Linux

笔记本： 我的第一个笔记本
创建时间： 2020/7/20 9:27 **更新时间：** 2020/8/23 22:25
作者： 1272209351@qq.com

---

uname -r 查看linux内核版本信息
vmstat 报告关于内核线程、虚拟内存、磁盘、陷阱和 CPU 活动的统计信息
sar 主要帮助我们掌握系统资源的使用情况，特别是内存和CPU的使用情况
stat 用于显示文件的状态信息

A top 查看cpu资源使用状态
B netstat 查看网络连接状态
C free 查看内存资源状态
D df 查看磁盘分区资源状态

一、基础指令

（一）命令的使用格式：命令名称 [操作选项] [操作对象]
（二）目录的命令：

（1）ls

ls ---- 浏览当前目录（windows 下的文件夹）下的文件信息
ls 默认情况下浏览 当前用户的家目录（当前主机的登录用户）
家目录------操作系统为每一个用户创建的受保护的目录
ls -a 选项：浏览目录下的所有文件，包含隐藏文件----Linux下文件名以 . 开头的文件默认不显示
ls -l 选项：查看目录下文件的详细信息
linux 下有 ----- 一切皆文件，linux 下所有东西都是文件，都可以通过操作文件的方式进行访问
linux 下文件类型并不以后缀名区别，目录也是文件只不过是目录类型文件
使用实例： ls -la Desktop/ --------- 选项可以组合使用
（不同文件显示颜色不同---蓝色：目录类型）
ls -lh：人性化显示

```
-h, --human-readable
       with -l, print sizes in human readable format (e.g., 1K 234M 2G)
```

```
man -ls ： 查看 ls 手册
man fopen ： 查看fopen 函数手册
```

给文件开辟空间大小：

```
drwxrwxr-x 7 san san 4096 Jul 15 21:52 workspace
[san@San ~]$ dd if=/dev/zero of=./hello.txt bs=500M count=1
1+0 records in
```

文件名 大小

（2）pwd

pwd --- 查看当前所在路径（打印工作路径）--- 打印出来的是一个绝对路径
/home/dev : 多层级路径的表达方式，最前边的 / 叫做根目录
绝对路径： 唯一路径，指的是以根目录为起始表达的路径
相对路径：多种多样，指的是以某一个路径作为参照路径---通常说的是以当前目录作为起始的路径

特殊文件：
. 表示目录自身
.. 表示目录的上一层目录

```
[dev@localhost ~]$ ls .
Desktop    Downloads  Music     Public     test        Videos
Documents  install    Pictures  Templates  third_part
[dev@localhost ~]$ ls /home/dev
Desktop    Downloads  Music     Public     test        Videos
Documents  install    Pictures  Templates  third_part
[dev@localhost ~]$ ls
Desktop    Downloads  Music     Public     test        Videos
Documents  install    Pictures  Templates  third_part
[dev@localhost ~]$
```

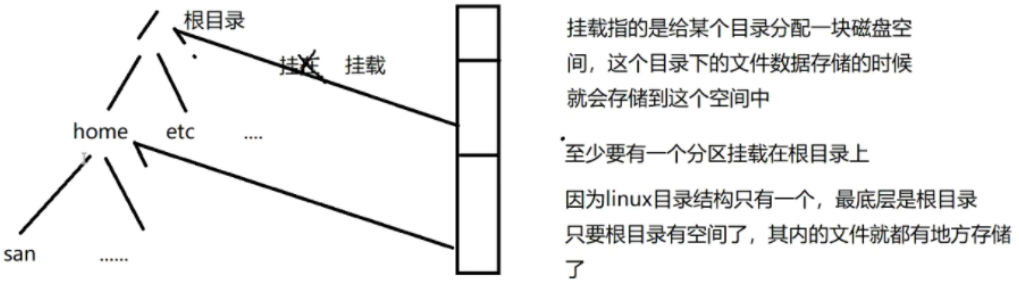
/home/san/workspace

san:./workspace home: ./san/workspace

Linux 下的目录结构：
磁盘：硬盘---存储文件
一个硬盘至少有两个分区：交换分区、文件系统分区----交换分区只有一个，文件系统分区可以有多个
交换分区：作为交换内存使用
文件系统分区：作为文件存储使用

程序运行，运行信息都占用的是内存。内存中的数据并非都是活跃数据
内存不够用---把非活跃内存区域中的数据取出来放到硬盘---交换分区，腾出来的内存就可以加载新的数据进行处理了

windows 下，磁盘分了多少分区，就可以有多少个盘符，每个盘符都是一个大目录，这个目录下的文件使用的磁盘空间就是这个分区的
windows 下的目录结构可以随着分区多少而改变；linux 下目录结构是唯一的，不会随分区多少而改变
Windows 就像是给某一个空间分配文件夹，linux 就像是给某一目录分配空间



（3）mkdir

mkdir ---- 创建目录（创建文件夹）
mkdir -p ----- 递归多层级创建目录，从外往内，哪一层不存在就创建哪一层

rmdir ---- 删除空目录
rmdir -p : 递归多层级删除目录，从内往外，哪一层为空就删除哪一层

rm ---- 删除文件
rm -r ： 递归删除目录下所有文件，最终删除目录
rm -i ： 有一个提示信息（rm -ir 删除提示）
rm -f ： 忽略提示信息（文件删除会提示是否删除？ y / n）
例如：rm -rf tmp

cp ： 将一个文件向另一个位置拷贝一份（默认拷贝普通文件）
cp -r: 递归将一个**目录及其内部文件**全部拷贝到指定位置
cp -r tem ./test

mv ： 移动一个文件或目录到另一个位置（剪切-----改名）

```
[san@San ~]$ mv ./test/ ./tmp
[san@San ~]$ ls
tmp  workspace
[san@San ~]$ ls tmp/
passwd  test.txt
[san@San ~]$
```

cd：改变工作路径，改变当前文件路径，进入某个目录
cd ~ ： 回到当前目录的家目录

常用操作：tab 键自动文件名补全-----避免手敲出错

（三）文件操作

（1）基本操作
touch --- touch 一个文件，若文件存在----刷新文件时间属性；若文件不存在---创建一个新文件
touch -d ： 使用指定时间刷新时间属性
例如：touch -d "2018-08-09 12:12:03" tmp
文件的时间属性：最后一次访问时间、最后一次修改时间、最后一次状态改变
touch -a：仅使用当前系统时间刷新**访问**时间
touch -m：仅使用当前系统时间刷新**修改**时间

