4 元字符

```
alias grep='grep --color=auto'
```

元字符 作用 示例 * 前一个字符匹配 0 次或任意多次 grep 1* reg.txt . 匹配除换行符外的任意一个字符 grep . reg.txt ^ 匹配行首。例如，^hello 会匹配以 hello 开头的行 grep ^a reg.txt $ 匹配行尾。例如，hello& 会匹配以 hello 结尾的行 grep a$ reg.txt [] 匹配中柄号中指定的任意一个字符，而且只匹配一个字符。

例如.[aoeiu]匹配任意一个元音字母， [0-9] 匹配任意一位数字，

[a-z][0-9] 匹配由小写字母和一位数字构成的两位字符 grep ab[bc]c reg.txt [^] 匹配除中括号中的字符以外的任意一个字符。例如，[^0-9] 匹配任意一位非数字字符，

[^a-z] 匹配任意一位非小写字母 grep a[^fg]c reg.txt \ 转义符，用于取消特殊符号的含义 grep .$ reg.txt {n} 表示其前面的字符恰好出现 n 次。例如，[0-9]{4} 匹配4位数字，[1][3-8][0-9]{9} 匹配手机号码 grep "a{1}" reg.txt {n,} 表示其前面的字符出现不少于 n 次。例如，[0-9]{2,} 匹配两位及以上的数字 grep "a{1,}" reg.txt {n,m} 表示其前面的字符至少出现 n 次，最多出现 m 次。例如，[a-z]{6,8} 匹配 6～8 位的小写字母 grep "a{2,3}" reg.txt

4.5 cut

- cut用来提取文本中的某一部分文本
- cut [选项] 文件名
    - -f 列号，用来指定要提取的列
    - -d 分隔符，按照指定分隔符分割列,默认分隔符是TAB制表符

```
cat /etc/passwd | grep /bin/bash | cut -f 1,2 -d :

df -h | grep /dev/sda5 | cut -f 5
```

4.6 printf

- 按规定格式输出
- printf 输出类型 输出内容

参数 含义 %ns 输出字符串,n是数字指代输出几个字符 %ni 输出整数,n是指输出几个数字 %m.nf 输出浮点数,m和n是数字，指代输出的整数位数和小数位数，如%6.2f代表输出6位位，2位小数，4位整数

```
printf "%s\t%s\t%s\t%s\t%s\t\%s\n" $(df -h | grep /dev/sda5) | cut -f 5
```

4.7 awk

- awk '条件1{动作1} 条件2{动作2}...' 文件名
- 条件(Pattern)
    - 一般使用关系表达式作为条件
    - x > 10 判断变量x是否大于10
    - x >=10 大于等于
    - x
- 动作(Action)
    - 格式化输出
- $0 整行 $1 第一列...

```
df -h | grep /dev/sda5 |awk '{print $5}'| cut -d '%' -f 1
```

** 4.7.1 begin end #**

- awk可以正确截取制表符和空格
- begin 在所有的输出之前打印
- end 在所有的输出之后打印

```
awk 'BEGIN{print "开始"}END{print "结束"}' numbers.txt
```

** 4.7.2 FS #**

- Field Separator，字段分隔符

```
awk 'BEGIN{FS=":"}{print $1"\t"$2}' /etc/passwd
```

** 4.7.3 声明变量 #**

```
awk 'BEGIN{sum=0}{sum=sum+$1}END{print sum}' numbers.txt
```

** 4.7.4 多条件 #**

```
awk '$2>90{print $1"\t优秀"}$2>80{print $2"\t良好"}' score.txt
```

** 4.7.5 NR #**

- NR,表示awk开始执行程序后所读取的数据行数

```
awk '{print NR,$0}' file1
awk 'NR=3{print $0}' file1
```

** 4.7.6 OFS #**

- OFS Out of Field Separator，输出字段分隔符

```
echo "i love you" | awk 'BEGIN{ FS=" ";OFS="-" }{ print $1,$2,$3}'
```

4.8 sed命令

- sed是一个轻量级编辑器，主要用来对数据进行选取、替换、替换和新增操作
- sed [选项] '[动作]' 文件名
- 所有的动作都必须用单号号括起来
- 类型类似于批量vi操作

