Linux

- **1、Linux介绍**
  - Linux是类Unix计算机操作系统的统称。
  - Linux操作系统的内核的名字也是“Linux”。
  - Linux这个词本身只表示Linux内核，但在实际上人们已经习惯了用Linux来形容整个基于Linux内核的系统。
  - Linux是由芬兰大学Linus Torvalds于1991年编写的。
- **2、Linux发行版组成**
  - Linux内核
  - 应用软件
    - 一些GNU程序库和工具
      - emacs
      - GCC
      - GNOME
    - 命令行shell
    - 图形桌面环境
      - KDE（qt编写）
      - GNOME（GTK编写）
      - Unity
    - 一些办公软件
      - OpenOffice
    - 编译器
      - gcc
      - g++
    - 文本编辑器到科学工具的应用软件
      - vi
      - gedit
- **3、Linux版本**
  - 商业公司维护的发行版本
    - **RedHat**（REHL）包管理方式采用基于RPM包的YUM包管理方式
  - 社区组织维护的发行版本
    - **Debian** 包管理方式采用的是：apt-get/dpkg xxx.deb
  - RedHat系列
    - **RHEL**(RedHat Enterprice Linux, 也就是所谓的RedHat Advance Server收费版本)，稳定。
    - **CentOS**(RHEL的社区克隆版本，免费)，稳定。
    - **FedoraCore**(由原来的RedHat桌面版本发展而来，免费版本)，稳定性差，最好只用于桌面应用。
  - Debian系列
    - **Debian**
    - **Ubuntu**
      - 命名规则
        - 前两位数字：发行时的年份的最后两位数字
        - 后两位数字：发行的月份
      - 版本
        - 桌面版（Desktop）：至少三年的技术支持
        - 服务器版（Server）：至少五年的技术支持
        - LTS版本（Long Term Support）
      - 版本发布频率，一年两次
        - 主版本号（年份）
          - 单数年：短期支持版
          - 双数年：长期支持版（LTS）
        - 副版本号（月份）
          - 四月份（xx.04）：该年度的稳定版
          - 十月份（xx.10）：该年度的测试版

学习方法

- **要求**
  - 记好课堂笔记
  - 只听不练学不会Linux，多动手实践
  - 课上跟上思路，简单的命令跟着敲，多思考
  - 分清主次，重点记忆重要命令
- **资料**
  - Linux自带帮助文档 - man
  - 鸟哥和私房菜

一、文件和目录操作

- **1、基本的shell操作**
  - shell命令行快捷键
    - 编辑命令

```
Ctrl + a          移到命令行首，ahead，与home键功能相同
Ctrl + e          移到命令行尾，end， 与end键功能相同
Ctrl + f          按字符前移（右向），forward，与方向键left功能相同
Ctrl + b          按字符后移（左向），back，与方向键right功能相同
Alt  + f          按单词前移（右向）
Alt  + b          按单词后移（左向）
Ctrl + xx         在命令行首和光标之间移动
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
Ctrl + w          从光标处删除至字首
Alt  + d          从光标处删除至字尾
Ctrl + d          删除光标处的字符，与delete键功能相同
Ctrl + h          删除光标前的字符，与backspace键功能相同
Ctrl + y          粘贴至光标后，yank
Alt  + c          从光标处更改为首字母大写的单词
Alt  + u          从光标处更改为全部大写的单词，uppercase
Alt  + l          从光标处更改为全部小写的单词，lowercase
Ctrl + t          交换光标处和之前的字符
Alt  + t          交换光标处和之前的单词
Alt  + Backspace  与 Ctrl + w 类似，分隔符有些差别
```

    - 重新执行命令

```
Ctrl + r          逆向搜索命令历史
Ctrl + g          从历史搜索模式退出
Ctrl + p          历史中的上一条命令，previous，与方向键up功能相同
Ctrl + n          历史中的下一条命令，next， 与方向键down功能相同
Alt  + .          使用上一条命令的最后一个参数
```

    - 控制命令

```
Ctrl + l          清屏，clear
Ctrl + o          执行当前命令，并选择上一条命令
Ctrl + s          阻止屏幕输出
Ctrl + q          允许屏幕输出
Ctrl + c          终止命令
Ctrl + z          挂起命令
```

    - Bang (!) 命令

```
!!                执行上一条命令
!blah             执行最近的以 blah 开头的命令，如 !ls
!blah:p           仅打印输出，而不执行
!$                上一条命令的最后一个参数，与 Alt + . 相同
!$:p              打印输出 !$ 的内容
!*                上一条命令的所有参数
```

```
!*:p            打印输出 !* 的内容
^blah           删除上一条命令中的 blah
^blah^foo       将上一条命令中的 blah 替换为 foo
^blah^foo^      将上一条命令中所有的 blah 都替换为 foo
```

- 虚拟终端(VT)
  - 简介

```
英文：Virtual Terminal
解释：连接在远地的分时共用计算机系统的远程终端，它具有使用户感到是在计算机旁使用终端的功能。
```

  - 常用工具

```
XManager
Putty
SecureCRT
```

- 命令和路径补齐

```
tab键按一次       当只有一个匹配时，自动补全命令或路径
tab键按两次       当有多个匹配时，显示命令或路径列表
```

- **2、Linux 标准目录结构**

初学Linux，首先需要弄清Linux 标准目录结构：

```
/
    root --- 启动Linux时使用的一些核心文件。如操作系统内核、引导程序Grub等。
    home --- 存储普通用户的个人文件
        ftp --- 用户所有服务
        httpd
        samba
        user1
        user2
    bin --- 系统启动时需要的执行文件（二进制）
    sbin --- 可执行程序的目录，但大多存放涉及系统管理的命令。只有root权限才能执行
    proc --- 虚拟，存在linux内核镜像；保存所有内核参数以及系统配置信息
        1 --- 进程编号
    usr --- 用户目录，存放用户级的文件
        bin --- 几乎所有用户所用命令，另外存在与/bin，/usr/local/bin
        sbin --- 系统管理员命令，与用户相关，例如，大部分服务器程序
        include --- 存放C/C++头文件的目录
        lib --- 固定的程序数据
        local --- 本地安装软件保存位置
        man --- 手工生成的目录
        info --- 信息文档
        doc --- 不同包文档信息
        tmp
        X11R6 ---  该目录用于保存运行X-Window所需的所有文件。该目录中还包含用于运行GUI要的配置文件和二进制文件。
        X386  --- 功能同X11R6，X11 发行版5 的系统文件
    boot --- 引导加载器所需文件，系统所需图片保存于此
    lib --- 根文件系统目录下程序和核心模块的公共库
        modules --- 可加载模块，系统崩溃后重启所需模块
    dev --- 设备文件目录
    etc --- 配置文件
        skel --- home目录建立，该目录初始化
        sysconfig --- 网络，时间，键盘等配置目录
    var
        file
        lib --- 该目录下的文件在系统运行时，会改变
        local --- 安装在/usr/local的程序数据，变化的
        lock --- 文件使用特定外设或文件，为其上锁，其他文件暂时不能访问
        log --- 记录日志
        run --- 系统运行合法信息
```

```
    spool --- 打印机、邮件、代理服务器等假脱机目录
    tmp
    catman --- 缓存目录
mnt --- 临时用于挂载文件系统的地方。一般情况下这个目录是空的。
        在我们将要挂载分区时在这个目录下建立目录，再将我们将要访问的设备挂载在这个目录上，这样我们就可访问文件了。
tmp --- 临时文件目录，系统启动后的临时文件存放在/var/tmp
lost+found --- 在文件系统修复时恢复的文件
```

/

```
根目录，一般根目录下只存放目录，不要存放文件。
/etc、/bin、/dev、/lib、/sbin应该和根目录放置在一个分区中。
```

/bin, /usr/bin

```
可执行二进制文件的目录，如常用的命令ls、tar、mv、cat等。
```

/boot

```
放置linux系统启动时用到的一些文件。/boot/vmlinuz为linux的内核文件，以及/boot/gurb。建议单独分区，分区大小100M即可。
```

/dev

```
存放linux系统下的设备文件，访问该目录下某个文件，相当于访问某个设备，常用的是挂载光驱mount:
/dev/cdrom、/mnt
```

/etc

```
系统配置文件存放的目录，不建议在此目录下存放可执行文件。
重要的配置文件有：/etc/inittab、/etc/fstab、/etc/init.d、/etc/X11、/etc/sysconfig、/etc/xinetd.d。
修改配置文件之前记得备份。

注：/etc/X11存放与x windows有关的设置。
```

/home

```
系统默认的用户家目录，新增用户账号时，用户的家目录都存放在此目录下。
~表示当前用户的家目录，~test表示用户test的家目录。
建议单独分区，并设置较大的磁盘空间，方便用户存放数据。
```

/lib，/usr/lib，/usr/local/lib

```
系统使用的函数库的目录，程序在执行过程中，需要调用一些额外的参数时需要函数库的协助，比较重要的目录为/lib/modules。
```

/lost+fount

```
系统异常产生错误时，会将一些遗失的片段放置于此目录下，通常这个目录会自动出现在装置目录下。
如加载硬盘于/disk 中，此目录下就会自动产生目录/disk/lost+found
```

/mnt，/media

```
光盘默认挂载点，通常光盘挂载于/mnt/cdrom下，也不一定，可以选择任意位置进行挂载。
```

/opt

```
给主机额外安装软件所摆放的目录。
如：FC4使用的Fedora 社群开发软件，如果想要自行安装新的KDE 桌面软件，可以将该软件安装在该目录下。
以前的 Linux 系统中，习惯放置在 /usr/local 目录下。
```

/proc

```
此目录的数据都在内存中，如系统核心，外部设备，网络状态，由于数据都存放于内存中，所以不占用磁盘空间。
```

比较重要的目录有/proc/cpuinfo、/proc/interrupts、/proc/dma、/proc/ioports、/proc/net/*等。

/root

系统管理员root的家目录，系统第一个启动的分区为/，所以最好将/root和/放置在一个分区下。

/sbin，/usr/sbin，/usr/local/sbin

放置系统管理员使用的可执行命令，如fdisk、shutdown、mount等。
与/bin不同的是，这几个目录是给系统管理员root使用的命令，一般用户只能"查看"而不能设置和使用。

/tmp

一般用户或正在执行的程序临时存放文件的目录,任何人都可以访问,重要数据不可放置在此目录下。

/srv

服务启动之后需要访问的数据目录，如www服务需要访问的网页数据存放在/srv/www内。

/usr

应用程序存放目录。
/usr/bin 存放应用程序。
/usr/share 存放共享数据。
/usr/lib 存放不能直接运行的，却是许多程序运行所必需的一些函数库文件。
/usr/local 存放软件升级包。
/usr/share/doc 系统说明文件存放目录。
/usr/share/man 程序说明文件存放目录。
使用 man ls时会查询/usr/share/man/man1/ls.1.gz的内容。
建议单独分区，设置较大的磁盘空间。

/var

放置系统执行过程中经常变化的文件。
/var/log 随时更改的日志文件。
/var/log/message 所有的登录文件存放目录。
/var/spool/mail 邮件存放的目录。
/var/run 程序或服务启动后，其PID存放在该目录下。
建议单独分区，设置较大的磁盘空间。

