230704 c高级

1. linux 的发展

```
贝尔实验室 、麻省理工 --------> multics 计划     失败
 汤姆森.肯     ----> unics
  ---> 贝尔实验室推广 ----> unix{ sys 5    BSD}

莱纳斯.托瓦斯  ----> 类unix ---> linux

linux 发行版本:
   ubuntu ----> “人道主义”

 GUN & GPL
    GUN is not linux
 GPL 通用协议

```

2. linux 的体系结构

```
应用层  ----------命令、程序、shell 脚本------
                    |
                    |shell   \   系统调用
                    |/
内核   ---------------------------------
   内核功能:
         文件管理
         设备管理
         进程管理
         内存管理
         网络管理
硬件 ------------------------------------
          摄像头、鼠标。。。。
内核:
  内核是Linux系统的最底层，提供了系统的核心功能并允许进程以一种有序
  的方式访问硬件。
  用于控制进程、输入、输出设备、文件系统操作、管理内存;

linux 系统支持多任务、多用户的模式运行;

```

3.shell 命令

```
shell是一种命令行解释器，
shell --> "贝壳" 对操作系统起到保护的作用;
用户 和 内核的交互:
  用户在命令行提示符下键入命令文本，开始与Shell进行交互。
  接着，Shell将用户的命令或按键转化成内核所能够理解的指令
  控制操作系统做出响应，直到控制相关硬件设备。
  然后，Shell将输出结果通过Shell提交给用户。

复习常见的shell命令:
ls 、 cd 、 pwd 、whoami
  ls 选项 参数
   选项:
        -a: 查看以点开头的隐藏文件;
        -l:查看文件的详细属性;
        -R: 递归查询
        -i: 查询inode号
eg:
  -rw-rw-r--  1 jiangcx jiangcx  903 Jul 23 22:48 te1.c
    -: 文件的类型
     linux 下的七大文件类型:（linux 一切皆文件）
      - 普通文件: XXX.c XXX.h XXX.txt
      d 目录文件: 相当于windows的文件夹
      b 块设备文件:磁盘(/dev/sda ...)
      c 字符设备文件:摄像头("/dev/video0")、串口(/dev/ttyUSB0)
      p 管道文件:(mkfifo fifo)用于本地进程间的通信;
      l 链接文件:（软链接:相当于windows的快捷方式）
      s 套接字文件: （socket）用于网络间通信;
   rw- rw- r--: 文件的权限 （0664）
      文件的三种权限:
        r --> 读权限  4
        w --> 写权限  2
        x --> 执行权限 1
   文件的权限包含:
          用户的权限
          组的权限
          其他权限
   一个权限可以用八进制表示:
        一个文件的最高权限: 0777
  1: 硬链接数
  jiangcx:用户名
  jiangcx:组名
  903: 文件的大小
  Jul 23 22:48:  文件最后修改的时间
  te1.c 文件名

 路径:
    相对路径:从当前出发的一条连续不断的路径;
    绝对路径:从根目录出发的一条连续不断的路径;
```

3.2 cd

```
cd 目录的路径 去到某个目录下
cd (cd ~) 去到家目录（/home/hqyj）
cd / 去到根目录
cd .. 返回上一级
cd - 返回上一步;
```

3.3 目录的创建

```
mkdir dirname
mkdir -p demo1/demo2/demo3 递归创建(创建多级目录)

删除:
  rm -r dirname 删除一个目录
  rmdir dirname 删除一个空目录
```

3.4 文件

```
文件的创建:
    ①touch filename 文件存则更新时间戳，不存在则创建一个空文件;
    ② vi filename 文件存在则打开不存在则创建;
文件的删除:
    rm filename
文件的查看:
    cat filename 文件的查看
    cat -n filename 查看文件自带行号
vi编辑器:
   三种模式:
    命令行模式:刚打开文件进入的模式;
      在任意模式下按 Esc 切换到命令行模式:
      nyy --> n代表行数 复制多行
      ndd --> 剪切多行
      p  --> 粘贴
      u  --> 撤销
      ctrl + r 反向撤销

      gg=G 代码缩进对齐

      /str 查找
        n:向下查找
        N:向上查找

      1G: 光标去到第一行
      G: 光标去到最后一行
    插入模式:
      在命令行模式的基础上，按 a / i / o;
    底行模式:
      在命令行模式的基础上，输入 : （shift + :）
        w: 保存
```

```
        q: 退出
        wq:保存并退出
        q!: 强制退出
        wq!:强制保存并退出

        设置行号: set number
        取消行号: set nonumber

        :num 去到第num行

        vsp filename 水平分屏
         ctrl + w + w 切换光标到分屏;

        替换:
        . ---> 当前
        $ ---> 末尾
        g ---> 每行的所有
        % 全文

        范围 s/str1/str2

        1, $ s/str1/str2
        //将第一行到最后一行的每行第一个str1替换为str2;
        1, $ s/str1/str2/g
        //将第一行到最后一行的所有str1替换为str2

        块复制，块剪切
          范围 y
          范围 d
          eg:
            . , $ y 复制当前到最后一行


```

3.5 文件的复制

```
cp 文件的路径1 文件的路径2
cp 文件的路径1 目录的路径2
   eg:
      cp test.c ../ 将test.c 拷贝到上一级的目录
      cp test.c ../my.c 将test.c 拷贝到上一级并重命名为my.c

cp -a 目录的路径1 目录的路径2
      cp demo1 demo2
```

3.6 文件移动

```
mv 路径1  路径2
  mv test.c my.c  重命名
```

3.7 其他常用命令

```
clear 清屏
exit 退出终端
su 切换用户
sudo 临时超级用户权限
diff
    diff file1 file2 比较两个文件是否一样
eog 图片查看器
   eog 1.jpg
history 查看历史输入记录
```

3.8 快捷方式

```
创建终端:
  ctrl + shift + n
终端的切换:
  Alt + Tab
Tab:
  按一下Tab 补全内容
  按两下: 显示待输入的命令
ctrl + l 清屏
crtl + c 结束
↑:向上查看历史输入记录
↓:向下查看历史输入记录
```

4. 软件包的管理

4.1 dpkg 离线安装

```
软件包名的构成:
sl   _5.02-1          _amd64      .deb
软件名 版本号   修订版本号    体系架构      后缀
                            i386: 32位
                            amd64(x64): 64位
运行软件: sl (软件名)

软件安装:
  sudo dpkg -i 软件包名
卸载:
   sudo dpkg -r 软件名
完全卸载:
  sudo dpkg -P 软件名
查看软件版本号:
  sudo dpkg -l 软件名
查看软件包的安装清单
  sudo dpkg -L 软件名
查看软件安装的状态
  sudo dpkg -s 软件名
```

