Linux

系统编程

Linux系统：”所见皆文件“

Linux系统目录：
    bin：存放二进制可执行文件
    boot：存放开机启动程序
    dev：存放设备文件
    home：存放用户
    etc：用户信息和系统配置文件
    lib：库文件
    root：管理员宿主和目录（家目录）
    usr：用户资源管理目录

Linux系统文件类型
    普通文件： -
    目录文件： d
    字符设备文件： c
    块设备文件： b
    软连接： l
    管道文件： p
    套接字： s
    未知文件。

软连接：ln -s file file.soft
    （快捷方式）为保证软连接可以任意搬移，创建时务必对源文件使用绝对路径

硬连接：ln file file.hard
    操作系统给每一个文件赋予唯一的 inode 当有相同 inode 的文件存在时，彼此同步。
    删除时， 只将硬连接计数减一，减到0时，inode被释放。

创建用户：
    sudo adduser 新用户名

修改文件所属用户：
    sudo chown 新用户名 待修改文件

删除用户：
    sudo deluser 用户名

创建用户组：
    sudo addgroup 新组名

修改文件所属用户组：
    sudo chgrp 新用户组名 待修改文件

删除组：
    sudo delgroup 用户组名

hhh

find命令：
    -type 按文件类型搜索 d/p/s/c/b/l f:文件
    -name 按文件名搜索
    -maxdepth 指定搜索深度 应作为第一个参数出现
    -size 按文件大小搜索 单位k、M、G
    -atime、mtime、ctime

grep命令：找文件内容
    grep -r ‘copy’ ./ -n
        -r: 递归遍历目录
        -n: 显示行号
    ps aux | grep ’cupsd‘ ---- 检索进程结果集

软件安装：
    1. 联网
    2. 更新软件资源列表到本地 sudo apt-get update
    3. 安装 sudo apt-get install 软件名
    4. 卸载 sudo apt-get remove 软件名
    5. 使用软件包(.deb)安装 sudo dpkg -i 安装包名

tar压缩：
    tar -zcvf 要生产的压缩包名 压缩材料
        tar zcvf test.tar.gz file1 dir2   使用gzip方式压缩
        tar jcvf test.tar.gz file1 dir2   使用bzip2方式压缩

tar解压：
    将压缩命令中的c--->x
        tar zxvf test.tar.gz   使用gzip方式解压

```
        tar jxvf test.tar.gz    使用bzip2方式解压
```

rar压缩:

```
    rar a -r 压缩包名 压缩材料
        rar a -r testrar.rar test.mp3
```

rar解压:

```
    unrar x 压缩包名
```

zip压缩:

```
    zip -r 压缩包名 压缩材料
```

zip解压:

```
    unzip 压缩包名
```

vim

跳转到指定行:

1. 88G（命令模式）
2. 88（末行模式）

跳转文件首尾:

```
    gg （命令模式）
    G （命令模式）
```

自动格式化程序:

```
    gg = G（命令模式）
```

大括号对应:

```
    %（命令模式）
```

光标移至行首：
    0（命令模式）执行结束，工作模式不变。

光标移至行尾：
    $（命令模式）执行结束，工作模式不变。

删除单个字符：
    x（命令模式）执行结束，工作模式不变。

替换单个字符：
    将待替换的字符用光标选中， r（命令模式），在按欲替换的字符。

删除一个单词：
    dw（命令模式）光标置于单词的首字母进行操作

删除光标至行尾：
    D 或者 d$（命令模式）

删除光标至行首：
    d0（命令模式）

删除指定区域：
    按 V（命令模式）切换为”可视模式“，使用hjkl挪移光标来选中待删除区域，按 d 删除该区域的数据

删除指定一行：
    在光标所在行，按 dd（命令模式）

删除指定N行：
    在光标所在删除首行， 按Ndd（命令模式）

复制一行：
    yy

粘贴：
    p：向后   P：向前

查找：
    1. 找 设想 内容：
        命令模式下，按 ”/“ 输入欲搜索关键字，回车。使用n检索下一个
    2. 找 看到的内容：
        命令模式下，将光标置于单词任意一个字符上，按 ”*“ / “#”

单行替换：
    将光标置于待替换行上，进入末行模式，输入 :s /原数据/新数据

通篇替换：
    末行模式 :%s/原数据/新数据/g g:不加，只替换每行首个

指定行的替换：

末行模式 :起始行号，终止行号s /原数据/新数据/g g:不加，只替换每行首个

```
:29,35s /printf/println/g
```

撤销、反撤销：

u、 ctrl+r（命令模式）

分屏：

sp：横屏分。Ctrl+ww 切换窗口

vsp：竖屏分。Ctrl+ww 切换窗口

跳转man手册：

将光标置于待查看函数单词上，使用 K（命令模式）跳转。指定卷， nK

查看宏定义：

将光标置于待查看宏定义单词上，使用 [d 查看定义语句。

在末行模式执行shell命令：

:!命令   :! ls -l

gcc编译：

4步骤：

预处理 编译 汇编 连接

-E .i -S .s -c .o 无参 a.out

-I：指定头文件所在目录位置

-c：只做预处理、编译、汇编、得到二进制文件！！！

-g：编译时添加调试语句。主要支持gdb调试

-Wall：显示所有警告信息

-D：向程序中”动态“注册宏定义

头文件守卫：防止头文件被重复包含

```
#ifndef _HEAD_H_
#define _HEAD_H_

......
#endif
```

静态库制作及使用步骤：

1. 将 .c 生成 .o 文件

```
gcc -c add.c -o add.o
```

2. 使用 ar 工具制作静态库

```
ar rcs lib库名.a dd.o sub.o div.o
```

3. 编译静态库到可执行文件中

```
gcc test.c lib库名.a -o a.out
```

动态库制作及使用：

1. 将 .c 生成 .o 文件 （生成与位置无关的代码 -fPIC）

```
gcc -c add.c -o add.o -fPIC
```

gdb调试工具： 大前提 程序是自己写的

基础指令：

-g：使用该参数编译可以执行文件，得到调试表。

gdb ./a.out

list: list l 列出源码 根据源码指定 行号设置断点

b: b 20 在20行位置设置断点。

run/r: 运行程序

n/next:下一条指令（会越过函数）

s/step:下一条指令（会进入函数）

p/print: p i 查看变量的值

continue: 继续执行断电后续指令

quit: 退出 gdb 当前调试

其他指令：

run: 使用 run 查找段错误出现位置

finish: 结束当前函数调用

set args: 设置main函数命令行参数

run 子串1 子串2 ...: 设置main函数命令行参数

info b: 查看断点信息表

b 20 if i=5: 设置条件断点

ptype: 查看变量类型

bt: 列出当前程序正存活着的栈帧

frame: 根据栈帧编号，切换栈帧。

makefile:

1 个规则：

目标：依赖条件

（一个tab缩进）命令

1. 目标的时间必须晚于依赖条件的时间，否则，更新目标

2. 依赖条件如果不存在，找寻新的规则去产生依赖条件

ALL: 指定 makefile 的终极目标

2 个函数：

src = $(wildcard ./*.c): 匹配当前工作目录下的所有 .c 文件。将文件名组成列表，赋值给变量src。 src = add.c sub.c div1.c

obj = $(patsubst %.c, %.o, $(src)): 将参数3中，包含参数1的部分，替换为参数2。 obj = add.o sub.o div1.o

clean: (没有依赖)

-rm -rf $(obj) a.out “-”:作用是，删除不存在文件时，不报错。顺序执行结束。

3 个自动变量：

$@: 在规则的命令中，表示规则中的目标。

$^: 在规则的命令中，表示所有依赖条件。

$<: 在规则的命令中，表示第一个依赖条件。如果将该变量应用在模式规则中，它可将依赖条件列表中的依赖依次取出，套用模式规则。

模式规则：

```
%.o:%.c
    gcc -c $< -o %@
```

静态模式规则：

```
$(obj):%.o%.c
    gcc -c $< -o %@
```

伪目标：

```
.PHONY: clean ALL
```

参数：

-n: 模拟执行make、make clean命令。

-f: 指定文件执行 make 命令

系统

open函数：

```
int open(char *pathname, int flags)
```

参数：

pathname：欲打开的文件路径名

flag: 文件打开方式： O_RDONLY | O_WRONLY | O_RDWR
O_CREAR | O_APPEND | O_TRUNC | O_EXCL | O_NONBLOCK...