选项

参数 含义 示例 -n 一般sed命令会把所有的数据都输出到屏幕上，如果加入此选项则只会把处理过的行输出到屏幕上 `sed -n '2p' score.txt`

-e 允许对输入数据应用多条sed编辑命令 sed -e 's/75/70/g;s/55/50/g' score.txt -i 用sed的修改直接修改编辑的文件，而不是在屏幕上输出 `sed -i '1i newline' score.txt`

动作

参数 含义 示例 a 追加，在每一行或者指定行下面添加一行或多行 `sed '1a newline' score.txt`

c 行替换，用c后面的字符串替换掉原始整个数据行 `sed 'c newline' score.txt`

s 字符串替换,用一个字符串替换另外一个字符串，格式为 "行范围s/旧字符串/新字符串/g" `sed '3s/lisi/lisisi/g' score.txt`

i 插入，在当前行插入一行或多行 `sed '1i newline' score.txt`

d 删除指定的行 `sed '1,2d' score.txt`

p 打印,输出指定的行 `sed -n '2p' score.txt`

4.9 排序命令sort

- sort [选项] 文件名
- 选项

选项 含义 -f 忽略大小写 sort -f -t ":" -n -k 5,5 /etc/passwd -n 以数值型进行排序，默认使用字符串顺序 sort -t ":" -n -k 3,3 /etc/passwd -r 反向排序，默认从小到大 sort -r /etc/passwd -t 指定分割符，默认分割符是制表符 sort -t ":" -k 3,3 /etc/passwd -k n[,m] 按照指定的字段范围排序。从第n个字段开始，到第m个字段结束，默认是到行尾 sort -t ":" -k 3,3 /etc/passwd

```
sort /etc/passwd
sort -r /etc/passwd
sort -t ":" -k 3,3 /etc/passwd
```

4.10 wc

- wc [选项] 文件名

选项 含义 -l 只统计行数 -w 只允许单词数 -m 只统计字符数

```
wc wc.txt
```

5. 流程控制

5.1 条件判断

** 5.1.1 按照文件类型进行判断 #**

选项 含义 -d 文件是否存在并且是目录 -e 文件是否存在 -f 文件是否存在并且是普通文件 -b 文件是否存在并且是块设备文件 -c 文件是否存在并且是字符设备文件 -L 文件是否存在并且是链接文件 -p 文件是否存在并且是管道文件 -s 文件是否存在并且是否为非空 -S 文件是否存在并且是套接字文件

```
test -e 1.txt
[-e 1.txt]
echo $?
```

```
# [ -e 1.txt ]&&echo "yes"|| echo "no"
yes
# [ -e 11.txt ]&&echo "yes"|| echo "no"
no
```

** 5.1.2 按照文件权限进行判断 #**

选项 含义 -r 文件是否存在，并且是否拥有读权限 -w 文件是否存在，并且是否拥有写权限 -x 文件是否存在，并且是否拥有执行权限

```
echo read  > read.txt
echo write  > write.txt
echo execute  > execute.txt

chmod u+w write.txt
chmod u+x execute.txt

[ -e read.txt ]&&echo "read yes"|| echo "no"
[ -e write.txt ]&&echo "write yes"|| echo "no"
[ -e execute.txt ]&&echo "execute yes"|| echo "no"
```

** 5.1.3 两个文件间的比较 #**

选项 含义 文件1 -nt 文件2 判断文件1的修改时间是否比文件2的新 文件1 -ot 文件2 判断文件1的修改时间是否比文件2的旧 文件1 -ef 文件2 判断文件1和文件2的inode号是否一致,可用于判断硬链接

```
[ write.txt -nt read.txt ]&&echo "write is older than read"|| echo "no"
[ read.txt -ot write.txt ]&&echo "read is older than write"|| echo "no"
ln execute.txt execute2.txt
[ execute.txt -ef execute2.txt ]&&echo "execute and execute2.txt are the same"|| echo "no"
```

