shell编程语法

一、**shell**介绍

1. 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++, java, go

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。

2. shell介绍

总结：

- shell就是人机交互的一个桥梁
- shell的种类

```
[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

3. 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
# 1.使用绝对路径执行脚本
[root@MissHou shell01]# /shell/shell01/1.sh
MissHou.itcast.cc
Sun Jul 22 14:41:00 CST 2018
# 2.使用相对路径进行执行脚本
[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；常用来在修改脚本之后通过source进行执行，查看能否有错
```

二、bash的特性

Bash是一个命令处理器，通常运行于文本窗口中，并能执行用户直接输入的命令。

Bash还能从文件中读取命令，这样的文件称为脚本。

和其他Unix shell 一样，它支持文件名替换（通配符匹配）、管道、here文档、命令替换、变量，以及条件判断和循环遍历的结构控制语句。

包括关键字、语法在内的基本特性全部是从sh借鉴过来的。

其他特性，例如历史命令，是从csh和ksh借鉴而来。

总的来说，Bash虽然是一个满足POSIX规范的shell，但有很多扩展。

1、命令和文件名自动补全

命令补全：

1. 查找内部命令；
2. 查找外部命令：

bash根据PATH环境变量定义的路径，自左而右在每个路径搜寻以给定命令名命名的文件；
第一次找到的命令即为要执行的命令，用户给定的字符串只有一条唯一对应的命令，直接补全，否则，再次Tab会给出列表。

```
查看环境变量
[root@zgs ~]# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
wh后双击tab键，列出所有wh开头命令
[root@zgs ~]# wh
whatis    which     whiptail  whoami
whereis   while     who
```

2、常见的快捷键

- Ctrl + l 清屏，相当于clear命令
- Ctrl + s 阻止屏幕输出，锁定
- Ctrl + q 允许屏幕输出
- Ctrl + c 终止命令
- Ctrl + z 挂起命令
- Ctrl + a 光标移到命令行首，相当于Home
- Ctrl + e 光标移到命令行尾，相当于End
- Ctrl + u 从光标处删除至命令行首
- Ctrl + k 从光标处删除至命令行尾
- Alt + r 删除当前整行

注意：**Alt**组合快捷键经常和其它软件冲突

3、常用的通配符

> * 匹配零个或多个字符
>
> ? 匹配任何单个字符
>
> ~ 当前用户家目录
>
> ~ 用户家目录
>
> ~+ 当前工作目录
>
> ~- 前一个工作目录
>
> *[0-9]* 匹配数字范围
>
> *[a-z]*：小写字母; 也能匹配大写
>
> *[A-Z]*：大写字母; 也能匹配小写
>
> *[abc]* 匹配列表中的任何的一个字符
>
> *[^abc]* 匹配列表中的所有字符以外的字符

4、bash中的引号

强引用:''

引用里面的内容直接打印,不会当做变量

```
[root@zgs ~]# echo '$PATH'
$PATH
```

弱引用:""

可以识别到里面的变量进行打印变量内容

```
[root@zgs ~]# echo "$PATH"
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin

```

命令引用:`

反向单引号，引用命令，因此需要echo $PATH完整的命令才行，才会进行正确打印

建议使用 **$(cmd)**

```
[root@zgs ~]# echo `$PATH`
-bash:
/usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/root/bin: No such file or
directory

[root@zgs ~]# echo `echo $PATH`
/usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/root/bin

```

5、命令展开

- ~：自动展开为用户的家目录，或指定的用户的家目录；
- {}: 可承载一个以,分隔的路径列表，并将其进行展开为多个路径。

  yum install -y tree

```
[root@zgs ~]# tree
.
├── his.txt
├── h.txt
├── pwd
└── res.txt

0 directories, 4 files

```

```
[root@zgs ~]# mkdir -p dir{1,2}/{a,b{1,2,3}}
[root@zgs ~]# tree
.
├── dir1
|   ├── a
|   ├── b1
|   ├── b2
|   └── b3
├── dir2
|   ├── a
|   ├── b1
|   ├── b2
|   └── b3
├── his.txt
├── h.txt
├── pwd
```

```
└── res.txt

10 directories, 4 files

```

6、命令历史

shell进程会在其会话中保存此前用户提交执行过的命令：history

定制`history`的功能，可通过命令历史相关环境变量实现：

- HISTSIZE：命令历史记录的条数, 最大数, 历史列表容量
- HISTFILE：指定历史文件，默认为~/.bash_history
- HISTFILESIZE：命令历史文件记录历史的条数; 最大条数

```
[root@zgs ~]# echo $HISTSIZE
1000
[root@zgs ~]# echo $HISTFILE
/root/.bash_history
[root@zgs ~]# echo $HISTFILESIZE
10000

```

查看历史

```
[root@zgs ~]# history
    1  root 2022/10/16 18:02:12 history
    2  root 2022/10/16 18:02:19 clear
    3  root 2022/10/16 18:02:20 ls
    4  root 2022/10/16 18:02:24 history
```

-c: 清空命令历史

```
[root@zgs ~]# history -c
[root@zgs ~]# history
    1  root 2022/10/16 18:03:23 history

```

-d offset: 删除历史中指定的第 几 个命令

```
[root@zgs ~]# history
    1  root 2022/10/16 18:03:23 history
    2  root 2022/10/16 18:10:16 history
    3  root 2022/10/16 18:15:38 ls
    4  root 2022/10/16 18:15:39 clear
    5  root 2022/10/16 18:15:41 history
[root@zgs ~]# history -d 3
[root@zgs ~]# history
    1  root 2022/10/16 18:03:23 history

```

```
    2  root 2022/10/16 18:10:16 history
    3  root 2022/10/16 18:15:39 clear
    4  root 2022/10/16 18:15:41 history
    5  root 2022/10/16 18:15:57 history -
d 3
    6  root 2022/10/16 18:16:00 history

```

n: 显示最近的n条历史

```
[root@zgs ~]# history 3
    6  root 2022/10/16 18:16:00 history
    7  root 2022/10/16 18:16:54 clear
    8  root 2022/10/16 18:22:15 history 3
[root@zgs ~]# history
    1  root 2022/10/16 18:03:23 history
    2  root 2022/10/16 18:10:16 history
    3  root 2022/10/16 18:15:39 clear
    4  root 2022/10/16 18:15:41 history
    5  root 2022/10/16 18:15:57 history -
d 3
    6  root 2022/10/16 18:16:00 history
    7  root 2022/10/16 18:16:54 clear
    8  root 2022/10/16 18:22:15 history 3
    9  root 2022/10/16 18:22:22 history
[root@zgs ~]# history 3
    8  root 2022/10/16 18:22:15 history 3

```

```
    9  root 2022/10/16 18:22:22 history
   10  root 2022/10/16 18:22:25 history 3

```

-a: 追加本次会话新执行的命令历史列表至历史文件

```
[root@zgs ~]# history -a his.txt
[root@zgs ~]# ls
his.txt  pwd  res.txt
[root@zgs ~]# cat his.txt
#1665916576
history -a his.txt
[root@zgs ~]# grep 'his' his.txt |
history -a his.txt
[root@zgs ~]# cat his.txt
#1665916576
history -a his.txt
#1665916634
grep 'his' his.txt | history -a his.txt

```

-r: 读历史文件附加到历史列表

```
   18  root 2022/10/16 18:37:14 grep 'his' his.txt | history -a his.txt
   19  root 2022/10/16 18:37:18 cat his.txt
   20  root 2022/10/16 18:37:52 clear
   21  root 2022/10/16 18:38:06 history
[root@zgs ~]# history -r his.txt
[root@zgs ~]# history
    ...
   20  root 2022/10/16 18:37:52 clear
   21  root 2022/10/16 18:38:06 history
   22  root 2022/10/16 18:38:15 history -r his.txt
   23  root 2022/10/16 18:36:16 history -a his.txt
   24  root 2022/10/16 18:37:14 grep 'his' his.txt | history -a his.txt
   25  root 2022/10/16 18:38:20 history
```

-w: 保存历史列表到指定的历史文件

```
[root@zgs ~]# history -w h.txt
[root@zgs ~]# head h.txt
#1665914603
history
#1665915016
history
#1665915339
clear
#1665915341
history
#1665915357
history -d 3