stat ---- 查看文件状态信息

cat ----将文件内容打印出来

more ---- 分页显示文件内容
向下按行滚动---回车；向下按页滚动---空格；退出显示---q 键
less ---- 分页显示文件内容
向下按行滚动--- 向下/ 回车； 向下按页滚动---空格 /f 键；
向上滚动-----向上/ b键； 退出显示----q 键
匹配查找字符串： /string 向下找； ?string 向上找

head --- 默认显示文件前十行内容
head -n 数字 -----指定显示行数
tail ---- 默认显示文件末尾十行内容
tail -n 数字 ----- 显示末尾行数
tail -f -------动态一直等待末尾的新数据显示（自己输入）
（ctrl + C 退出）

ifconfig --- 查看网卡信息
echo --- 将后续字符串打印出来（打印字符串）-- 将数据写入标准输出--显示器设备
打印字符串：将字符串显示到界面上----操作系统控制，从键盘读取数据，要把数据写入显示器
linux 下一切皆文件-----所有东西都当做文件进行访问---对显示器操作也是个文件操作---将数据写入显示器文件

（2）重要操作

>> 或者 > ： 叫做重定向符号---进行数据流的重定向---文件重定向
>> 或 > ： 将要操作的数据，不在写入原本的文件而是写入到新文件中
例如：echo "asdf" >> test.txt ---把原本要写入到标准输出文件的数据写入到test.txt 文件中

> ：清空重定向，将数据重定向到指定文件中，但在这之前会清空文件原有内容；
>> ： 追加重定向，将数据重定向到指定文件中，并将新数据追加到该文件末尾。

管道符 | ： 连接两个命令，将前边命令的输出结果作为后边命令的输入数据，让后边命令进行处理
例如：打印23行内容：head -n 23 ./passwd | tail -n 1

关机命令：shutdown -h now 立即关机　　/　　halt 命令
su root --- 切换到管理员用户，需要输入管理员用户密码

（3）压缩打包

压缩：将一个文件按照一些压缩算法，将文件数据从多变少
zip（压缩）/ unzip（解压）：需要指定压缩包的名称（.zip）
gzip（压缩）/gunzip（解压缩）：不需要指定名称（Linux 下一般 .gz 后缀）
bzip2（压缩）/ bunzip2（解压缩）：不需要指定名称（.bz2）
（linux 下文件格式并不以后缀名区分，后缀名只是便于用户区分文件功能性质）

打包：将多个文件合成一个文件
tar : linux 下最常用打包工具---将多文件打包成一个文件，提供解包功能，并且打包解包同时可以进行压缩解压缩
-c ： 打包
-x ： 解包
-z：打包或解包同时进行 gzip 格式压缩或解压缩
-j：打包或解包同时进行bzip2 格式压缩解压缩
-v :显示 打包解包信息
-f ： 用于指定 tar 包名称，通常是作为最后一个选项，因为后边要加上包名称

tar -czvf * .tar.gz ** （打包）------ tar -xzvf *** .tar.gz （解包）**
tar -cjvf * .tar.gz ** （打包）------- tar -xjvf *** .tar.gz （解包）**
防止解压出错，常不指定压缩包形式：
tar -cvf * .tar.gz ** （打包） tar -xvf *** .tar.gz （解包）**
（打包：*** 表示压缩包名称，** 代表将要打包的文件名）

！！默认解压的文件都在当前所在目录下，不一定在压缩包所在目录

（4）匹配查找

在终端执行命令，单引号与双引号区别：
大多数情况下，意义相同，都是为了括起一串数据，表示一个整体
单引号会消除括起来的数据的特殊字符的特殊含义；双引号不会

匹配查找命令：
grep：从文件内容中匹配包含某个字符串的行，常用于在某个文件中找函数
-i ： 匹配忽略大小写
-v：反向匹配，匹配不包含字符串的行
-R： 对指定目录下的文件递归逐个进行内容匹配
例如：grep -R 'san' ./
grep -v 'nologin' passwd
grep -i 'root' passwd

find：从指定目录中查找指定名称/大小/时间/类型的文件
find ./ -name "*test*" 通过文件名查找文件
find ./ -size "*test*" 通过大小查找文件
find ./ -time "*test*" 通过时间查找文件

find ./ -type d 通过文件类型查找文件------

（f：普通文件；d：目录文件；c：字符设备 b：块设备 p：管道文件 l：符号链接文件 s：套接字文件）

find ./ size -10M 通过文件大小找文件（10 M 以内文件）
+10M 超过10M 大小的文件
find ./ mmin -10 通过时间找文件
cmin、mmin、amin--分钟为单位
ctime、 mtime、atime----以天为单位
（文件的时间属性：最后一次访问时间 c、最后一次修改时间 m、最后一次状态改变 a）

bc： 计算器
date：显示时间信息
date +"%s"： 时间戳。从1970年1月1日0时0分0秒到现在的秒数
date + "%Y-%m-%d %G : %M: %S" ： 年月日 时分秒
date -s "" ： 设置系统时间

```
[san@San sprint]$ date +"%Y-%m-%d"
2020-07-22
[san@San sprint]$ date +"%Y-%m-%d %H:%M:%S"
2020-07-22 11:12:28
[san@San sprint]$
```

cal ： 日历

```
CAL(1)              General Commands Manual              CAL(1)

NAME
       cal - 显示一个日历

总览
       cal [ -mjy ] [ 月份 ] [ 年份 ]

描述
       显示一个简单的日历..  如果没有指定参数，则显示当前月份.
       选项如下所列:

       -m     显示星期一作为一周的第一天..  (缺省为星期日.)

       -j     显示儒略历的(Julian)日期 (以 1 为基的天数，从  1
              月 1 日开始计数) .

       -y     显示当前年份的日历..

       一个单一的参数指定要显示的年份    (1    -    9999)    ;
       注意年份必须被完全地指定: cal 89 不会 显示1989年的日历.
       两个参数表示月份  (1  -  12) 和年份.  如果没有指定参数,
       则显示当前月份的日历.
```

su： 切换用户
su username
平常尽量避免直接使用root用户，因为root 用户是管理员，可以在系统中为所欲为

sudo：为普通用户的某个操作权限进行临时权限（不用每次切换管理员用户）：需配置

sudo需要进行配置之后才能使用：
配置过程：
1. su root 切换到管理员用户
2. visudo 打开sudo配置
3. :90 跳转到文档第90行，在90行附近 ---

```
## Allow root to run any commands anywhere
root    ALL=(ALL)       ALL
test    ALL=(ALL)       ALL
```