返回值：

成功：打开文件所得到对应的 文件描述符（整数）

失败：-1 设置errno

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

参数：
pathname：欲打开的文件路径名
flags：文件打开方式：O_RDONLY | O_WRONLY | O_RDWR
O_CREAR | O_APPEND | O_TRUNC | O_EXCL | O_NONBLOCK...
mode：参数3使用的前提，参数2指定了O_CREAT 取值8进制数，用来描述文件的访问权限 rwx 0644
创建文件最终权限 = mode & ~umask
返回值：
成功：打开文件所得到对应的 文件描述符（整数）
失败：-1 设置errno

close函数：
int close(int fd);

错误处理函数：　　与 errno 相关
printf("xxx error: %d\n", errno);

char *strerror(int errnum);
printf("xxx error: %s\n", strerror(errno));

void perror(const char *s);
perror("open error");

read函数：
ssize_t read(int fd, void *buf, size_t count);
参数：
fd：文件描述符
buf：存数据的缓冲区
count：缓冲区大小
返回值：
0：读到文件末尾
成功：读到的字节数
失败：-1 设置 errno
-1：并且 errno = EAGIN 或 EWOULDBLOCK，说明不是read失败，而是read在以非阻塞方式读一个设备文件（网络文件），并且文件无数据。

write函数：
ssize write(int fd, const void *buf, size_t count);
参数：
fd：文件描述符
buf：待写出数据的缓冲区
count：数据大小
返回值：
成功：写入的字节数
失败：-1 设置 errno

文件描述符：
PCB进程控制块：本质 结构体。
成员：文件描述符表
文件描述符： 0/1/2/3/4/.../1023 表中可用的最小的
0 - STDIN_FILENO

    1 - STDOUT_FILENO
    2 - STDERR_FILENO

阻塞、非阻塞：是设备文件、网络文件的属性
    产生阻塞的场景，读设备文件、读网络文件。（读常规文件无阻塞概念）
    /dev/tty -- 终端文件
    open("/dev/tty", O_RDWR | O_NONBLOCK) 设置/dev/tty非阻塞状态（默认为阻塞状态）

fcntl:
    int flgs = fcntl(fd, F_GETFL);
    flgs |= O_NONBLOCK;
    fcntl(fd, F_SETFL, flgs);
    获取文件状态：F_GETFL
    设置文件状态：F_SETFL

lseek函数：

传入参数：
    1．指针作为函数参数。
    2．通常有 const 关键字修饰。
    3．指针指向有效区域，在函数内部做读操作。

传出参数：
    1．指针作为函数参数。
    2．在函数调用之前，指针指向的空间可用无意义，但必须有效。
    3．在函数内部，做写操作
    4．函数调用结束后，充当函数返回值。

传入参数：
    1．指针作为函数参数。
    2．在函数调用之前，指针指向的空间有实际意义。
    3．在函数内部，先做读操作，后做写操作。
    4．函数调用结束后，充当函数返回值。

stat/lstat函数：
    int stat(const char *path, struct stat *buf);
    参数：
        path：文件路径
        buf：(传出参数)存放文件属性
    返回值：
        成功： 0
        失败： -1 errno
    获取文件大小：buf.st_size
    获取文件类型：buf.st_mode
    获取文件权限：buf.st_mode
    符号穿透： stat会 lstat不会

信号

信号共性：

简单、不能携带大量信息、满足条件才发送

信号的特质：

信号是软件层面上的“中断”。一旦信号产生，无论程序执行到什么位置，必须立即停止运行，处理信号，处理接收，再继续执行后续指令。

所有信号的产生及处理全部都是由【内核】完成的

信号相关的概念：

产生信号：

1．按键产生
2．系统调用产生
3．软件条件产生
4．硬件异常产生
5．命令产生

概念：

未决：产生与递达之间的状态
递达：产生并且送达到进程，直接被内核处理掉
信号处理方式：执行默认处理动作、忽略、捕捉（自定义）

阻塞信号集（信号屏蔽字）：本质是位图，用来记录信号的屏蔽状态，一旦被屏蔽的信号，在接触屏蔽前，一直处于未决状态。

未决信号集：本质也是位图，用来记录信号的处理状态，该信号集中的信号，表示已经产生，但尚未被处理。

信号四要素：

信号使用之前，应先确定其四要素，然后再用！！！
信号编号、信号名称、信号对应事件、信号默认处理动作

kill命令和kill函数：
int kill(pid_t pid, int signum)
pid: > 0: 发送信号给指定进程
= 0: 发送信号给跟调用kill函数的那个进程处于同一进程组的进程
< -1:取绝对值，发送信号给该绝对值所对应的进程组的所有组成员
= -1:发送信号给，有权限发送的所有进程
signum: 待发送的信号
返回值：
成功：0
失败：-1 errno

alarm函数：使用自然计时法
定时发送 SIGALRM 给当前进程
unsigned int alarm(unsigned int seconds);
seconds: 定时秒数
返回值：上次定时剩余时间 无错误现象
alarm(0); 取消闹钟
time命令：查看程序执行时间
实际时间 = 用户时间 + 内核时间 + 等待时间 （优化瓶颈IO）

目录

递归遍历目录：ls-R.c
1. 判断命令行参数，获取用户想要查询的目录名。 argv[1]
argc == 1 --- ./
2. 判断用户指定的是否是目录。 stat S_ISDIR(); 封装函数 isFile
3. 读目录：
opendir(dir)

```
        while(readdir()){
            普通文件，直接打印
            目录：
                拼接目录访问绝对路径。sprintf(path, “%s/%s”, dir, d_name)
                递归调用自己。 --- opendir(path) readdir closedir
        }
        closedir()
```

dup和dup2：

```
    int dup(int oldfd);     文件描述符复制
        oldfd:已有文件描述符
        返回：新文件描述符
    int dup2(int oldfd, int newfd);     文件描述符复制，重定向
```

funtl函数实现dup：

```
    int fcntl(int fd, int cmd, ...)
    cmd: F_DUPFD
    参3:  被占用的，返回最小可用的
          未被占用的， 返回=该值的文件描述符
```

进程

进程:
    程序：死的。只占用磁盘空间。 --- 剧本
    进程：活的。运行起来的程序。占用内存、CPU等系统资源。 --- 戏

PCB进程控制块:
    进程id
    文件描述符表
    进程状态：初始态、就绪态、运行态、挂起态、终止态。
    进程工作目录位置
    *umask掩码
    信号相关信息资源
    用户id和组id

fork函数:
    pid_t fork(void);
    创建子进程。父进程各自返回。父进程返回子进程pid。子进程返回0.
    getpid();   getppid();
    循环创建N个子进程模型。每个子进程标识自己的身份

父子进程相同:
    刚fork后，data段、text段、堆、栈、环境变量、全局变量、宿主目录位置、进程工作目录位置、信号处理方式。

父子进程不同:
    进程id、返回值、各自的父进程、进程创建时间、闹钟、未决信号集。

父子进程共享:
    读时共享、写时复制。 --- 全局变量
    1. 文件描述符
    2. mmap映射区

gdb调试:
    设置父进程调试路径：set follow-fork-mode parent
    设置子进程调试路径：set follow-fork-mode child

exec函数族:
    使进程执行某一程序。成功无返回值，失败返回-1
    int execlp(const char *file, const char *arg, ...); 借助PATH环境变量找寻待执行程序
        参1：程序名
        参2：argv0
        参3：argv1
        ... argvN
        哨兵：NULL
    int execl(const char *path, const char *arg, ...); 自己指定待执行程序路径

ps ajx --- 查看pid ppid gid sid

孤儿进程:
    父进程先于子进程终止，子进程沦为“孤儿进程”，会被init进程领养。

僵尸进程：
    子进程终止，父进程尚未对子进程回收，在此期间，子进程为“僵尸进程”

wait函数： 回收子进程退出资源
    pid_t wait(int *status);
    参数：（传出）回收进程的状态
    返回值：成功：回收进程的pid
           失败：-1 errno