** 5.1.4 两个整数间的比较 #**

选项 含义 整数1 -eq 整数2 判断整数1是否和整数2相等 整数1 -ne 整数2 判断整数1是否和整数2不相等 整数1 -gt 整数2 判断整数1是否大于整数2 整数1 -lt 整数2 判断整数1是否小于整数2 整数1 -ge 整数2 判断整数1是否大于等于整数2 整数1 -le 整数2 判断整数1是否小于等于整数2

```
[ 2 -eq 2 ]&&echo "2==2"|| echo "no"
[ 3 -ne 2 ]&&echo "2!=2"|| echo "no"
[ 3 -gt 2 ]&&echo "2>2"|| echo "no"
[ 1 -lt 2 ]&&echo "2|| echo "no"
[ 2 -ge 2 ]&&echo "2>=2"|| echo "no"
[ 2 -le 2 ]&&echo "2|| echo "no"
```

** 5.1.5 字符串的判断 #**

选项 含义 -z 字符串 判断字否串是否为空 -n 字符中 判断字符串是否为非空 字符串1 == 字符串2 判断字符串1是否和字符串2相等 字符串1 != 字符串2 判断字符串1是否和字符串2不相等

```
name=zhufeng
[ -z "$name" ]&&echo "yes"|| echo "no"
name2=zhufeng
[ "$name" == "$name2" ]&&echo "yes"|| echo "no"
```

** 5.1.6 多重条件判断 #**

选项 含义 判断1 -a 判断2 逻辑与 判断1 -o 判断2 逻辑或 !判断 逻辑非

```
[ 2 -gt 1 -a 3 -gt 2 ]&&echo "yes"|| echo "no"
[ 2 -gt 1 -a 3 -gt 4 ]&&echo "yes"|| echo "no"
[ 2 -gt 1 -o 3 -gt 4 ]&&echo "yes"|| echo "no"
[ ! 3 -gt 4 ]&&echo "yes"|| echo "no"
```

5.2 单分支if语句

- if语句使用 fi结尾
- [条件判断式]就是使用 test命令进行判断，所以中括号和条件判断式之间必须有空格
- then后面跟符合条件之后执行的程序，可以放在[]之后，用;分隔，也可以换行，不用;

** 5.2.1 语法 #**

```
if [条件判断];then
 代码体
fi
```

```
if [条件判断]
then
 代码体
fi
```

```
if [ 2 -gt 1 ];then echo bigger; fi
```

** 5.2.2 判断当前用户是否是root用户 #**

```
user=$(whoami)
user=`whoami`
if [ "$user" == root ]
then
echo "I am root"
fi
```

5.3 双分支if语句

** 5.3.1 语法 #**

```
if [条件判断]
then
 代码体1
else
 代码体2
fi
```

** 5.3.2 判断是否目录 #**

```
read -t 10 -p "please input a filename" dir
if [ -d "$dir" ]
then
  echo "is directory"
else
  echo "not a direcotry"
fi
```

5.4 多分支if语句

** 5.4.1 语法 #**

```
if [条件判断1]
then
 代码体1
elif [条件判断2]
 代码体2
else
 代码体3
fi
```

```
read -p "please input a grade" grade
if [ "$grade" -gt 90 ]
 then
  echo great
elif [ "$grade" -gt 80 ]
  then
   echo good
else
   echo bad
fi
```

5.5 case 语句

- case 和if 都是多分支判断语句,if能判断多个条件,case只能判断一个条件**5.5.1 语法#**

```
case 变量名 in
值1)
  代码块1
  ;;
值2)
  代码块2
 ;;
*)
  代码块3
 ;;
esac
```

```
read -p "yes or no?" -t 30 choose
case $choose in
  "yes")
    echo 'yes'
    ;;
  "no")
    echo "no"
   ;;
  *)
    echo other
   ;;
esac
```

5.6 for循环

5.6.1 语法

```
for 变量 in 值1 值2 值3
do
代码块
done
```

```
for i in 1 2 3
do
 echo $i
done
```