4. yy复制root这一行，p粘贴
5. 将用户名从root改变成自己的用户名称
6. :wq保存退出

sudo的使用演示： sudo yum install lrzsz

tab 键----自动补全（补不出来证明没有这个文件）
ctrl+c ---- 中断当前操作（q 键）
man ** ---- 查看帮助手册

（5）shell：为什么在终端输入一个字符串，回车会被当成命令执行，完成某个功能？

操作系统内核与用户之间的内核----命令行解释器
用户不能直接访问系统内核----直接访问内核太危险；
操作系统会提供一些接口---系统调用接口，用户只能通过这些接口完成内核某个特定功能的访问

shell 会捕捉用户的标准输入得到字符串，通过字符串判断用户想要干什么

浏览目录这种功能会涉及很多系统调用接口的使用，为了方便用户使用，因此将系统调用接口封装了一些直接完成常用功能的程序
功能程序：被称为shell命令程序

终端能够执行命令，就是因为终端打开之后默认运行了一个**程序----shell---命令行解释程序**
shell 程序多种多样：base、dash、csh

（用户-->终端-->shell --> 内核）shell 是一个用户与内核之间的沟通桥梁，在linux 下就是一个命令行解释器

权限：限制用户权力的东西
操作系统中的操作权限：在Linux系统中将用户分为两类：　　普通用户---只能干受限的操作
　　管理员用户--为所欲为
使用su 命令切换用户---为了获取这个用户的操作权限

root ->useradd 添加用户　userdel 删除用户

（6）文件权限：

文件访问对用户的分类：
　　文件所有者（u）、文件所属组(g)、 其他用户（o）
文件访问对操作的分类：
　　可读（r）、可写（w）、可执行（x）
文件访问权限表示为：
　rwxrwxrwx 描述了三类用户各自对文件所能进行的操作
例如： **rw-rw-r--：所有者对文件可读可写不可执行，所属组成员对文件可读可写不可执行，其他用户只读、不能写也不能执行**
在系统中权限存储：使用二进制比特位图---0、1
rw-rw-r--：110 110 100　--- 占空间小操作更方便
为了方便表示和记忆：可以使用三个八进制数字分别表示三类用户权限：rw-rw-r-- 》 110 110 100 》 664

　例如：753------》111 101 011
　　u: 可读可写可执行（所有者）
　　g:可读不可写可执行（所属组）
　　o:不可读可写可执行（其他用户）
目录：可浏览 -r 、可在目录下删除创建文件 -w 、可进入-x

（7）文件访问权限指令

创建一个文件的默认权限：
umask： 查看或者设置文件的创建权限掩码也就是说掩码决定了一个文件的创建的默认权限
　-S ： 人性化显示

```
777 ---- 是shell 中 默认给定的文件权限
```

```
计算方法：777（满权限）-（减去）八进制掩码
正规计算方法：给定权限 &  （~掩码）---- 777 &（~002）（~ :取反）
```

例如：现在shell 中umask 掩码的值为 002，问创建一个文件后三个权限
777 & （~ 022）= 111 111 111 & 111 101 101=755

创建好的文件权限的修改：
chmod 777 hello.text ---- 直接用八进制数进行修改
chmod a-x hello.txt ----- 删除所有用户可执行权限
针对某用户进行权限的增删 ：
chmod [a/u/g/o/] +/- [r/w/x] filename

文件用户信息修改：
chown username filename 修改文件所有者（只能使用 root 修改）
chgrp groupname filename 修改文件所属组

文件权限的沾滞位：
特殊的权限位 --- 主要用于设置目录沾滞位，其他用户在这个目录下能够创建文件，可以删除自己的文件，但是不能删除别人的文件
chmod + t filename

二、常用工具

（一）软件包管理工具：安装其他的软件工具
yum --- 类似于手机上的应用管家
redhat系列 ----- Linux 中默认的管理工具
提供软件包的查看、安装、移除等管理操作

三板斧操作：
查看：yum list ---查看所有软件包 yum search --- 搜索指定软件包 例如：yum search gcc
安装：su root（管理员权限) ----- yum install --安装包安装
移除：(卸载) su root (管理员权限) -- yum remove --- 软件包移除

yum makecache ---- 将软件包信息保存到本地
安装常用软件工具：
编译器：
gcc/gcc-c++、编辑器：vim、
调试器：gdb、版本管理工具：git

查看版本信息： gcc-v 、gdb-v 、 git-v 、
查看常用工具是否有版本信息（是否安装）：
vim --version、gcc--version、gdb--version、git--version
(对应安装：yum install vim、yum install gcc gcc-c++、yum install gdb、yum install git)

（二）编辑器：写代码
vim -- Linux下常用的编辑器
vim 命令行编辑器，不能使用鼠标（鼠标无效）--- 在命令行中实现光标的移动、文本的操作、文本的编辑，vim 具有多种操作模式（12种），**常用三种：插入模式、普通模式、底行模式**

插入模式：进行文本数据的编辑插入
普通模式：进行文本常见操作，复制、粘贴、剪切、删除、撤销、返回、文本对齐、光标快速移动
底行模式：进行文本的保存退出，以及文本的匹配查找替换操作

vim 的基本使用：
打开文件：vim filename -- 打开一个已有文件，若没有自动创建，打开之后默认进入普通模式
注：所在目录必须具有可写权限
vim 打开文件，每次其实是打开一个临时文件，作为中间交换文件然后关闭源文件，编辑的操作都是在中间文件中完成的，只有正规退出，：wq 的时候才会将改变的数据写入源文件中并且删除

中间文件，否则中间文件存在的情况下，下次vim 打开文件时就会有提示信息（删除这个中间文件后就没有）

操作模式的切换：所有模式都是围绕普通模式进行切换的
普通模式--》插入模式：i -- 进入插入模式； a -- 光标后移一个字符进入插入模式；o -- 光标所在行下方创建新行进入插入模式
普通模式--》底行模式：使用英文 （冒号）：进入底行模式实现文本的保存和退出
底行模式--》其他模式：ESC （任何模式下按ESC键都可以返回到普通模式）