    函数作用1： 阻塞等待子进程退出
    函数作用2： 清理子进程残留在内核的pcb资源
    函数作用3： 通过传出参数，得到子进程结束状态

    获取子进程正常终止值：
        WIFEXITED(status) -- 为真 -- 调用WEXITSTATUS(status) -- 得到子进程退出值
    获取导致子进程异常终止信号：
        WIFSIGNALED(status) -- 为真 -- 调用WTERMSIG(status) -- 得到导致子进程异常终止的信号编号。

waitpid函数： 指定某一个进程进行回收，可以设置非阻塞 waitpid(-1, &status, 0) == wait(&status)
    pid_t waitpid(pid_t, int *status, int optoins);
    参数：
        pid：指定回收的子进程pid
            > 0: 待回收的子进程pid
            -1: 任意子进程
            0:   同组的子进程
        status：（传出）回收进程的状态
        options：WNOHANG指定回收方式为 非阻塞。
    返回值：
        > 0: 表成功回收的子进程pid
        0 : 函数调用时，参3指定了WNOHANG，并且没有子进程结束
        -1 : 失败 errno

总结：
    wait、waitpid 一次调用，回收一个子进程
                   回收多个 需 while 循环

进程间通信的常用方式、特征；
    管道：简单
    信号：开销小
    mmap映射：非血缘关系进程间
    socket（本地套接字）：稳定

管道：
    实现原理：内核借助环形队列机制，使用内核缓冲区实现。
    特质：
        1. 伪文件

2. 管道中的数据只能一次读取
3. 数据在管道中，只能单向流动

局限性：

1. 自己写，不能自己读
2. 数据不可以反复读
3. 半双工通信

pipe函数： 创建并打开管道

int pipe(int fd[2]);

参数：

fd[0]: 读端

fd[1]: 写端

返回值：

成功：0

失败：-1 errno

管道的读写行为;

读管道：

1. 管道有数据，read返回实际读到的字节数
2. 管道无数据，1）无写端，read返回0（类似文件读到文件尾）
   2）有写端，read阻塞等待

写管道：

1. 无读端，异常终止（SIGPIPE导致的）
2. 有读端：1）管道已满，阻塞等待
   2）管道未满，返回写出的字节个数

守护进程：

deamon进程。通常运行于操作系统后台，脱离控制终端。一般不与用户直接交互。周期性的等待某个事件发送或周期性执行某一动作。不受用户登录注销影响，通常采用以 d 结尾的命名方式

守护进程创建步骤：

1. fork子进程，让父进程终止。
2. 子进程调用 setsid() 创建新回话。
3. 通常根据需要，改变工作目录位置 chdir() 防止目录被卸载。
4. 通常根据需要，重设 umask文件权限掩码，影响新文件的创建权限
5. 通常根据需要，关闭/重定向 文件描述符
6. 守护进程 业务逻辑 while()

线程

线程概念：

进程：有独立的 进程地址空间，有独立的pcb。　　分配资源的最小单位

线程：有独立的pcb，没有独立的进程地址空间。　　最小单位的执行

ps -Lf 进程id --- 线程号 LWP --- cpu执行的最小单位

线程共享：

线程同步：

协同步调，对公共区域数据按序访问。防止数据混乱，产生与时间有关的错误

锁的使用：

建议锁！ 对公共数据进行保护，所有线程【应该】在访问公共数据前先拿锁 再访问，但 锁本身不具备强制性。

使用mutex（互斥量、互斥锁）基本步骤：pthread_mutex_t

1. pthread_mutex_t lock； 创建锁
2. pthread_mutex_init； 初始化
3. pthread_mutex_lock； 加锁
4. 访问共享数据（stdout）
5. pthread_mutex_unlock();解锁
6. pthread_mutex_destory；销毁锁

注意事项：

尽量保证锁的粒度，越小越好（访问共享数据前，加锁。访问结束【立即】解锁）

互斥锁，本质是结构体，可以看成整数，初始值为 1 （pthread_mutex_init() 函数调用成功）

加锁： --操作　　阻塞线程

解锁： ++操作　　唤醒阻塞在锁上的线程

try锁：尝试加锁，成功--　失败返回，同时设置错误号 EBUSY

restrict关键字：

用来限定指针变量。被该关键字限定的指针变量所指向的内存操作，必须由本指针完成。

死锁：

是使用锁不恰当导致的现象。

1. 对一个锁反复lock
2. 两个线程，各自持有一把锁，请求另一把。

读写锁：

锁只有一把，以读方式给数据加锁 -- 读锁。 以写方式给数据加锁 -- 写锁。
读共享，写独占
写锁优先级高
相较于互斥量而言，当 读 线程多的时候，提高访问效率。

网络编程

协议：一组规则

分层模型结构：
OSI七层模型： 物数网传会表应
TCP/IP四层模型：网网传应
应用层：http、ftp、nfs、ssh、telnet
传输层：TCP、UDP
网络层：IP、ICMP、IGMP
网络接口层：以太网帧协议、ARP

	C/S(client-server)	**B/S(browser-server)**
优点	缓存大量数据、协议选择灵活，速度快	安全性、跨平台、开发工作量较小
缺点	安全性、跨平台、开发工作量较小	不能缓存大量数据、严格遵守http

网络传输流程：
数据没有封装之前，是不能在网络中传递的。
数据->应用层->传输层->网络层->链路层
以太网帧协议：

ARP协议：根据IP地址获取mac地址
以太网帧协议：根据mac地址，完成数据包传输

IP协议：
版本：IPv4、IPv6 -- 4位
TTL：time to live 设置数据包在路由节点中的跳转上线，每经过一个路由节点，该值-1，减为0的路由，有义务将该数据包丢弃
源IP：32位 -- 4字节 192.168.1.108 --- 点分十进制IP地址（string） ---二进制
目的IP：32位 -- 4字节

IP地址：可以在网络环境中，唯一标识一台主机
端口号：可以在网络的一台主机上，唯一标识一个进程
IP地址 + 端口号：可以在网络环境中，唯一标识一个进程

UDP：
16位：源端口号 2^16 = 65536
16位：目的端口号

TCP协议：
16位：源端口号 2^16 = 65536
16位：目的端口号
32序号
32确认序号
6个标志位
16位窗口大小 2^16 = 65536

网络套接字：socket
一个文件描述符指向一个套接字（该套接字内部由内核借助两个缓冲区实现）
在通信过程中，套接字一定是成对出现的

网络字节序：
小端法：(pc本地存储) 高位存高地址，低位存低地址 int a = 0x12345678
大端法：(网络存储) 高位存低地址，低位存高地址
htonl → 本地-->网络(IP)
htons → 本地-->网络(port)
htonl → 网络-->本地(IP)
htonl → 网络-->本地(port)

TCP通信流程

TCP通信流程分析：
server：
1. socket() 创建socket
2. bind () 绑定服务器地址结构

3. listen() 设置监听上限
4. accept() 阻塞监听客户端连接
5. read(fd) 读socket获取客户端数据
6. 小--大写   toupper()
7. write(fd)
8. close()

client:

1. socket()   创建socket
2. connect()   与服务器建立连接
3. write()   写数据到socket
4. read()   读转换后的数据
5.显示读取结果
6.close()