4.2 apt 在线安装

```
软件源：
    /etc/apt/sources.list （服务器镜像地址）
 创建服务器的索引（清楚镜像地址上有哪些软件资源）
  sudo apt-get update
  软件包的下载缓存地址：
    /var/cache/apt/archives
  软件的安装：
    sudo apt-get install 软件名;
  软件的卸载：
      sudo apt-get remove 软件名
  清空软件包
      sudo apt-get clean
  检查系统中依赖关系的完整性
      sudo apt-get check
  将系统中的所有包升级到最新版本
      sudo apt-get upgrade
  重新安装：
    sudo apt-get  --reinstall install 软件名
  完全卸载：
    sudo apt-get --purge remove 软件名
 只下载不安装：
     sudo apt-get -d install 软件名
 获取软件包的安装状态：
     sudo apt-cache policy 软件名

2.配置网络 ---> 桥接模式
      ping www.baidu.com

```

day2

1.文件互传（windows 和ubuntu）

1.1 重装安装vm-tools

```
1.确保有安装包
    VMwareTools-10.3.22-15902021.tar.gz
   设置 ---> 重新安装Vmware-tools ,点击就可以找到安装包;
2.解压：
    sudo tar -xvf VMwareTools-10.3.22-15902021.tar.gz
   会生成一个目录： vmware-tools-distrib
3.cd vmware-tools-distrib
4.sudo ./vmware-install.pl
   一直回车，如果在安装的过程中出错(.....[No]),就把相应已经存在的
  文件删除(新建终端删除)，接着安装完然后再重复第4步重新安装;
```

```
重新安装Vm --- tools 是灰色的解决办法:
  1.将虚拟机关机 ---> 找到设置 -->CD/DvD -->选择使用ISO映像
    文件；去Vmware的安装路径下浏览，找到linux.iso选择；
  2.打开虚拟机 ---> CD/DVD 里面就有Vmware的安装包；
```

1.2 安装共享文件夹

```
1.将虚拟机关机 --> 设置 --> 选项 ---> 共享文件夹 --> 总是启用
    ----> 添加 --> 主机路径: windows的路径 --->
      设置在ubuntu 上此文件夹的名称 --> 应用;
2.打开虚拟机，去到 /mnt/hgfs里面就能找到共享文件夹；

 1.可以将文件拷贝到共享文件夹，那么两边系统都能看的；
 2.可以将windows（ubuntu）上的文件直接鼠标复制，
   去到ubuntu(windows)的图形界面 鼠标粘贴;
 3.对于普通文件从windows传到ubuntu，可以拷贝然后到ubuntu的终端，
    右键粘贴文件名，使用命令mv 拷贝；
     eg:
       mv '/tmp/VMwareDnD/6RuSze/非递归先序.png' ./
```

2.文件相关的命令

2.1 文件的压缩

```
在对文件压缩的时候，压缩文件会把源文件覆盖，解压也是一样;
.gz: 压缩率最低，压缩速度最快
    压缩:
       gzip file.txt ---> file.txt.gz
    解压:
       gunzip file.txt.gz ---> file.txt
.bz2: 压缩率次之， 压缩速度次之
  压缩:
    bzip2 file.txt ---> file.txt.bz2
  解压:
    bunzip2 file.txt.bz2 ----> file.txt
.xz: 压缩率最高，压缩速度最慢
  压缩:
       xz file.txt ---> file.txt.xz
  解压:
       unxz file.txt.xz ---> file.txt
```

2.3 文件的归档

```
归档:  对批量文件进行打包，归档的对象是目录;
 归档之后源文件依旧在，生成新的归档文件;
```

```
 拆包之后，归档的文件依旧在，会重新生成源文件;
归档、压缩：
   tar 选项 归档后的文件名 目录
拆包、解压：
   tar 选项 归档后的文件名
   选项:
      归档并压缩:
        -c 创建归档(打包)
        -v 显示归档的过程
        -f f后面固定跟随归档后的文件名
        -z 归档并压缩为.gz的文件
        -j 归档并压缩为 .bz2的文件
        -J 归档并压缩为 .xz的文件
        注意:选项中f放在最后，其他选项位置随意;
     拆包并解压:
       -x 创建归档(打包)
        -v 显示归档的过程
        -f f后面固定跟随归档后的文件名
        -z 拆包并解压.gz的文件
        -j 拆包并解压 .bz2的文件
        -J 拆包并解压 .xz的文件

       万能拆包：
          tar -xvf 归档后的文件名;
```

2.4 文件查看

```
cat filename 查看文件的内容
cat -n filename 查看的内容会自带行号;

head -num filename 查看文件的前num行;
tail -num filename 查看文件末尾的num行;

more / less 以一种比例的方式查看文件，适合较大的文件查看;

file filename 查看文件类型;
```

2.5 重定向和追加

```
>:
 重定向
   eg:
      cat /etc/passwd > test.txt
      注意: test.txt 不存在则会创建，并把/etc/passwd里面的内容
        重定向到test.txt中，如果test.txt存在，那里面的所有内容会
        被替换;
>>: 追加
  eg:
    cat /etc/passwd >> test.txt
```

```
    注意：test.txt 不存在则会创建，并把/etc/passwd里面的内容
        追加到test.txt中，如果test.txt存在，也会将内容从文件的
        末尾追加写入；
 eg：
    将/etc/passwd 里的第45行显示在终端上；
  方法1:
    head -45 /etc/passwd > test.txt
    tail -1 test.txt
  方法2：
      head -45 /etc/passwd | tail -1
| 管道:
  将前面的结果作为后面的输入；

路径：
    绝对路径：从根目录出发的一条连续不断的路径
    相对路径：从当前出发的一条连续不断的路径
```

2.6 在文件中搜索字符串

```
grep "string" 文件名 选项
  -n 搜索的结果待行号
  -i 不区分大小写
  -R 递归搜索，给个目录，搜索整个目录下的所有文件
  -w 搜索更加严谨，字符串的前后有其他字符就搜索不到；
选项可以多个一起用；
    eg:
      grep "str" test.txt -ni
        表示是在test.txt 中搜索 str 这个串，不区分大小写以及
        搜索结果显示行号
      grep "^str" ./ -niR
        表示：在当前目录的所有文件中搜索以str开头的串,不区分大小写
        以及搜索结果显示行号
      grep "str$" test.txt -n
        表示搜索以str结尾的串，结果显示行号；
      grep "str" test.txt -wni
       表示：搜索 str前后没有其他字符的串，不区分大小写以及搜索结果
         显示行号
     grep "arr\[5\]" ./ -nR
        表示搜索数组 arr[5];