ctrl + z 暂停插入
底行模式下常见操作：
：w --- 保存 ：q ---- 退出 ：wq ---- 保存并退出
：q! -- 强制退出，不保存

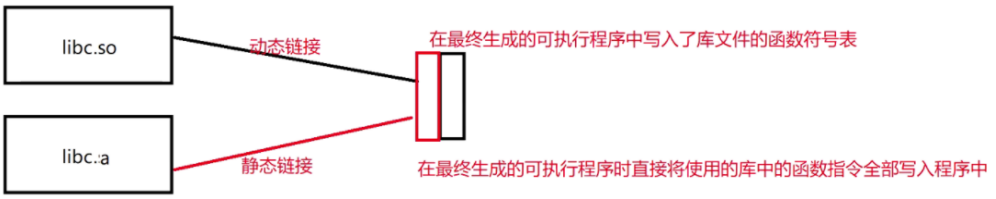
普通模式下的操作指令：
光标移动：h (上)、 j(左)、 k(右) 、 l (下)、 wb（按单词移动光标）、ctrl + f/b （上下翻页）、gg （光标自动到文档首行）、G（文档尾行）
文本操作：yy / nyy -- 复制
p/P ---- 粘贴（粘贴到下方/上方）
（例如，n=2 时候2yy :复制两行内容）
vim 中没有删除--删除就是剪切，dd/ndd （剪切，n代表剪切行数
x :删除字符；dw :删除单词
其他操作：u: 撤销上部操作，ctrl+r 撤销的反向操作

gg=G --- 全文对齐

（三）编译器：将我们所写的高级语言代码解释成机器指令集gcc：C语言编译器 g++：C++编译器
编译器作用：C、C++是高级语言，机器无法识别这些代码，需要编译器将高级语言代码解释为机器指令生成可执行程序文件才能可执行。
可执行文件：一段功能的机器指令集
编译型语言:C/C++ 脚本型：python、lua、shell
编译型语言：程序编译之后才可以执行 --- 运行性能高，但是编码较慢
脚本语言:编写完毕直接执行----逐行解释型，由解释工具逐行解释然后执行功能（不同脚本语言有不同解释器）----编码较快但是运行效率低

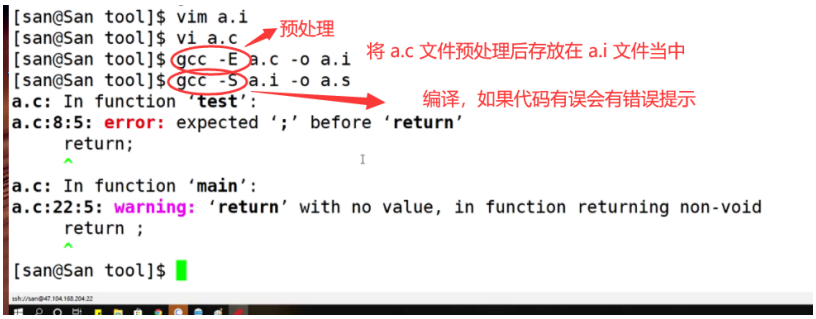
编译过程:
（1）预处理：
展开所有代码（引入头文件、宏替换、删除注释..）**gcc -E**
（2）编译：
纠正、没有错误了将代码解释成汇编代码 **gcc -S**
（3）汇编：
将汇编代码解释为机器指令
（4）链接：
将所有用到的机器指令文件打包到一起，生成可执行程序

链接库文件时的两种链接方式：(gcc默认链接：动态链接)
（1）动态链接：libc.so 链接动态库，在可执行程序中记录函数符号信息表，生成的可执行程序比较小，但是运行时需要加载动态库，多个程序可以在内存中使用同一个相同的库不会在内存中造成代码冗余
（2）静态链接：libc.a 链接静态库，直接将使用的函数实现写入可执行程序中，生成的可执行程序比较大，但是运行时不需要额外依赖加载库文件，但是如果多个程序使用了相同的静态库，运行时会在内存中造成代码冗余。



不仅仅是我们自己写的代码文件，还包括库文件
库文件：大佬已经写好的常见功能代码，打包起来的文件
例如： printf 函数 就是一个库函数，程序中如果调用printf 函数，我们的可执行程序运行时，就必须找到printf 对应的实现指令
gcc 编译器，在链接生成可执行程序时会默认链接标准C库

gcc 编译器常见操作选项：
-E：只进行预处理；-o：指定要生成的文件名称；
-S：只进行编译； -c：只进行汇编；



vi ---- 打开进入某一个文件（代码）
vnew --- 打开一个新文件窗口

未定义：告诉编译器有这个东西（变量/函数），声明 了这个变量/函数，但是编译器未找到这个东西
未声明：从未告诉编译器有这个东西就直接使用了

.h 文件--- 在C语言中自己要用于进行变量/函数的声明，告诉编译器有这个变量/函数的存在，你先编译具体变量的定义/函数的实现有可能在其他的 .c 文件中
通过 .h 文件只是为了告诉编译器有什么，编译链接时候必须把所有的 .c 文件的指令以及库文件中的指令打包到一起才行
在一个 .c 文件中引入其他 .h 文件，只是为了告诉编译器在编译这个文件时候有哪些其他文件中定义的函数/变量可以使用

（四）调试器：调试程序的运行过程 （gdb）
调试一个程序的运行过程能够让我们从运行过程中发现程序哪里有问题，可以适当的改变数据到达某种调试目的
并不是所有程序都可以进行调试，调试器只能调试具有调试符号信息的程序 --- debug 版本的程序

gcc 生成可执行程序，默认会生成 release 版本的程序，程序中没有调试符号信息，想要生成 debug 版本需要加上 -g 选项

调试一个程序的前提：
这个程序是一个 debug 版本程序
是 gcc -g a.c b.c -o main 生成的程序（a.c / b.c 是两个需要链接起来调试的文件名，生成一个新的 main 文件）

1、调试器加载程序
gdb ./main 直接使用 gdb 加载程序
./main （当前目录下的main 文件）
gdb -p 30507 直接对一个正在运行中的程序进行调试
（ -p 用于指定进程的 id ）

2、开始调试程序
run 直接运行 start 开始进行逐步调试

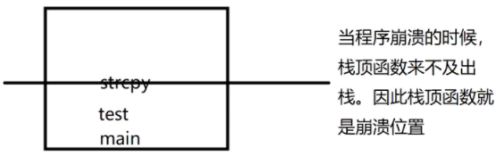
3、常用调试指令
（1）流程控制：
next / n 下一步，遇到函数直接运行，不会跟踪进入函数
step / s 下一步，遇到函数则跟踪进入函数
list / l 默认查看调试行附近代码，也可以指定行:list file :line
until: 直接运行到指定行也可以指定文件: until file :line
例如：list a.c:20 until a.c:26
continue / c 继续运行直到断点处停下