- **3、文件和目录操作相关命令**
  - 用户目录
    - 绝对路径：从根目录开始写，如/usr/bin
    - 相对路径：相对于当前的工作目录而言
      - **.** 当前目录
      - **..** 当前目录的上一级目录
      - **-** 在临近的两个目录直接切换
    - root@ubuntu:~#
      - root: 当前用户名
      - @：at, 在
      - ubuntu：主机名
      - ~ : 当前用户home目录
      - #：超级用户
      - $：普通用户
  - **ls**

```
ls -al
    -a, --all           不隐藏任何以. 开始的项目
    -l                  使用较长格式列出信息
```

```
-rw-r--r--  1 super root     3771 6月   7  2016 .bashrc
drwx------ 22 super root     4096 12月 15 08:02 .cache
lrwxrwxrwx  1 super root       31 11月 23 07:42 .vimrc -> /home/super/.spf13-vim-3/.vimrc

以最后一个为例：
lrwxrwxrwx
第1个字符：表示文件类型
    -   普通文件
    d   目录
    l   链接符号
    b   块设备
    c   字符设备
    s   socket文件
    p   管道
第2-4个字符：文档所有者权限
第5-7个字符：同组用户权限
第8-10个字符：其他人权限
r: read 4
w: write 2
x: execute 1

1               文件的硬链接数
super           该文件或目录的所有者
root            该文件或目录所属的组
31              占用的存储空间
11月 23 07:42   文件最后创建或修改的时间
.vimrc          文件名
```

- **cd**

```
cd ~        切换到用户home目录
cd /        切换到根目录
cd          切换到用户home目录
cd..        切换到上一级目录
```

- **tree**

```
tree path   显示指定路径的目录树结构
```

- **文件或目录颜色一般情况**

```
白色        普通文件
蓝色        目录
绿色        可执行文件
红色        压缩文件
青色        链接文件
黄色        设备文件
灰色        其他文件
```

- **mkdir**

```
mkdir dirname               创建目录。
mkdir dir1/dir2/dir3 -p     按层次创建目录。
```

- **rmdir**

```
rmdir dirname       目录必须为空才能删除，所以实际使用时不常用。
```

- **rm**

```
rm -rf dirname      递归强制删除文件夹下的目录和文件，实际使用中常用。
rm -ri dirname      递归删除并提示用户删除了哪些目录或文件。
```

- **touch**

```
touch test.txt      创建文件。
```

- **cp**

```
cp hello.c temp     在当前目录下生成一个temp文件，并把hello.c的内容写入文件。
                    如果temp不存在，则创建文件；如果存在，则覆盖已有文件。

cp mydir newdir -r  在当前目录下生成一个newdir目录，并把mydir目录里面的目录或文件拷贝过去。
                    如果目录不存在，则创建；如果存在，则mydir目录本身也拷贝到newdir目录下。
```

- **cat**

```
cat test.txt        查看文件的内容，缺点内容太多时查看困难。
```

- **more**

```
more stdio.h        分屏查看文件，Enter下翻一行，Space下翻一页，缺点不能往回看。
```

- **less**

```
less stdio.h        分屏查看文件。比more更实用，可以来回翻页。
                    ctrl + p    上翻一行
                    ctrl + n    下翻一行
                    ctrl + b    上翻一页
                    ctrl + f    下翻一页
                    q           退出
```

- **head**

```
head -5 stdio.h     查看前五行内容，不加参数默认是10行。
```

- **tail**

```
tail -5 stdio.h     查看后5行内容， 不加参数默认是10行。
```

- **mv**

```
mv test.c hello.c   移动到。多用于重命名目录或文件。
```

- **ln**

```
ln -s ~/.vim/.vimrc .vimrc      创建软链接，相当于windows下的快捷方式,可用于目录或文件。
                                路径要用绝对路径，这样软链接拷贝到哪里都是有效的。
                                如果删除原文件，则软链接失效。

ln .vimrc .vimrc.hard           创建硬链接，只能用于文件。不必使用绝对路径。
                                创建后生成的硬链接与原文件引用同一个inode，硬链接本身不占硬盘空间。
                                如果删除原文件，硬链接仍然有效。
```

- **wc**

```
wc test.c       统计文件的行数，字数，字节数。
```

- **od**

```
od -tx test.c       以十六进制形式查看文件。
od -tc test.c       以ASCII字符形式查看文件。
od -td test.c       以有符号十进制形式查看文件。
od -tf test.c       以浮点数形式查看文件。
od -to test.c       以八进制形式查看文件。
od -tu test.c       以无符号十进制形式查看文件。
```

- **du**

```
du -h       查看当前文件夹下的目录或文件所占用磁盘的大小。
```

- **df**

```
df -h       查看当前文件夹下的目录或文件所使用磁盘空间的详细情况。
```

- **which**

```
which ls    查看外部命令的位置，内部命令无法查看。
```

- **pwd**

```
pwd         查看当前所在目录
```

- **4、文件权限、用户、用户组相关命令**
  - 查看当前用户

```
whoami      查看当前用户名。
id          查看当前用户信息，如用户名，所属群组，UID，GID等。
```

  - 修改目录或文件权限
    - 文字设定法

```
chmod [who] [+|-|=] [mode]
who : 不指定的话，默认是所有
    文件所有者：u
    文件所属组：g
    其他人：o
    所有人：a
mode
    r : 读
    w : 写
    x : 执行
chmod o+w temp      给其他人加写权限。
chmod a-x temp      给所有人去除执行权限。
chmod a=r temp      所有人只有读权限。
```

    - 数字设定法

```
-   0   没有权限
r   4   读
w   2   写
x   1   执行

chmod 755 temp      -rwxr-xr-x
chmod -001 temp     -rwxr-xr--
```

  - 修改目录或文件所有者与所属组

```
chown root temp         把文件temp的所有者修改为root。
chown super:root temp   把文件temp的所有者修改super,所属组修改为root。
```

  - 修改目录或文件所属组

```
chgrp root temp         修改文件所属组为root。
```

  - 目录必须有执行权限，否则不能打开目录。

- **5、查找和检索相关命令**
  - 按文件属性查找
    - 文件名

```
find ~ -name "hello.c"      按完整名称匹配
find ~ -name "*.h"          *代表任意长度字符
find ~ -name "tem?"         ? 代表一个任意字符
```

    - 文件大小

```
find ~ -size +10k               查找大于10k的文件
find ~ -size -10M               查找小于10M的文件
find ~ -size +10M -size -100M   查找大于10M小于是100M的文件
```

    - 文件类型

```
find ~ -type p      查找类型为管道的档案
find ~ -type f      查找类型为文件的档案
find ~ -type d      查找类型为目录的档案
find ~ -type l      查找类型为链接的档案
```

  - 按文件内容查找

```
    grep -r "stdio.h" ~     在用户家目录中查找文件内容里包括stdio.h的所有文件
```

- **6、软件安装和卸载**
  - 在线安装
    - apt-get

```
安装      apt-get install tree        在线下载安装
移除      apt-get remove tree
更新      apt-get update              更新软件列表
清理      apt-get clean               清理所有软件安装包，实际上清理的是/var/cache/apt/archives目录下的.deb文件
```

    - aptitude

```
安装       aptitude install tree
重新安装    aptitude reinstall tree
更新       apt-get update
移除       aptitude remove tree
显示状态    aptitude show tree
```

  - deb包安装
    - 安装

```
dpkg -i xxx.deb
```

    - 删除

```
dpkg -r xxx
```

  - 源码安装

```
1）解压缩源代码包。
2）进入安装目录。
3）检测文件是否缺失，创建Makefile，检测编译环境：./configure。
4）编译源码，生成库和可执行程序：make。
5）把库和可执行程序，安装到系统目录下：make install。
```

```
6）删除和卸载软件：make distclean。
7）上述安装步骤并不是绝对的，应该先查看附带的README文件。
```

- **7、磁盘管理**
  - 磁盘设备种类

```
hd      Hard Disk          硬盘
fd      Floppy Disk        软盘
sd      SCSI Device        小型计算机系统接口（英语：Small Computer System Interface; 简写：SCSI）。
                           一种用于计算机和智能设备之间（硬盘、软驱、光驱、打印机、扫描仪等）系统级接口的独立处理器标准。
                           SCSI是一种智能的通用接口标准。
设备都保存在/dev中
```

  - sd后面加小写字母，分别代表不同的硬盘，如：

```
sda     第一个硬盘
sdb     第二个硬盘
sdc     第三个硬盘
sdd     第四个硬盘
```

  - 每个硬盘又分为主分区和逻辑分区，一个硬盘最多允许四个主分区，第一个逻辑分区从sda5开始。

```
主分区：
sda1    主分区1
sda2    主分区2
sda3    主分区3
sda4    主分区4

扩展分区：
sda5    逻辑分区1
sda6    逻辑分区2
sda7    逻辑分区3
......
```

  - 查看设备信息

```
fdisk -l    当你不知道usb的名称时，可以用这个命令在获取
```

  - 挂载USB

```
mount /dev/sdb1 /mnt
注：  如果挂载的目录不是一个空目录，则目录里面的内容会被临时覆盖掉，导致目录原有文件无法操作，卸载后会恢复原有文件。
建议使用空目录，  如/mnt。
```

  - 卸载USB

```
umount /mnt
```

二、打包压缩和常用服务器

- **1、压缩包管理**
  - 屌丝版
    - **gzip** -- .gz格式压缩包

```
gzip *.txt      假设当前目录中有a.txt和b.txt，则会生成a.txt.gz和b.txt.gz，txt源文件会被删除。
                这个命令不能用于目录。
gunzip *.gz     解压
```

    - **bzip2** -- .bz2格式的压缩包

```
bzip2 -k *.txt  假设当前目录中有a.txt和b.txt，则会生成a.txt.bz2和b.txt.bz2文件，加-k参数可以将源文件保留。
                这个命令不能用于目录。
bunzip2 *.bz2   解压
```

- 高富帅版
  - **tar** -- 不使用z或j参数，该命令只能对文件和目录打包
    - 参数

```
c       创建--压缩时使用。
x       释放--解压缩时使用。
v       显示提示信息--压缩和解压缩都可以使用，可以省略。
f       指定压缩文件的名字

z       使用gzip的方式压缩文件
j       使用bzip2的方式压缩文件
```

    - 压缩

```
用法：tar zcvf 生成的压缩包的名字（xxx.tar.gz） 要压缩的文件或目录
示例：tar zcvf alltxt.tar.gz *.txt test/       同时对文件和目录进行打包压缩

用法：tar jcvf 生成的压缩包的名字（xxx.tar.bz2） 要压缩的文件或目录
示例：tar jcvf alltxt.tar.bz2 *.txt test/      同时对文件和目录进行打包压缩
```

    - 解压缩

```
tar zxvf 压缩包的名字                  解压到当前目录
tar jxvf 压缩包的名字 -C 目录           解压到指定目录
```

  - **rar**
    - 参数

```
a       Add压缩
x       Extract解压缩
r       压缩时递归目录，不加这个参数默认就是递归的
```

    - 压缩

```
用法：rar a 生成的压缩文件的名字 压缩的文件或目录

示例：
rar a all *.txt         打包压缩文件
rar a dir dir/          打包压缩目录
```

    - 解压缩

```
用法：rar x 压缩文件名 [解压缩目录]
示例：rar x all.rar
```

  - **zip**
    - 参数

```
r       压缩目录时加个参数用于递归目录，不加不能压缩
```

    - 压缩

```
用法：zip 压缩包的名字 压缩的文件或目录
示例：zip all *.txt
     zip -r myzip mytest/
```

- 解压缩

```
用法：unzip 压缩包的名字
     unzip 压缩包的名字 -d 解压的目录
示例：unzip all.zip
     unzip all.zip -d test/
```