```

-n: 读历史文件中未读过的行到历史列表

```

```

!!:执行上一条命令

```
[root@zgs ~]# history
    1  root 2022/10/16 18:03:23 history
    2  root 2022/10/16 18:10:16 history
    3  root 2022/10/16 18:15:39 clear
    4  root 2022/10/16 18:15:41 history


```

```
    5  root 2022/10/16 18:15:57 history -d 3
    6  root 2022/10/16 18:16:00 history
    7  root 2022/10/16 18:16:54 clear
    8  root 2022/10/16 18:22:15 history 3
    9  root 2022/10/16 18:22:22 history
   10  root 2022/10/16 18:22:25 history 3
   11  root 2022/10/16 18:22:50 clear
   12  root 2022/10/16 18:29:22 history
[root@zgs ~]# !!
history
    1  root 2022/10/16 18:03:23 history
    2  root 2022/10/16 18:10:16 history

```

7.标准I/O和管道

```
程序：指令+数据
  读入数据：Input
  输出数据：Output
```

```
STDOUT和STDERR可以被重定向到文件
命令 操作符号 文件名
支持的操作符号包括：
1）>  把STDOUT重定向到文件
2）>> 把STDERR重定向到文件
```

```
> 将内容覆盖重定向到文件中
[root@zgs ~]# echo good > f1
[root@zgs ~]# cat f1
good
>> 将内容追加重定向到文件中
[root@zgs ~]# echo good123 >> f1
[root@zgs ~]# cat f1
good
good123
[root@zgs ~]# echo hello > f1
[root@zgs ~]# cat f1
hello
```

输入重定向:<

```
[root@zgs ~]# cat < b
123
[root@zgs ~]# cat < a
123
[root@zgs ~]# cat <a >c
[root@zgs ~]# cat c
123

```

多行输入重定向

```

```

```
[root@zgs ~]# cat > f1 << END
> a
> v
> s
> f
> END
[root@zgs ~]# cat f1
a
v
s
f
[root@zgs ~]# cat > f1 <<EOF
> 1
> 2
> 3
> EOF
[root@zgs ~]# cat f1
1
2
3

```

使用管道符 |

```
[root@zgs ~]# cat pwd | grep root
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
[root@zgs ~]# cat pwd | grep root | cut -d: -f1
root
operator
[root@zgs ~]#
[root@zgs ~]# ll /etc/ | less
```

三、shell变量使用

1.定义变量

注意: 变量与值之间不能有空格; 不能写成 age = 12

```
[root@zgs ~]# name='zhangsan'
[root@zgs ~]# echo name
name
[root@zgs ~]# echo $name
zhangsan
[root@zgs ~]# name="lisi"
[root@zgs ~]# echo name
name
[root@zgs ~]# echo $name
lisi

```

bash 会将所有的变量当做字符串处理

打印变量的完整写法; 区分大小写

```
[root@zgs ~]# age=19
[root@zgs ~]# echo $age
19
[root@zgs ~]# echo ${age}
19
[root@zgs ~]# Age=20
[root@zgs ~]# echo $age
19
[root@zgs ~]# echo $Age
20
[root@zgs
```

单双引号问题:

- 单引号变量, 不识别特殊语法; 强引用
- 双引号变量, 能识别特殊符号; 弱引用
- 反引号变量(``), 里面能够识别命令; 命令引用

```
[root@zgs ~]# address='henan zhongmou heyi'
[root@zgs ~]# echo $address
henan zhongmou heyi
[root@zgs ~]# clear
[root@zgs ~]# d='$address'
[root@zgs ~]# echo $d
$address
[root@zgs ~]# s="$address"
[root@zgs ~]# echo $s
henan zhongmou heyi

# 单引号 是强引用
# 双引号 是弱引用, 会将值中的$翻译出来, 能够输出变量
# 反引号 是命令引用
[root@zgs ~]# c=`cat /etc/passwd`
[root@zgs ~]# echo $c
root:x:0:0:root:/root:/bin/bash
```

2.变量作用域

全局变量:

在当前的shell中任何地方使用的变量

在当前shell中任何地方都能使用，不同shell中的全局变量互不影响，在shell中定义的变量默认为全局变量。

打开两个shell窗口，在两个shell中定义名字相同,值不同的变量，然后输出，两个shell互不影响。

全局变量的范围是shell会话（进程）而不是shell脚本，也就是全局变量的作用域是其所在进程。

```
# 当前的用户的环境变量()
[root@zgs ~]# cat  $HOME/.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

```

```
PATH=$PATH:$HOME/bin

export PATH

```

```
# 当前用户的bash信息(aliase, umask)
[root@zgs ~]# cat  $HOME/.bashrc
# .bashrc

# User specific aliases and functions

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

```

```
# 每个用户退出当前shell时最后读取的文件
[root@zgs ~]# echo $HOME/.bash_logout
/root/.bash_logout
[root@zgs ~]# cat $HOME/.bash_logout
# ~/.bash_logout

```

```
# 使用bash shell系统全局变量
[root@zgs ~]# cat /etc/bashrc | head -3
# /etc/bashrc

# System wide functions and aliases

```

```
# 系统和每个系统的环境变量信息
[root@zgs ~]# cat /etc/profile | head -4
# /etc/profile

# System wide environment and startup
programs, for login setup
# Functions and aliases go in /etc/bashrc

```

用户登陆读取文件的顺序

```
用户登录系统读取相关文件的顺序：
/etc/profile-->$HOME/.bash_profile-->$HOME/.bashrc-->/etc/bashrc-->$HOME/.bash_logout
```

局部变量：

shell中支持自定义函数，与其他语言不同的是，shell中定义的变量默认也是全局变量，在函数外一样可以调用，想要定义局部变量，需要在定义时在变量名前加上local命令。

这样定义的变量就是局部变量，函数外就不能访问了。

```
[root@zgs ~]# name='tom'
[root@zgs ~]# echo $name
tom
[root@zgs ~]# sh
sh-4.2# echo $name

sh-4.2# exit
exit
[root@zgs ~]# echo $name
tom
# 在函数中使用
[root@zgs shells]# vim f.sh
[root@zgs shells]# ./f.sh
hello Tom
```

```
Tom
[root@zgs shells]# cat f.sh
#!/bin/bash
# 定义函数 hello()
hello(){
    name="Tom"
    echo "hello" $name
}
# 调用函数 hello()
hello
# 函数外使用name
echo $name

```

环境变量:

可以在任何shell中使用的变量

默认情况下变量的作用域是当前shell，如果用**export**命令将其导出，那么此变量在其所有的子**shell**中也生效，这种变量就是环境变量。

环境变量只能向下传递，即父shell可以传递给子shell，反过来则不行。

注意这里的环境变量不是变量在所有shell中都有效，而是在export变量时的shell的所有子shell中有效。

export 将当前变量变成环境变量

```
# 将变量A变成环境变量
[root@zgs ~]# export A=hello
# env 用来查看当前用户的环境中有哪些变量
[root@zgs ~]# env | grep '^A'
A=hello
[root@zgs ~]# B='bug'
[root@zgs ~]# export B
[root@zgs ~]# echo B
B

# set 查看当前用户的所有变量(临时/环境变量); 内容太多不建议查看所有
[root@zgs ~]# set | head
A=hello
ABRT_DEBUG_LOG=/dev/null
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_fignore:histappend:interactive_comments:login_shell:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_COMPLETION_COMPAT_DIR=/etc/bash_completion.d
```

```
22 BASH_LINENO=()
23
```

```
1 # 在子进程中查看是否有A变量
2 [root@zgs ~]# echo $A
3 hello
4 [root@zgs ~]# sh
5 sh-4.2# echo $A
6 hello
7
```

```
1 # 将变量写入文本，永久生效
2 [root@zgs ~]# vim /etc/profile
3 [root@zgs ~]# vim ~/.bashrc
4
```

系统变量(在内置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* 命令没有找到
> - 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

```
[root@zgs ~]# grep -q 'a' a.txt
[root@zgs ~]# echo $?
0
[root@zgs ~]# zgs 123
-bash: zgs: command not found
[root@zgs ~]# echo $?
127
[root@zgs ~]# cat ff.txt
cat: ff.txt: No such file or directory
[root@zgs ~]# echo $?
1
[root@zgs ~]# cd f
-bash: cd: f: No such file or directory
[root@zgs ~]# echo $?
1