(查看 a.c 15 行 附近代码)

```
}
(gdb) list a.c:15
10              return;
```

（2）断点操作 break / b
break file :line (例如：break a.c:9)：给指定文件指定行打断点，程序运行到断点就会停下来
break function ：给函数打断点
info break ： 查看断点信息
watch a ： 给变量 a 打断点，当变量数据发生改变时停下来
delete / d ： 删除断点

（3）内存操作
print / p 查看变量数据：例如：print a
print a=10 设置变量数据
backtrace / bt 查看程序运行调用栈信息，程序一旦崩溃，查看栈可以快速定位崩溃位置---栈顶函数



for(int a=10;a<20;a++）变量 a 在使用时定义，出错的处理方法：gcc -std=c99

```
[san@San tool]$ gcc -std=c99 a.c b.c -o main
```

（五）项目的自动化构建工具：自动化的将某个项目构架成功
make / Makefile ---------- （vi Makefile）
Makefile :普通的文本文件,用于记录项目的构建规则流程
make :Makefile 解释程序，逐行解释Makefile 中的项目构建规则，执行构建指令完成项目的构建

Makefile 的编写规则：
目标对象 : 依赖对象
制表符\t 执行指令 （tab 键）

目标对象：要生成的可执行程序的名称
依赖对象：生成目标对象所需要的源码文件

make 在解释执行的时候会找到当前所在目录中的Makefile 文件，在这个文件中找到第一个目标对象，执行这个目标对象对应的指令

gcc 要是不使用 -o 指定生成文件名称，则默认生成 a.out

make 的解释执行规则：
（1）make 一旦执行，则运行make 解释程序，会在当前所在目录寻找Makefile 文件
（2）make 的执行规则中，要生成目标对象，首先要保证依赖对象已经生成，则会递归向下寻找依赖对象的生成规则
（3）在Makefile 中找到第一个目标对象，根据与依赖对象的时间关系判断是否需要重新生成（只生成第一个目标对象就会退出）
（4）若是需要重新生成，则执行对应下方的指令（指令不一定非是生成目标对象指令）
生成第一个目标对象后，make 就会退出不会生成第二个目标对象

为了生成两个目标对象-----all : 目标名1 目标名2

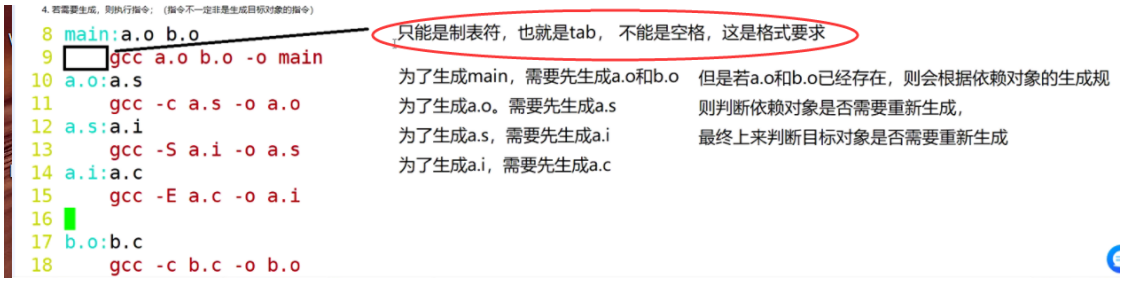
```
8 all:main main1
9
0 main:a.c b.c
1     gcc a.c b.c -o main
2 main1:a.c b.c
3     gcc a.c b.c -o main
```

为了生成 main 需要先生成 a.o 和 b.o
为了生成 a.o 需要先生成 a.s
为了生成 a.s 需要先生成 a.i
为了生成 a.i 需要先生成 a.c　　（递归过程）

（若a.o 和 b.o 已经存在，则会根据依赖对象的生成规则判断依赖对象是否需要重新生成，最终来判断目标对象是否需要重新生成）

```
 8 main:a.o b.o
 9     gcc a.o b.o -o main
10 a.o:a.s
11     gcc -c a.s -o a.o
12 a.s:a.i
13     gcc -S a.i -o a.s
14 a.i:a.c
15     gcc -E a.c -o a.i
16
17 b.o:b.c
18     gcc -c b.c -o b.o
```

```
[san@San tool]$ ls
a.c  b.c  b.h  Makefile
[san@San tool]$ make
gcc -E a.c -o a.i
gcc -S a.i -o a.s
gcc -c a.s -o a.o
gcc -c b.c -o b.o
gcc a.o b.o -o main
[san@San tool]$
```



Makefile 编写规则进阶：

make 中的预定义变量：预定义变量通常是在执行指令中使用

$@：表示目标对象

$^：表示所有依赖对象

$<：表示所有依赖对象中的第一个依赖对象

```
main:a.o b.o
    gcc $^ -o $@
a.o:a.s
    gcc -c $^ -o $@
a.s:a.i
    gcc -S $^ -o $@
a.i:a.c
    gcc -E $^ -o $@

b.o:b.c
    gcc -c $^ -o $@
```

或：% 通配符，匹配识别任意字符串

```
main:a.o b.o
    gcc $^ -o $@        % :通配符
%.o:%.c
    gcc -c $< -o $@
```

wildcard / patsubst 关键字的使用： **(shell 中 # 注释)**

```
src=$(wildcard ./*.c) #从当前目录下找到以.c结尾的文件名称，放到src变量中
#$src = a.c b.c
obj=$(patsubst %.c, %.o, $(src)) #将src变量中的数据，后缀名由.c修改为.o将结果赋值给obj变量
#$obj = a.o b.o
main:$(obj)
    gcc $^ -o $@
%.o:%.c
    gcc -c $< -o $@
#makefile中的清理动作,这个对象是第二个目标对象，不会自动执行，需要我们制定执行
#声明伪对象，伪对象不管如何每次都要重新生成
.PHONY:clean
clean:
    rm -rf main $(obj)
```

执行时 make 默认只执行第一个main 文件，要想指定执行第二个 clean 文件需要输入 make clean ,如果当前文件中存在clean 文件则需要在vim 中声明伪代码

伪对象的使用：

.PHONY : 目标对象名称

伪对象的作用：不管对象是否最新每次都要重新生成

（六）项目版本管理工具：可以实现项目的回滚、合并等管理操作

git ： 项目的版本管理工具

版本管理工具：对项目的开发周期进行管理，每一次提交的修改都会有对应的版本号，能够让我们在程序出问题的时候回滚回去

1、从远程仓库克隆到本地： git clone https连接路径

2、本地提交修改信息：git add ./*
3、提交本地版本管理：git commit -m "删除了一个无效文件"
4、将本次版本提交到远程服务器仓库：
git push origin master
origin ---- 用于指定分支名称
master ---- 主分支