- **2、进程管理**
  - 查看在线用户状态

```
who                 查看当前在线用户的状态

tty1~tty6           文字界面终端
tty7                图形界面终端
pts                 虚拟终端
Alt+Ctrl+F1~F7      切换终端，切换后各个终端互不影响
```

  - 查看进程

```
ps aux                  查看在线用户下所有进程
ps aux | grep bash      进程过滤
```

  - 终止进程

```
kill -l                 查看所有信号
kill -SIGKILL 5179      终止进程
kill -9 4678            终止进程
```

  - 查看当前进程的环境变量

```
env                     查看所有环境变量
env | grep PATH         只查看PATH环境变量
```

  - 查看系统状态

```
top
```

- **3、网络管理**
  - 获取网络接口信息

```
ifconfig
```

  - 测试与目标主机连通性

```
ping www.baidu.com -c 4     发送4个数据包后终止
```

  - 查看服务器域名对应的IP地址

```
nslookup www.baidu.com
```

- **4、用户管理**
  - 创建用户

```
adduser super       用户名只能小写，不能大写
useradd -s /bin/bash -g super -d /home/super -m super       用户名可以大写
-s      指定新用户登陆时shell类型
-g      指定所属组，该组必须已经存在
-d      用户家目录
```

```
-m      用户家目录不存在时，自动创建该目录
```

- 设置用户组

```
groupadd super
```

- 删除用户

```
deluser super
userdel -r super        选项-r的作用是把用户的主目录一起删除
```

- 切换用户

```
su super
```

- 修改用户密码

```
passwd super
```

- 退出当前用户

```
exit
```

- 查看所有用户信息

```
vi /etc/passwd
```

- **5、ftp服务器搭建**
  - **vsftpd**作用：文件的上传和下载
  - **服务器端**
    - 安装vsftpd

```
apt-get install -y vsftpd       安装完成后会随系统启动而自动启动
```

    - 修改配置文件

```
vi /etc/vsftpd.conf

listen=YES
anon_root=/home/super/anonFtp/      #指定匿名用户根目录，不指定默认是/srv/ftp/
anonymous_enable=YES    #允许匿名用户登录
write_enable=YES        #实名用户拥有写权限（上传数据）
local_umask=022         #设置本地掩码为022
anon_upload_enable=YES      #匿名用户可以向ftp服务器上传数据
anon_mkdir_write_enable=YES     #匿名用户可以在ftp服务器上创建目录
```

    - 创建匿名用户根目录

```
cd ~
mkdir anonFtp
chown ftp anonFtp
chmod 777 anonFtp
```

    - 重启服务

```
service vsftpd retart
```

  - **客户端**
    - 实名用户登录

```
ftp IP(server)
输入用户名
输入密码

文件的上传和下载
    文件的上传： put file
    文件的下载： get file
    不允许操作目录，如果想操作目录 -- 打包tar/rar/zip
```

- 匿名用户登录

```
ftp IP(server)
用户名：anonymous
密码：直接回车

不允许匿名用户在任意目录直接切换
只能在一个指定目录范围内工作
需要在ftp服务器上创建一个匿名用户的目录 -- 匿名用户的根目录
```

- 退出

```
quit        推荐
bye         推荐
exit
```

- **lftp客户端访问ftp服务器**
  - 一个ftp客户端工具，可以上传和下载目录
  - 安装

```
apt-get install -y lftp
```

  - 登录服务器
    - 匿名

```
lftp IP(sever) 回车
login
```

    - 实名

```
lftp username@IP(sever) 回车
输入服务器密码
```

  - 操作

```
put file                上传文件
mput file1 file2        上传多个文件
get file                下载文件
mget file1 file2        下载多个文件
mirror dir              下载整个目录及其子目录
mirror -R dir           上传整个目录及其子目录
```

- **6、nfs服务器搭建**
  - net file system : 网络文件系统，它允许网络中的计算机之间通过TCP/IP网络共享资源。
  - 服务器端
    - 安装

```
apt-get install nfs-kernel-server       安装成功后会随系统启动而自动启动
```

        - 创建共享目录

```
如: /home/super/NfsShare
```

        - 修改配置文件

```
vi /etc/exports
在文件最后加入下面设置：
/home/super/NfsShare *(rw, sync)
```

        - 重启服务

```
service nfs-kernel-server restart
```

    - 客户端

        - 挂载服务器共享目录

```
用法： mount serverIP:ShareDir /mnt
示例： mount 192.168.10.100:/home/super/NfsShare /mnt
```

- **7、ssh（Secure Shell）服务器**

    - 服务器端

        - 安装ssh

```
apt-get install -y openssh-server
aptitude show openssh-server
```

    - 客户端

        - 远程登录

```
ssh Username@ServerIP
```

        - 退出登录

```
logout
```

- **8、scp（Super Copy）命令**

    - 使用该命令的前提条件

```
目标主机已经安装了openssh-server
```

    - 使用方法

```
scp -r 目标用户名@目标主机IP地址：/目标文件的绝对路径  /保存到本机的绝对（相对）路径
scp -r super@192.168.10.100:/home/super/SCP/ ./mydir/
```

- **9、其他命令**

    - 翻页

```
Shift+PageUp        上翻页
Shift+PageDown      下翻页
```

    - 清屏

```
clear
Ctrl＋l
```

- 创建终端

```
Ctrl+Alt+T      Ubuntu
Ctrl+Shift+T    添加新标签页
```

- 看手册

```
man man

一共有九个章节，1,2,3,5是重点：
1   可执行程序或 shell 命令
2   系统调用(内核提供的函数)
3   库调用(程序库中的函数)
4   特殊文件(通常位于 /dev)
5   文件格式和规范，如 /etc/passwd
6   游戏
7   杂项(包括宏包和规范，如man(7)，groff(7))
8   系统管理命令(通常只针对root用户)
9   内核例程 [非标准
```

- 设置查看别名

```
alias                   查看
alias l='ls -al'        设置
```

- 在显示器上显示数据

```
echo "hello"    打印字符串
echo $PATH      打印环境变量
echo $?         显示上一次程序退出值
```

- **10、关机重启**
  - **poweroff**
  - **reboot**
  - **shutdown**
    - 参数

```
-t      秒数:设定在切换至不同的runlevel之前，警告和删除两讯号之间的延迟时间（秒）
-k      仅送出警告讯息文字，但不是真的要shutdown
-r      shutdown之后重新开机
-h      shutdown之后关机
-n      不经过init，由shutdown指令本身来做关机动作（不建议使用）
-f      重新开机时，跳过fsck指令，不检查档案系统
-F      重新开机时，强迫做fsck检查
-c      将已经正在shutdown的动作取消
```

    - 示例

```
shutdown -r now     #立即重启
shutdown -h now     #立即关机
shutdown -k now 'Hey! Go away! now...'      #发出警讯息，但没有真的关机
shutdown -t3 -r now     #立即重启，但在警告和删除processes之间，延迟3秒钟
shutdown -h 10:42 'Hey! Go away!'   #10：42关机
shutdown -r 10 'Hey! Go away!'      #10分钟后关机
shutdown -c     #取消shutdown指令，必须切换至其他tty，登入之后，才能下些命令
shutdown now    #切换至单人操作模式（不加任何选项时）
```

三、vim和函数库

1、vim编辑器的使用

1.1、vim简介

- Vim是从 vi 发展出来的一个文本编辑器。代码补全、编译及错误跳转等方便编程的功能特别丰富，在程序员中被广泛使用，和Emacs并列成为类Unix系统用户最喜欢的文本编辑器。

1.2、工作模式

- 命令模式 -- 打开文件之后，默认进入命令模式。
  - 移动光标

```
h           左
j           下
k           下
l           右
0           行首
$           行尾
gg          文件头部
G           文件尾部
5G          行跳转，移到第五行
```

  - 删除操作（不是真正意义上的删除，更象是剪切操作）

```
x           删除光标后一个字符
X           删除光标前一个字符
dw          删除光标后一个单词
d0          删除光标到行首的所有字符
D(d$)       删除光标到行尾的所有字符
dd          删除一行
5dd         删除五行
```

  - 撤消操作

```
u           撤消
ctrl+r      还原前一个操作
```

  - 复制粘贴

```
p           小写的p，粘贴到光标的下一行
P           大写的p，粘贴到光标的所在行
yy          复制一行
5yy         复制五行
```

  - 可视模式

```
v           切换到可视块模式
h/j/k/l     调整选中块
y           复制
d           删除
p           粘贴
```

  - 查找模式

```
/           切换到查找模式，向下查找
?           切换到查找模式，向上查找
n           下一个匹配
N           上一个匹配
#           移到要查找的单词上，按#号即可开启查找模式
```

  - 字符替换

```
r           单个字符替换
```

- 行缩进

```
>>          向右缩进
<<          向左缩进
```

- 查看函数man文档

```
K           先输入3（库函数章节），再输入K，以查看指定函数的man文档
```

- 编辑模式 -- 需要输入一些命令，切换到编辑模式。

```
a       在光标所在位置的后边插入
A       在当前行的尾部插入

i       在光标所在位置的前边插入
I       在光标所在行的行首插入

o       在光标所在行的下边开辟一个新的行
O       在光标所在行的上边开辟一个新的行

s       删除光标后边的字符
S       删除光标所有的行
```

- 末行模式 -- 在末行模式下可以输入一些命令。

```
:s/tom/jack/g           把光标所在行的tom替换成jack
:%s/tom/jack/g          把整个文档的tom替换成jack
:25,30s/tom/jack/g      把25到30行的tom替换成jack

:q          退出
:q!         强制退出
:w          保存
:wq         保存并退出
:x          相当于:wq
ZZ          相当于:wq
```

- 分屏操作

```
:sp [file]      将屏幕水平分成两部分
:vsp [file]     交屏幕垂直分成两部分
:wqall          同时保存打开文件
ctrl+ww         切换两个屏幕
```

- vim打造成IDE

```
/etc/vim/vimrc      系统级配置文件
~/.vim/vimrc        用户级配置文件
```

2、gcc编译器

- gcc编译的四个阶段：

```
hello.c  -------->   hello.i  -------->    hello.s  -------->    hello.o  ------->    hello.out
        预处理编译器（cpp）      编译器（gcc）           汇编器（as）            链接器（ld）
        gcc -E                  gcc -S                  gcc -c                  gcc
```

- 编译工具链：

```
        预处理编译器：cpp     gcc -E hello.c -o hello.i   头文件展开，宏替换，注释去掉
             编译器：gcc      gcc -S hello.i -o hello.s   c文件变成汇编文件
             汇编器：as       gcc -c hello.s -o hello.o   汇编文件变成二进制文件
```

```
链接器：ld       gcc hello.o -o hello          将函数库中相应的代码组合到目标文件中
```

- 直接编译生成可执行文件

```
gcc hello.c -o hello
```

- 编译时指定头文件目录-I

```
gcc hello.c -I ./include hello
```

- 编译时指定宏，通常用于调试代码-D

```
gcc hello.c -D DEBUG

在需要打印日志的代码中加上下面的语句：
#ifdef DEBUG
    printf("debug info");
#endif
```

- 编译时对代码进行优化-O3

```
gcc hello.c -o hello -O3
级别：
0   没有优化
1   缺省值
3   优化级别最高
```