5.6.2 语法

```
for((i=1;i10;i++));
do
 echo $(($i));
done
```

5.7 while循环

- while循环是不定循环，也称为条件循环，只要条件判断成立，就会一直继续

```
while [条件判断式]
do
 代码块
done
```

```
i=1
result=0
while [ $i -le 100 ]
 do
   result=$(($result+i))
   i=$(($i+1))
 done
echo $result
```

5.8 until循环

- 直到条件不成立停止

```
i=1
result=0
until [ $i -gt 100 ]
 do
   result=$(($result+i))
   i=$(($i+1))
 done
echo $result
```

6. 函数

- linux shell 可以用户定义函数，然后在shell脚本中可以随便调用
- 可以带function fun() 定义，也可以直接fun() 定义,不带任何参数
- 调用函数不需要加()

6.1 简单函数

```
[ function ] funcname [()]
{
    action;
    [return int;]
}
```

```
start(){
>  echo start
> }
 start
```

```
start(){
 echo start
}
start
```

6.2 返回值

- 参数返回，可以显示加：return 返回，如果不加，将以最后一条命令运行结果，作为返回值

```
> sum4(){
> r=$(($1+$2))
> return $r
> }
# sum4 2 3
echo $?

5
```

6.3 参数说明

参数处理 说明 $# 传递到脚本的参数个数 $* 以一个单字符串显示所有向脚本传递的参数 $@ 与$*相同，但是使用时加引号，并在引号中返回每个参数 $ 脚本运行的当前进程ID号 $! 后台运行的最后一个进程的ID号 $- 显示Shell使用的当前选项，与set命令功能相同 $? 显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误

6.4 profile

- parameter-substitution (http://tldp.org/LDP/abs/html/parameter-substitution.html)
- shell帮助文档 (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_06_02)

```
echo $-
echo "${-#*i}"
```

6.4.1 脚本# cat /etc/profile

```
for i in /etc/profile.d
```

- 如果现在执行的命令(或脚本)是交互式（interactively），那么加载之
- 如果执行的命令（或脚本）是非交互式（non-interactively），那么加载之，并且将输出重定向到 /dev/null,目的是防止有些执行某些程序（如scp），需要登录到远端机器，读取远端机器的环境变量有输出，导致程序中断！

** 6.4.2 打印当前shell的选项 #**

```
#  echo $-
himBH
```

- $-记录着当前设置的shell选项， himBH是默认值，你可以通过 set 命令来设置或者取消一个选项配置。例如：

```
set -x
```

- 这个可以打开 shell 的调试开关，调试 shell 脚本非常有用，这个时候再检查下 $- 变量的值，可以看到多了 x 字符：

```
[root@localhost ~]# echo $-
+ echo himxBH
himxBH
```

** 6.4.3 选项 #**

- 查看当前 shell 的选项配置

```
set -o
```

6.4.3.1 i - interactive

- 包含这个选项说明当前的 shell 是一个交互式的 shell，何为交互式？你输入命令，shell 解释执行后给你返回结果，我们在 Terminal 下使用的 shell 就是交互式的 如果我们在一个脚本里面 echo $-，结果是不会包含 i 的

** 6.4.3.2 H - history expand #**

- history expand 就是展开历史列表中的命令，可以通过!感叹号来完成，例如"!!"返回上最近的一个历史命令，"!n"返回第 n 个历史命令，等等

** 6.4.3.3 B - brace expansion #**

- 大括号扩展，是可以让bash生成任意字符串的一种扩展功能

```
echo g{a,b,c}$
```

** 6.4.3.4 m - monitor mode #**

- 打开监控模式,可以控制进程的停止、继续，后台或者前台执行
- 正常情况下，在交互式模式下，该选项默认是打开的，所以再执行一个比较耗时的命令时，你可以按下 CTRL+Z 让它在后台运行，然后可以用 fg 命令将后台运行的任务恢复到前台执行

```
 sleep 10s
```

** 6.4.3.5 h - hashal #**

- Remember the location of commands as they are looked up for execution. This is enabled by default. 记录命令的位置