> 项目的版本管理工具：git
> 三板斧操作： git clone-克隆远程仓库到本地； git add --all 目录名称 ---记录修改信息； git commit -m "备注" --提交本地版本
> git push - 提交版本到远程仓库

git config -- g lobal user.name
git config -- g lobal user.emil

三、Linux 系统编程：

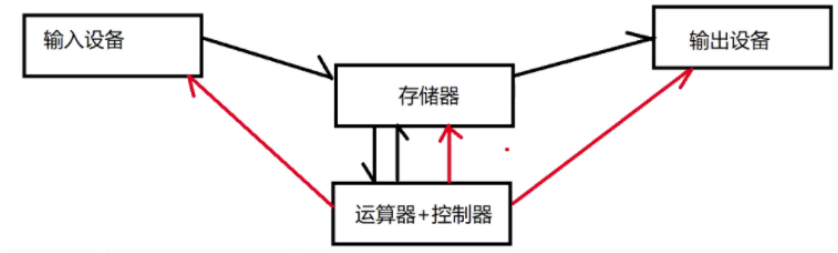
进程的各个操作----以及所实现的功能，以及实现的原理流程

（一）进阶概念：深入了解什么是进程

冯诺依曼体系结构----现代计算机的硬件体系结构

现代计算机的硬件体系结构：计算机应该包含五大硬件单元：
输入设备：采集数据，例如键盘、网卡接受网络中的数据
输出设备：进行数据输出，例如显示器、网卡向网络发送数据
存储器：进行中间数据缓冲，
运算器：进行数据运算
控制器：进行设备控制
运算器+控制器=CPU中央处理器

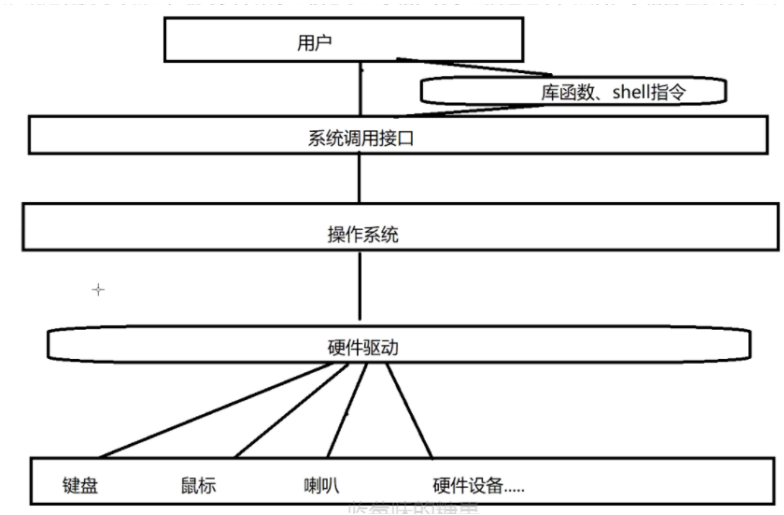
所有设备都是围绕存储器工作的------
CPU 不会直接从输入设备获取数据进行处理，而是先把数据放到存储器中，CPU 从存储器中获取数据处理
CPU不会直接将数据交给输出设备进行输出，而是先把数据存放到存储器中，控制输出设备从存储器中获取输出数据



存储器实际上就是内存，为什么不是硬盘?
因为硬盘的数据吞吐量太低了：机械--200MB/s
内存的数据吞吐量：是硬盘的数十倍
内存速度这么快，然而内存只用于缓冲不使用内存存储数据而是使用硬盘存储？ 因为硬盘与内存的存储介质不同，内存是易失性介质，数据掉电就会丢失，而硬盘掉电后数据不会丢失。

连续读取
L1CACHE大概是100~500GB/s的水平
L3CACHE 大概是10~50G/s的水平
DDR4内存大概是3GB/s的水平
nvme ssd大概是2000MB/s的水平
SATA ssd大概是450MB/s的水平
机械硬盘大概是100~150MB/s的水平
随机读写不清楚，总之也是差n个数量级，只会比连续读写差距更大。

（1）操作系统：
一个搞管理的软件（安装在计算机上的一个程序），管理计算机上的软硬件资源
操作系统：内核（负责系统的核心功能---软件管理、硬件管理、内存管理、文件管理.....）+ 外部应用
用户不能直接访问系统内核（危险太高），因此为了控制风险----系统让干什么才能干什么
操作系统向外提供访问内核的接口----每个接口完成的功能都是固定的---系统调用接口
操作系统如何进行软硬件的管理：先描述在组织



系统调用接口：操作系统向用户提供的访问内核的接口
库函数与系统调用接口的关系：库函数封装了系统调用接口

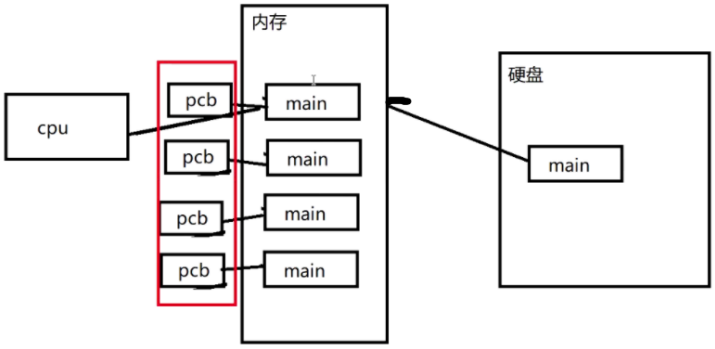
（2）进程：运行中的程序
一个程序运行起来，有数据以及指令需要被CPU 执行处理，根据冯诺依曼体系结构---CPU不会直接去硬盘上找到程序文件执行处理，需要先将程序数据信息加载到内存中，然后CPU从内存中获取数据以及指令进行执行处理----程序运行会被加载到内存中

操作系统中进程都是同时运行的，CPU只有一个如何做到多个程序同时运行?
CPU的分时机制：实现系统同时运行多个程序的技术
CPU只负责执行指令，处理数据（处理哪个程序其实CPU并不关心），因此操作系统的进程管理就体现出来，对程序的运行调度进行管理。
CPU进程程序运行处理的时候，并不会一次性将一个程序运行完毕才会运行下一个，而是每个程序都只运行一段很短的时间（给一个程序分配一个时间片），时间片运行完毕则由操作系统进行调度，让另一个程序的代码数据在CPU上进行处理。
CPU 分时机制实现CPU轮询处理每一个运行中的程序，而程序运行调度由操作系统进行管理。
管理思路:操作系统将每一个程序的运行信息保存下来，进行调度管理的时候才能知道这个程序上一次运行到了哪里
程序的指令、数据在内存中是死的，谈不上进程-运行中的程序