```

- $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 从第10个位置开始，变量必须用{}大括号括起来

```
[root@zgs shells]# touch f2.sh
[root@zgs shells]# vim f2.sh
[root@zgs shells]# cat f2.sh
#!/bin/bash
echo 脚本名 $0
echo 参数1: $1
echo 参数个数: $#
echo 参数做整体输出: $*
echo 参数独立输出: $@

[root@zgs shells]# ls
a8.sh  f2.sh  f.sh
[root@zgs shells]# chmod +x f2.sh
[root@zgs shells]# ./f2.sh 1 2 3
脚本名 ./f2.sh
参数1: 1
参数个数: 3
参数做整体输出: 1 2 3
参数独立输出: 1 2 3
[root@zgs shells]#
```

> 注意: 分析$* 和 $@

> $* 在外边带双引号会将所有参数当做一个整体输出, 没有双引号会将传入的参数当做每个独立的参数输出
>
> $@ 无论是否添加双引号都会将参数当做独立参数输出

```
[root@zgs shells]# cat f3.sh
#!/bin/bash
# 挨个输出参数
for i in "$@"
do
  echo "\$@" $i
done
echo '------'
# 当做整体输出参数
for i in "$*"
do
  echo "\$*" $i
done
[root@zgs shells]# ./f3.sh 1 2 3
$@ 1
$@ 2
$@ 3
------
$* 1 2 3

```

3.变量使用规则

变量赋值

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

变量获取

> $变量名
>
> ${变量名} -- 完整写法

```
[root@zgs ~]# abc=123
[root@zgs ~]# echo abc
abc
[root@zgs ~]# q = ok
-bash: q: command not found
[root@zgs ~]# q=qq
[root@zgs ~]# echo q
q
```

```
[root@zgs ~]# echo $q
qq
[root@zgs ~]# echo $abc
123
[root@zgs ~]# p="how are you! "
[root@zgs ~]# echo $p
how are you!

```

获取某部分变量

- 这里 ${s:4:3} 中 4表示从第4个字符开始，3表示获取后边三个字符(第5,6,7个字符)

```
[root@zgs ~]# s="how are you"
[root@zgs ~]# echo ${s}
how are you
[root@zgs ~]# echo ${s:2:3}
w a
[root@zgs ~]# echo ${s::3}
how
[root@zgs ~]# echo ${s:4:}

[root@zgs ~]# echo ${s:4:3}
are

```

取消变量

> 全局变量可以取消, 环境变量也能

```
[root@zgs ~]# a=123
[root@zgs ~]# b=321
[root@zgs ~]# echo $a
123
[root@zgs ~]# echo $b
321
[root@zgs ~]# unset a
[root@zgs ~]# echo $a

[root@zgs ~]# export b
[root@zgs ~]# env | grep ^b
b=321
[root@zgs ~]# unset b
[root@zgs ~]# echo $b

```

变量区分大小写

```
[root@zgs ~]# qq='541863389'
[root@zgs ~]# echo $qq
541863389
[root@zgs ~]# Qq='10086'
[root@zgs ~]# echo $Qq
10086

```

变量命名

字母,数字,下划线, 但是不能数字或者特殊字符开头

```
[root@zgs ~]# a123_='abc_123_'
[root@zgs ~]# echo $a123_
abc_123_
[root@zgs ~]# 1abc='abc'
-bash: 1abc=abc: command not found

```

命令结果可以保存到变量中

注意这里使用 反引号

```
[root@zgs ~]# grep -i 'root' pwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
[root@zgs ~]# res=`grep -i 'root' pwd`
[root@zgs ~]# echo $res
root:x:0:0:root:/root:/bin/bash operator:x:11:0:operator:/root:/sbin/nologin
[root@zgs ~]#
```

```
# uname: 获取系统类型
[root@zgs ~]# uname
Linux
# -r: 获取当前系统版本信息
[root@zgs ~]# uname -r
3.10.0-1160.62.1.el7.x86_64
# -n: 获取当前系统名称
[root@zgs ~]# uname -n
zgs
# 通过$进行命令结果赋值
[root@zgs ~]# name=$(uname -n)
[root@zgs ~]# echo $name
zgs

```

有类型变量

declare -i 将变量定义整数变量

```
[root@zgs ~]# a=1
[root@zgs ~]# b=1
[root@zgs ~]# echo $a+$b
1+1
[root@zgs ~]# declare -i a=2
[root@zgs ~]# declare -i b=2
[root@zgs ~]# declare -i c=$a+$b
[root@zgs ~]# echo  $c
4
[root@zgs ~]# declare -i res=x+y
[root@zgs ~]# echo $res
330
[root@zgs ~]# declare -i res=x*y
[root@zgs ~]# echo $res
26600

```

declare -r 使变量只能读

```
[root@zgs ~]# declare -r say='read only'
[root@zgs ~]# echo $say
read only
[root@zgs ~]# say="read me"
-bash: say: readonly variable

```

declare -x 标记变量是环境变量

通过环境导出 export

```
[root@zgs ~]# declare -x x='env123'
[root@zgs ~]# env | grep env123
x=env123
[root@zgs ~]# X='env321'
[root@zgs ~]# env | grep env321
[root@zgs ~]# export X
[root@zgs ~]# env | grep env321
X=env321

```

declare -a 指定为索引数组(普通数组); 查看普通数组

(先学什么是数组吧)

```
[root@zgs ~]# declare -a arry='([0]="abc" [1]="qwr" [2]="zxc")'
[root@zgs ~]# echo $arry
abc
[root@zgs ~]# echo ${arry[1]}
qwr
[root@zgs ~]# echo ${arry[2]}
zxc
[root@zgs ~]# echo ${arry[@]}
abc qwr zxc

```

```
[root@zgs ~]# declare -a
declare -a BASH_ARGC='()'
declare -a BASH_ARGV='()'
declare -a BASH_LINENO='()'
declare -ar BASH_REMATCH='()'
declare -a BASH_SOURCE='()'
declare -ar BASH_VERSINFO='([0]="4" [1]="2" [2]="46" [3]="2" [4]="release" [5]="x86_64-redhat-linux-gnu")'
...

```

declare -A 指定为关联数组; 查看关联数组

```
[root@zgs ~]# declare -A
declare -A BASH_ALIASES='()'
declare -A BASH_CMDS='()'

```

交互式定义变量

让用户定义变量值

> 使用*read* 命令，让键盘给变量输入值

```
[root@zgs ~]# read name
zgs
[root@zgs ~]# echo $name
zgs
```

-p 提示信息

```
[root@zgs ~]# read -p '输入数字:' number
输入数字:789
[root@zgs ~]# echo $number
789

```

-n 字符数(限制变量字符个数)

```
[root@zgs ~]# read -n 6 -p '账户: '
username
账户: root
[root@zgs ~]# echo $username
root
[root@zgs ~]# read -n 6 -p '账户: '
username
账户: 541863[root@zgs ~]#
[root@zgs ~]# echo $username
541863

```

-s 不显示, 安静模式, 输入不显示字符

```
[root@zgs ~]# read -n 6 -p '密码: '
password
密码: 123321[root@zgs ~]#
[root@zgs ~]# read -sn 6 -p '密码: '
password
密码: [root@zgs ~]#
[root@zgs ~]# echo $password
123456

```

-t 超时(默认单位秒)(限制用户输入变量值的超时时间)

```
[root@zgs ~]# read -s -t3 -n6 -p'密码: ' password
密码: [root@zgs ~]#
[root@zgs ~]#

```

-a 后跟一个变量，该变量会被认为是个数组，然后给其赋值，默认是以空格为分割符。

```
[root@zgs ~]# read -p'数组: '  -a nlist
数组: 123 213 312
[root@zgs ~]# echo $nlist
123
[root@zgs ~]# read -p'请输入一个列表: ' -a array1
请输入一个列表: apple banana orange
[root@zgs ~]# echo $array1
apple
[root@zgs ~]# echo ${array1[*]}
apple banana orange

```

变量取值操作

> (取变量部分, 替换修改部分值, 使用新值)

1. 取出目录下的目录与文件：dirname 和 basename

```
[root@zgs ~]# path=/etc/sysconfig/network-scripts/ifcfg-eth0
[root@zgs ~]# echo $path
/etc/sysconfig/network-scripts/ifcfg-eth0
[root@zgs ~]# dirname $path
/etc/sysconfig/network-scripts
[root@zgs ~]# basename $path
ifcfg-eth0
[root@zgs ~]#
[root@zgs ~]# path2=`dirname $path`
[root@zgs ~]# echo $path2
/etc/sysconfig/network-scripts
[root@zgs ~]# dirname $path2
/etc/sysconfig
[root@zgs ~]# basename $path2
network-scripts

```

2. 变量'内容'的删除.