```

2.7 搜索文件

```
find 搜索路径 -name filename
```

2.8 字符的提取

```
cut
  eg:
    cut -d ':' -f 1,5 /etc/passwd
     : 是分隔符
     1,5 提取的位置，第一位置和第5位置;
     /etc/passwd 文件名
eg:
①  提取当前用户的组id 和用户id // 1000:1000
  head -45 /etc/passwd | tail -1 | cut -d ':' -f 3,4

②  在终端上显示当前用户行号 // 45
    grep "hqyj" /etc/passwd -n | cut -d ':' -f 1

```

2.9 通配符

```
* 代表所有字符
?  通配一个字符
[ab] 通配中括号里面的其中一个字符
[^ab] 通配一个字符，除了a或者b的其他字符;
[a-z] 通配这个范围里面的一个字符
例如:
    test1.c test2.c testa.c testb.c testab.c testc.c
命令:
    ls *.c
  输出:
    test1.c test2.c testa.c testb.c testab.c testc.c
命令:
    ls test?.c
  输出:
    test1.c test2.c testa.c testb.c testc.c
命令:
    ls test[ab].c
  输出:
      testa.c testb.c
命令:
    ls test[^ab].c
     输出:
      test1.c test2.c  testc.c

 eg:
   char buf[32] = {0};

    scanf("%[^\n]", buf);

    puts(buf);
 输入:
  hello world
 输出:
  hello world

```

2.10 修改文件的权限

```
一个文件的权限分为：
  用户的权限      u
  组的权限        g
  其他用户的权限 o
  用户的权限、组的权限、其他权限 可以用a来表示

  文件的权限：
    读   r    4
    写   w    2
    可执行 x   1

chmod 0777 filename 给一个文件赋最高权限
chmod 0666 filename
  eg:
   chmod a+x file.txt 将file.txt的权限所有者都赋可执行权限
   chmod uo-x file.txt 将file.txt 的用户的权限、其他权限去掉
     可执行权限；
   chmod u+rw file.txt 将file.txt 的用户的权限赋读、写的权限

 修改一个文件所在的用户：
  eg：
      chown root file.txt
      将fiel.txt 的用户名修改为root
  修改一个文件所在的组：
   eg:
    chgrp root file.txt
      将file.txt 的组名修改为root
```

day3

1.软链接

```
软链接创建：
    ln -s 源文件  目标文件
 相当于windows的快捷方式；
 一般源文件给绝对路径；
 软链接的文件类型是链接文件；
 当源文件给了绝对路径以后，链接文件可以移动到其他路径下使用；
 源文件给相对路径，链接文件不能移动；
 如果源文件移动了那么，链接文件会失效；
   eg:
    ln -s /home/jiangcx/class/230704/day3/a.out print
                                源文件                  链接文件

```

2.硬链接

```
硬链接的创建：（相当于对文件取别名）
  ln 源文件 目标文件
    硬链接是根据inode创建的文件，硬链接文件和源文件inode号一样；
    当硬链接文件被修改，源文件也会被修改；
    创建硬链接硬链接数会加一；
    当原文件啊被删除，硬链接数会减一；
    源文件被移动，硬链接数不变， 硬链接正常使用；
```

3.man 手册

```
man 函数名 （默认查看第一页）
man num 函数名 （查看第num页）

一般第一页查询 命令
    第二页      系统调用函数的接口 （open）
    第三页      库函数

man -a 函数名 （q退出当前页，回车进入下一页）

```

4.用户管理

4.1 adduser

```
1.创建用户
  sudo adduser 用户名

  /etc/skel 存放用户的模板
  /etc/passwd 存放用户信息
  /etc/group 存放用户的信息
  /etc/shadow 存放用户密码的文件

  sudo su 去到root用户下
  su 用户名 去到某个普通用户

  关机命令：
  sudo shutdown -r now(时间) 重启
  sudo reboot 重启
  sudo shutdown -h now(时间) 关机

  新添加的用户用不了sudo,解决办法：
 （切换到root用户或者本地用户）
    sudo vi /etc/sudoers
      添加一行代码：
      linux   ALL=(ALL:ALL) ALL
     （新用户名）
修改用户密码：
        sudo passwd 用户名
```

4.2 usermod

```
  1.
    sudo usermod -aG linux test
      将test追加到linux 这个组里面
  2.
    sudo usermod -c LINUX linux
      将linux这个用户在/etc/passwd里的第五个参数修改为LINUX
      //修改登录界面名称
  3.sudo usermod -g test linux
    //将linux 这个用户的组改为test;
  4.sudo usermod -d /home/test linux
    //将linux 这个用户的家目录改为/home/test
  5.sudo usermod -l Linux linux
      //将linux这个用户的用户名改为Linux

```

4.3 用户的删除及组的删除

```
sudo deluser 用户名 删除用户
sudo deluser  --remove-home 用户名 删除用户及家目录
sudo delgroup 组名 删除组
```

5. 磁盘相关的命令

```
/dev/sda开头 磁盘相关的文件
/dev/sdb开头  移动硬盘相关的文件
/dev/sr0 光驱 （CD/DVD）
  sudo fdisk -l 查看磁盘
  sudo df -h 查看分区

  挂载光驱:
    sudo mount /dev/sr0 ~/myDVD
  取消挂载:
    sudo umount /dev/sr0
   挂载优盘:
      sudo mount /dev/sdb ~/myDVD
   取消挂载
      sudo umount /dev/sdb

```

6 .环境变量

```
env 查看环境变量

HOME:  存放家目录
PATH: 保存可执行文件的路径
  如何输入可执行文件名就能运行:
    ①导入临时的环境变量:
      export PATH=$PATH:./
    ②让环境变量长久有效
```

```
        将export PATH=$PATH:./ 放在家目录下的.bashrc文件中
     让脚本生效:
          source .bashrc
    ③打开/etc/environment 文件
           sudo vi /etc/environment
      PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:
            /usr/bin:/sbin:/bin:/usr/games:/usr/local
              /games:/snap/bin:./"（加上./）
      source /etc/environment
```

7. 网络相关的命令

```
ifconfig 查询ip地址
ping (ping 两台主机之间是否能通信)

 解决ubuntu网络图标消失的问题:
 1.删除文件:
 sudo rm  /var/lib/NetworkManager/NetworkManager.state
 2.将文件里的false改为true
    sudo vi /etc/NetworkManager/NetworkManager.conf
 3.重启网络服务
    sudo service network-manager restart
```