操作系统通过对一个程序运行的描述，让一个程序运行起来，让一个程序动起来，对于操作系统来说，进程就是PCB，是一个程序运行的**动态描述**，通过PCB 才能实现程序的运行调度管理

PCB 描述信息：
linux 下的PCB 就是一个 struct task_struct 结构体
内存指针--能够让操作系统调度程序运行的时候，知道进程对应的指令以及数据在内存中的位置
上下文数据--程序运行过的指令和数据，程序运行中的指令和数据，程序即将执行的指令和数据，操作调度切换进程运行的时候能够让CPU这个进程接着切换之前继续运行，继续处理之前没有处理完的数据，切换时保存正在执行的指令以及数据信息
程序计数器--也属于上下文数据，保存的就是指令位置，切换回来知道从哪里继续运行
标识符--能够让操作系统识别唯一的运行中的程序
（IO信息、状态信息、优先级、记账信息）

> 对于校长管理学生来说，学生的学籍信息就是学生
> 对操作系统来说，管理程序的运行，就将程序的运行描述起来，然后组织起来进行管理，描述的运行信息对于操作系统来说就是运行中的程序就是进程。



调度：
操作系统将程序的信息放到CPU寄存器中，让CPU直到执行什么指令、处理什么数据
若多个程序运行，CPU分时机制实现程度的切换运行----如何切换--这就是调度
CPU调度：分配CPU资源给进程--就是运行这个进程，管理CPU资源
资源调度站：分配资源、管理资源
CPU调度---分配CPU资源，切换运行程序

并发：CPU资源不够的情况下，采用CPU分时机制任务轮询处理
并行:多核CPU上，多个进程同时占据CPU进行数据处理

（进程部分复习）

五大硬件单元：输入设备，输出设备，存储器，运算器，控制器
所有的设备都是围绕内存工作，内存在各个设备之间起到一个数据缓冲作用
操作系统：一个在计算机上管理软硬件资源的软件
操作系统：系统内核+外部应用
管理：先描述，再组织
系统调用：系统内核向外提供访问内核的接口。； 系统调用接口有时候并不太好用，大佬们针对典型功能对系统调用接口进行封装--库函数
库函数和系统调用接口的关系：上下级的封装调用关系
进程概念：
进程：运行中的程序，但是在操作系统的层面，进程就是PCB，是操作系统对一个程序运行的动态描述，通过这些描述让程序运行起来，实现操作系统对程序运行的调度管理。
PCB描述信息：pcb在linux下是task_struct结构体，进程标识符，内存指针，上下文数据，程序计数器
cpu分时机制：将cpu处理程序运行的过程划分成为时间片，每个程序在cpu上只运行一段很短的时间，时间片完毕后切换下一个进程
并发：cpu资源不够的情况下，采用cpu分时机制，任务轮询处理
并行：多核cpu上，多个进程同时占据cpu进行数据处理。

（3）进程状态
每个进程PCB中都会描述一个运行的状态信息，通过状态信息告诉操作系统这个进程现在应该干什么
时间片：操作系统给每一个程序分配的CPU处理时间，时间片运行完了就切换另一个程序，实际上就是CPU与运行处理时间段，一个进程拿到时间片表示分配到了CPU的处理时间段
课本中的状态：就绪---拿到时间片就能运行、

运行---正在被CPU处理执行、
阻塞---暂时不满足某些条件则不允许

查看进程信息：ps　　；
ps -ef：　ps -ef | grep
ps -aux :查看更详细的信息
前台进程：当前占据了终端的进程
fg + 进程名：将后台进程提到前台运行

Linux 中对状态更加细分:
运行状态 R：包含就绪以及运行，也就是正在运行的，以及拿到时间片就能运行的都称之为**运行状态**，操作系统遇到pcb 这个状态就会调度运行；
休眠：暂时不需要CPU调度运行，让出CPU资源，休眠也有唤醒条件，操作系统调度程序运行时会查看状态，如果是休眠就会看唤醒条件是否满足，如果满足则置为运行状态进行处理，如果不满足则切换下一个进程
停止 T：停止与休眠不一样（休眠操作系统会去查看进程唤醒条件是否满足，而停止是只能手动换醒）
可中断休眠状态 S：可以被打断的休眠，通常的满足运行条件或被一些中断打断休眠之后进入运行状态
不可中断休眠状态 D：只能通过满足条件自然醒，进入运行状态，不会被一些中断打断休眠状态
僵尸状态 Z：描述的是一个进程退出了，但是进程资源没有被完全释放，等待处理的一种状态

进程创建：pid_t fork(void) ----通过复制调用进程，创建一个新的进程　　[例如：　pid_t pid=fork()]
进程就是PCB，创建一个进程就是创建一个PCB，复制了调用fork 的这个进程PCB 的信息（内存指针、程序计数器、上下文数据）
这个新的进程，运行的代码与调用fork 的进程一样并且运行位置也相同

僵尸进程：处于僵尸状态的进程，退出了但是资源没有完全被释放的进程
　危害：资源泄漏，可能会导致正常进程起不来）
　产生：一个进程先于父进程退出，父进程没有关注子进程退出状态导致子进程资源无法被释放，进入僵尸状态
　解决：进程等待-----进程控制讲---一直关注子进程，退出了就能直接发现
　孤儿进程：父进程先于子进程退出，子进程就会成为孤儿进程
　特性：让出终端进入系统后台运行，并且父进程成为 1号进程
　（守护进程、精灵进程）
1 号进程，在 centos 7 之前，叫 init 进程，在 centos 7之后就叫做 systemd ，系统中父进程是 1 号进程，通常会以 d 结尾，表示自己是个在后台默默运行服务

ulimit -a ： 查看用户创建的最大进程数

```c
#include <stdio.h>
#include <unistd.h>

int main (int argc, char *argv[])
{
    pid_t pid = fork();//创建子进程
    //因为子进程复制了父进程，因此子进程与父进程运行的代码以及运行的位置都是一样的
    //从代码运行的角度来看，就都是从fork函数之后开始运行的
    //因为父子进程运行的代码数据都一样，因此无法直接分辨，只能通过返回值判断
    //对于父进程fork返回值>0;  对于子进程返回值==0
    //虽然父子进程代码相同，但是因为返回值不同，因此会各自进入不同的判断执行体
    if (pid > 0) {
        //this is parents
        printf("this is parents:%d\n",getpid());
    }else if (pid == 0) {
        //this is child
        printf("this is child:%d\n");
    }else {
        //error
        printf("error\n");
    }

-- INSERT --
```