- 编译时显示告警信息-Wall

```
gcc hello.c -o hello -Wall
```

- 编译时包含调试信息-g（gdb）

```
gcc hello.c -o hello -g
```

- 查看编译版本

```
gcc -v
gcc --version
```

3、静态库的创建和使用

- 准备

```
目录结构
myCalc
├── include
│   └── head.h
├── lib
│   └── libMyCalc.a
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mul.c
    ├── sub.c
```

- 命名格式
  - 以lib开头。
  - 静态库名。
  - .a结尾。

```
示例 ：  libMyCalc.a
```

- 描述
  - 优点
    - 寻址方便、速度快。
    - 库被打包到可执行程序中，直接发布可执行程序即可使用。
  - 缺点
    - 静态库的代码在编译过程中已经被载入可执行程序，因此体积较大。
    - 如果静态函数库改变了，那么你的程序就必须重新编译。
  - 使用场合
    - 在核心程序上使用，保证速度，忽略空间。
    - 主流应用于80、90年代，现在很少使用。
- 制作
  - 得到与位置有关的`*.o`

```
cd ~/myClac/src
gcc *.c -c -I ../include
```

  - 将生成的`*.o`文件打包成静态库`libMyCalc.a`

```
ar rcs libMyCalc.a *.o

ar工具不包含在一gcc中。
参数：
r        将文件夹插入到静态库中。
c        创建静态库，不管库是否存在。
s        写入一个目标文件索引到库中，或者更新一个存在的目标文件。

制作好以后移动到lib目录中
mv libMyCalc.a ../lib
```

  - 查看库中的符号（函数）

```
nm libMyCalc.a
```

  - 发布和使用静态库
    - 发布静态库，把`include`和`lib`目录打包给用户。
    - 使用
      - 在`main.c`中包含头文件

```
#inlcude "head.h"
```

      - 两种编译方式

```
gcc main.c lib/libMyCalc.a -I include -o myapp
gcc main.c -I include -L lib -l MyCalc -o myapp
```

      - 执行`myapp`

```
./myapp
```

4、动态库的创建和使用

- 准备

```
目录结构
myCalc
```

```
├── include
│   └── head.h
├── lib
│   └── libMyCalc.so
├── main.c
└── src
    ├── add.c
    ├── div.c
    ├── mul.c
    ├── sub.c
```

- 命名格式
  - 以`lib`开头。
  - 静态库名。
  - `.so`结尾。

    ```
    示例 ：  libMyCalc.so
    ```

- 描述
  - 优点
    - 执行程序体积小。
    - 动态库更新了，不需要重新编译程序，前提是函数接口不变。
  - 缺点
    - 发布程序的时候，需要将动态库提供给用户。
    - 动态库没有被打包到应用程序中，加载速度相对较慢。
  - 使用场合
    - 现代应用程序中常用。
- 制作
  - 生成与位置无关的`*.o`

    ```
    cd ~/myClac/src
    gcc -fPIC *.c -c -I ../include
    ```

  - 将生成的`*.o`文件打包成共享库（动态库）`libMyCalc.so`

    ```
    gcc -shared -o libMyCalc.so *.o -I ../include

    制作好以后移动到lib目录中
    mv libMyCalc.so ../lib
    ```

  - 查看库中的符号（函数）

    ```
    nm libMyCalc.so
    ```

  - 发布和使用静态库
    - 发布静态库，把`include`和`lib`目录打包给用户。
    - 使用
      - 在`main.c`中包含头文件

        ```
        #inlcude "head.h"
        ```

      - 两种编译方式

        ```
        gcc main.c lib/libMyCalc.so -I include -o myapp
        gcc main.c -I include -L lib -l MyCalc -o myapp
        ```

      - 执行`myapp`

```
./myapp
```

- 查看可执行文件夹所依赖的动态库

```
ldd myapp
```

- 解决依赖的动态库无法加载的问题
  - 把`libMyCalc.so`拷贝到`/lib`目录下(不推荐使用，容易与系统库冲突)

    ```
    cp lib/libMyCalc.so /lib
    ```

  - 设置环境变量LD_LIBRARY_PATH（测试时使用）

    ```
    export LD_LIBRARY_PATH=./lib     只对当前会话有效
    ```

  - 设置环境变量LD_LIBRARY_PATH到~/.bashrc中（测试时使用）

    ```
    vi ~/.bashrc
    export LD_LIBRARY_PATH=/home/super/myCalc/lib        会话每次开启就会生效
    ```

  - 设置配置文件`/etc/ld.so.conf`（推荐使用），重启终端
    - 编辑动态连接器的配置文件

      ```
      vi /etc/ld.so.conf
      ```

    - 动态库的路径写到配置文件中

      ```
      /home/super/myCalc/lib
      ```

    - 更新动态库

      ```
      ldconfig -v
      ```

四、makefile和文件IO

1、gdb调试

- 1.1、启动gdb

```
gdb app
```

- 1.2、查看代码

```
>>>(gdb) l          #list
>>>(gdb) l 行号或函数名
>>>(gdb) l 文件名:行号或函数名
```

- 1.3、设置断点
  - 设置当前文件断点

    ```
    >>>(gdb) b      #break
    >>>(gdb) b 行号或函数名
    >>>(gdb) b 文件名:行号或函数名
    ```

  - 设置条件断点

```
>>>(gdb) b 10 if value==19
```

- 删除断点

```
>>>(gdb) d 断点的编号        #delete或del
```

- 1.4、查看设置的断点

```
>>>(gdb) i b            #info，可以获取断点的编号，删除断点时用
```

- 1.5、开始执行gdb调试

```
>>>(gdb) start      #只执行一步
>>>(gdb) c          #continue，直接停在断点的位置
>>>(gdb) run        #有断点会停在断点处，没有会直接跑完程序，较少用
```

- 1.6、单步调试

```
>>>(gdb) s          #进入函数体内部
>>>(gdb) finish     #从函数体内部跳出
>>>(gdb) n          #不进入函数体内部
>>>(gdb) u          #退出当前循环
```

- 1.7、查看变量的值

```
>>>(gdb) p      print
```

- 1.8、查看变量的类型

```
>>>(gdb) ptype 变量名
```

- 1.9、设置变量的值

```
>>>(gdb) set var 变量=值
```

- 1.10、设置追踪变量

```
>>>(gdb) display 变量        #设置追踪变量
>>>(gdb) info display        #获取变量的编号
>>>(gdb) undisplay 编号      #取消追踪变量
```

- 1.10、退出gdb调试

```
>>>(gdb) quit
```

2、makefile的编写

2.1、makefile的命名

```
makefile
Makefile

注：两种任选其中一种，其他名称不行。
```

2.2、makefile的规则

- 规则中的三要素：目标、依赖、命令
  - makefile文件内容格式

```
目标：依赖文件
    命令
```

- makefile文件内容简单示例

```
app:main.c add.c sub.c mul.c div.c
    gcc main.c add.c sub.c mul.c div.c -o app
```

- 第二行必须有一个tab缩进。

- 子目标和终极目标
  - 示例

```
#终极目标
app:main.o add.o sub.o div.o
    gcc main.o add.o sub.o div.o -o app
#子目标
main.o:main.c
    gcc -c main.c
#子目标
add.o:add.c
    gcc -c add.c
#子目标
sub.o:sub.c
    gcc -c sub.c
#子目标
mul.o:mul.c
    gcc -c mul.c
#子目标
div.o:div.c
    gcc -c div.c
```

  - 好处
    - 当其他一个文件被修改时，只会编译修改的文件，再打包生成app，编译效率高。
- 更新目标的原则

2.3、makefile的两个函数

- `wildcard`：取得目录下的所有文件
- `patsubst`：将目标下的所有文件作替换
- 示例

```
#自定义变量
src=$(wildcard ./*.c)
obj=$(patsubst ./%.c, ./%.o, $(src))
target=app
#终极目标
$(target):$(obj)
    gcc $(obj) -o $(target)
#子目标
%.o:%.c
    gcc -c $< -o $@
```

2.4、makefile的三个自动变量

- 普通变量
  - 变量取值：`foo = $(obj)`
  - 由makefile维护的一些变量
    - 通常格式都是大写
      - `CC`: 默认值是`cc`
    - 有些有默认值，有些没有

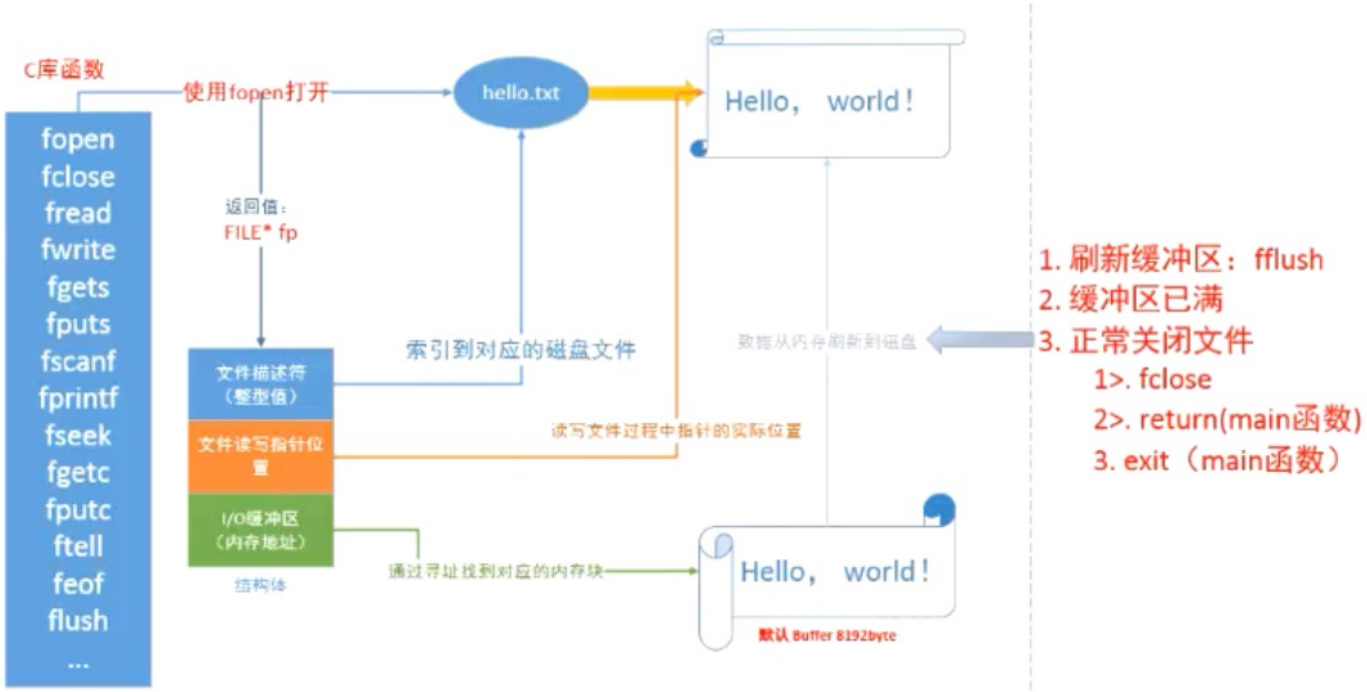
        - `CPPFLAGS`: 预处理器需要的选项，如：`-I`
        - `CFLAGS`: 编译的时候使用的参数 `-Wall -g -c`
        - `LDFLAGS`: 链接库使用的选项 `-L -l`
    - 用户可以修改这些变量的默认值
        - `CC = gcc`
- 自动变量
    - 变量
        - `$<` 规则中的第一个依赖条件
        - `$@` 规则中的目标
        - `$^` 规则中的所有依赖条件
        - 以上三种变量只能在规则的命令中使用
    - 模式规则
        - 在规则的目标定义中使用%
        - 在规则的依赖条件中使用%
        - 示例

```
#自定义变量
src=$(wildcard ./*.c)
obj=$(patsubst ./%.c, ./%.o, $(src))
target=app
#makefile维护的变量
CC = gcc
CPPFLAGS = -I
#终极目标
$(target):$(obj)
    $(CC) $^ -o $@
#子目标
%.o:%.c
    $(CC) -c $< -o $@
#伪目标
.PHONY:clean
clean:
    #命令前面加-表示，如果命令执行失败则忽略这个命令
    -makir /release
    rm -f $(obj) $(target)
```