8.进程相关的命令

```
1.进程:动态的概念，一个程序的执行过程;
 进程号: pid

 kill : 向内核发送信号;
  kill -l 查看信号
    eg:
        kill -9 pid 向一个进程发送SIGKILL 的信号
 ps  -aux  查看进程
 top 动态查看进程的状态
```

day4

1.shell 脚本

```
shell 是命令行解释器
  ubuntu 默认使用bash
  #! /bin/bash  表示此脚本由bash 解释;
  sh
```

```
  csh
  ksh
shell 脚本
  shell命令的有序集合；
  shell 是一门解释性的语言,不需要编译;
创建脚本的三步：
    第一步：创建一个xxx.sh的文件
      vi xxx.sh
    第二步：给初始文件赋权限
      chmod 0777 xxx.sh
    第三步：执行shell脚本
      ./xxx.sh
  eg：
    1.在家目录下创建一个目录 mydir
    2.将/etc/passwd 和 /etc/groff 拷贝到mydir
    3.再将mydir 归档并压缩为 mydir.tar.gz
  执行脚本的三种方法：
    第一种：./xxx.sh
    第二种：  bash xxx.sh
    第三种：source xxx.sh

   ./xxx.sh 适用于所有脚本的运行，需要赋权限； bash xxx.sh 默认
    由/bin/bash解释
   ./xxx.sh 和 bash xxx.sh 执行的时候都是创建一个子终端执行，
      再将结果返回到当前终端;
   source 是在当前终端执行的，一般用于让脚本生效;
```

2.shell 变量

```
自定义变量：
  1.命名规则：  见名知意
  2.字母数字下划线不以数字开头
  3.变量的前面不需要数据类型
  4.变量内容都视为字符串
  5.变量不需要提前定义，可以直接使用
  6.变量的后面不需要给分号结束
  7.在对变量赋值的时候，等号两边不能有空格
 位置变量(相当于命令行参数)
  $0 $1.....
 预定义变量
  $? $$
 环境变量
  PATH HOME ...
```

2.1 变量的引用

```
在引用变量的时候，需要在变量前面加上 $
eg：
    $var 或者 ${var}
```

2.2 变量的赋值

```
var=123
var=hello //变量的重新赋值
当字符串有空格的时候，赋值的时候需要加引号，双引号单引号都可以；
如果引号里面有引用变量的情况，必须是双引号；
eg:
  var1='hello world'
  var2="hello world"
  var="$var world" //变量的追加赋值
 eg:
  编写一个shell脚本，定义两个变量保存环境变量 HOME 和 PATH 的值；
  将两个变量的值交换；
```

2.3 变量的取消赋值

```
unset var
```

2.4 命令置换

```
将一个命令的结果赋值给一个变量
  eg:
    var=`ls`(注意：  ` 是Esc下面的按键 )
    或者
    var=$(ls)
```

2.5 位置变量

```
命令行参数
  $0 $1 $2 $3.....${10} ${11}.....
  $0表示脚本文件名，如果用source执行$0为bash
  $@ $* 表示所有的命令行参数
  $$ 脚本执行的进程号 (预定义变量)

  set 将命令置换的结果赋值给命令行参数
 eg:
    #ret=`grep "hello" test.txt -ni`
   // test.txt 一个已经创建好的文件；
    ret=`grep $1 $2 -ni` //将grep搜索的结果赋值给ret变量
      //终端执行的时候应该加上命令行参数 如：
      // bash XXX.sh "str" 文件名
    echo $ret
      //打印grep搜索的结果

    set $ret //将命令置换的结果设置给命令行参数

    echo $1 //搜索的第一条结果
    echo $2//搜索的第二条结果
    echo $3 //.....
    echo $4
```

2.5 echo 输出

```
echo str 直接输出 并且自带换行
echo -n str 输出不换行
echo -e "hello\n" 识别特殊字符'\n'
```

2.6 字符串

2.6.1 字符串的拷贝

```
str1=hello
str2=world
str1=$str2
```

2.6.2 字符串的连接

```
str1=hello
str2=world
str1="$str1$str2"
```

2.6.3 求字符串的长度

```
${#str}
```

2.6.4 字符串的提取

```

  str="www.baidu.comwww.baidu.com"
1. ${str:start:len}
 从左往右数到第start的位置，提取后面的len个字符;
  eg:
    ${str:4:3} --> bai
2. ${str:start}
从左往右数到start的位置，提取后面的所有字符;
eg:
    ${str:4}   --->baidu.comwww.baidu.com
3.${str:0-start:len}
从右往左数到start的位置，提取该位置及后面的len-1个字符
eg:
  ${str:0-5:4} ---> u.co
4.${str:0-start}
 从右往左数到start的位置，提取该位置及后面的所有字符
   eg:
    ${str:0-5} --> u.com
 查找子串
5.${str#*sub}
从左往右数，提取第一个子串后面的所有字符
  eg:
    ${str#*bai} ---> du.com www.baidu.com

```

```
6.${str##*sub}
从左往右提取最后一个子串后面的所有字符
  eg:
  ${str##*bai} ---->du.com
7.${str%sub*}
 从右往左数，提取第一个子串前面的所有内容
eg:
  ${str%bai*} --->www.baidu.com www.
8. ${str%%sub*}
从右往左数，提取最后一个子串前面的所有内容
  eg:
  ${str%%bai*} ---> www.

```

3.注释

```
# 单行注释
:<<!
   注释的内容
!

:<<"EOF"
   注释的内容
EOF

```

4.数组

4.1 数组的初始化

```
数组只支持一维数组，里面元素依旧是字符串
1.根据下标依次赋值
  arr=(1 2 3 4 5)
2.不按照顺序赋值
  arr=([0]=10 [2]=20 [3]=30 [5]=50 [10]=100)

  没有被赋值到的元素值为空;数组的长度只会统计有效的长度为 5

```

4.2 数组的引用

```
${数组名[下标]} 下标依旧是从0开始
  eg:
    arr[1] 表示数组的第二个元素
    ${arr[1]} 表示数组的第二个元素的值
    ${arr[@]} 和 ${arr[*]} 表示数组的所有元素

  arr[1]=100 //对数组的第二个元素赋值