** 6.4.4 删除 #**

从左往右看，删除掉 $- 变量的值中第一个 i 字符以及之前的内容

```
${-#*i}
```

%与#号的意义刚好相反，从右往左看，删除掉 $- 变量的值中最后一个 i 字符以及之后的内容

```
${-%i*}
```

7.shell实战

7.1 注意事项

- 开头加解释器: #/bin/bash和注释说明
- 命名建议规则: 变量名大写、局部变量小写，函数名小写，名字体现出实际作用
- 默认变量是全局的，在函数中变量local指定为局部变量，避免污染其它作用域
- set -e 遇到执行非0时退出脚本, set -x 打印执行过程
- 写脚本一定要先测试再上生产环境

7.2 获取随机字符串或数字

- -c, --characters=LIST select only these characters

** 7.2.1 获取随机8位字符串 #**

- MD5全称是报文摘要算法（Message-Digest Algorithm 5），此算法对任意长度的信息逐位进行计算，产生一个二进制长度为128位(十六进制长度就是32位)的报文摘要,不同的文件产生相同的报文摘要的可能性是非常非常之小的

```
echo $RANDOM |md5sum|cut -c 1-8
cat /proc/sys/kernel/random/uuid | cut -c 1-8
```

** 7.2.2 获取随机8位数字 #**

- cksum命令是确保文件从一个系统传输到另一个系统地过程中没有被损坏。这个测试要求校验和在源系统中被计算出来,在目的系统中又被计算一次，两个数字比较，如果校验和相等，则该文件被认为是被正确传

输了
- %N 十亿分之一秒，纳秒 [000000000，999999999]

```
echo $RANDOM |cksum|cut -c 1-8
date +%N | cut -c 1-8
```

7.3 定义一个颜色输出字符串函数

- 终端的字符颜色是用转义序列控制的，是文本模式下的系统显示功能
- 转义序列是以ESC开头,即用\033来完成（ESC的ASCII码用十进制表示是27，用八进制表示就是033）
- 书写格式: 开头部分： \033[显示方式;前景色;背景色m + 结尾部分：\033[0m 显示方式

数值 含义 0 默认值 1 高亮 22 非粗体 4 下划线 24 非下划线 5 闪烁 25 非闪烁 7 反显 27 非反显

前景色

数值 含义 30 黑色 31 红色 32 绿色 33 黄色 34 蓝色 35 洋红 36 青色 37 白色

背景色

数值 含义 40 黑色 41 红色 42 绿色 43 黄色 44 蓝色 45 洋红 46 青色 47 白色

```
echo -e "\033[m hello \033[m"
显示方式：正常     字体前景色：红色   背景色：青色
echo -e "\033[0;31;46m hello \033[m"
```

```
#!/bin/bash
#description: test
function echo_color(){
  if [ $1 == "green" ]; then
     echo -e "\033[32;40m$2\033[0m"
  elif [ $1 == "red" ]; then
     echo -e "\033[31;40m$2\033[0m"
  fi
}

echo -e "\033[32;40mshell\033[0m"
echo -e "\033[33;40mshell\033[0m"

echo_color red hello
```

7.4 批量创建用户

- /dev/null 2>&1 这条命令的意思是将标准输出和错误输出全部重定向到/dev/null中,也就是将产生的所有信息丢弃
- command > file 2>file 的意思是将命令所产生的标准输出信息,和错误的输出信息送到 file中。 command > file 2>file 这样的写法,stdout和stderr都直接送到file中, file会被打开两次,这样stdout和stderr会互相覆盖,这样写相当使用了FD1和FD2两个同时去抢占file 的管道
- command >file 2>&1 这条命令就将stdout直接送向file, stderr 继承了FD1管道后,再被送往file,此时,file 只被打开了一次,也只使用了一个管道FD1,它包括了stdout和stderr的内容

符号 含义 > 代表重定向到哪里 /dev/null 代表空设备文件 2 表示stderr标准错误 & 表示等同于的意思，2>&1，表示2的输出重定向等同于1 1 1表示stdout标准输出，系统默认值是1，所以">/dev/null"等同于"1>/dev/null"

```
USER_FILE=users.txt
for USER in user{1..5}; do
  if ! id $USER &>/dev/null; then
    PASS=$(echo $RANDOM | md5sum | cut -c 1-8)
    useradd $USER
    echo $PASS | passwd --stdin $USER &> /dev/null
    echo -e "$USER\t$PASS" >> $USER_FILE
    echo "$USER user create successfully."
  else
    echo_color red "$USER already exists.";
  fi
done
```