IP地址转换函数:

int inet_pton(int af, const char *src, void *dst);   本地字节序(string IP) → 网络字节序

af: AF_INET、AF_INET6
src: 传入，IP地址(点分十进制)
dst: 传出，转换后的 网络字节序的 IP地址
返回值:
成功: 1
异常: 0，说明src指向的不是一个有效的IP地址
失败: -1

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size); 网络字节序 → 本地字节序(string IP)

af: AF_INET、AF_INET6
src: 网络字节序IP地址
dst: 本地字节序(string IP)
size: dst的大小
返回值:
成功: dst
失败: NULL

sockaddr地址结构:

```
struct sockaddr_in addr;
addr.sin_family = AF_INET / AF_INET6    man 7 ip
addr.sin_port = htons(9527);
    int dst;
    inet_pton(AF_INET, “192.157.22.45”, (void *)&dst);
addr.sin_addr.s_addr = dst;
```

【*】addr.sin_addr.s_addr = htonl(INADDR_ANY);   取出系统中有效的任意IP地址，二进制类型

```
bind(fd, (struct sockaddr *)&addr, size);
```

socket函数:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

创建一个套接字

domain: AF_INET、AF_INET6、AF_UNIX
type: SOCK_STREAM、SOCK_DGRAM

protocol: 0
返回值:
成功: 新套接字所对应文件描述符
失败: -1 errno

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen) 给socket绑定一个地址结构(IP + port)
sockfd: socket 函数返回值
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_port = htons(8888);
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr: 传入参数(struct sockaddr *)&addr
addrlen: sizeof(addr) 地址结构的大小
返回值:
成功: 0
失败: -1 errno

int listen(int sockfd, int backlog); 设置同时与服务器建立连接的上限数(同时进行3次握手的客户端数量)
sockfd: socket 函数返回值
backlog: 上限数值，最大值128
返回值:
成功: 0
失败: -1 errno

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen); 阻塞等待客户端建立连接，成功的话，返回一个与客户端成功连接的socket文件描述符
sockfd: socket 函数返回值
addr: 传出参数，成功与服务器建立连接的那个客户端的地址结构(IP + port)
socklen_t clit_addr_len = sizeof(addr);
addrlen: 传入传出 &clit_addr_len
入: addr的大小。 出:客户端addr实际大小。
返回值:
成功: 能与服务器进行数据通信的socket对应的文件描述符
失败: -1 errno

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
sockfd: socket 函数返回值
struct sockaddr_in srv_addr; 服务器地址结构
srv_addr.sin_family = AF_INET;
srv_addr.sin_port = 9527; 跟服务器bind时设定的 port 完全一致。
srv_addr.sin_addr.s_addr
inet_pton(AF_INET, “服务器的IP地址”, &srv_addr.sin_addr.s_addr)
addr: 传入参数。服务器的地址结构
addrlen: 服务器的地址结构的大小
返回值:
成功: 0
失败: -1 errno

如果不是有bind绑定客户端地址结构，采用“隐式绑定”

TCP概念笔记

三次握手：

主动发起连接请求端，发送 SYN 标志位，请求建立连接。携带序号号、数据字节数（0）、滑动窗口大小。

被动接受连接请求端，发送 ACK 标志位，同时携带 SYN 请求标志位。携带序号、确认序号、数据字节数（0）、滑动窗口大小。

主动发起连接请求端，发送 ACK 标志位，应答服务器连接请求。携带确认序号。

四次挥手：

主动关闭连接请求端，发送 FIN 标志位。

被动关闭连接请求端，应答 ACK 标志位。 ---半关闭完成

被动关闭连接请求端，发送 FIN 标志位。

主动关闭连接请求端，应答 ACK 标志位。 ---连接全部关闭

滑动窗口：

发送给连接对端，本端的缓冲区大小（实时），保证数据不会丢失。

错误处理函数：

封装目的：

在server.c编程过程中突出逻辑，将出错处理与逻辑分开，可以直接跳转man手册。

【wrap.c】

存放网络通信相关常用 自定义函数

命名格式：系统调用函数首字符大写，方便查看man手册。如：Listen()、Accept()

函数功能：调用系统调用函数，处理出错场景

在 server.c 和 client.c 中调用 自定义函数

联合编译 server.c 和 wrap.c 生成 server

client.c 和 wrap.c 生成 client

【wrap.h】

存放 网络通信相关常用 自定义函数原型(声明)

readn：

读 N 个字节

readline：

读一行。

read 函数的返回值：

1. > 0 实际读到的字节数
2. = 0 已经读到结尾（对端已经关闭） 【! 重! 点! 】
3. -1 应进一步判断 errno 的值：

errno = EAGAIN or EWOULDBLOCK： 设置了非阻塞方式 读。 没有数据到达。
errno = EINTR 慢速系统调用被 中断。
errno = “其他情况” 异常。

TCP状态时序图:
结合三次握手、四次挥手 理解记忆
1. 主动发起连接请求端： CLOSE -- 发送SYN -- SEND_SYN -- 接收ACK、SYN -- SEND_SYN -- 发送ACK -- ESTABLISHED(数据通信态)
2. 主动关闭连接请求端： ESTABLISHED(数据通信态)-- 发送FIN -- FIN_WAIT_1 -- 接收ACK -- FIN_WAIT_2(半关闭) -- 接收对端发送FIN
-- FIN_WAIT_2(半关闭) -- 回发ACK -- TIME_WAIT(只有主动关闭连接方，会经历该状态) -- 等2MSL时长 -- CLOSE

3. 被动接收连接请求端： CLOSE -- LISTEN -- 接收SYN --LISTEN-- 发送ACK、SYN -- SYN_RCVD -- 接收ACK -- ESTABLISHED(数据通信态)
4. 被动关闭连接请求端： ESTABLISHED(数据通信态) -- 接收FIN -- ESTABLISHED(数据通信态) -- 发送ACK
-- CLOSE_WAIT(说明对端【主动关闭连接端】处于半关闭状态) -- 发送FIN -- LAST_ACK -- 接收ACK -- CLOSE

重点记忆： ESTABLISHED、FIN_WAIT_2 <--> CLOSE_WAIT、TIME_WAIT(2MSL)
netstat -apn | grep 端口号

2MSL时长:
一定出现在【主动关闭连接请求端】 --- 对应 TIME_WAIT 状态
保证 最后一个 ACK 能成功被对端接收。（等待期间，对端没收到我发的ACK，对端会再次发送FIN请求）

端口复用:
```
int opt = 1;         // 设置端口复用
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
```

半关闭:
通信双方中，只有一端关闭通信。 --- FIN_WAIT_2
close(cfd);
shutdown(int fd, int how);
how: SHUT_RD 关读端
SHUT_WR 关写端
SHUT_RDWR 关读写
shutdown 在关闭多个文件描述符应用的文件时，采用全关闭方法。close 只关闭一个

多并发服务器

多进程并发服务器:
1. Socket(); 创建 监听套接字 lfd
2. Bind(); 绑定地址结构 Struct sockaddr_in addr

```
3. Listen();
4. while(1){
    cfd = Accept();    接收客户端连接请求
    pid = fork();
    if (pid == 0){    子进程 read(cfd) --- 小->大 --- write(cfd)
        close(lfd);   关闭用于建立连接的套接字 lfd
        read();
        小->大
        write();
    } else if (pid > 0){
        close(cfd);    关闭用于与客户端通信的套接字 cfd
        continue;
    }
}
5. 子进程:
       close(lfd)
       read()
       小->大
       write()
   父进程:
       close(cfd)
       注册信号捕捉函数:   SIGCHLD
       在回调函数中，完成子进程回收
           while ( waitpid() )
```