```

4.3 求数组的长度

```
数组的长度:${#arr[@]} ${#arr[*]

单个元素的长度:  ${#arr[下标]}
```

4.4

```
在数组的后面追加赋值
 arr=(${arr[@] "AAA"})

在数组的前面插入赋值
  arr=("AAA" ${arr[@]})
  注意:
    顺序初始化和跳跃初始化，在数组的整体前后赋值，
    数组里的元素都会重新排列，有空格的字符串会被
    拆分开作为多个元素，下标是跳跃的也会变得有顺序;
```

4.5 取消赋值

```
unset arr[下标]
unset arr
```

day5

1.read 输入

```
read 变量1 变量2 ....
  //当输入多个字符串的时候，遇到空格就会分割字符串
read -p "提示的内容" 变量
read -a 数组名  //输入的数据依次对数组赋值
read -n num 变量 //输入的字符个数等于num就自动结束输入
read -s 变量 静默输入 //比如在输入密码的时候不会显示输入的字符
read -t 秒数 变量 //在规定时间内正常输入，否则结束输入
```

2.shell运算

```
shell 变量保存的是字符串，要进行运算需要加特定的符号;
 + - * /  %
 += -= *= /=
 > < == !=
 && || !
 & | ~ ^ >> <<
```

2.1 (())

```
1.支持整数的运算，不支持浮点数的运算;
2.在运算的时候，取变量的值可以加$也可以不加
3.可以将计算结果整体赋值给一个变量,需要加$
 eg:
    ret=$((var1 + var2))
  $?  表示上一条语句的执行情况，执行成功返回0，失败返回非0
4.可以像C语言一样进行复杂的运算;
5.((表达式1，表达式2，表达式3))
  以表达式3作为整个表达式的运算结果
```

2.2 $[ ]

```
1.支持整数运算，不支持浮点运算
2.在运算的时候，取变量的内容可以加$也可以不加
3.运算结果一定要赋值给一个变量保存
4.基本上和(())用法一样
5.ret=$[表达式1，表达式2，表达式3]
   //以表达式3作为整个表达式的运算结果
```

2.3 expr

```
1.支持整数运算不支持浮点运算
2.在运算的时候需要加上空格
  eg:
    expr $var1 + $var2
3.引用变量的时候必须加上$
4.expr 可以输出结果
5.如果需要将结果赋值给一个变量，那么需要命令置换
      eg:
        var=`expr $var1 + $var2`
6.expr 的算术运算
          + -  \*  /
  关系运算
       \> \< \>= \<= ==
  如果需要加上括号运算，括号需要转义
          \(  \)
7.expr 不能进行幂运算
```

```
expr 处理字符串
  str=hello
  1.匹配字符串
  expr match $str "string"
  从第一个字符开始匹配，如果第一个不一样返回0，
    第一个一样返回一样的个数
  2.提取字符
  expr substr $str start len
   从第start 的位置开始提取len个字符
  3.求字符串的长度
  expr length $str
```

3.shell中的选择结构

```
if 判断式
then
  语句块;
fi

if 判断式
then
   语句块1
else
   语句块2
 fi

阶梯结构
if 判断式1
then
   语句块1;
elif 判断式2
then
   语句块2;
  ......
elif 判断式n
then
   语句块n;
else
   语句块n+1
fi

嵌套:
  if 判断式1
  then
      if 判断式2
      then
          语句3 //判断式1 和 判断式2 都成立执行语句3
      else
         语句2 //判断式1成立， 判断式2不成立
      fi
```

```
    else
        语句块1 //判断式1不成立
   fi
```

4. shell 中的判断式

```
第一种：
    [ 判断式 ]  [ 和 ] 左右必须有空格；
第二种：
    test 判断式   test 左右也必须有空格
```

5. shell整数的判断

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

6.shell 与 或 非

```
&&   -a
||   -o
!     !
```

```
1.输入一个成绩输出对应的等级；
2.输入一个年份判断是否是闰年
  (能被4整除但不能被100整除  或者  能被400整除)
```

7.结束脚本

```
exit 0 正常结束
exit 1 不正常结束

重新执行此脚本
  bash $0
```

8 shell 字符串的判断

```
-z 判断字符串是否为空
-n 判断字符串不为空
= 判断字符串相等
! = 判断字符串不等
 /> 判断字符串大于
 /< 判断字符串小于
   对字符串的判断，在引用字符串内容的时候加上引号;
eg:
    "$str"
 eg:
   从终端输入两个字符串，比较两个字符串的大小
```

9.shell 对文件的判断

9.1 对文件类型的判断

```
-f 判断文件是否存在并判断是否为普通文件
-d 判断文件是否存在，并判断是否为目录文件
-L 判断文件是否存在，并判断是否为链接文件
-b 判断文件是否存在，并判断是否为块设备文件
-c 判断文件是否存在，并判断是否为字符设备
-p 判断文件是否存在，并判断是否为管道文件
-S 判断文件是否存在，并判断是否为套接字文件
-s 判断文件是否存在，文件长度大于0为真
-e 判断文件是否存在
```

```
输入一个文件判断类型
```

9.2对文件权限的判断

```
-w 判断文件是否具备写权限
-r 判断文件是否具备读权限
-x 判断文件是否具备执行权限

eg:
  输入一个文件，判断文件是否存在，如果存在那么再判断文件是否具备
     写权限，如果写权限就往这个文件写入一句“hello world”,如果
     不具备写权限，那么赋上写权限再写入hello world;
```

9.3 对文件时间戳的判断

```
-nt 判断文件更新
-ot 判断文件更旧
```

10 case ....in ....esac

```
case 表达式 in
  模式1)
```

```
    shell语句块1
    ;;
  模式2)
   shell 语句块2
    ;;
  .....
  *)
    shell 表达式n;
     ;;
esac

  注意:模式可以通配
    * 通配所有，以上模式和表达式都不匹配就执行*下面的内容
    [a-zA-Z]通配a-z 或者A-Z
    "str1"| "str2"|.... 通配str1或者 .....strn

    请输入一个软件名，输入yes 下载，输入no拒接下载
```

day6

1.循环结构

1.1 while

```
while 判断式 //表达式成立执行循环体
do
    shell 语句
done

 辅助控制语句
 break 跳出当前循环
 continue 结束本次循环继续下一次循环

实现从1加到100求和
```