7.5 检查主机存活状态

```
for IP in $@; do
  if ping -c 1 $IP &>/dev/null; then
     echo "$IP is ok."
  else
     echo "$IP is wrong!"
  fi
done
```

7.6 系统监控

** 7.6.1 系统命令 #**** 7.6.1.1 vmstat #**

- 借助 vmstat工具来分析CPU统计信息
- vmstat(Virtual Memory Statistics 虚拟内存统计) 命令用来显示Linux系统虚拟内存状态，也可以报告关于进程、内存、I/O等系统整体运行状态
- vmstat [刷新延时 刷新次数]

```
vmstat 1 3
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0    532 329932  99388 459768    0    0    16    81   59   50  3  1 96  0  0
```

7.6.1.1.1 (procs)进程信息字段

分类 参数 含义 procs r 等待运行的进程数，数量越大，系统就越繁忙 procs b 不可被唤醒的进程数量，数量越大，系统越繁忙

7.6.1.1.2 memory(内存信息字段)

分类 参数 含义 memory swpd 使用的Swap空间的大小，单位KB memory free 空闲的内存容量，单位KB memory buff 缓冲的内存容量，单位KB memory cache 缓存的内存容量，单位KB

7.6.1.1.3 swap(交换分区信息)

- 如果说si和so数越大说明数据经常要在磁盘和内存之间数据交换，系统性能就会越差

分类 参数 含义 swap si(in) 从磁盘中交换到内存中的数据的数量，单位KB swap so(out) 从内存中交换到硬盘中的数据的数量，单位KB

7.6.1.1.4 io(磁盘读写)

- bi和bo数越大，说明磁盘的I/O越繁忙

分类 参数 含义 io bi(in) 从块设备读入数据的问题，单位是块 io bo(out) 写到块设备的数据的总量，单位是块

7.6.1.1.5 system(系统信息字段)

- in和cs数越大，说明系统与接口设备的通信越繁忙

分类 参数 含义 system in(interrupt) 每秒被中断的进程次数 system cs(switch) 每秒钟进行的事件切换次数

7.6.1.1.6 CPU(CPU信息字段)

分类 参数 含义 CPU us(user) 非内核进程消耗CPU运算时间的百分比 CPU sy(system) 内核进程消耗CPU运算时间的百分比 CPU id(idea) 空闲CPU的百分比 CPU wa(wait) 等待I/O所消耗的CPU百分比 CPU st(steal) 被虚拟机偷走的CPU百分比
** 7.6.1.2 free #**

- 查看内存使用状态
- free [-b|-k|-m|-g]

选项 |参数名称|参数含义| |:----|:----| |-b|以字节为单位| |-k|以KB字节为单位| |-m|以MB字节为单位| |-g|以GB字节为单位|

```
# free -m
             total       used       free     shared    buffers     cached
Mem:          1006        687        319          0         98        449
-/+ buffers/cache:        139        866
Swap:         1999          0       1999
```

7.6.1.2.1 第一行

分类 参数 含义 total 内存总数,total=used+free used 已经使用的内存数 free 空闲的内存数 shared 多个进程共享的内存数,0，废弃，永远为0 buffers 缓冲区内存数,buffer用于作为写入磁盘的内容缓冲区 cached 缓存内存数,用于从磁盘中读取的内容缓存

7.6.1.2.2 第二行

- 用于指示应用程序看到的内存情况，也就是应用程序获取内存大小

参数 算法 含义 - buffers/cache 第一行的used-buffers-cached 已经使用的要减去缓存和缓冲的内存量 + buffers/cache 第一行的free+buffers+cached 空闲的要加上缓存和缓冲的内存量

7.6.1.2.3 第三行

分类 参数 含义 total swap总数，默认单位是K used 已经使用的swap数，默认单位是K free 空闲的swap数，默认单位是K
** 7.6.1.3 查询网络状态 #**