3、系统IO函数

3.1、一些概念

- C库IO函数工作流程



- 硬盘为什么慢

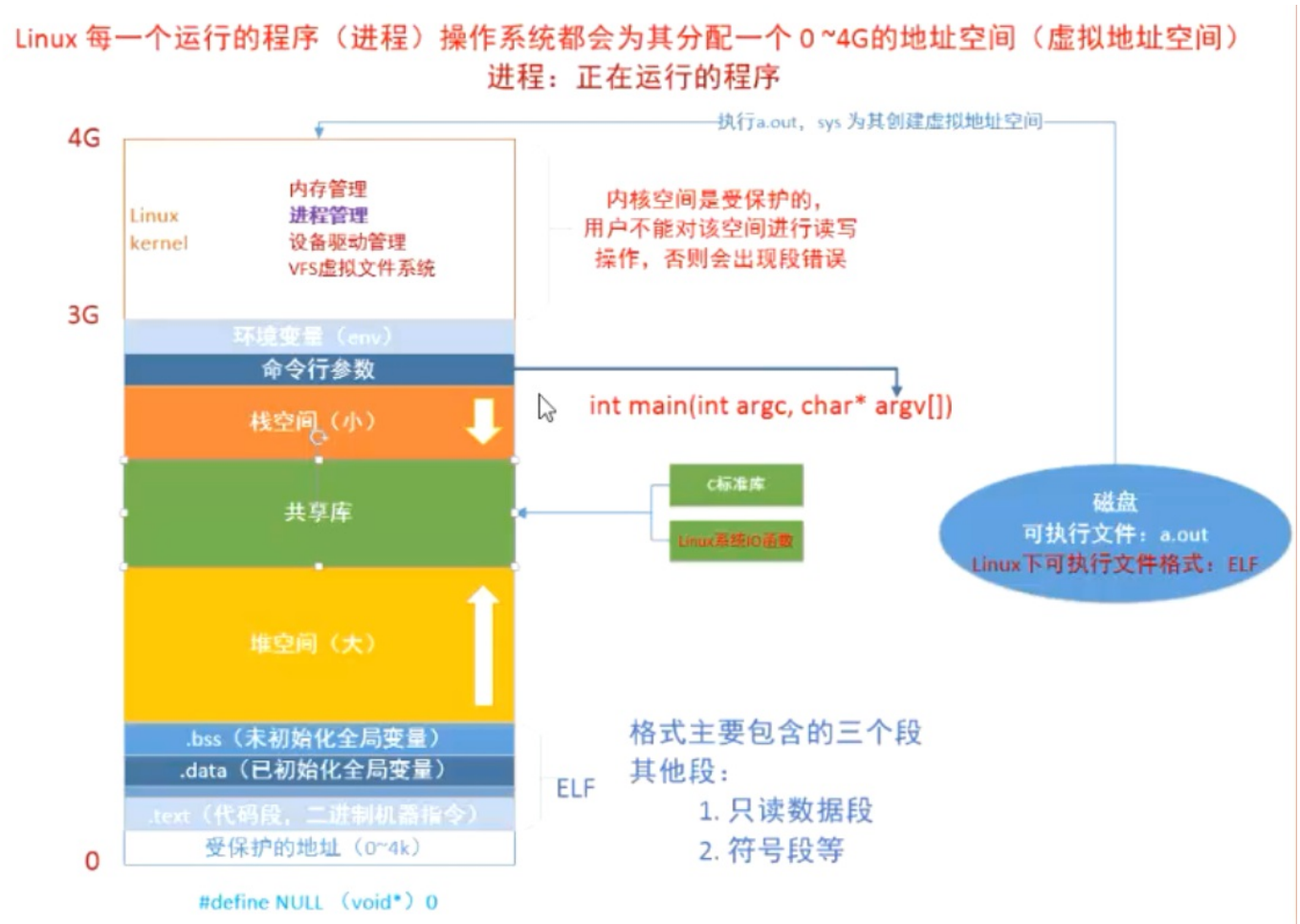
  大部分硬盘都是机械硬盘，读取寻道时间和写入寻道时间都是是在毫秒级（ms）。

  相对来说内存读写速度都非常快，因为内存术语电子设备，读写速度是纳秒（ns）级别的。

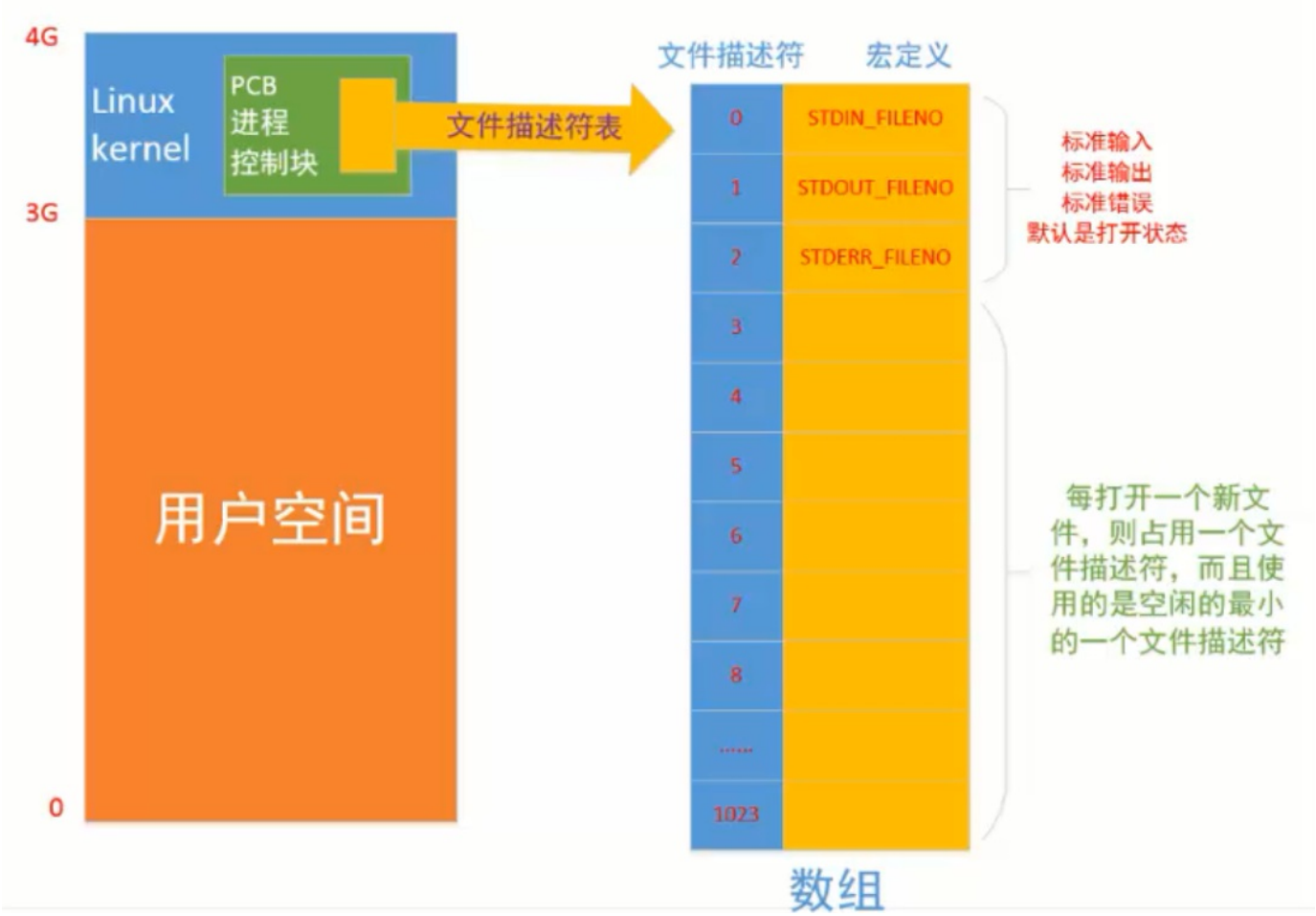
  1秒 = 1000毫秒(ms)
  1秒 = 1,000,000 微秒(μs)
  1秒 = 1,000,000,000 纳秒(ns)