1.2 until

```
while 判断式 //表达式不成立执行循环体
do
    shell 语句
done
```

1.3 for循环

```
1.一般用法
 //     初值        判断      递增递减
```

```
  for((表达式1；表达式2；表达式3))
  do
     循环体
  done
2.常规用法
 for 变量 in 单词列表
 do
      循环体
 done


 1.散列的单词列表
 2.连续的单词列表 {start..end}
 3.把一个命令经过命令置换作为单词列表


eg:
  <<!
for var in apple huawei sanxing oppo vivo xiaomi meizu
do
        ((i++))
        echo "i = $i ---> var: $var"
done
!

:<<!
for var in {1..100}
do
        ((sum += $var))
done
echo "sum = $sum"
!

for var in `ls $1`
do
        ((i++))
        echo "$var"
done

echo "i = $i"


 练习:
 1.输入两个目录名称 mydir myfile,判断这两个目录是否
  存在，如果存在输入【yes/no】询问要不要删除，
  如果删除则再创建出来；
2.再输入一个已经存在的目录，将这个目录里的所有文件
 拷贝到myfile中，所有目录拷贝到mydir
3.统计文件的个数，目录的个数；
```

2. shell 函数

```
一般形式
  function 函数名（）{
    函数体
  }
shell 函数 function 是shell函数的标配
shell 函数没有数据类型
shell 函数没有参数说明列表
shell 里的变量默认为全局变量
shell 中的局部变量 需要加上local

函数调用：
  函数名
    或者
  函数名 arg1 arg2 ....(参数)
   $0    $1   $2
  在函数体内部的位置变量通过调用函数的时候传参；
  在函数体外部的位置变量通过命令行参数传递
  eg：
    :<<!
function add(){
        ret=$((10+20))
}

add

echo "ret: $ret"
!

:<<"!"
function add(){
        ret=$(($1+$2))
}

#add 20 20
add $1 $2 # 执行脚本： bash 08_func.sh 10 20

echo "ret: $ret"
!


function add(){
        local ret=$((10+20))

        return $ret #返回结果
}
add #函数调用
echo ret: $? #结束返回值，注意只能接收255以内
```

4. C语言

4.1 虚拟4G内存

```
C语言的本质：内存
C语言是如何分配内存的：
  1.根据数据类型，自动分配内存
  2.在堆区上手动分配空间

  虚拟4G内存：
    内核层 3G-4G
    用户层 0-3G
     堆区 手动申请 手动释放 （malloc / free）
     (映射区 、mmap)
     栈区 自动申请、自动释放

    静态数据区
       dada 段 全局或者static 修饰的 初始化的变量
       bss 段  全局或者static 修饰的未被初始化的变量
       test 段
       .ro / rodata段 只读 const 修饰 、字符串常量
             const char *p = "hello world"

```

4.1 类型

```
存储类型：
    自动存储 auto 一般默认为自动存储
    寄存器存储 register 运行速率快
       为什么不都定义成寄存器存储？
          不一定能申请到，寄存器数量是有限制的；
    外部存储： extern 一般用作外部说明
    静态存储：static
       修饰局部变量 没有初始化初值为0
       修饰局部变量 延长生命周期
       修饰全局变量或者函数 限制作用域,仅能在本文件使用
       static 和 extern 不能一起使用；
      头文件：
       <include.h> 默认在系统目录下搜索这个头文件
       "include.h" 默认在当前目录下搜索头文件

 数据类型：
  基本类型
       char  1字节
       short  2字节
       int  4字节
       long  8字节
       longlong 8字节
       float 4字节
       double 8字节
       enum  4字节
  构造类型
       数组 结构体 共用体
  指针类型
```

```
        char * 、int * 、struct stu *....
  无类型 void

```

4.2 const

```
1.const 修饰常量吗？
   const 修饰的是变量
2.const 修饰的变量一定不能修改吗？
    const 修饰的局部变量可以用指针间接修改；
3.
 1） const 数据类型 *指针变量名 ；
  指针变量不能修改目标的值,指针变量本身的值可以修改；
eg：
   int var1 = 10, var2 = 20;
    const int *p = &var1;
    *p = 100; // false read-only
    p = &var; // true
2) 数据类型 * const  指针变量名；
 指针变量可以修改目标的值，指针变量本身的值不能修改；
eg:
  int var1 = 10, var2 = 20;
  int * const p = &var1; // 只能初始化
  p = &var2 // flase
  *p = 100; // true

 3) const 数据类型 * const 指针变量名；
 指针变量既不能修改目标的值，指针变量本身的值也不能修改；
eg:
  int var1 = 10, var2 = 20;
  int * const p = &var1; // 只能初始化
  p = &var2 // flase
  *p = 100; // false

```

4.3 malloc 动态分配内存

```
 #include <stdlib.h>

 void *malloc(size_t size);
{
  功能：  在堆区上申请一片空间；
  参数：  size：表示空间的大小
  返回值：返回空间首地址，void *类型
    因为工程师可能需要不同类型的空间，所有返回void *类型的地址，
    可以强制类型转换为其他任意类型的地址；
    申请失败返回NULL；

}
```

```
 void free(void *ptr);
{
  功能：释放空间
  参数：空间的首地址
    注意：malloc 申请的空间需要手动释放，如果不释放可能会造成宕机
   把空间释放之后，指向申请的这片空间的指针依旧指向这片空间，需要将
    它指向零地址(待指状态);
}

输入一片数据在堆区上存储：

什么是段错误？
  访问了非法内存；
如何规避段错误？
  不能使用野指针；
  malloc 释放了空间之后将指针指向NULL;
```

5 .二维数组

```
可以由多个一维数组组成：
  int arr[2][3] = {1, 2, 3, 4, 5, 6}
    可以看成由 两个一维数组组成；
  arr[0]
  arr[1]

  arr[2][3] --> {arr[0], arr[1]}
元素的访问：
  arr[0] == *(arr + 0)
  arr[1] == *(arr + 1)

  arr[0] 一维数组 ---> {1,  2,  3}
 元素的访问：
  *(arr[0] + 0) == 1 == *(*(arr + 0) + 0) == **arr
  *(arr[0] + 1) == 2 == *(*(arr + 0) + 1)
  *(arr[0] + 2) == 3 == *(*(arr + 0) + 2)

    第i行的第j列：
   *(*(arr + i) + j)
  == arr[i][j]
  == (*(arr + i))[j]
  == *(arr[i] + j)


```

6.指针访问二维数组

```
二级指针可不可以访问二维数组？
   不能，二维数组的数组名是行指针，二级指针和二维数组名的类型不一样；
```

```
行指针的一般形式：
  <存储类型> <数据类型> （*行指针变量名）[常量表达式];
                                          //列数
 eg：
    int arr[2][3] = {0};
    int (*p)[3] = arr;

第i行的第j列：
   *(*(p + i) + j)
  == p[i][j]
  == (*(p + i))[j]
  == *(p[i] + j)

  // char * arr[32]