多线程并发服务器:

```
1. Socket()     创建 监听套接字 lfd
2. Bind()       绑定地址结构 Struct sockaddr_in addr
3. Listen()
4. while(1){
    cfd = Accept(lfd, );
    pthread_create(&tid, NULL, tfn, (void *)cfd);
    pthread_detach(tid);    // pthread_join(tid, void **) 新线程--专用于回收子线程。
}
5. 子线程:
    void *tfn(void * arg){
        // close(lfd)   不能关闭。主线程要使用lfd
        read(cfd)
        小->大
        write(cfd)
        pthread_exit( (void *)10 )
    }
```

select多路IO转接

select多路IO转接：

原理：借助内核，select来监听，客户端连接、数据通信事件。

```
void FD_ZERO(fd_set *set);      ---清空一个文件描述符集合。
    fd_set rset;
    FD_ZERO(&rset);

void FD_SET(int fd, fd_set *set);   ---将待监听的文件描述符 添加到监听集合中
    FD_SET(3, &rset);
    FD_SET(5, &rset);
    FD_SET(6, &rset);

void FD_CLR(int fd, fd_set *set);   ---将一个文件描述符从监听集合中 移除。
    FD_CLR(4, &rset);

int  FD_ISSET(int fd, fd_set *set); ---判断一个文件描述符是否在监听集合中。
    返回值:    在 1     不在 0
    FD_ISSET(4, &rset);


    int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
        nfds:       监听的所有文件描述符中，最大文件描述符+1
        readfds:    读 文件描述符监听集合。          传入、传出参数
        write:      写 文件描述符监听集合。          传入、传出参数      NULL
        exceptfds:  异常 文件描述符监听集合          传入、传出参数      NULL
        timeout:    > 0:    设置监听超时时长
                    NULL:   阻塞监听
                    0:      非阻塞监听 轮询
        返回值:
                    > 0:    所有监听集合（3个）中，满足对应事件的总数
                    0:      没有满足监听条件的文件描述符
                    -1:     errno
```

思路分析：

```
    int maxfd = 0;              创建套接字
    lfd = socket();
    maxfd = lfd;
    bind();                     绑定地址结构
    listen();                   设置监听上线
    fd_set rset, allset;        创建 读 监听集合
    FD_ZERO(&allset);           将 读 监听集合清空
    while(1){
        rset = allset;          保存监听集合
        ret = select(lfd+1, &rset, NULL, NULL, NULL);   监听文件描述符集合对应的事件
        if(ret > 0){                                    有监听的描述符满足对应事件
            if(FD_ISSET(lfd, &rset)){                   // 1 在    0 不在
                cfd = accept();                         建立连接，返回用于通信的文件描述符
                maxfd = cfd;
```

```
                FD_SET(cfd, &allset);                       添加到监听通信描述符集合
中
            }
            for(i = lfd+1; i <= maxfd; i++){
                FD_ISSET(i, &rset);                         有read、write事件
                read();
                小---大
                write();
            }
        }
    }
```

select优缺点：

缺点：监听上限受文件描述符限制。最大1024

检测满足条件的 fd，自己添加业务逻辑提高小。提高了编码难度

优点：跨平台。Win Linux macOS Unix 类Unix mips

poll

poll:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
    fds: 监听的文件描述符【数组】
        struct pollfd{
            int fd:         待监听的文件描述符
            short ecents:   待监听的文件描述符对应的监听事件
                            取值：POLLIN、POLLOUT、POLLERR
            short revents:  传入时，给 0 如果满足对应事件的话，返回 非 0 --- POLLIN、
POLLOUT、POLLERR
        }
    nfds： 监听数组的 实际有效监听个数。
    timeout： >0 : 超时时长。单位毫秒
              -1:  阻塞等待
              0:   不阻塞
    返回值：返回满足对应监听事件的文件描述符 总个数。

优点：
    自带数组结构。可以将 监听事件集合 和返回事件集合 分离。
    拓展 监听上限。超出 1024 限制
缺点：
    不能跨平台。 Linux
    无法直接定位满足监听事件的文件描述符，编码难度大
```

read 函数的返回值：

1. > 0 实际读到的字节数

2. = 0 已经读到结尾（对端已经关闭） 【! 重! 点! 】
3. -1 应进一步判断 errno 的值：
errno = EAGAIN or EWOULDBLOCK： 设置了非阻塞方式 读。 没有数据到达。需要 再次读
errno = EINTR 慢速系统调用被 中断。被异常中断 需要重启
errno = ECONNRESET 说明连接被 重置。需要close()，移除监听队列

突破 1024 文件描述符限制：

cat /proc/sys/fs/file-max -- 当前计算机所能打开的最大文件个数，受硬件影响
ulimit -a 当前用户下的进程，默认打开文件描述符个数，缺省为 1024
修改：
打开 sudo vi /etc/security/limits.conf 写入：
* soft nofile 65536 设置默认值，可以直接借助命令修改（注销用户使其生效）
* hard nofile 100000 命令修改上限

epoll

epoll实现多路IO转接思路：

```
lfd = socket();     监听连接事件lfd
bind();
listen();

int epfd = epoll_create(1024);      epfd 监听红黑树的树根
struct epoll_event tep, ep[1024];
tep 用来设置单个 fd 属性，ep是epoll_wait()传出的满足监听事件的数组
tep.events = EPOLLIN;       初始化 lfd 的监听属性
tep.data.fd = lfd;

epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &tep);  将 lfd 添加到监听红黑树上。
while(1){
    ret = epoll_wait(epfd, ep, 1024, -1)    实施监听
    for(i = 0; i < ret; i++){
        if(ep[i].data.fd == lfd){
            cfd = Accept();

            tep.events = EPOLLIN;           初始化 cfd 的监听属性。
            tep.data.fd = cfd;

            epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &tep);
        } else{         cfd们 满足读事件， 有客户端写数据来。
            n = read(ep[i].data.fd, buf, sizeof(buf));
            if(n == 0){
                epoll_ctl(epfd, EPOLL_CTL_DEL, ep[i].data.fd, NULL);       将关闭的
cfd，从监听树上删除
                close(ep[i].data.fd);
            } else if(n > 0){
                小 -- 大
                write(ep[i].data.fd, buf, n);
```

```
            }
        }
    }
}
```

epoll事件模型：

ET模式：边沿触发

缓冲区剩余未读尽的数据不会导致 epoll_wait 返回。新的事件满足，才会触发

```
struct epoll_event event;
event.events = POLLIN | EPOLLET;
```

LT模式：水平触发（默认）

缓冲区剩余未读尽的数据会导致 epoll_wait 返回。

结论：

```
epoll 的 ET 模式，高效模式，但是只支持非阻塞模式  轮询
struct epoll_event event;
event.events = POLLIN | EPOLLET;
epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &event);
int flg = fcntl(cfd, F_GETFL);
flg |= O_NONBLOCK;
fcntl(cfd, F_SETFL, flg);
```

优点：

高效，突破1024文件描述符。

缺点：

不能跨平台。Linux

epoll:

```
int epoll_create(int size);      创建一颗监听红黑树
```

size: 创建的红黑树的监听节点数量（仅供内核参考）

返回值：

成功：指向新创建的红黑树的根节点的 fd

失败：-1 errno

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);      操作监听红黑树
```

epfd: epoll_create 函数的返回值

op: 对该监听红黑树所做的操作

EPOLL_CTL_ADD   添加 fd 到监听红黑树

EPOLL_CTL_MOD   修改 fd 在监听红黑树上的监听事件

EPOLL_CTL_DEL   将一个 fd 从监听红黑树上摘下（取消监听）

fd: 待监听的 fd

event: 本质 struct epoll_event 结构体 地址

events: EPOLLIN、EPOLLOUT、EPOLLERR

data: 联合体

int fd: 对应监听事件的 fd

void *ptr

```
struct evt{
    int fd;
    void(*func)(int fd);
} *ptr;
```

```
                uint32_t u32
                uint64_t u64
        返回值: 成功0  失败-1 errno

    int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int
timeout);   阻塞监听
        epfd: epoll_create 函数的返回值
        events: 传出参数【数组】，满足监听条件的那些 fd 结构体。
        maxevents:数组 元素的总个数 1024
            struct epoll_events[1024];
        timeout： >0 : 超时时长。单位毫秒
                   -1:  阻塞等待
                   0:   不阻塞
        返回值:
            > 0: 满足监听的总个数，可以用作循环上限
            0:   没有 fd 满足监听事件
            -1:  失败 errno
```

epoll反应堆模型:

```
    epoll ET模式 + 非阻塞、轮询 + void *ptr。

    原来:
        socket、bind、listen -- epoll_createa创建监听红黑树 -- 返回epfd -- epoll_ctl()向树
上添加一个监听fd -- while(1)
        -- epoll_wait()监听 -- 对应监听fd有事件产生 -- 返回监听满足数组 -- 判断返回数组元素 --
lfd满足 -- accept()
        -- cfd满足 -- read() -- 小变大 -- write回去。

    反应堆：不但要监听cfd的读事件，还要监听cfd的写事件。
        socket、bind、listen -- epoll_createa创建监听红黑树 -- 返回epfd -- epoll_ctl()向树
上添加一个监听fd -- while(1)
        -- epoll_wait()监听 -- 对应监听fd有事件产生 -- 返回监听满足数组 -- 判断返回数组元素 --
lfd满足 -- accept()
        -- cfd满足 -- read() -- 小变大 -- cfd从监听红黑树上摘下 -- EPOLLOUT -- 回调函数 --
epoll_ctl()
        -- EPOLL_CTL_ADD重新放到红黑树监听写事件 -- 等待epoll_wait返回 -- 说明cfd可写 --
write回去
        -- cfd从监听红黑树上摘下 -- EPOLLOUT -- 回调函数 -- epoll_ctl() -- EPOLL_CTL_ADD重新
放到红黑树监听写事件
        -- 等待epoll_wait返回。


    eventset函数:
        设置回调函数:
            lfd -- acceptconn()
            cfd -- recvdata()
            cfd -- senddata()
    eventadd函数:
```

讲一个 fd 添加到监听红黑树。设置监听读事件，或是写事件

网络编程中：

read -- recv()

write -- send()

libevent

特性：

基于“事件”异步通信模型 --- 回调

libevent框架：

1.创建 event_base （乐高底座）

```
struct event_base *event_base_new(void);
struct event_base *base = event_base_new();
```

2.创建 事件event

常规事件：event -- event_new();

带缓冲区的事件：bufferevent -- bufferevent_socket_new();

3.将事件添加到 base 上

```
int event_add(struct event *ev, const struct timeval *tv);
```

4.循环监听事件满足

```
int event_base_dispatch(struct event_base *base);
```

5.释放 event_base

```
event_base_free(base);
```

创建事件event：

```
struct event *ev;
struct event *event_new(struct event_base *base, evutil_socket_t fd, short what,
event_callback_fn cb; void *arg);
```

base: event_base_new()的返回值。

fd: 绑定到 event 上的文件描述符。

what: 对应的事件(r w e)

EV_READ 一次读事件

EV_WRITE 一次写事件

EV_PERSIST 持续触发。结合 event_base_dispatch 函数使用，生效。

cb: 一旦事件满足监听条件，回调的函数

```
typedef void(*event_callback_fn)(evutil_socket_t fd, short, void *)
```

arg: 回调的函数的参数。

返回值：成功创建的 event

添加事件到 event_base

```
int event_add(struct event *ev, const struct time *tv);
```

ev: event_new()的返回值。

tv: NULL

从 event_base 上摘下事件：　　【了解】

```
    int event_del(struct event *ev);
        ev: event_new()的返回值。
```

销毁事件：

```
    int event_free(struct event *ev);
        ev: event_new()的返回值。
```

未决和非未决：

```
    非未决：没有资格被处理
    未决：有资格被处理，但尚未被处理
        event_new() -- event -- 非未决 -- event_add() -- 未决 --dispatch() && 监听事件被触发 -- 激活态 -- 执行回调函数
        -- 处理态 -- 非未决 event_add && EV_PERSIST -- 未决 -- event_del() -- 非未决
```

带缓冲区的事件bufferevent

```
    #include <event2/bufferevent.h>
    read / write 两个缓冲，借助队列实现。
```

创建、销毁bufferevent：

```
    struct bufferevent *ev;
    struct bufferevent *bufferevent_socket_new(struct event_base *base, evutil_socket_t fd, enum bufferevent_options options);
        base: events_base
        fd: 封装到bufferevent内的fd
        options: BEV_OPT_CLOSE_ON_FREE
    返回：成功创建的bufferevent对象
    void bufferevent_socket_free(struct bufferevent *ev);
```

给bufferevent设置回调：

```
    对比event： event_new(fd, callback);       event_add -- 挂到event_base上
                bufferevent_socket_new(fd)   bufferevent_setcb(callback)

    void bufferevent_setcb(struct bufferevent *bufev,
                           bufferevent_data_cb readcb,
                           bufferevent_data_cb writecb,
                           bufferevent_event_cb eventvb,
                           void *cbarg);
    bufev: bufferevent_socket_new() 返回值
    readcb: 设置 bufferevent 读缓冲，对应回调 read_cb{ bufferevent_read()读数据 }
    writecb: 设置 bufferevent 写缓冲，对应回调 write_cb{ } -- 给回调者发送写成功通知。可以NULl
    eventcb: 设置 事件回调。也可以传NULL
        type void(*bufferevent_event_cb)(struct bufferevent *bev, short events, void *ctx);
        void event_cb(struct bufferevent *bev, short events, void *ctx)
        {
            ......
        }
```

```
        events: BEV_EVENT_CONNECTED

    cbarg: 上述回调函数使用的参数。
    read 回调函数类型:
        type void(*bufferevent_data_cb)(struct bufferevent *bev, void *ctx);
        void read_cb(struct bufferevent *bev, void *cbarg)
        {
            ......
            bufferevent_read(); --- read()
        }
    bufferevent_read()函数的原型:
        size_t bufferevent_read(struct bufferevent *bev, void *buf, size_t bufsize);

    write回调函数类型:
        int bufferevent_write(struct bufferevent *bufev, const void *data, size_t
size);
```

启动、关闭 bufferevent 的缓冲区:

```
    void bufferevent_enable(struct bufferevent *bufev, short events);   启动
    events: EV_READ、EV_WRITE、EV_READ | EV_WRITE
    默认: write缓冲是enable、read缓冲是disable
        bufferevent_enable(ev, EV_READ);    --开启读缓冲。
```

连接客户端:

```
    socket(); connect();
    int bufferevent_socket_connect(struct bufferevent *bev, struct sockaddr *address,
int addrlen);
        bev: bufferevent 事件对象（封装了fd）
        address、len: 等同于 connect() 的 参数2/3
```

创建监听服务器:

```
    ------ socket();  bind();  listen();  accept();
    struct evconnlistener *listener
    struct enconnlistener *evconnlistener_new_bind(
        struct event_base *base,
        evconnlistener_cb cb,
        void *ptr,
        unsigned flags,
        int backlog,
        const struct sockaddr *sa,
        int socklen);

    base:    event_base
    cb:      回调函数。一旦被回调，说明在其内部应该与客户端完成数据读写操作 进行通信。
    ptr:     回调函数的参数。
    flags:   LEV_OPT_CLOSE_ON_FREE | LEV_OPT_REUSEABLE
    backlog: listen() 的参数2。 -1 表最大值
    sa:      服务器自己的地址结构
    socklen: 服务器自己的地址结构的大小
```