#是 去掉左边(键盘上#在 $ 的左边)

%是去掉右边(键盘上% 在$ 的右边)

单一符号是最小匹配；两个符号是最大匹配

> 一个“%”代表从右往左去掉一个/key/; 一个表示去掉一个
>
> 两个“%%”代表从右往左最大去掉/key/; 两个表示最大去掉,省一个
>
> 一个“#”代表从左往右去掉一个/key/; 一个表示去掉一个
>
> 两个“##”代表从左往右最大去掉/key/; 两个表示最大去掉,省一个

```
[root@zgs ~]# url=www.baidu.com
# 获取变量的长度
[root@zgs ~]# echo ${#url}
13

# 注意: 这里的 * 号表示匹配要删除的内容
# 这里表示将变量 以点分割 删除第1个点及之前的字符
[root@zgs ~]# echo ${url#*.}
baidu.com
# 这里表示将变量 以点分割 删除第2个点及之前的字符
[root@zgs ~]# echo ${url##*.}
com
# 这里表示将变量 以点分割 删除第2个点及之后的字符
[root@zgs ~]# echo ${url%.*}
www.baidu
# 这里表示将变量 以点分割 删除第1个点及之后的字符
[root@zgs ~]# echo ${url%%.*}
www
```

3. 变量内容替换 使用 /

```
# 单个替换
[root@zgs ~]# url=https://pypi.douban.com/simple
[root@zgs ~]# echo ${url/http/HTTP}
HTTPs://pypi.douban.com/simple
# 多个匹配，贪婪匹配
[root@zgs ~]# echo ${url//./-}
https://pypi-douban-com/simple
[root@zgs ~]# url='https://pypi.douban.com/simple'
[root@zgs ~]# echo ${url}
https://pypi.douban.com/simple
[root@zgs ~]# echo ${url/./=}
https://pypi=douban.com/simple
[root@zgs ~]# echo ${url//./=}
https://pypi=douban=com/simple
[root@zgs ~]# echo ${url///|}
https://pypi.douban.com/simple
[root@zgs ~]# echo ${url/\//|}
https:|/pypi.douban.com/simple
[root@zgs ~]# echo ${url////|}
https:||pypi.douban.com|simple

```

4. 使用- 或者 = 替换变量

> *${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; 不会被真赋值, 本身还是没有被赋值
>
> 变量有被赋值（包括空值）： 不会被替代

```
[root@zgs ~]# echo ${n}

[root@zgs ~]# echo ${n-123}
123
[root@zgs ~]# echo ${n}

[root@zgs ~]# m=998
[root@zgs ~]# echo ${m-123}
998
[root@zgs ~]# echo $m
998

```

```
[root@zgs ~]# echo ${y=110}
110
[root@zgs ~]# y=120
[root@zgs ~]# echo ${y=110}
120

```

> *echo ${t=777}* t变量没有被赋值, 使用之后会被赋这个新值
>
> 如果被赋值: 不会影响原来的值, 输出值也不会变

```
[root@zgs ~]# echo $t

[root@zgs ~]# echo ${t=777}
777
[root@zgs ~]# echo $t
777

```

```
[root@zgs ~]# echo $y

[root@zgs ~]# y=200
[root@zgs ~]# echo $y
200
[root@zgs ~]# echo ${y=250}
200
[root@zgs ~]# echo $y
200

```

> *${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; 不会真被赋值, 变量还是空值
>
> 变量有被赋值： 不会被替代; 也不会改变被赋值

```
[root@zgs ~]# echo $i

[root@zgs ~]# echo ${i:-233}
233
[root@zgs ~]# echo $i

```

```
[root@zgs ~]# echo $j

[root@zgs ~]# j=666
[root@zgs ~]# echo ${j:-555}
666
[root@zgs ~]# echo $j
666

```

> *${变量名:=新的变量值}*
>
> 没有被赋值: 使用之后会使用新值, 本身被赋新值

> 有被赋值:变量被不会受到影响

```
[root@zgs ~]# echo $a

[root@zgs ~]# echo ${a:=123}
123
[root@zgs ~]# echo $a
123

```

```
[root@zgs ~]# echo $b

[root@zgs ~]# b=321
[root@zgs ~]# echo $b
321
[root@zgs ~]# echo ${b:=234}
321
[root@zgs ~]# echo $b
321

```

> *${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： 会被新值替代, 不会改变变量本身

```
[root@zgs ~]# echo ${u+132}

[root@zgs ~]# echo $u

[root@zgs ~]# u=111
[root@zgs ~]# echo $u
111
[root@zgs ~]# echo ${u+132}
132
[root@zgs ~]# echo $u
111

```

> *${*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）： 会被新值替代, 变量本身还是原来数值

```
[root@zgs ~]# echo ${q:+900}

[root@zgs ~]# echo $q

[root@zgs ~]# q=180
[root@zgs ~]# echo $q
180
[root@zgs ~]# echo ${q:+900}
900
[root@zgs ~]# echo $q
180

```

> *${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@zgs ~]# echo ${d?111}
-bash: d: 111
[root@zgs ~]# echo $d

[root@zgs ~]# d=220
[root@zgs ~]# echo ${d?111}
220
[root@zgs ~]# echo ${d?}
220
[root@zgs ~]# echo ${w?}
-bash: w: parameter null or not set
[root@zgs ~]#
```

> *${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

```
[root@zgs ~]# echo ${abc:?123}
-bash: abc: 123
[root@zgs ~]# abc=998
[root@zgs ~]# echo ${abc:?123}
998
[root@zgs ~]# echo $abc
998
[root@zgs ~]# echo ${abc:?}
998

```

四、数组使用

普通数组：

> 只能使用整数作为数组索引(元素的下标) -- 类似*python*列表

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

```
[root@zgs ~]# list[0]=123
[root@zgs ~]# list[2]=456
[root@zgs ~]# list[3]=789
[root@zgs ~]# echo $list
123
[root@zgs ~]# echo $list[@]
123[@]
[root@zgs ~]# echo ${list[@]}
123 456 789

```

- 一次赋多个值

  变量名=(值1 值2 值3 . . .)

```
[root@zgs ~]# lt=(1 2 3 11 22 33)
[root@zgs ~]# echo $lt
1
[root@zgs ~]# echo ${lt}
1
[root@zgs ~]# echo ${lt[@]}
1 2 3 11 22 33

```

```
# 使用命令结果赋值
[root@zgs ~]# array1=(`cat /etc/passwd`)//将文件中每一行赋值给array1数组
[root@zgs ~]# array2=(`ls /root`)
[root@zgs ~]# array3=(harry amy jack "Miss Hou")
[root@zgs ~]# array4=(1 2 3 4 "hello world" [10]=linux)

```

```
# 内容太多删除一部分
[root@zgs ~]# echo ${array1[@]}
(root:x:0:0:root:/root:/bin/bash bin:x:1:1:bin:/bin:/sbin/nologin
...
[root@zgs ~]# echo ${array2[@]}
ab.txt a.txt B.c b.txt c.txt demo.py d.py f1 f2 his.txt h.txt pp.txt pwd res.txt shells t.txt w.sh
[root@zgs ~]# echo ${array3[@]}
harry amy jack Miss Hou
[root@zgs ~]# echo ${array4[@]}
1 2 3 4 hello world linux
[root@zgs ~]# echo ${array4[4]}
hello world
[root@zgs ~]# echo ${array4[5]}
# 注意这种写法，会将数值赋值给第10个索引位置
```

```
[root@zgs ~]# echo ${array4[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:
echo ${array[0]}          // 获取第一个元素
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
```

```
[root@zgs ~]# lt=(1 2 3 11 22 33)
[root@zgs ~]# echo ${lt[0]}
1
[root@zgs ~]# echo ${lt[*]}
1 2 3 11 22 33
[root@zgs ~]# echo ${lt[@]}
1 2 3 11 22 33
[root@zgs ~]# echo ${#lt[*]}

```