```

day7

---

1. 指针数组

```
一般形式：
    <存储类型> <数据类型> * 指针数组名[元素个数];
   eg：
      int * parr[5]; //表示一个可以存储5个整型地址的数组

    int var1, var2, var3;
      parr[0] = &var1;
      parr[1] = &var2;
      parr[2] = &var3;

    int *p = &var1, *q = &var2, *k = &var3;
      parr[0] = p;
      parr[1] = q;
      parr[2] = k;

    sizeof(parr) == 8 * 5
用指针数组保存二维数组：
    int a[2][3];
    parr[0] = a[0];
    parr[1] = a[1];

   a[i] == *(parr + i)
   a[i][j] == *(*(parr + i) + j)

    parr++// 数组名地址常量

 用二级指针访问指针数组：
      //int a[32] = {0};

```

```
      //int *p = a;

    int **p = parr;
  a[i][j] == *(*(p + i) + j)
```

```
思考：
char *str = "12345678";
char p[5] = {0};
short *t = (short *)&p[2];
*t = atoi(str + 4); // atoi("12345") == 12345;
求：
  p[0]:0
  p[1]:0
  p[2]:0x2e
  p[3]:0x16
  p[4]:0

 小端:数据的低位存在低地址，高位存在高地址
 大端：数据的高位存在低地址，低位存在高地址
```

2.函数

2.1 函数的传参

```
1.函数的传参：
    复制传递：将实参的值赋值给形参，形参不能修改实参的值
    地址传递：将实参的地址赋值给形参，形参可以修改实参的值；
    全局变量：
```

2.2 数组作为参数传递

```
一维数组：
 ①地址传递：
    函数原型：
          void func(int *p, int size);
          void func(char *p);
    一般在传递数组的时候，传递数组的首地址和大小；
    在传递字符串数组的时候，可以不用传大小，
    因为知道首地址就能找到结尾的'\0'；
 ②复制传递：
        函数原型：
          void func(int p[100], int size);
          void func(int q[], int size);
          sizeof(p) = ? //8
          sizeof(q) = ? //8
```

```
 数组的复制传递个地址传递只是表现形式不同，实质上都是将数组的首地址
 赋值给形参，形参可以修改实参的值；

 二维数组：
       int arr[3][2] = {1,  2, 3, 4, 5, 6};
       void func(int (*p)[2], int c);
 指针数组：
      int * parr[3] = {arr[0], arr[1], arr[2]};
      void func(int **p, int c, int l);

```

2.3 指针函数

```
本质：  返回值为一个地址的函数；
一般形式：
    <数据类型>   * <函数名>(形式参数说明列表)
    {
          函数体；

          return 地址量；
    }
eg：
    char * strcpy(char *dest, const char *src);
    返回值返回dest；
    char str1[32] = "hello";
    char str2[32] = "world";
/*    strcpy(str1, str2);
      puts(str1);*/
    puts(strcpy(str1, str2));

```

2.4 函数指针

```
本质：是一个指针，一个可以指向函数的指针；
 函数名：入口地址；

函数指针的一般形式：
  数据类型 (*函数指针变量名)(形参);
  数据类型和形参应该和所指向的函数保持一致；
函数指针数组的一般形式：
  数据类型 （*函数指针变量名[元素个数]）(形参);

int add(int x, int y)
{
  return x + y;
}

int sub(int x, int y)
{
  return x - y;
```

```c
}

int testfunc(int x, int y, int (*pfunc)(int, int))
{

   return pfunc(x, y);
}

int main()
{
  int a = 10, b = 20;
  printf("%d\n", add(a, b)); // 30
  printf("%d\n", sub(a, b)); // -10

  //通过函数指针来访问函数
  int (*pfunc)(int, int) = add;
  printf("%d\n", pfunc(a, b)); // 30
  pfunnc = sub;
  printf("%d\n", pfunc(a, b));//-10
   //函数指针数组
   int (*parrfunc[3])(int, int) = {add, sub};
   printf("%d\n", parrfunc[0](100, 200));//300
   printf("%d\n", parrfunc[1](100, 200));//-100

  //回调函数
  printf("%d\n", testfunc(10, 20, add));

  return 0;
}

```

3. gdb 调试

```
编译:  gcc file.c -g
调试:
   gdb a.out

 l 查看代码默认10行
 r 运行
 b 行号 设置断点
 info b 查看断点
 c 继续运行
 n 单行运行
 p var 查看变量的值
 q 退出
```

4. typedef 和define

```
1.define 宏定义，宏替换 在预处理的阶段进行的;
  将后面的内容用前面的内容来替换;
   不需要加分号;
```

```
2.typedef 取别名，在编译的阶段进行的;
  将前面的内容取个别名放在后面;
  语句结束需要加分号;

eg:
    #define INT_t int
     INT_t var;
     typedef int int_t;

eg:


#if 0
        typedef int * int_p;
        int var = 10;
        int_p p = &var;
        printf("*p = %d\n", *p);
#endif
#if 0
        typedef int Arr_t[10];
        Arr_t arr={0};
        arr[0] = 100;
        arr[9] = 200;
        int i = 0;
        for(; i < 10; i++){
                printf("%-4d", arr[i]);
        }
        puts("");
        printf("sizeof(arr) = %ld\n", sizeof(arr));

#endif

#if 0
        typedef int *Parr_t[10];
        Parr_t parr;
        printf("sizeof(parr) = %d\n", sizeof(parr));

#endif

#if 1
        typedef int (*Pfunc_t)(int);
        Pfunc_t pfunc;
        pfunc = func;

        printf("%d\n", pfunc(10));
        typedef int (*PfuncArr_t[3])(int);
        PfuncArr_t pfuncarr = {func};
        printf("%d\n", pfuncarr[0](20));

#endif


```

5.枚举

```
用于对有序数据的罗列，比如月份、星期；
一般形式：
  enum Week{
      Mon,
      Tue,
      Wed,
      Thu,
      Fri,
      Sat,
      Sun
  };

 1.枚举里面的成员是常量；
 2. 对枚举里面的成员赋值，那么下面的成员会依次递增，
    如果枚举里的成员没有赋值第一个默认为0；
 3.枚举定义的变量用枚举里面的成员来赋值；
 4.枚举是基本类型占4个字节；
 5.枚举的成员之间用,隔开；

enum Week day = Sun;
//enum Week day = 7; // 一般不这样赋值