- netstat 选项

选项 含义 -t 列出TCP协议端口 -u 列出UDP协议端口 -n 不使用域名与服务名，而使用IP地址和端口号 -l 仅列出在监听状态网络服务 -a 列出所有的网络连接

```
netstat -tlun
netstat -an | more
netstat -unt | grep  ESTABLISHED
```

** 7.6.2 函数 #**** 7.6.2.1 cpu #**

```
cpu(){
    local user system idle cwait
    user=$(vmstat | awk 'NR==3{print $13}')
    system=$(vmstat | awk 'NR==3{print $14}')
    idea=$(vmstat | awk 'NR==3{print $15}')
    cwait=$(vmstat | awk 'NR==3{print $16}')
    echo "user cpu: $user%"
    echo "system cpu: $system%"
    echo "idle cpu: $idea%"
    echo "wait: $cwait%"
}
cpu
```

** 7.6.2.2 memory #**

```
memory(){
  local total used free
  used=$(free -m | awk 'NR==3{print $3}')
  free=$(free -m | awk 'NR==3{print $4}')
  total=$(($used+$free))
  echo "内存总计: ${total}M"
  echo "内存使用: ${used}M"
  echo "内存剩余: ${free}M"
}
memory
```

** 7.6.2.3 disk #**

```
disk(){
  local mount total used used_percent free
  partitions=$(df -h|awk 'BEGIN{OFS="="}/^\/dev/{print $6,$2,$3,$4,$5}')
  echo $p
  for p in  $partitions; do
    mount=$(echo $p | cut -d"=" -f1)
    total=$(echo $p | cut -d"=" -f2)
    used=$(echo $p | cut -d"=" -f3)
    free=$(echo $p | cut -d"=" -f4)
    used_percent=$(echo $p | cut -d"=" -f5|cut -d"%" -f1)
    if [ $used_percent -ge 5 ]; then
      echo "挂载点=$mount,总大小=$total,使用大小=$used,空闲大小=$free,使用百分比=$used_percent%"
    fi
  done
}
```

7.7 监控网络流量

```
network(){
    local old_in old_out new_in new_out
    old_in=$(ifconfig ens33 | awk '/RX/&&/bytes/{print $5}')
    old_out=$(ifconfig ens33 | awk '/TX/&&/bytes/{print $5}')
    sleep 1s
    new_in=$(ifconfig ens33 | awk '/RX/&&/bytes/{print $5}')
    new_out=$(ifconfig ens33 | awk '/TX/&&/bytes/{print $5}')
    in=$(($new_in-$old_in))
    out=$(($new_out-$old_out))
    echo "$1 入口流量=${in}bytes/s 出口流量=${out}bytes/s"
}
network
```

7.8 监控网站状态

- -w 可以获取一些变量的值

```
curl -o /dev/null -s -w "%{http_code}" http:

function check_url(){
    HTTP_CODE=$(curl -o /dev/null -s -w "%{http_code}" $1)
    if [ $HTTP_CODE -ne 200 ]; then
      echo "$1不可达"
    else
      echo "$1状态正常"
    fi
}
```

7.9 监控nginx状态

** 7.9.1 本地安装nginx #**

```
rpm -Uvh http:
yum install -y nginx
systemctl start nginx.service
systemctl status  nginx.service
```

** 7.9.2 监控脚本 #**

```

local nginx
nginx=`ps -ef |grep nginx|grep -v grep|wc -l`
 if [ $nginx -gt 2 ];then
    echo "your nginx is running"
    exit 0
 else
    /bin/systemctl start nginx.service
    exit 1
fi
```

7.10 监控mysql状态

** 7.10.1 本地安装mysql #**

```
wget -i -c http:
yum -y install mysql57-community-release-el7-10.noarch.rpm
yum -y install mysql-community-server
systemctl start  mysqld.service
systemctl status mysqld.service

mysql -uroot -p
```