  两者相差一百万倍！！！

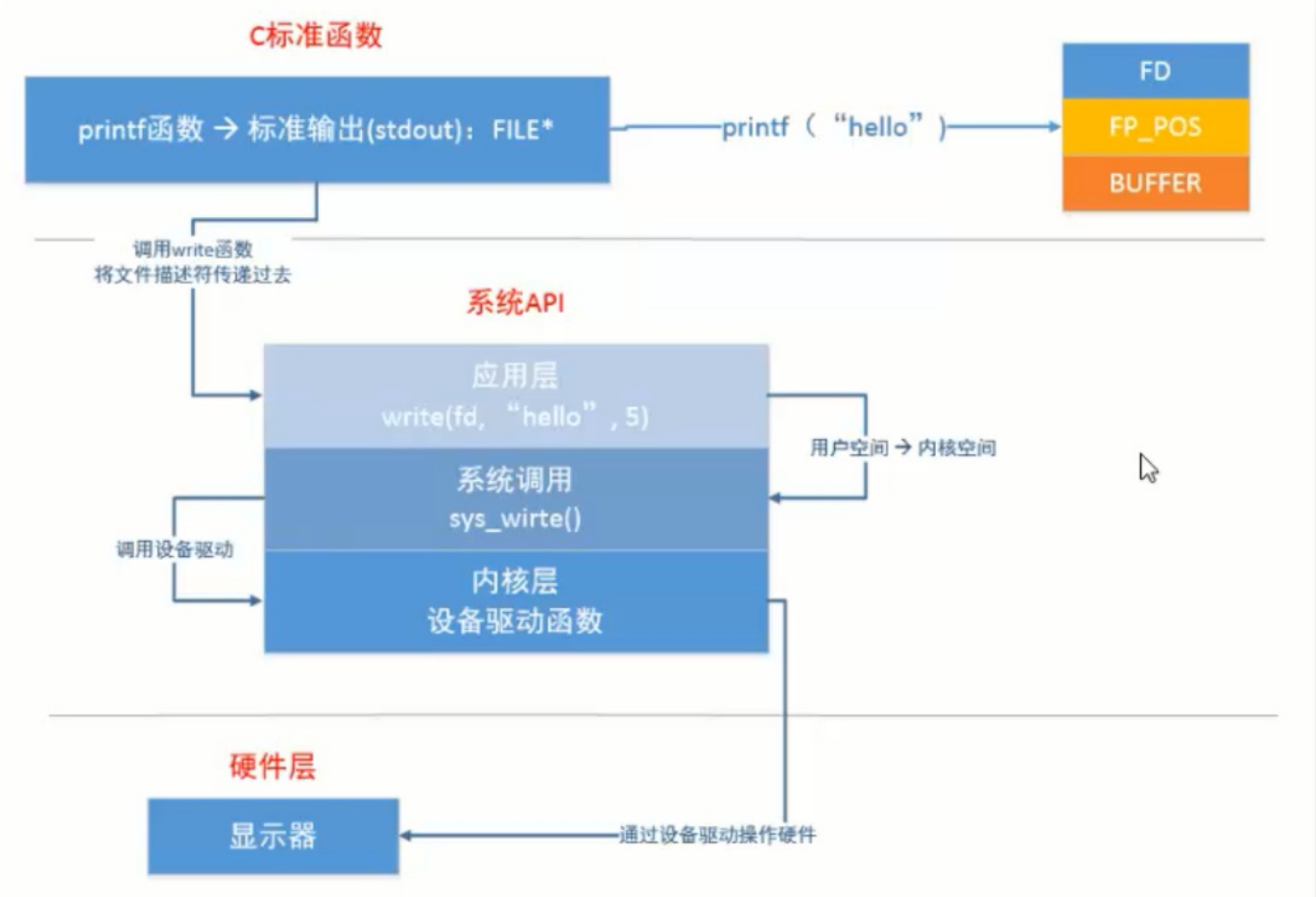
- 虚拟内存地址空间


Linux 每一个运行的程序（进程）操作系统都会为其分配一个 0 ~4G的地址空间（虚拟地址空间）
进程：正在运行的程序
执行a.out，sys 为其创建虚拟地址空间
4G
3G
0
Linux
kernel
内存管理
进程管理
设备驱动管理
VFS虚拟文件系统
内核空间是受保护的，
用户不能对该空间进行读写
操作，否则会出现段错误
环境变量（env）
命令行参数
栈空间（小）
int main(int argc, char* argv[])
共享库
C标准库
Linux系统IO函数
磁盘
可执行文件：a.out
Linux下可执行文件格式：ELF
堆空间（大）
.bss（未初始化全局变量）
.data（已初始化全局变量）
.text（代码段，二进制机器指令）
受保护的地址（0~4k）
ELF
格式主要包含的三个段
其他段：
1. 只读数据段
2. 符号段等
#define NULL （void*）0

- pcb与文件描述符



- 库函数与系统函数之间的关系



- CPU为什么要使用虚拟地址空间与物理地址空间映射？解决了什么问题？
  - 方便编译器和操作系统安排程序的地址分布。
    - 程序可以使用一系列相邻的虚拟地址来访问物理内存中不相邻的大内存缓冲区
  - 方便进程之间隔离
    - 不同进程使用的虚拟地址彼此隔离，一个进程中的代码无法更改正在由另一个进程使用的物理内存。
  - 方便OS使用你那可怜的内存
    - 程序可以使用一系列虚拟地址来访问大于可用物理内存的内存缓冲区。当物理内存的供应量变小时，内存管理器会将物理内存页（通常大小为4KB）保存到磁盘文件。数据或代码页会根据需要在物理内存和磁盘之间移动。
- 查看帮助文档

```
man 2 open
man 3 printf
```

- errno
  - 定义在头文件`/usr/include/error.h`中
    - 全局变量
    - 任何标准C库函数都能对其进行修改（Linux系统函数更可以）
- 错误宏定义位置
  - 第1-34个错误定义：`/usr/include/asm-generic/errno-base.h`
  - 第35-133个错误定义：`/usr/include/asm-generic/errno.h`
- 是记录系统的最后一次错误代码。代码是一个int型的值。
  - 第个errno值对应着以字符串表示的错误类型。
  - 当调用“某些”函数出错时，该函数会重新设置errno的值。
- perror函数
  - 头文件`stdio.h`
  - 函数定义`void perror(const char * s)`
  - 函数说明
    - 用来将上一个函数发生的错误的原因输出到标准设备（stderr）。

- 参数s所指的字符串会先打印出，后面再加上错误原因字符串。
- 些错误原因依照全局变量error的值来决定要输出的字符串。

3.2、open

- 函数定义

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- 功能

```
打开文件。
```

- 参数

```
pathname    文件路径，如./hello.c
flags       文件访问模式，O_RDONLY, O_WRONLY, O_RDWR,  更多见man 2 open
mode        文件权限设置，实际结果是mode & umask
```

- 返回值

```
返回一个文件描述fd，-1表示打开文件时发生错误。
```

- 示例

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    int fd;
    //打开文件
    fd = open("hello.c", O_RDWR);
    //打开文件，不存在则创建并设置权限为：755 & umask
    fd = open("hello.c", O_RDWR | O_CREAT, 0755);
    //打开文件，不存在则创建并设置权限为：755 & umask，另判断文件是否存在
    fd = open("hello.c", O_RDWR | O_CREAT | O_EXCL, 0755);
    //打开文件并清空文件里的内容
    fd = open("hello.c", O_RDWR | O_TRUNC);

    printf("fd = %d\n", fd);
    //打开失败则打印错误并退出程序
    if(fd == -1)
    {
        perror("open file failure");
        exit(1);
    }

    int ret = close(fd);
    printf("ret = %d\n", ret);
    //关闭失败则打印错误并退出程序
    if(ret == -1)
    {
        perror("close file failure");
        exit(1);
    }
    return 0;
}
```

3.3、read

- 函数定义

```
ssize_t read(int fd, void *buf, size_t count);
```

- 功能

```
读取文件。
```

- 参数

```
fd          文件描述符
buf         缓冲区
count       读取多少字节数
```

- 返回值

```
有符号int型值
-1      表示读文件失败
0       表示文件读完了
>0      表示读取的字节数
```

3.4、write

- 函数定义

```
ssize_t write(int fd, const void *buf, size_t count);
```

- 功能

```
写入文件。
```

- 参数

```
fd          文件描述符
buf         缓冲区
count       读取多少字节数
```

- 返回值

```
有符号int型值
-1      表示写文件失败
0       表示什么都没写
>0      表示写了多少字节数
```

- 示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
    //以只读方式找开文件
    int fd = open("english.txt", O_RDONLY);
    if(fd == -1)
    {
        perror("open");
```

```
        exit(1);
    }
    //以写的方式找开文件，文件不存在则创建
    int fd1 = open("newfile", O_CREAT | O_WRONLY, 0664);
    //打开失败，打印错误并退出程序
    if(fd1 == -1)
    {
        perror("open1");
        exit(1);
    }

    char buf[2048] = {0};
    //每次读取2048字节
    int count = read(fd, buf, sizeof(buf));
    //读取失败，打印错误并退出程序
    if(count == -1)
    {
        perror("read");
        exit(1);
    }
    while(count)
    {
        //写入文件
        int ret = write(fd1, buf, count);
        printf("write bytes %d\n", ret);
        //继续读取内容
        count = read(fd, buf, sizeof(buf));
    }
    //关闭文件
    close(fd);
    close(fd1);
    return 0;
}
```

3.5、close

- 函数定义

```
int close(int fd);
```

- 功能

```
关闭文件。
```

- 参数

```
fd      文件描述符
```

- 返回值

```
0       表示关闭成功
-1      表示关闭失败
```

3.6、lseek

- 函数定义

```
off_t lseek(int fd, off_t offset, int whence);
```

- 功能

```
1、获取文件大小。
2、移动文件指针。
```

```
3、文件拓展。
```

- 参数

```
fd          文件描述符
offset      文件指针偏移量
whence      来源，主要值有：SEEK_SET、SEEK_CUR、SEEK_END等
```

- 返回值

```
有符号int型值
-1      表示读文件失败
0       表示文件读完了
>0      表示读取的字节数
```

- 示例

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int fd = open("aa", O_RDWR);
    if(fd == -1)
    {
        perror("open file");
        exit(1);
    }

    int ret = lseek(fd, 0, SEEK_END);
    printf("file length = %d\n", ret);

    //文件拓展
    ret = lseek(fd, 2000, SEEK_END);
    printf("return value = %d\n", ret);

    //实现文件拓展，需要再最后做一次写操作
    write(fd, "a", 1);

    close(fd);
    return 0;
}
```

五、一些系统函数的使用

1、Linux文件操作相关函数

stat函数

- 函数定义

```
int stat(const char *pathname, struct stat *buf);
```

- 功能

```
获取文件属性（从inode上获取）。
```

- 参数

```
pathname        文件名
buf             结构体指针stat
```

- 返回值

```
0       成功
-1      失败
```

- 文件属性

```
struct stat
{
       dev_t     st_dev;         /* 文件的设备编号 */
       ino_t     st_ino;         /* 节点 */
       mode_t    st_mode;        /* 文件的类型和存取的权限 */
       nlink_t   st_nlink;       /* 链接到该文件的硬链接数目，刚建立的文件值为1 */
       uid_t     st_uid;         /* 用户ID */
       gid_t     st_gid;         /* 组ID */
       dev_t     st_rdev;        /* （设备类型）若此文件为设备文件，则为其设备编号 */
       off_t     st_size;        /* 文件字节数（文件大小） */
       blksize_t st_blksize;     /* 块大小（文件系统的I/O，缓冲区的大小） */
       blkcnt_t  st_blocks;      /* 块数 */
       struct timespec st_atim;  /* 最后一次访问时间 */
       struct timespec st_mtim;  /* 最后一次修改时间 */
       struct timespec st_ctim;  /* 最后一次改变时间（属性） */
}
```

- 特性
  - 能够穿透（跟踪）符号链接。如果是软链接，会统计原始文件信息，而不是软链接的信息。
- 示例

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <time.h>
#include <pwd.h>
#include <grp.h>

int main(int argc, char * argv[])
{
    if(argc < 2)
    {
        printf("./a.out filename\n");
        exit(1);
    }

    struct stat st;
    int ret = stat(argv[1], &st);
    if(ret == -1)
    {
        perror("stat");
        exit(1);
    }
    //存储文件类型和访问权限
    char perms[11] = {0};
    //判断文件类型
    switch(st.st_mode & S_IFMT)
    {
        case S_IFLNK:
            perms[0] = 'l';
            break;
        case S_IFDIR:
            perms[0] = 'd';
            break;
```

```
        case S_IFREG:
            perms[0] = '-';
            break;
        case S_IFBLK:
            perms[0] = 'b';
            break;
        case S_IFCHR:
            perms[0] = 'c';
            break;
        case S_IFSOCK:
            perms[0] = 's';
            break;
        case S_IFIFO:
            perms[0] = 'p';
            break;
        default:
            perms[0] = '?';
            break;
    }

    //判断文件的访问权限
    //文件所有者
    perms[1] = (st.st_mode & S_IRUSR) ? 'r' : '-';
    perms[2] = (st.st_mode & S_IWUSR) ? 'w' : '-';
    perms[3] = (st.st_mode & S_IXUSR) ? 'x' : '-';
    //文件所属组
    perms[4] = (st.st_mode & S_IRGRP) ? 'r' : '-';
    perms[5] = (st.st_mode & S_IWGRP) ? 'w' : '-';
    perms[6] = (st.st_mode & S_IXGRP) ? 'x' : '-';
    //其他人
    perms[7] = (st.st_mode & S_IROTH) ? 'r' : '-';
    perms[8] = (st.st_mode & S_IWOTH) ? 'w' : '-';
    perms[9] = (st.st_mode & S_IXOTH) ? 'x' : '-';

    //硬链接计数
    int linkNum = st.st_nlink;
    //文件所有者
    char * fileUser = getpwuid(st.st_uid)->pw_name;
    //文件所属组
    char * fileGrp = getgrgid(st.st_gid)->gr_name;
    //文件大小
    int filesize = (int)st.st_size;
    //修改时间
    char * time = ctime(&st.st_mtime);
    char mtime[512] = {0};
    strncpy(mtime, time, strlen(time) - 1);

    char buf[1024];
    sprintf(buf, "%s %d %s %s %d %s %s", perms, linkNum, fileUser, fileGrp, fileSize, mtime, argv[1]);

    printf("%s\n", buf);

    return 0;
}
```