```
6
[root@zgs ~]# echo ${!lt[*]}
0 1 2 3 4 5
[root@zgs ~]# echo ${lt[*]:2:4}
3 11 22 33

```

关联数组：

> 可以使用字符串作为数组索引(元素的下标) -- 类似*python*字典

声明关联数组

```
[root@zgs ~]# declare -A array1
[root@zgs ~]# declare -A array2
[root@zgs ~]# declare -A array3
[root@zgs ~]# echo $array3

```

数组赋值

> 数组名[索引|下标]=变量值

- 单赋值

```
[root@zgs ~]# array1[python]=one
[root@zgs ~]# array1[mysql]=two
[root@zgs ~]# array1[linux]=333
[root@zgs ~]# echo $array1

[root@zgs ~]# echo $array1[*]
[*]
[root@zgs ~]# echo ${array1[*]}
333 two one

```

- 多赋值

```
[root@zgs ~]# array2=([name1]=apple [name2]=banana [name3]=orange)
[root@zgs ~]# echo $array2[*]
[*]
[root@zgs ~]# echo ${array2[*]}
orange banana apple
[root@zgs ~]# array3=([num1]=998 [num2]=688 [num3]=299)
[root@zgs ~]# echo ${array3[*]}
998 688 299
[root@zgs ~]# echo ${#array3[*]}
3
[root@zgs ~]# echo ${!array3[*]}
num1 num2 num3

```

五、判断条件(2课时)

运算符

算术运算符: 默认情况下, shell就只能支持简单的整数运算

> 加+ 减- 乘* 除/ %(取余数) **幂
>
> *Bash shell* 的算术运算有四种方式:

> 使用*let* 命令
>
> 使用 *$(( ))*
>
> 使用*$[ ]*
>
> 使用 *expr* 外部程式

```
[root@zgs ~]# n=1
[root@zgs ~]# let n+=1
[root@zgs ~]# echo $n
2
[root@zgs ~]# let n=n+1
[root@zgs ~]# echo $n
3

```

shell用于小数运算；将计算结果以数学结果呈现

```
[root@zgs ~]# echo 1+1.5 | bc
2.5
[root@zgs ~]# echo 1+1.5
1+1.5
[root@zgs ~]# echo 2^10 | bc
1024
[root@zgs ~]# echo "scale=10;1/27" | bc
.0370370370

```

> *i++ 和 ++i* 的理解

- 影响变量

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

```

- 对表达式的值的影响

```
[root@zgs ~]# unset i j
[root@zgs ~]# i=1;j=1
[root@zgs ~]# let x=i++
[root@zgs ~]# let y=++j
[root@zgs ~]# echo $i
2
[root@zgs ~]# echo $j
2
[root@zgs ~]# echo $x
1
[root@zgs ~]# echo $y
2

```

```
[root@zgs ~]# echo $((1+1))
2
[root@zgs ~]# echo $((2*5))
10
[root@zgs ~]# echo $((2/5))
0
[root@zgs ~]# echo $((5/5))
1
[root@zgs ~]# echo $((4/2))
2
[root@zgs ~]# echo $((4-2))
2

```

```
[root@zgs ~]# echo $[2+1]
3
[root@zgs ~]# echo $[2/1]
2
[root@zgs ~]# echo $[2*1]
2
[root@zgs ~]# echo $[2%1]
0
[root@zgs ~]# echo $[2**10]
1024

```

```
# 使用计算符号
[root@zgs ~]# expr 2 + 3
5
[root@zgs ~]# expr 5 \* 2
10
[root@zgs ~]# expr 5 \/ 2
2
# 计算长度
[root@zgs ~]# expr length hello
5
[root@zgs ~]# expr length 541863389
9
# 逻辑判断
[root@zgs ~]# expr 4 > 3
[root@zgs ~]# expr 4 \> 3

```

```
1
[root@zgs ~]# expr 4 \> 6
0

```

> *echo $LANG*
>
> 输出：*en_US.UTF-8*（此时为英文）
>
> *LANG=zh_CN.UTF-8*
>
> *echo $LANG*
>
> 输出：*zh_CN.UTF-8*（此时为中文）

条件判断

语法格式

```
#!/bin/bash

if 条件
then
   命令1
else
   命令2
fi

```

文件判断

> *ll* 命令查看文件类型
>
> 使用*“ls -l”*命令查看文件名，看第一个字符
>
> 开头为*“-”*的是普通文件（如文本文件、二进制文件、压缩文件、图片等）；
>
> 开头为*“d”*的是目录文件（蓝色）；
>
> 开头为*“b”*的是设备文件（块设备），存储设备硬盘、*U*盘、*/dev/sda*、*/dev/sda1*；
>
> *“c”*表示设备文件（字符设备），打印机、终端、*/dev/tty1*、*/dev/zero*；

> “s”表示套接字文件；
>
> “p”表示管道文件；
>
> “l”表示链接文件（浅蓝色）。

- -e 判断文件是否存在, 存在就为真, 否则就为假

  shell对于真假判断的逻辑, 提供了 && 和 || (相当于 python的 and 和 or 逻辑运算符)

  > 条件 A && 条件B 同时成立则成立, 一个不成立则不成立
  >
  > 条件A || 条件B 有一个成立则成立, 都不成立则不成立

```
[root@zgs ~]# test -e f.txt
[root@zgs ~]# echo $?
1
[root@zgs ~]# ls
t1.sh  t2.sh  test.sh
[root@zgs ~]# test -e t1.sh
[root@zgs ~]# echo $?
0
# 注意这里不存在显示1; 存在显示0
```

```
[root@zgs ~]# test -e t1.sh && echo '文件存在则输出'
文件存在则输出
[root@zgs ~]# test -e t123.sh && echo '文件存在则输出'
[root@zgs ~]# test -e t123.sh || echo '文件不存在则输出'
文件不存在则输出
# 符号连用
[root@zgs ~]# test -e f.txt && echo '文件存在'
[root@zgs ~]# test -e f.txt && echo '文件存在' || touch f.txt
[root@zgs ~]# ls
f.txt  t1.sh  t2.sh  test.sh

```

- -d 判断文件是否存在并且是一个目录类型

```
[root@zgs ~]# ls
1.sh  2.sh  3.sh  a.txt  b.txt  c.txt
dir1  dir2  dir3  test.sh  word.txt
[root@zgs ~]# test -d dir1; echo $?
0
[root@zgs ~]# test -d a.txt; echo $?
1

```

- -f 判断文件是否存在并且是一个普通文件类型

```
[root@zgs ~]# test -f 1.sh ; echo $?
0
[root@zgs ~]# test -f a.sh ; echo $?
1
[root@zgs ~]# test -f dir1 ; echo $?
1

```

判断的三个语法写法

1. test -e file 只要文件在条件为真

```
[root@zgs ~]# test -e t1.sh
[root@zgs ~]# echo $?
0
[root@zgs ~]# test -e tt.sh
[root@zgs ~]# echo $?
1

```

2. 使用[判断条件] 判断目录是否存在，存在为真

   注意：[前后有空格]

```
[root@zgs ~]# [ -d ./dir1/ ]; echo $?
0
[root@zgs ~]# [ -f ./dir1/ ]; echo $?
1

```

3. 使用 [[判断条件]] 判断文件是否存在, 存在为真

   注意: [[前后有空格]]

```
[root@zgs ~]# [[ -d ./t1.sh ]]; echo $?
1
[root@zgs ~]# ls
dir1  f.txt  t1.sh  t2.sh  test.sh
[root@zgs ~]# [[ -f ./t1.sh ]]; echo $?
0

```

- -L 判断文件是否存在并且是一个软链接

```
[root@zgs /]# test -L ./root; echo $?
1
[root@zgs /]# test -L ./bin; echo $?
0
[root@zgs /]# test -L ./sbin; echo $?
0
[root@zgs /]# test -L ./etc; echo $?
1

```

- -b　　#判断文件是否存在并且是一个设备文件

```
[root@zgs dev]# [ -b ./vda ]; echo $?
0
[root@zgs dev]# [ -b ./ssh ]; echo $?
1
[root@zgs dev]# [ -b ./snd ]; echo $?
1