释放监听服务器：

```
void evconnlistener_free(struct evconnlistener *lev);
```

服务器端 libevent 创建TCP连接：

1. 创建event_base
2. 创建bufferevent事件对象。bufferevent_socket_new();
3. 使用bufferevent_setcb()函数给bufferevent的 read、write、event设置回调函数。
4. 当监听的事件满足时，read_cb会被调用，在其内部bufferevent_read();读
5. 使用evconnlistener_new_bind创建监听服务器，设置其回调函数，当有客户端连接时，这个回调函数会被调用。
6. 封装listener_cb()在函数内部，完成与客户端通信。
7. 设置读缓冲、写缓冲的使能状态enable、disable
8. 启动循环event_base_dispatch();
9. 释放连接。

Libevent实现TCP服务器流程：

1. 创建event_base;
2. 创建服务器连接监听器 evconnlistener_new_bind();
3. 在 evconnlistener_new_bind 的回调函数中，处理接收连接后的操作。
4. 回调函数被调用，说明有一个新客户端连接上来。会得到一个新 fd 用于跟客户端通信（读写）
5. 使用 bufferevent_socket_new() 创建一个新 bufferevent 事件，将 fd 封装到这个事件对象中。
6. 使用 bufferevent_setcb 给这个事件对象的 read、write、event 设置回调。
7. 设置 bufferevent 的读写缓冲区 enable / disable;
8. 接收、发送数据 bufferevent_read() / bufferevent_write();
9. 启动循环监听 event_base_sidpatch();
10. 释放资源

Libevent实现TCP客户端流程：

1. 创建 event_base;
2. 使用 bufferevent_socket_new() 创建一个用跟服务器通信的 bufferevent 事件对象。
3. 使用 bufferevent_socket_connect() 连接服务器。
4. 使用 bufferevent_setcb() 给 bufferevent 对象的read、write、event 设置回调。
5. 设置 bufferevent 对象的读写缓冲区 enable / disable;
6. 接收、发送数据 bufferevent_read() / bufferevent_write();
7. 启动循环监听 event_base_sidpatch();
8. 释放资源

使用技巧

Linux基本指令

1.创建文件 vi filename
        按i键进入编辑模式，esc或shift+zz退出编辑模式。
2.打开文件 vi filename
        esc键退出编辑模式后移动光标
        h键向左移动
        j键向下移动
        k键向上移动
        l键向右移动
3.查看文件内容 cat filename
        显示行号 cat -b filenamem
4.统计单词数目 wc filename(可一次统计多个文件)
        第一列 文件的总行数
        第二列 单词数目
        第三列 文件大小
        第四列 文件名
5.复制文件 cp filename copyfile
6.重命名文件 mv filename newfile
7.删除文件 rm filename(彻底删除，可一次删除多个)

######Linux文件和目录管理######

1.切换目录 cd [相对路径或绝对路径]
        根目录 /
        当前目录 .
        上一层目录 ..
        上次所在目录 -
        当前登录用户的主目录 ~

2.显示当前路径 pwd

3.查看目录下的文件 ls [选项参数] [目录名称]
        ls -a 显示全部文件，包括隐藏文件
        ls -l 显示文件的详细信息 (=ll)
        ls -lh 文件大小显而易见 KB MB GB

4.创建目录（文件夹）mkdir [-mp] 目录名
        mkdir -m 手动配置目录权限
        mkdir -p 递归创建所有目录，一层一层自动创建
        rmdir 删除空目录，非空会报错

5.创建文件 touch [选项] 文件名

修改文件的时间戳

stat 文件名 查看文件的３个时间参数

6.在文件之间建立链接 ln [选项] 源文件 目标文件

ln -s 建立软链接文件。如果不加则建立硬链接文件。

ln -f 强制。如果目标文件已经存在，则删除目标文件后再建立链接

软硬链接的区别特点:

软:常用，源文件必须写绝对路径(windows中的快捷方式)

软链接和硬链接在原理上最主要的不同在于:

硬链接不会建立自己的 inode 索引和 block（数据块），而是直接指向源文件的 inode 信息和 block，所以硬链接和源文件的 inode 号是一致的;

而软链接会真正建立自己的 inode 索引和 block，所以软链接和源文件的 inode 号是不一致的，而且在软链接的 block 中，写的不是真正的数据，而仅仅是源文件的文件名及 inode 号。

7.复制文件和目录 cp [选项] 源文件 目标文件

-a 相当于-d -p -r的集合

-d 源文件为软链接(对硬链接无效)，则复制出的也是软链接

-i 如果目标文件存在，则会询问是否覆盖

-l 建立硬链接，不复制文件

-s 建立软链接，不复制文件

-p 保留源文件的所有属性(权限、时间)

-r 递归复制，复制目录

-u 源文件和目标文件有差异时，更新目标文件

8.删除文件或目录 rm [选项] 文件或目录

-f 强制删除，和-i相反直接删除

-i 删除前询问

-r 递归删除，删除目录及包含的所有内容

rm -rf 最常用，删除文件和目录都可以

9.移动文件或改名 mv [选项] 源文件 目标文件

-f 若已存在，强制覆盖

-i 交互移动，若已存在询问是否覆盖

-n 若已存在，则不动

-v 显示移动过程

-u 对目标文件升级

mv filename newname 重命名

10.shell通配符

* 匹配任意数量的字符

? 匹配任意一个字符

[]匹配[]内的任意一个字符

[a-d] 匹配abcd

[1-5] 匹配12345

######Linux打包和压缩######

1.打包命令 tar [选项] 源文件或目录
        -c 将多个文件或目录打包
        -A 追加tar文件到归档文件
        -f 指定包的文件名
        -v 显示打包文件的过程

        习惯用法:tar -cvf 文件名.tar 文件名
        gzip test.tar 把test.tar压缩成test.tar.gz

2.解打包操作 tar [选项] 压缩包
        -x 对tar包解打包
        -f 指定包名
        -t 只查看不解压
        -C 指定位置
        -v 查看过程

3.打包压缩 tar [选项] 压缩包 源文件或目录
        -z 压缩解压缩.tar.gz格式
        -j 压缩解压缩.tar.bz2格式

4.压缩文件或目录 zip [选项] 压缩包名 源文件或目录列表
        -r 递归压缩目录
        -m 文件压缩后删除源文件
        -v 显示过程
        -q 不显示命令执行过程
        -u 往压缩文件中添加新文件
        -压缩级别 -1 速度快 -9效果好

5.解压zip文件 unzip [选项] 压缩包名
        -d目录名 解压到指定目录
        -n 解压时不覆盖
        -o 解压时覆盖
        -v 查看压缩文件的详细信息，不解压
        -t 测试好坏， 不解压
        -x文件列表 解压但不包括列表指定的文件

6.压缩gzip [选项] 源文件
        -c 保留源文件压缩
        -d 解压缩
        -r 递归压缩指定目录下的子文件
        -v 显示文件名和压缩比

7.解压缩gunzip [选项] 文件.gz
        -r 递归解压缩指定目录下的子文件

-c 解压后输出
-f 强制解压
-l 显示内容
-v 显示命令执行过程
-t 测试好坏

8.压缩bzip2 [选项] 源文件.bz2 （比gzip好）
-d 解压缩
-k 保留源文件
-f 强制覆盖
-t 测试是否完整
-v 显示详细信息

9.解压缩bunzip2 [选项] 源文件
-k 保留源文件
-f 强制
-v 显示命令执行过程
-L 显示压缩文件内容

#####VIM文本编辑器######

1.打开文件
vim filename 打开或新建文件，光标在第一行
vim -r filename 恢复上次打开崩溃的文件
vim -R filename 把文件只读放入vim
vim + filename 打开文件，光标位于最后一行
vim +n filename 打开文件，光标在第n行
vim +/pattern filename 打开，光标位于与pattern匹配的位置
vim -c command filename 编辑前执行command

2.查找文本
/abc 在光标位置向前查找abc
/^abc 查找以abc为首的行
/abc$ 查找以abc为尾的行
?abc 在光标位置向后查找abc
n 向同一方向重复上次的查找指令
N 向相反方向重复上次的查找指令

:set ic 忽略大小写
:set noic 区分大小写(默认)