- st_mode
  - 该变量占两字节，共16位。
  - 掩码的使用：st_mode & 掩码
  - 其他人权限（0-2 bit）

```
S_IROTH 00004 读权限
S_IWOTH 00002 写权限
S_IXOTH 00001 执行权限
S_IRWXO 00007 掩码
```

- 所属组权限（3-5 bit）

```
S_IRGRP 00040 读权限
S_IWGRP 00020 写权限
S_IXGRP 00010 执行权限
S_IRWXG 00070 掩码
```

- 文件所有者权限（6-8 bit）

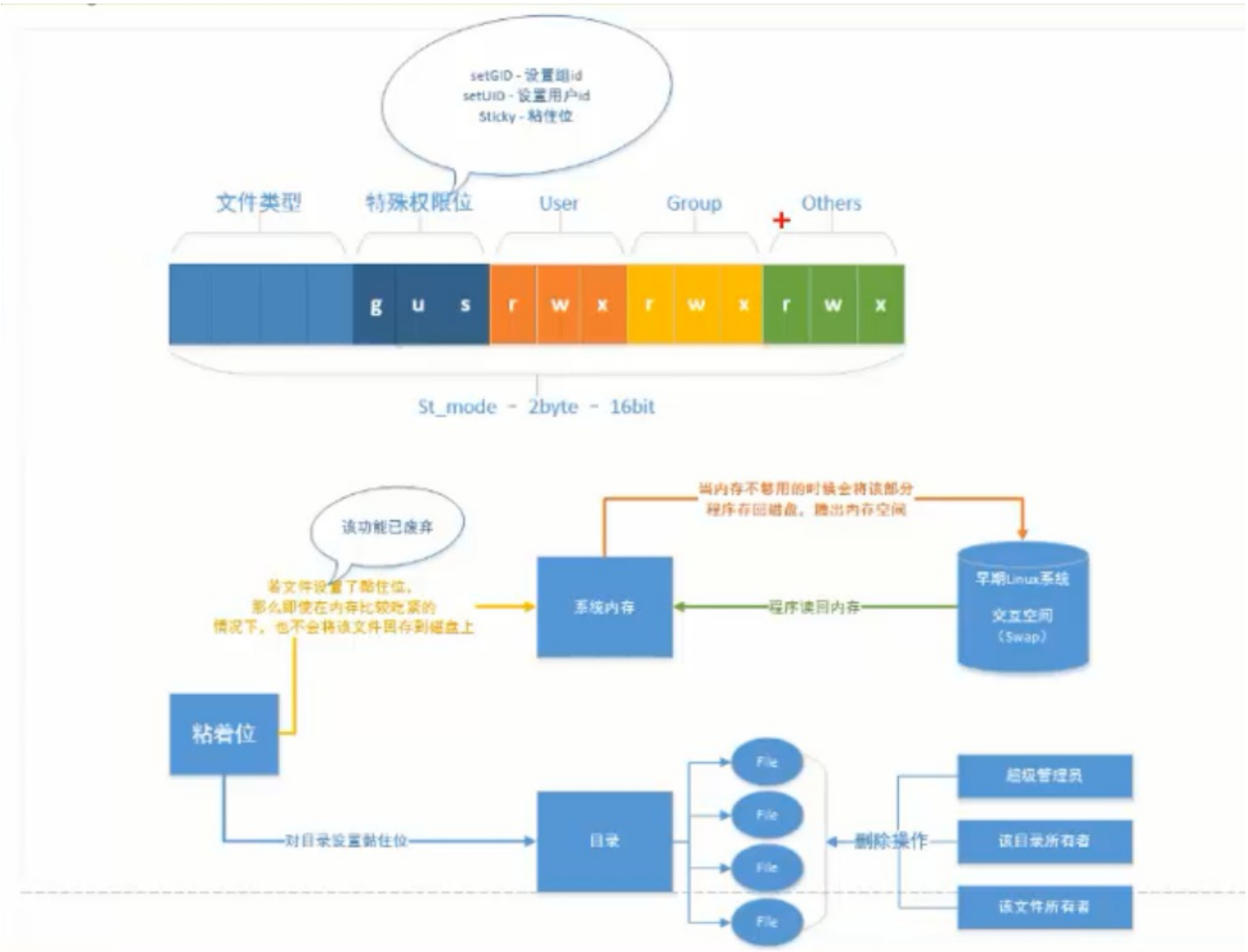
```
S_IRUSR 00400 读权限
S_IWUSR 00200 写权限
S_IXUSR 00100 执行权限
S_IRWXU 00700 掩码
```

- 特殊权限位（9-11 bit），很少用

```
S_ISUID 0004000 设置用户ID
S_ISGID 0002000 设置组ID
S_ISVTX 0001000 黏住位
```

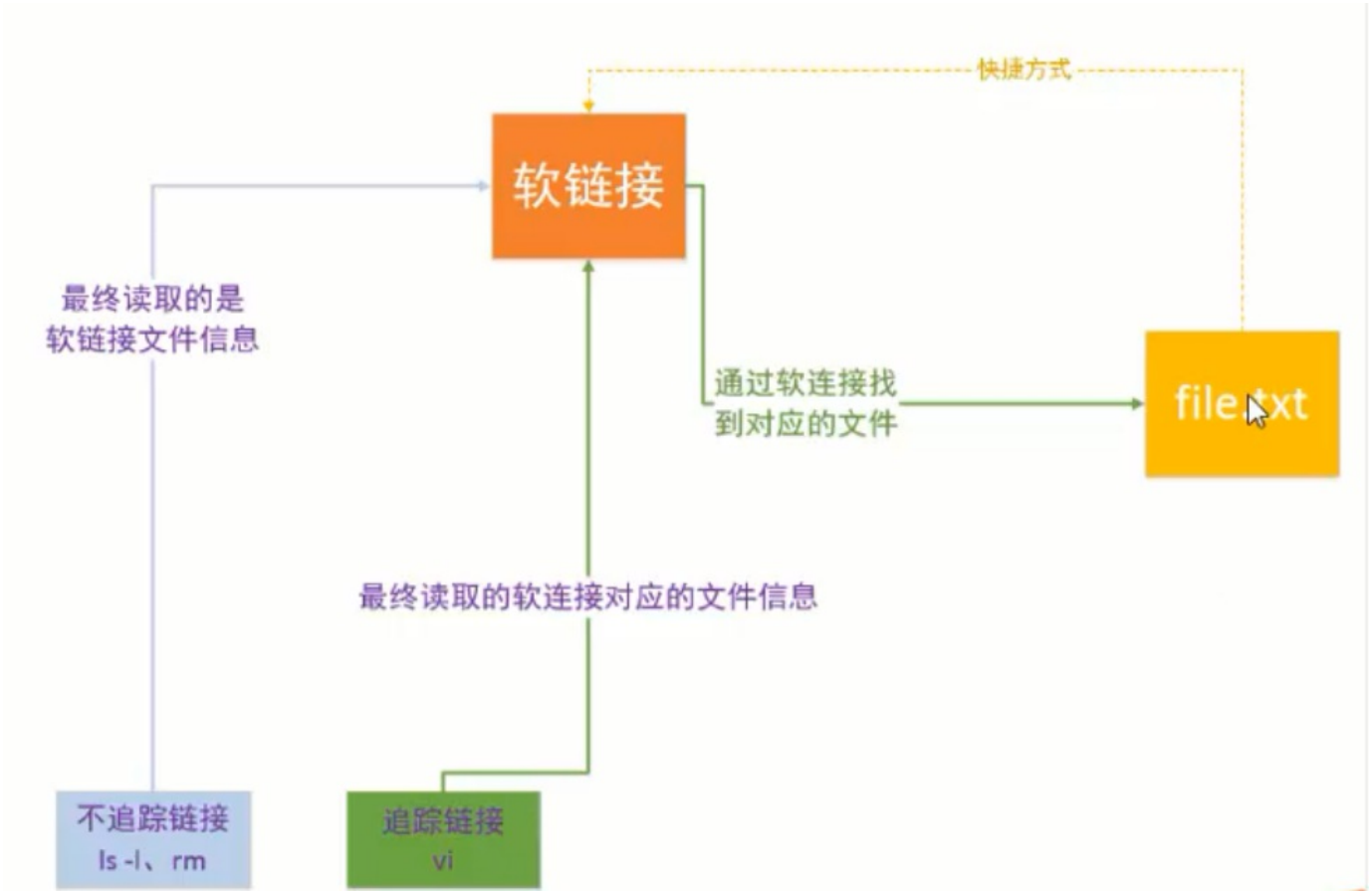
- 文件类型（12-15 bit）

```
S_IFSOCK 0140000 套接字
S_IFLNK 0120000 符号链接（软链接）
S_IFREG 0100000 普通文件
S_IFBLK 0060000 块设备
S_IFDIR 0040000 目录
S_IFCHR 0020000 字符设备
S_IFIFO 0010000 管道
S_IFMT 0170000 掩码
```



lstat函数

- 功能和用法与stat相同
- 特性：不穿透（跟踪）符号链接。如果是软链接，会统计软链接的信息，而不是原始文件的信息。
- 链接的追踪



access函数

- 函数定义

```
int access(const char *pathname, int mode);
```

- 功能

```
测试指定文件是否拥有某种权限。
```

- 参数

```
pathname        文件名
mode            权限类别
                R_OK        是否有读权限
                W_OK        是否有写权限
                X_OK        是否有执行权限
                F_OK        测试一个文件是否存在
```

- 返回值

```
0       所有欲查核的权限都通过了检查
-1      有权限被禁止
```

- 示例

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

int main(int argc, char * argv[])
{
    if(argc < 2)
```

```
    {
        printf("access filename\n");
        exit(1);
    }

    int ret = access(argv[1], W_OK);
    if(ret == -1)
    {
        perror("access");
        exit(1);
    }
    printf("you can write this file.\n");
    return 0;
}
```

chmod函数

- 函数定义

```
int chmod(const char *pathname, mode_t mode);
```

- 功能

```
改变文件的权限
```

- 参数

```
pathname        文件名
mode            权限，必须是一个8进制数字，转换用strtol函数
```

- 返回值

```
0       改变成功
-1      失败
```

- 示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main(int argc, char * argv[])
{
    if(argc < 2)
    {
        printf("chmod filename\n");
        exit(1);
    }

    int ret = chmod(argv[1], 0755);
    if(ret == -1)
    {
        perror("chmod");
        exit(1);
    }
    return 0;
}
```

chown函数

- 函数定义

```
int chown(const char *pathname, uid_t owner, gid_t group);
```

- 功能

```
改变文件的所有者
```

- 参数

```
pathname        文件名
owner           文件所有者ID
group           文件所属组ID

查看UID和GID：/etc/passwd
```