```

- -S　　#判断文件是否存在并且是一个套接字文件(跟网络编程有关scoket)

```
[root@zgs run]# [[ -S /run/docker.sock ]]; echo $?
0
[root@zgs run]# [[ -S /run/docker ]]; echo $?
1

```

- -c　　#判断文件是否存在并且是一个字符设备文件

```
[root@zgs dev]# [ -c ./zero ]; echo $?
0
[root@zgs dev]# [ -c ./tty ]; echo $?
0
[root@zgs dev]# [ -c ./vfio/ ]; echo $?
1
[root@zgs dev]#

```

- -p　　#判断文件是否存在并且是一个命名管道文件
- -s　判断文件(包含目录文件)是否有内容, 非空文件条件满足

  -s 表示非空; 文件存在且有内容

！ -s 表示空文件;文件没有内容或者文件不存在都是空文件

```
[root@zgs ~]# test -s f.txt; echo $?
1
[root@zgs ~]# test -s t1.sh; echo $?
0
[root@zgs ~]# test -s t2.sh; echo $?
0
[root@zgs ~]# test ! -s t2.sh; echo $?
1
[root@zgs ~]# test ! -s f.txt; echo $?
0

```

文件权限判断

> 讲到这里咱们需要回顾课本中的权限管理;了解高级权限

```
[root@zgs ~]# cd /home/
[root@zgs home]# ls
zgs
[root@zgs home]# touch f
[root@zgs home]# ll
total 4


```

```
-rw-r--r-- 1 root root    0 Nov  6 23:22 f
drwx------ 4 zgs  zgs  4096 Oct 28 09:41 zgs
[root@zgs home]# chmod 700 f
[root@zgs home]# ll
total 4
-rwx------ 1 root root    0 Nov  6 23:22 f
drwx------ 4 zgs  zgs  4096 Oct 28 09:41 zgs

```

- -r 当前用户对其是否可读

```
[root@zgs home]# test -r f; echo $?
0
[root@zgs home]# su zgs
[zgs@zgs home]$ ll
total 4
-rwx------ 1 root root    0 Nov  6 23:22 f
drwx------ 4 zgs  zgs  4096 Oct 28 09:41 zgs
[zgs@zgs home]$ test -r f; echo $?
1


```

- -w 当前用户对其是否可写

```
[zgs@zgs home]$ test -w f; echo $?
1
[zgs@zgs home]$ exit
exit
[root@zgs home]# test -w f; echo $?
0

```

- -x 当前用户对其是否可执行

```
[root@zgs home]# test -x f; echo $?
0
[root@zgs home]# su zgs
[zgs@zgs home]$ test -x f; echo $?
1

```

- -u 是否有suid

```
[zgs@zgs home]$ test -u f; echo $?
1
[zgs@zgs home]$ ll
total 4
-rwx------ 1 root root    0 Nov  6 23:22 f
drwx------ 4 zgs  zgs  4096 Oct 28 09:41 zgs
[zgs@zgs home]$ test -g f; echo $?
1
[zgs@zgs home]$ exit
exit
[root@zgs home]# test -u f; echo $?
1
[root@zgs home]# test -g f; echo $?
1

```

- -g　是否sgid

```
[zgs@zgs home]$ test -u f; echo $?
1
[zgs@zgs home]$ ll
total 4
-rwx------ 1 root root    0 Nov  6 23:22 f
drwx------ 4 zgs  zgs  4096 Oct 28 09:41 zgs
[zgs@zgs home]$ test -g f; echo $?
1
[zgs@zgs home]$ exit
exit
[root@zgs home]# test -u f; echo $?
1
[root@zgs home]# test -h f; echo $?
1
[root@zgs home]# test -g f; echo $?
1

```

- -k　是否有t位

```
[root@zgs home]# test -t zgs; echo $?
1
[root@zgs home]# test -t f; echo $?
1

```

文件比较

- a -nt b 比较a文件是否比b文件新
- a -ot b 比较a文件是否比b文件旧
- a -ef b 比较是否为同一个文件

```
[root@zgs ~]# [ a.txt -nt b.txt ]; echo $?
1
[root@zgs ~]# [ a.txt -ot b.txt ]; echo $?
1
[root@zgs ~]# [ a.txt -ef b.txt ]; echo $?
1
[root@zgs ~]# touch c.txt
[root@zgs ~]# [ c.txt -nt b.txt ]; echo $?
0
[root@zgs ~]# [ c.txt -ot b.txt ]; echo $?
```

```
1
[root@zgs ~]# [ a.txt -ot c.txt ]; echo $?
0
[root@zgs ~]# [ c.txt -ef c.txt ]; echo $?
0

```

判断数字

- -eq 相等

```
[root@zgs ~]# [ 2 -eq 3 ]; echo $?
1
[root@zgs ~]# [ 2 -eq 2 ]; echo $?
0
[root@zgs ~]# [ 2 -eq 1 ]; echo $?
1

```

- -ne 不等

```
[root@zgs ~]# [[ 123 -ne 123 ]]; echo $?
1
[root@zgs ~]# [[ 123 -ne 12 ]]; echo $?
0

```

- -gt 大于

```
[root@zgs /]# [ 2 -gt 3 ]; echo $?
1
[root@zgs /]# [ 3 -gt 3 ]; echo $?
1
[root@zgs /]# [ 4 -gt 3 ]; echo $?
0

```

- -lt 小于

```
[root@zgs /]# [ 4 -lt 3 ]; echo $?
1
[root@zgs /]# [ 3 -lt 3 ]; echo $?
1
[root@zgs /]# [ 2 -lt 3 ]; echo $?
0

```

- -ge 大于等于

```
[root@zgs /]# [ 3 -ge 3 ]; echo $?
0
[root@zgs /]# [ 4 -ge 3 ]; echo $?
0
[root@zgs /]# [ 1 -ge 3 ]; echo $?
1

```

- -le 小于等于

```
[root@zgs /]# [ 1 -le 3 ]; echo $?
0
[root@zgs /]# [ 3 -le 3 ]; echo $?
0
[root@zgs /]# [ 4 -le 3 ]; echo $?
1

```

判断字符串

- -z 是否为空字符串;字符串长度为0成立

```
[root@zgs ~]# test -z "";echo $?
0
[root@zgs ~]# test -z " ";echo $?
1
[root@zgs ~]# test -z "123";echo $?
1
[root@zgs ~]# test -z "abc";echo $?
1

```

- -n 是否为非空字符串;值字符串非空就成立

```
[root@zgs ~]# test -n "abc";echo $?
0
[root@zgs ~]# test -n "";echo $?
1
[root@zgs ~]# test -n " ";echo $?
0

```

- = 判断是否相等

```
[root@zgs ~]# test 'abc' = 'abc'; echo $?
0
[root@zgs ~]# test '123' = '132'; echo $?
1
[root@zgs ~]# test 123 = 132; echo $?
1
[root@zgs ~]# test 123 = 123; echo $?
0


```

- !=判断是否不相等

```
[root@zgs ~]# test 123 != 123; echo $?
1
[root@zgs ~]# test 122 != 123; echo $?
0
[root@zgs ~]# test 'abc' != 123; echo $?
0
[root@zgs ~]# test 'abc' != abc; echo $?
1
[root@zgs ~]# test 'abc' = abc; echo $?
0

```

> 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

```
[root@zgs ~]# [ 1 -lt 2 ] && [ a = 'a' ]; echo $?
0
[root@zgs ~]# [ 1 -lt 2 ] && [ a = 'a' ]; echo $?
0
[root@zgs ~]# [ 1 -lt 2 ] || [ a = 'a' ]; echo $?
0
[root@zgs ~]# [ 1 -lt 2 ] || [ a = 'abc' ]; echo $?
0
[root@zgs ~]# [ 1 -lt 2 ] && [ a = 'abc' ]; echo $?
1

```

```
[root@zgs ~]# [ $a \> 8 -a $a \< 11 ];
echo $?
1
[root@zgs ~]# [ $a -gt 8 -a $a \< 11 ];
echo $?
0
[root@zgs ~]# [ $a -gt 8 -o $a \< 11 ];
echo $?
0
[root@zgs ~]# [ $a -lt 8 -o $a \< 11 ];
echo $?
0
[root@zgs ~]# [ $a -lt 8 -o $a -gt 11 ];
echo $?
1