```

6.结构体

```
1.结构体属于构造类型，它的成员可以是相同数据类型的集合，
  也可以是不同数据类型的集合；
2.结构体里面的成员是变量；
3.结构体里面的成员一定是固定大小的成员；
4.结构体的成员之间用 ； 隔开；
5.只有在定义了结构体变量的时候系统才分配空间；
6.结构体变量之间可以直接赋值；
7.结构体的关键字是struct
8结构体的访问：结构体变量名.成员 如果是指针，结构体指针->成员
```

```
结构体的一般形式：
  struct 结构体名{
    成员列表1;
    成员列表2;
     ....
  };
 定义结构体变量：
    struct 结构体名 结构体变量名;
```

6.1 结构体的定义方式

```
① struct STU{
    char name[32];

```

```
    int id;
    float score;
  };

② struct STU{
    char name[32];
    int id;
    float score;
 }stu1, stu2;

③ struct{
   char name[32];
    int id;
   float score;
 }stu1, stu2; //不能单独定义结构体变量

④ typedef struct STU{
     char name[32];
    int id;
    float score;
 }stu_t;

⑤ typedef struct{
     char name[32];
    int id;
    float score;
 }stu_t;

```

7.结构体赋值

```
① 结构体变量.成员直接赋值;
 strcpy(stu.name, "TOM");
 stu.id = 1;
 stu.score = 98.5;

② 定义结构体变量的时候整体赋值作为初始化;
    struct STU stu3 = {"xiaoming", 1, 97, 'B'};
③ 对变量赋值的时候，整体赋值那么需要强转类型换行
    stu2 = (struct STU){"xiaofang", 3, 100, 'G'};
④部分赋值:
     struct STU stu4 = {
                    .name = "小张",
                    .score= 70
     };

```

8.结构体数组

```
eg:
  typedef struct{
```

```
    char name[32];
    int id;
    float score;
    char sex;
  }stu_t;

stu_t stu[3];
1.逐个赋值：
 strcpy(stu[0].name, "TOM")

  2. 完全初始化
   stu_t stu[3] = {
    {"小明", 1, 90, 'B'},
    {"小红", 2,  83,  'B'},
    {
      .name = "小芳",
      .sex = 'G'
    }
  };
//跳跃赋值
3. stu_t stu[3] = {
    [0] = {"小明", 1, 90, 'B'},
    [2] = {"小红", 2,  83,  'B'},
 };
作业：
    定义一个结构体数组，实现对学生信息的存储以及增删改查;
```

day8

1.结构体指针

```
通过指针访问结构体变量用箭头;
typedef struct{
  char name[32];
  int id;
  char sex;
}stu_t;

stu_t stu1;
stu_t *p = &stu;
strcpy(p->name, "Tom");
p->id = 1;
p->sex = 'B';

```

2.结构体字节数对齐

```
64位操作系统：
    当结构体中字节数最大的成员大于4字节按照8字节对齐；
    字节数最大成员小于4字节就按照字节数最大的成员对齐；
注：
   在存储的时候如果先存储低字节的数据，后面对齐的字节数足够存一个高字
   节数据的时候，那么存储完低字节数据会空出相应的地址空间；
   eg:
    struct STU{
      char var;
      short var1;
      int var2;
    };
  | var|空|var1 |   | 4字节
  |var2|  |     |   | 4字节
```

3. 联合体 union

```
程序开发用的相对比较少；
特点：  内存是共用的；
缺点：修改一个成员其他的数据会被覆盖；
优点：节约空间
共用体空间大小：由最大的成员所占字节数来确定；
```

4.条件编译

```
格式：
 #ifdef ....#endif
 #ifdef .... else .... #endif
 #ifdef ....elif ... elif ....#endif
 #ifndef ...#endif
 eg1：
    #define AAA
      int main()
    {
      #ifdef AAA
      printf("hello\n");
      #else
      printf("world\n");
      #endif
      return 0;
    }
eg2:
  //#define AAA
  #define BBB 1
  int main()
  {
    #ifdef AAA
      printf("AAA\n");
```

```
    #elif BBB
      printf("BBB\n");
    #elif CCC
     printf("CCC\n");
    #else
      printf("DDD\n");
    #endif
      return 0;
  }

```

5. Makefile

```
Makfile 是一个普通文件，里面放着对工程的编译规则;

make 是一个可执行程序，对Makefile中编译规则的解析;

Makefile 可以根据文件的时间戳推导，编译新修改的文件;

makefile的体验版本：

 ①体验版本

(标签)
all:
            编译指令
<一个Tab的位置>

编译：
      make 或者make all
编译的四个阶段:
  1.预处理：展开头文件和宏
    gcc -E file.c -o file.i
  2.编译:检查语法错误生成汇编文件
   gcc -S file.i -o file.s
  3.汇编: 生成二进制文件
   gcc -c file.s -o file.o
  4.链接:链接库，生成可执行文件
   gcc file.o -o app

 目标:依赖
     命令;
eg1:
app:file1.o file2.o main.o
        gcc file1.o file2.o main.o -o app
file1.o:file1.c
        gcc -c file1.c -o file1.o
file2.o:file2.c
        gcc -c file2.c -o file2.o
main.o:main.c
        gcc -c main.c -o main.o
```

```
clean:
        rm *.o app

makefile 中的变量:
 普通变量的引用:
  ${var} 或者 $(var)
 赋值:
    普通赋值:  //如果后面值被更新那么会取最新的值
       var=abc
    询问赋值:
      var ?= abc // 如果之前赋过值那么就不会再重新赋值
    追加赋值
      var += abc
    直接赋值
      var := abc //如果后面值被更新那么取当前值
       eg:
        var1 = abc
        var2 = $(var1)
        var1 = def

        var ?= abc
        var ?= def

        str = abc
        str1 = def
        str += $(str1)

        a := abc
        b := $(a)
        a := def

all:
@echo "var1 = $(var1)" # def
  @echo "var2 = $(var2)" # def makefile 直接赋值会取最新的值

  @echo "var = $(var)" # abc
  @echo "str = $(str)" # adcdef
  @echo "b = $(b)" # adcdef

  特殊变量:
  $@ 目标
  $< 第一个依赖
  $^ 所有依赖

 eg2:
CC = gcc
TARGET = app
OBJS = file1.o file2.o main.o
FLAGS = -c -o

$(TARGET):$(OBJS)
```

```
90          $(CC) $^ -o $@
91  file1.o:file1.c
92          $(CC) $(FLAGS) $@ $<
93  file2.o:file2.c
94          $(CC) $(FLAGS) $@ $<
95  main.o:main.c
96          $(CC) $(FLAGS) $@ $<
97  clean:
98          rm $(OBJS) $(TARGET)
99
```