- 返回值

```
0       成功
-1      失败
```

- 示例

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

int main(int argc, char * argv[])
{
    if(argc < 2)
    {
        printf("chown filename\n");
        exit(1);
    }

    int ret = chown(argv[1], 116, 125);
    if(ret == -1)
    {
        perror("chown");
        exit(1);
    }
    return 0;
}
```

truncate函数

- 函数定义

```
int truncate(const char *path, off_t length);
```

- 功能

```
将参数path指定的文件大小改为参数length指定的大小。如果原来的文件大小比参数length大，则超过部分会被删去。
```

- 参数

```
path        文件路径
length      指定文件的大小
```

- 返回值

```
0       成功
-1      失败
```

连接函数

- link函数
  - 函数定义

    ```
    int link(const char *oldpath, const char *newpath);
    ```

  - 功能

    ```
    创建一个硬链接。
    ```

- symlink函数
  - 函数定义

    ```
    int symlink(const char *target, const char *linkpath);
    ```

  - 功能

    ```
    创建一个软链接。
    ```

- readlink函数
  - 函数定义

    ```
    ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
    ```

  - 功能

    ```
    读软链接对应的文件名，而不是读内容。
    ```

  - 示例

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

    int main(int argc, char * argv[])
    {
        if(argc < 2)
        {
            printf("readlink filename\n");
            exit(1);
        }

        char buf[512];
        int ret = readlink(argv[1], buf, sizeof(buf));
        if(ret == -1)
        {
            perror("readlink");
            exit(1);
        }
        buf[ret] = 0;
        printf("buf = %s\n", buf);

        return 0;
    }
    ```

- unlink函数
  - 函数定义

    ```
    int unlink(const char *pathname);
    ```

- 功能

  删除一个文件的目录并减少它的链接数，若成功则返回0，否则返回-1，错误原因存于errno。
  如果想调用这个函数来成功删除文件，你就必须拥有这个文件的所属目录的写和执行权限。

- 使用

  如果是符号链接，删除符号链接。
  如果是硬链接，硬链接数减1，，当减为0时，释放数据块和inode。
  如果文件硬链接数为0，但有进程已打开该文件，并持有文件描述符，则等该进程关闭该文件时，kernel才真正去删除该文件。

- 示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
    int fd = open("tempfile", O_CREAT | O_RDWR, 0664);
    if(fd == -1)
    {
        perror("open");
        exit(1);
    }
    //删除临时文件
    int ret = unlink("tempfile");

    //write file
    write(fd, "hello\n", 5);

    //重置文件指针
    lssek(fd, 0, SEEK_SET);

    //read file
    char buf[24] = {0};
    int len = read(fd, buf, sizeof(buf));

    //将读出的内容写在屏幕上
    write(1, buf, len);

    //close file
    close(fd);
    return 0;
}
```

rename函数

- 函数定义

```
int rename(const char *oldpath, const char *newpath);
```

- 功能

  文件重命名。

2、Linux目录操作相关函数

chdir函数

- 函数定义

```
int chdir(const char *path);
```

- 功能

```
修改当前进程的路径。
```

getcwd函数

- 函数定义

```
char *getcwd(char *buf, size_t size);
```

- 功能

```
获取当前进程工作目录。
```

mkdir函数

- 函数定义

```
int mkdir(const char *pathname, mode_t mode);
```

- 功能

```
创建目录。
注意：创建的目录需要有执行权限，否则无法进入目录。
```

rmdir函数

- 函数定义

```
int rmdir(const char *pathname);
```

- 功能

```
删除一个空目录。
```

opendir函数

- 函数定义

```
DIR *opendir(const char *name);
```

- 功能

```
打开一个目录。
```

readdir函数

- 函数定义

```
struct dirent *readdir(DIR *dirp);
```

- 功能

```
读目录。
```

- 返回值

```
struct dirent {
    ino_t          d_ino;       /* 此目录进入点的inode */
    off_t          d_off;       /* 目录文件开头至此目录进入点的位移 */
    unsigned short d_reclen;    /* d_name的长度, 不包含NULL字符 */
    unsigned char  d_type;      /* d_name所指的文件类型 */
    char           d_name[256]; /* 文件名 */
};

d_type:
DT_BLK      This is a block device.
DT_CHR      This is a character device.
DT_DIR      This is a directory.
DT_FIFO     This is a named pipe (FIFO).
DT_LNK      This is a symbolic link.
DT_REG      This is a regular file.
DT_SOCK     This is a UNIX domain socket.
DT_UNKNOWN  The file type is unknown.

-D_BSD_SOURCE 编译时添加宏定义
```

colsedir函数

- 函数定义

```
int closedir(DIR *dirp);
```

- 功能

```
关闭目录。
```

- 示例

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>

int get_file_count(char * root)
{
    DIR * dir = NULL;
    dir = opendir(root);
    if(dir == NULL)
    {
        perror("opendir");
        exit(1);
    }

    struct dirent * ptr = NULL;
    char path[1024] = {0};
    int total = 0;
    while(ptr = readdir(dir) != NULL)
    {
        //过滤.和..
        if(strcmp(ptr->d_name, ".") == 0 || strcmp(ptr->d_name, "..") == 0)
        {
            continue;
        }
        //如果是目录
        if(ptr->d_type == DT_DIR)
        {
            sprintf(path, "%s/%s", root, ptr->d_name);
            total += getFileNum(path);
        }
```

```
        //如果是文件
        if(ptr->d_type == DT_REG)
        {
            total++;
        }
    }

    closedir(dir);
    return total;
}

int main(int argc, char * argv[])
{
    if(argc < 2)
    {
        printf("./a.out dir\n");
        exit(1);
    }

    int total = get_file_count(argv[1]);
    printf("%s has file numbers %d\n", argv[1], total);

    return 0;
}
```

6、fcntl函数

- 函数定义

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

- 功能

```
改变已经打开的文件的属性。
1、复制一个现有的描述符 -- cmd
        F_DUPFD
2、获取/设置文件描述符标记 -- cmd
        F_GETFD
        F_SETFD
3、获取/设置文件状态标记 -- cmd
        F_GETFL
                    O_RDONLY        只读打开
                    O_WRONLY        只写打开
                    O_RDWR          读写打开
                    O_EXEC          执行打开
                    O_SEARCH        搜索打开目录
                    O_APPEND        追加写
                    O_NONBLOCK      非阻塞模式

        F_SETFL, 可更改的几个标识
                    O_APPEND
                    O_NONBLOCK
4、获取/设置异步I/O所有权 -- cmd
        F_GETOWN
        F_SETOWN
5、获取/设置记录锁 -- cmd
        F_GETLK
        F_SETLK
        F_SETLKW
```

- 示例

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
```

```
#include <unistd.h>
#include <string.h>

int main(int argc, char * argv[])
{
    int fd;
    int flag;

    //测试字符串
    char * p = "we are family!";
    cahr * q = "yes, man.";

    //只写的方式找开文件
    fd = open("test.txt", O_WRONLY);
    if(fd == -1)
    {
        perror("open");
        exit(1);
    }

    //输入新的内容，该部分会覆盖原来旧的内容
    if(write(fd, p, strlen(p)) == -1)
    {
        perror("write");
        exit(1);
    }

    //使用F_GETFL命令得到文件状态标志
    flag = fcntl(fd, F_GETFL, 0);
    if(flag == -1)
    {
        perror("fcntl");
        exit(1);
    }

    //将文件状态标志添加“追加写”选项
    flag |= O_APPEND;
    //将文件状态修改为追加写
    if(fcntl(fd, F_SETFL, flag) == -1)
    {
        perror("fcntl -- append write");
        exit(1);
    }

    //再次输入新内容，该内容会追加到旧内容的后面
    if(write(fd, q, strlen(q)) == -1)
    {
        perror("write again");
        exit(1);
    }

    //关闭文件
    close(fd);
    return 0;
}
```

7、dup, dup2函数

- 函数定义

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- 功能

复制现有的文件描述符。

- 返回值

dup返回的是文件描述符中没有被占用的最小的文件描述符

- 注意事项

dup2两种情况：
1、old --> new，如果new是一个被打开的文件描述符，在拷贝前先关掉new。
2、old和new是同一个文件描述符，不会关掉new，直接返回old。

- dup示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    int fd = open("a.txt", O_RDWR);
    if(fd == -1)
    {
        perror("open");
        exit(1);
    }

    printf("file open fd = %d\n", fd);

    //找到进程文件描述表中==第一个==可用的文件描述符
    //将参数指定的文件复制到该描述符后，返回这个描述符
    int ret = dup(fd);
    if(ret == -1)
    {
        perror("dup");
        exit(1);
    }
    printf("dup fd = %d\n", ret);
    char * buf = "你是猴子派来的救兵吗？？？\n";
    char * buf1 = "你大爷的，我是程序猿！！！\n";

    write(fd, buf, strlen(buf));
    write(ret, buf1, strlen(buf1));

    close(fd);
    return 0;
}
```

- dup2示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    int fd = open("english.txt", O_RDWR);
```

```
    if(fd == -1)
    {
        perror("open");
        exit(1);
    }

    int fd1 = open("a.txt", O_RDWR);
    if(fd1 == -1)
    {
        perror("open");
        exit(1);
    }

    printf("file open fd = %d\n", fd);
    printf("file open fd1 = %d\n", fd1);

    //调用dup2后，fd1和fd同时指向了a.txt
    int ret = dup2(fd1, fd);
    if(ret == -1)
    {
        perror("dup2");
        exit(1);
    }
    printf("current fd = %d\n", ret);
    char * buf = "主要看气质^_^！！！！！！！！！！\n";

    write(fd, buf, strlen(buf));
    write(fd1, "hello, world!", 13);

    close(fd);
    close(fd1);
    return 0;
}
```

- 解决gcc编译过程中c99语法报错的问题

```
alias gcc='gcc -std=gnu99'
```

- 索引节点inode：保存的其实是实际的数据的一些信息，这些信息称为“元数据”（也就是对文件属性的描述）。例如：文件大小，设备标识符，用户标识符，用户组标识符，文件模式，扩展属性，文件读取或修改的时间戳，链接数量，指向存储内容的磁盘区块的指针，文件分类等等。（注意数据分成：元数据+数据本身）

- 注意inode怎样生成的：每个inode节点的大小，一般是128字节或256字节。inode节点的总数，在格式化时就给定（现代OS可以动态变化），一般每2KB就设置一个inode。一般文件系统中很少有文件小于2KB的，所以预定按照2KB分，一般inode是用不完的。所以inode在文件系统安装的时候会有一个默认数量，后期会根据实际的需要发生变化。

- 注意inode号：inode号是唯一的，表示不同的文件。其实在Linux内部的时候，访问文件都是通过inode号来进行的，所谓文件名仅仅是给用户容易使用的。当我们打开一个文件的时候，首先，系统找到这个文件名对应的inode号；然后通过inode号，得到inode信息，最后，由inode找到文件数据所在的block，现在可以处理文件数据了。

- inode与文件的关系：当创建一个文件的时候，就给文件分配了一个inode。一个inode只对应一个实际文件，一个文件也会只有一个inode。inode最大数量就是文件的最大数量。