```

逻辑符号总结：

1. && ||都可以用来分割命令或者表达式
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

综合使用

数值比较

> *id -u id命令查看当前用户的id, uid , gid; -u只看uid号码*

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

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

```
[zgs@localhost root]$ a=123
[zgs@localhost root]$ [ $a = 123 ];echo $?
0
[zgs@localhost root]$ [ "$a" = 123 ];echo $?
0
[zgs@localhost root]$ [ "${a}" = 123 ];echo $?
0
```

```
[zgs@localhost root]$ [ "${a:1:2}" = 123 ];echo $?
1
[zgs@localhost root]$ echo ${a:1:2}
23
[zgs@localhost root]$ h="a b c"
[zgs@localhost root]$ test $h == "abc"; echo $?
bash: test: 参数太多
2
[zgs@localhost root]$ test "$h" == "abc"; echo $?
1

```

单方括号[]与双方括号[[]] 区别

```
# 如果在 [] 使用 > <  需要使用 \ 取消转义(重定义的意思)
# 如果在 [[]] 使用 > < 可以正常使用
[root@localhost ~]# [ 1 > 2 ]; echo $?
0
[root@localhost ~]# [ 1 \> 2 ]; echo $?
1
[root@localhost ~]# ls
1.sh  2  2.sh  3.sh  4.sh  files  f.txt
[root@localhost ~]# [[ 1 > 4 ]]; echo $?

```

```
1
[root@localhost ~]# ls
1.sh  2  2.sh  3.sh  4.sh  files  f.txt
# [] 里面的变量有空格 需要使用双引号; [[]] 里面变量有空格不需要使用双引号
[root@localhost ~]# hw='hello word'
[root@localhost ~]# [ "hello word" = $hw ]; echo $?
-bash: [: 参数太多
2
[root@localhost ~]# [[ "hello word" = $hw ]]; echo $?
0
[root@localhost ~]# [ "hello word" = "$hw" ]; echo $?
0

```

六、判断语句(2课时)

语法结构

```
#!/bin/bash

if 条件
then
  命令1
else
  命令2
fi
```

> *shell*的判断语法结构及嵌套与*python*极其相似, 可以类比学习

1. 判断以if 开头, fi结尾
2. [条件判断] 例如上边的test命令判断, 空格隔开
3. if 语句后接 then 需要在其后添加 ; elif 后边也是接then

```
[root@zgs ~]# vim 1.sh
[root@zgs ~]# cat 1.sh
#!/bin/bash
a=14
b=14
if [ $a -eq $b ]
  then
    echo 'a等于b'
  else
    echo 'a不等b'
```

```
fi
[root@zgs ~]# sh 1.sh
a等于b

```

shell 中if的结构语法分三类:

- 单分支

```
if  [ 条件表达式 ]；then
            命令
fi
```

```
age=22
if [ $age -gt 18 ];then
echo 你已成年
fi
```

- 双分支

```
if [ 条件表达式 ] ; then
    命令
else
    命令
fi
```

```
sum=88
if [ $sum -gt 80 ];then
    echo 优秀
else
    echo 中等
fi

```

- 多分支

```
if [ 条件表达式 ]; then
    命令
elif [ 条件表达式 ]; then
   命令
else
   命令
fi

```

```
time=8
if [ $time -gt 18 ];then
    echo 傍晚
elif [ $time -gt 12 ];then
    echo 下午
elif [ $time -gt 7 ];then
    echo 上午
else
    echo 清晨
fi

```

判断案例

1. 案例一 文件内容判断

判断这个文本word.txt里面是否存在有字母 I

```
[root@zgs ~]# vim word.txt
[root@zgs ~]# cat word.txt

how are you
fine think you
are you ok
I am ok
123312
```

```
v
t
y
x
w


```

```
[root@zgs ~]# cat 2.sh
#!/bin/bash
# grep筛选文本; &> 内容重定向, 无论输出是否正确都扔进黑洞(/dev/null)
grep 'I' /root/word.txt &>/dev/null
# 将结果保存给变量; 或者不必
res=$?
if [ $res -eq 0 ];then
    echo '存在字母I'
else
    echo '不存在字母I'
fi

[root@zgs ~]# source 2.sh
存在字母I

```

练习: 判断 /etc/passwd 中第一行是否存在root

```
#!/bin/bash

# 1. 筛选判断文件passwd中第一行是否以root开头
head -1 /etc/passwd | grep '^root' &>/dev/null
echo 命令判断的结果: $?

if test $? -eq 0
then
        echo 'passwd文件的第一行存在 root'
else
        echo '不存在!'
fi
```

2. 案例二 判断文件存在于权限

检查用户root目录中的 test.sh 文件是否存在，并且检查是否有执行权限

```
#!/bin/bash

if [ -f ./1.sh ]
then
  echo 该普通文件存在
  # 判断是否有执行权限
  if test -x ./1.sh; then
    echo 当前用户有可执行权限
```

```
  else
    echo 当前用户没有执行权限
  fi
else
  echo 文件不存在当前目录
fi

```

结果如下:

```
[root@zgs ~]# ls
1.sh  2.sh  a.txt  b.txt  c.txt  dir1
dir2  dir3
[root@zgs ~]# sh 2.sh
该普通文件存在
当前用户有可执行权限

```

3.案例三 成绩判断

提示用户输入本次考试成绩（百分制），要求判断分数大于等于80分的进入下一阶段竞赛，其余同学都淘汰，进入竞赛的同学，再进一步判断学科类别，java组和shell组，如果输入错误则提示错误

```
read -p "请输入您本次的考试成绩：" second

```

```
if [ $second -ge 80 -a $second -le 100 ];then
    echo "恭喜你进入竞赛"
    read -p "请输入你的学科: " dev
    if [ $dev = java ];then
        echo "你将进入java学科比赛"
    elif [ $dev = python ];then
        echo "你将进入python学科比赛"
    else
        echo "输入有误"
   fi
elif [ $second -lt 80 ];then
    echo "淘汰"
else
    echo "输入错误"
fi
```

```
#!/bin/bash
read -n3 -p "请输入您的考试成绩" mark
if [ $mark -ge 80 ] && [ $mark -le 100 ];then
    echo 开始竞赛
    read -p '输入你的学科: ' obj
    if test $obj == "python"; then
    echo 进入python赛场
    elif test $obj == "shell"; then
    echo 进入shell赛场
```

```
    else
    echo "进入其他赛场"
    fi
elif [$mark -lt 80]; then
    echo "淘汰"
else
    echo "输入错误"
fi

[root@zgs ~]# ls
1.sh  2.sh  3.sh  a.txt  b.txt  c.txt
dir1  dir2  dir3
[root@zgs ~]# chmod +x 3.sh
[root@zgs ~]# ./3.sh
请输入您的考试成绩99
开始竞赛
输入你的学科: shell
进入shell赛场

```

4.案例四 判断交互内容

用户输入内容, 使用if判断内容是否为整数

```
#!/bin/bash

read -p "请输入一个整数:" num
```

```
//判断输入是否为0，
if [ $num -eq 0  ]; then
    echo "$num是整数"
else
    expr $num + 0 &> /dev/null
    //判断是否为整数
    if [ $? -eq 0 ];then
        echo "您输入的$num 是一个整数"
    else
        echo "您输入的$num 不是一个整数"
    fi
    //判断奇偶性
    expr ${num} % 2  &> /dev/null
    if [ $? -ne 0 ];then
        echo "您输入的$num 是一个偶数"
    else
        echo "您输入的$num 是一个奇数"
    fi
fi
```

```
#!/bin/bash
read -p '输入一个整数: ' num
if [ $num -eq 0 ]; then
    echo "$num 是整数"
else
    expr $num + 0 &> /dev/null
    #判断是否为整数
```

```
    if [ $? -eq 0 ]; then
        echo "输入的$num 是整数"
    else
        echo "输入的$num 不是整数"
    fi
    #判断奇偶性
    expr ${num} % 2 &> /dev/null
    if [ $? -ne 0 ];then
        echo 输入的 $num 是偶数
    else
        echo 输入的 $mum 是奇数
    fi
fi
```

结果

```
[root@zgs ~]# ls
1.sh  2.sh  3.sh  4.sh  a.txt  b.txt
c.txt  dir1  dir2  dir3
[root@zgs ~]# chmod +x 4.sh
[root@zgs ~]# ls
1.sh  2.sh  3.sh  4.sh  a.txt  b.txt
c.txt  dir1  dir2  dir3
[root@zgs ~]# ./4.sh
输入一个整数: 11
输入的11 是整数
输入的  是奇数