进程创建：pid_t fork(void)--通过复制调用进程，创建一个新的进程

进程就是pcb，创建一个进程就是创建了一个pcb，复制了调用fork的这个进程pcb的信息（内存指针、程序计数器、上下文数据）

这个新的进程，运行的代码与调用fork的进程一样，并且运行位置也相同。

两个进程运行的程序相同：哪个是调用进程（父进程），哪个是新建进程（子进程）

在父进程中返回子进程的pid，是大于0的；　在子进程中返回0；返回-1表示创建子进程失败　通过返回值可以分辨父子进程

./test　运行程序，程序代码数据会被加载到内存中，操作系统创建pcb实现对test程序运行的调度管理

fork创建子进程之后，父子进程谁先运行，不一定，大家都是pcb，操作系统调度到谁谁就运行

fork 创建子进程之后，父子进程谁先运行不一定，大家都是PCB，操作系统调度到谁谁就运行

子进程与父进程的工作一样：如果有任务子进程运行若干崩溃子进程会崩溃，父进程不会崩溃

```c

int main()
{
    pid_t pid = fork();
    if (pid > 0) {
        //parents
        printf("parents:%d\n", getpid());
        sleep(3);
        exit(0);
    }else if (pid == 0) {
        //child
        printf("child:%d\n", getpid());
    }else {
        //error
        exit(0);
    }
    printf("----------------------%d\n", getpid());
    while(1) {
    sleep(10000);//因为父进程直接exit退出，因此只有子进程会调用sleep（）
    }
    return 0;
}
- INSERT --
```

```
[san@San progress]$ vi test.c
[san@San progress]$ make
make: `fork' is up to date.
[san@San progress]$ make test
gcc test.c -o test
[san@San progress]$ ./test
parents:18374
child:18375
----------------------18375
[san@San progress]$
```

exit 退出一个进程，谁调用谁退出

（4）环境变量：终端 shell 中进行系统运行环境配置的变量

作用：

（1）可以使系统环境配置更加灵活（不像修改配置文件后还得加载配置问题）

（2）可以通过环境变量向子进程传递数据

操作指令：env 查看所有环境变量

echo--直接打印某个变量的内容

例如：echo $PATH ($ 表示后边的字符串是一个变量名称)

set --- 查看所有变量，不止环境变量

export --- 声明定义转换环境变量

例如：export PATH=$PATH:

unset --- 删除变量，包括环境变量

注意：shell 中的普通变量可以起到环境配置的作用，但是无法进行数据传递

典型环境变量：PATH --- 存储程序默认的搜索运行路径

运行一个程序时候，如果没有指定程序路径，只有程序名称，则shell 会去PATH 环境变量保存的路径中去找这个程序，若过没有找到就报错，没有这个命令，如果将我们自己的程序所在路径加入到PATH 环境变量中，则这个程序就可以直接运行了

代码操作：

char *getenv (char *key) 通过环境变量名称获取变量内容

在终端中运行的程序，父进程都是bash，也就是shell 程序；
一个终端打开默认运行一个程序--shell程序--命令行解释器，当shell 捕捉到字符串，就会到PATH 环境变量指定的路径下去找到对应名称的程序，进而执行这个程序完成功能(其实是shell 程序创建了一个进程，让这个进程去调度我们要运行的程序)
在终端中运行程序所创建的进程都是shell 创建的，也就是说，我们运行的程序的父进程都是shell

(获取环境变量，三种)

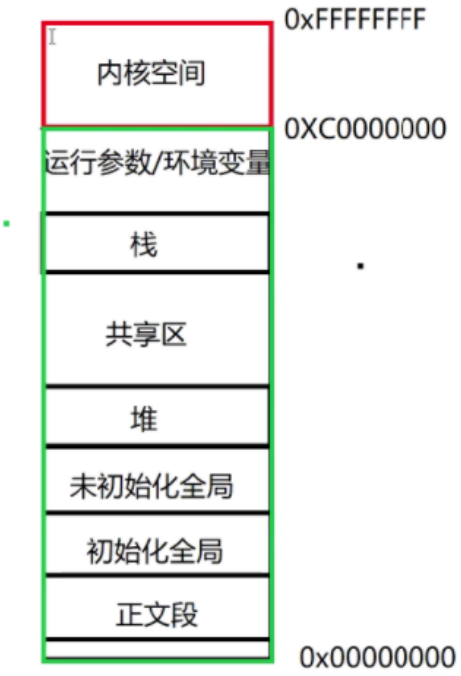
```c
int main (int argc, char *argv[], char *env[])
{
    //getenv(),通过环境变量名称获取环境变量数据，通过返回值返回，若没有这个环境变量返回NULL
    char * tmp = getenv("MYVAL");
    if (tmp == NULL) {
        printf("have no env var:MYVAL\n");
    }else {
        printf("MYVAL:[%s]\n", tmp);
    }

    //int i;
    //for(i = 0; env[i] != NULL; i++) {
    //    printf("environment: [%s]\n", env[i]);
    //}

    extern char **environ;//声明变量，注意这个变量在库中有，只需要声明就能使用
    int i;
    for(i = 0; environ[i] != NULL; i++) {
        printf("environment: [%s]\n", environ[i]);
    }
```

```c
int main (int argc, char *argv[], char *env[])
{
    //argc 是程序运行参数的个数
    //argv[]  是用于指向程序中各个环境变量的一个字符串指针数组
    //./env -l -a    [./env]就是第0个参数；  [-l]就是第一个参数 [-a]就是第二个参数
    //env[]   是用于指向程序中各个环境变量的一个字符串指针数组
```

main 函数的第三个参数
    int main(int argc,char *argv[] ,char *env[])
extern char **environ ; 库中的全局变量，每个节点指向一个环境变量，但是使用时需要声明，告诉编译器有这个变量

（5）程序地址空间

32位操作系统最大支持的内存大小就是4G，因为寻址空间只有这么大（指针大小只有 4 个字节）



（二）进程控制：多进程的实现
（三）基础IO：进程中IO操作
（四）进程间通信：进程间的数据通信
（五）进程信号：进程中信号的操作
（六）多线程：进程中多线程的操作