** 7.10.2 默认密码和忘记密码 #**

```
cat  /var/log/mysqld.log | grep  temporary

1. 修改/etc/my.conf, 添加参数skip-grant-tables
systemctl restart mysqld.service
mysql -uroot
use mysql
update user set authentication_string=PASSWORD('123456') where User='root';
alter user 'root'@'localhost' identified by 'Zfpx2019@qq.com';
```

** 7.10.3 监控脚本 #**

```

PortNum=`netstat -lnt|grep 3306|wc -l`
if [ $PortNum -eq 1 ]
then
 echo "mysqld is running."
else
 echo "mysqld is stoped."
fi
```

7.11 mysql备份脚本

```
DATE=$(date +%F_%H-%M-%S)
HOST=127.0.0.1
DB=test
USER=root
PASS=Zfpx2019@qq.com
MAIL="83687401@qq.com"
BACKUP_DIR=/data/db_backup
if [ ! -d "$BACKUP_DIR" ];then
 mkdir -p $BACKUP_DIR
fi
SQL_FILE=${DB}_FULL_$DATE.sql
BAK_FILE=${DB}_FULL_$DATE.zip
cd $BACKUP_DIR
if mysqldump -h$HOST -u$USER -p$PASS -B $DB > $SQL_FILE; then
  zip $BAK_FILE $SQL_FILE && rm -rf $SQL_FILE
  if [ ! -s $BAK_FILE ]; then
    echo "$DATE 备份失败" | mail -s "备份失败" $MAIL
  fi
else
  echo "$DATE 备份失败" | mail -s "备份失败" $MAIL
fi
find $BACKUP_DIR -name '*.zip' -ctime +14 -exec rm {} \;
```

8. 附录

8.1. Linux特殊文件

** 8.1.1 块设备 #**

- 块设备：系统能够随机无序访问固定大小的数据片的设备，这些数据片称为块。块设备是以固定大小长度来传送资料的，它使用缓冲区暂存数据，时机成熟后从缓存中一次性写入到设备或者从设备中一次性放到缓存区。常见的块设备有硬盘、CD-ROM驱动器、Flash闪存等等，它们也是通过文件形式存在于Linux中的。Linux以 b(block)表示块设备

```
# ll /dev/sr0
brw-rw----.   /dev/sr0
```

** 8.1.2 字符设备 #**

- 字符设备：按照字符流方式被有序访问，以不定长度的字符传送资料，不存在缓冲区，所以对这种设备的读写都是实时的，比如键盘等等。Linux以 c(char)表示字符设备

```
# ll /dev/input/mouse0
crw-r-----.   /dev/input/mouse0
# ll /dev/input/event0
crw-r-----.  /dev/input/event0
```

** 8.1.3 管道文件 #**

- 管道文件是linux下的一种特殊缓存文件,所谓缓存就是只会存在于进程执行的时候，进程关闭管道文件也关闭了,管道文件要在读写的时候，先阻塞在写操作的open函数，当有读操作进来的时候，通信链路建立，写操作开始往管道里写东西，写完成之后，在读操作会把东西读出来，最后两个进程都正常结束退出,Linux以 p(pipe)表示管道设备

```
find / -type p
ll /var/run/autofs.fifo-misc
```

** 8.1.4 套接字 #**

- 套接字Socket看做是不同主机之间的进程进行双向通信的端点，简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。套接字Socket是连接应用程序和网络驱动程序的桥梁，套接字Socket在应用程序中创建，通过绑定与网络驱动建立关系。此后，应用程序送给套接字Socket的数据，由套接字Socket交给网络驱动程序向网络上发送出去。计算机从网络上收到与该套接字Socket绑定IP地址和端口号相关的数据后，由网络驱动程序交给Socket，应用程序便可从该Socket中提取接收到的数据，网络应用程序就是这样通过Socket进行数据的发送与接收的。Linux以 s(Socket)表示套接字设备

```
[root@localhost input]# ll /tmp/mysql.sock
srwxrwxrwx. 1 mysql mysql 0 4月   6 12:18 /tmp/mysql.sock
[root@localhost input]# ll /var/run/cups/cups.sock
srwxrwxrwx. 1 root root 0 4月   6 17:51 /var/run/cups/cups.sock
```