```

5.案例五 判断进程是否存在

手动输入一个进程名, 判断进程是否存在

> *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	说明
-d	定义多个进程之间的分隔符, 如果不定义则使用换行符。
-P	根据父进程PID，找出所有字进程的pid
-n	表示如果该程序有多个进程正在运行，则仅查找最新的，即最后启动的。

选项	说明
-o	表示如果该程序有多个进程正在运行，则仅查找最老的，即最先启动的（多个进程时即父进程PID）。
-G	其后跟着一组group id，该命令在搜索时，仅考虑group列表中的进程。
-u	其后跟着一组有效用户ID(effetive user id)，该命令在搜索时，仅考虑该effective user列表中的进程。
-U	其后跟着一组实际用户ID(real user id)，该命令在搜索时，仅考虑该real user列表中的进程。
-x	表示进程的名字必须完全匹配, 以上的选项均可以部分匹配。
-l	将不仅打印pid,也打印进程名。
-f	一般与-l合用, 将打印进程的参数。

```
# 定义变量
read -p "请输入需要判断的进程名: " process
# 通过命令来查看进程是否存在
pgrep $process &>/dev/null
# 通过命令执行的状态来判断是否存在
if [ $? -eq 0 ];then
    echo "进程$process存在"
else
    echo "进程$process不存在"
fi

```

6.案例六 判断用户是否存在

输入一个用户，用脚本判断该用户是否存在;并且判断是超级用户(0)系统用户(1-999)普通用户(>=1000)

- 脚本

```
#!/bin/bash

read -n5 -p'输入用户的名字: ' user

id $user &> /dev/null
if [ $? == 0 ]; then
    # 账户存在判断是哪种用户
    echo "$user 存在!"
    uid=`id -u $user` &>/dev/null
    if [ $uid == 0 ];then
        echo "$user 是超级用户"
    elif [ $uid -ge 1 ] && [ $uid -lt 1000 ]; then
        echo "$user 是系统用户"
    else
        echo "$user 是普通用户"
    fi
else
    echo "$user 不存在!"
fi

```

- 结果

```
[root@zgs ~]# ls
1.sh  2.sh  3.sh  4.sh  5.sh  a.txt
b.txt  c.txt  dir1  dir2  dir3
[root@zgs ~]# chmod +x 5.sh
[root@zgs ~]# ls
1.sh  2.sh  3.sh  4.sh  5.sh  a.txt
b.txt  c.txt  dir1  dir2  dir3
[root@zgs ~]# vim 5.sh
[root@zgs ~]# sh 5.sh
输入用户的名字: root
root 存在!
root 是超级用户
[root@zgs ~]# sh 5.sh
输入用户的名字: zgs
zgs 存在!
zgs 是普通用户
[root@zgs ~]# sh 5.sh
输入用户的名字: adm
adm 存在!
adm 是系统用户
[root@zgs ~]# sh 5.sh
输入用户的名字: ming
ming 不存在!

```

7.案例七 判断内核版本

判断当前内核主版本是否大于2，如果满足则输出当前内核版本，不满足则提示

```
uname -r
echo $kernel|cut -d. -f1
```

- 脚本

```
#!/bin/bash

kernel=`uname -r | cut -d. -f1`

if [ $kernel \> 2 ];then
    echo 当前版本 `uname -r`
else
    echo 当前版本小于等于2
fi

```

- 结果

```
[root@zgs ~]# ls
1.sh  2.sh  3.sh  4.sh  5.sh  6.sh  a.txt
b.txt  c.txt  dir1  dir2  dir3
[root@zgs ~]# chmod +x 6.sh
[root@zgs ~]# ls
1.sh  2.sh  3.sh  4.sh  5.sh  6.sh  a.txt
b.txt  c.txt  dir1  dir2  dir3
[root@zgs ~]# ./6.sh
当前版本 3.10.0-1160.62.1.el7.x86_64

```

七、循环语句(2课时)

语法结构

- 列表循环结构

```
for variable in {list}
     do
          command
          command
          …
     done
或者
for variable in a b c
     do
          command
          command
     done
```

- 循环写法

```
#!/bin/bash

for i in {1,2,3}
do
    echo $i
done

for j in a b c d
do
    echo $j
done

```

- 带点点写法

```
#!/bin/bash

for v in {1..10}
do
    echo $v
done

```

- 使用seq写法

```
#!/bin/bash

for a in `seq 6`
do
    echo $a
done

for c in $(seq 5)
do
    echo $c
done

```

- 步长使用

```
#!/bin/bash

for x in {0..10..2}
do
    echo $x
done

for y in {2..6..2}
do
    echo $y
done

```

- 降序使用

```
#!/bin/bash

for i in {10..1}
do
    echo $i
done

for j in {10..1..-1}
do
    echo $j
done

```

- seq降序

```
#!/bin/bash

for s in `seq 10 -2 1`
do
    echo $s
done

```

- 不带列表循环

```
# 不带列表的for循环执行时由用户指定参数和参数的个数，下面给出了不带列表的for循环的基本格式：
for variable
do
    command
    command
    …
done
```

```
#!/bin/bash

for var
do
    echo $var
done

echo "脚本后参数: $@ 共$#个"

```

- c语言风格写法

```
for(( expr1;expr2;expr3 ))
    do
        command
        command
        …
    done
for (( i=1;i<=5;i++))
    do
        echo $i
    done


expr1：定义变量并赋初值
expr2：决定是否进行循环（条件）
expr3：决定循环变量如何改变，决定循环什么时候退出
```

```
#!/bin/bash

for ((i=1;i<=10;i++))
do
    echo $i
done

for ((i=1;i<=10;i+=2))
do
    echo $i
done

```

循环案例

案例计算奇数和

计算1到100的奇数之和，方法不止一种

> 思路：
>
> 1. 定义一个变量来保存奇数的和 *sum=0*
> 2. 找出*1-100*的奇数，保存到另一个变量里 *i*
> 3. 从*1-100*中找出奇数后，再相加，然后将和赋值给*sum*变量
> 4. 遍历完毕后，将*sum*的值打印出来

```bash
#!/bin/bash
#定义一个变量来保存奇数的和
sum=0
#打印1-100的奇数并且相加重新赋值给sum
for i in {1..100..2}
do
    sum=$[ $i + $sum ]
done

#打印1-100的奇数和
echo "1-100的奇数和为:$sum"
```

```bash
#!/bin/bash
#定义一个变量来保存奇数的和
sum=0
#打印1-100的奇数并且相加重新赋值给sum
for (( i=1;i<=100;i+=2))
do
    let sum=sum+$i
    或者
    let sum=sum+i
    或者
    let sum=$sum+$i
done
#打印1-100的奇数和
echo "1-100的奇数和为:$sum"
```

```bash
#!/bin/bash
sum=0
for ((i=1;i<=100;i++))
do
    if [ $[$i%2] -ne 0 ];then
    let sum=sum+$i
    fi
done
echo "1-100的奇数和是:$sum"
```

```bash
#!/bin/bash
sum=0
for ((i=1;i<=100;i++))
do
    [ $[$i%2] -eq 0 ] && true  || let sum=sum+$i
done
echo "1-100的奇数和是:$sum"
```

循环控制：

循环体： do....done之间的内容

- continue：继续；表示循环体内下面的代码不执行，重新开始下一次循环

- break：打断；马上停止执行本次循环，执行循环体后面的代码
- exit：表示直接跳出程序

```
#!/bin/bash

for i in {1..6}
do
# 判断当循环到2执行break, 否则执行 touch命令
    test $i -eq 4 && break || touch ./mysh/$i.sh
done

```

案例判断质数

输入一个正整数,判断是否为质数(素数）
质数：只能被1和它本身整除的数叫质数。

> 思路：
>
> 0、让用户输入一个数，保存到一个变量里 *read num*
>
> 1、如果能被其他数整除就不是质数——>*numi* 是否等于0 $i = 2$ *num-1*

> 2、如果输入的数是1或者2取模根据上面判断又不符合，所以先排除1和2
>
> 3、测试序列从2开始，输入的数是4——>得出结果num不能和i相等，并且num不能小于 i