3.替换文本
r 替换光标所在位置的字符
R 输入内容会覆盖掉后面的文本，Esc结束
:s/a1/a2/g 所在行中所欲a1用a2替换

:n1,n2s/a1/a2/g n1到n2行中的所有a1用a2替换
:g/a1/a2/g 文件中所有a1都用a2替换

4.删除文本
x 删除光标所在位置的字符
dd 删除光标所在行
ndd 删除当前行及后n行的文本
dG 删除光标所在行直到文件末尾
D 删除光标所在位置到行尾
:a1,a2d 删除a1行到a2行的文本内容

被删除的内容存在剪贴板中，p键粘贴

5.复制和粘贴文本
p 粘贴到光标后
P() 粘贴到光标前
y 复制
yy 讲所在行复制，nyy复制n行
yw 复制单词

两行拼成一行 J

6.保存退出文本
:wq 保存并退出编辑器
:wq! 保存并强制退出
:q 不保存就退出编辑器
:q! 不保存强制退出编辑器
:w 保存但是不退出编辑器
:w! 强制保存文本
:w filename 另存到filename文件
x! 保存文本并退出(常用)
ZZ 直接退出编辑器

7.光标移动快捷键
0或^ 移动到当前行的行首
$ 移动到行尾
gg 移动到文件开头
G 移动到文件末尾
nG 移动到第n行
:n 编辑模式定位到指定行
% 括号配对

8.批量注释
:起始行，终止行s/^/#/g
例 :1,10s/^/#/g 第一行到第十行行首加'#'注释，'^'为行首，'g'是否询问，改成'c'则不询问

:起始行，终止行s/^#//g
意为将行首的'#'替换为空，即删除取消注释

9.自定义快捷操作

:map 快捷键 执行命令

:unmap 快捷键 取消快捷键定义

例 :map ^p l#<Esc>

^p 表示Ctrl+P

l 在光标所在行的行首插入

要输入的字符

<Esc> 表示退回命令模式

######文本处理三剑客######

1.内容查看

cat [选项] 文件名

cat file1 file2 > file3 链接合并文件

2.分屏显示文件内容 more [选项] 文件名

more 只能向后翻看

less 前后翻看 less [-N]显示行号

less交互指令:

/abc 向下搜索abc

?abc 向上搜索abc

head 显示文件的前几行 head [-n] 前n行

tail 显示文件的后几行 tail [-f] 监听新增内容

3.输入重定向

命令<文件 将指定文件作为命令的输入设备

命令<<分界符 连续输入直到遇到分界符

命令<文件1>文件2 将1的执行结果输出到2

4.输出重定向

命令>文件 将执行的标准输出结果重定向输出到文件中，覆盖原文件中的数据

命令2>文件 将错误输出写到文件中，覆盖原文件中的数据

命令>>文件 标准输出写入文件，追加写入

命令2>>文件 错误输出写入文件，追加写入

命令>>文件 2>&1

命令&>>文件 标准或错误输出，追加写入文件

5.grep支持的正则表达式的通配符

c* 匹配零个或多个字符c

. 匹配任意一个字符，只能是一个

[xyz] 匹配xyz中的任意一个字符

[^xyz] 匹配除xyz以外的所有字符

^ 锁定行的开头
$ 锁定行的结尾

如匹配特殊字符需加反斜杠\ * \+ \{

6.查找文件内容 grep [选项] 模式 文件名
-c 仅列出匹配的行数(个数?)
-i 忽略模式中的字母大小写
-l 列出带有匹配行的文件名
-n 列出行号
-v 列出没有匹配的行
-w 完全匹配字符，忽略部分匹配

7.sed [选项] [脚本命令] 文件名
难难难
略略略
8.awk

9.

######软件安装######

apt install	apt-get install	安装软件包
apt remove	apt-get remove	移除软件包
apt purge	apt-get purge	移除软件包及配置文件
apt update	apt-get update	刷新存储库索引
apt upgrade	apt-get upgrade	升级所有可升级的软件包
apt autoremove	apt-get autoremove	自动删除不需要的包
apt full-upgrade	apt-get dist-upgrade	在升级软件包时自动处理依赖关系
apt search	apt-cache search	搜索应用程序
apt show	apt-cache show	显示安装细节

rpm dpkg yum apt(ubuntu)
1.搜索软件 sudo apt-cache search package_name(正则表达式)

2.查看软件包信息 sudo apt-cache show package_name

3.查看软件包依赖关系 sudo apt-cache depends package_name

4.查看每个软件包的简要信息 sudo apt-cache dump

5.安装软件 sudo apt-get install package_name

6.更新已安装的软件包 sudo apt-get upgrade

7.更新软件包列表 sudo apt-get update

8.卸载一个软件包但是保留相关配置文件 sudo apt-get remove package_name

9.卸载一个软件包同事删除配置文件 apt-get -purge remove package_name

10.删除软件包的备份 apt-get clean

######用户和用户组的管理######

1./etc/passwd内容解释
用户名:密码:UID(用户ID):GID(组ID):描述下信息:主目录:默认Shell

2./etc/shadow(影子文件)内容解析
用户名:加密密码:最后一次修改时间:最小修改时间间隔:密码有效期:密码需要变更前的警告天数:密码过期后的宽限时间:账号失效时间:保留字段

3./etc/group文件解析
组名:密码:GID:该用户组中的用户列表

4./etc/gshadow文件内容解析
组名:加密密码:组管理员:组附加用户列表

5.useradd命令详解 useradd [选项] 用户名
        -u UID 指定用户的UID(大于500)
        -d 主目录 指定用户的主目录(绝对路径)
        -c 用户说明 指定/etc/passwd文件中的用户信息(第5个字段)
        -g 组名 指定用户的初始组 默认建立同用户名相同的初始组
        -G 组名 指定用户的附加组
        -s shell 指定用户登录的Shell，默认是/bin/bash
        -e 日期 指定用户的失效日期，格式为"YYYY-MM-DD"
        -o 允许创建的用户的UID相同
        -m 建立用户时强制建立用户的家目录
        -r 创建系统用户 UID在1~499之间

6.修改/etc/default/useradd文件 useradd -D[选项] 参数
        -b HOME 设置创建的主目录所在的默认目录
        -e EXPIRE 设置密码失效时间YYYY-MM-DD
        -f INACTIVE 设置密码过期的宽限天数
        -g GROUP 设置新用户的初始组
        -s SHELL 设置新用户的默认shell(完整路径)

7.修改用户密码 passwd [选项] 用户名
        -S 查询用户密码的状态(root权限)
        -l 暂时锁定用户，/etc/shadow密码前加"!"
        -u 解锁用户
        --stdin 管道符输出的数据作为用户密码(批量添加用户时使用)

-n 天数 多久时间不能再次修改密码4
-x 天数 设置用户的密码有效期5
-w 天数 设置密码过期前的警告天数6
-i 日期 设置用户密码失效日期7

8.修改用户信息 usermod [选项] 用户名
-c 说明 修改用户的说明信息
-d 主目录 修改用户的住目录
-e 日期 修改用户的失效日期
-g 组名 修改用户的初始组
-u UID 修改用户的UID
-G 组名 修改用户的附加组
-l 用户名 修改用户名
-L 临时锁定用户
-U 解锁用户
-s shell 修改用户的登录Shell

9.修改用户密码状态: chage [选项] 用户名
-l 列出用户的密码详细状态
-d 日期 最后一次修改密码的日期
-m 天数 修改密码最短保留天数
-M 天数 修改密码的有效期
-W 天数 修改密码到期前的警告天数
-i 天数 修改密码过期后的宽限天数
-E 日期 修改账号失效日期

10.删除用户 userdel -r 用户名

11.查看用户的UID和GID: id 用户名

12.用户间切换: su [选项] 用户名
- 不仅切换用户的身份，同时切换工作工作环境(PATH、MAIL等)，省略用户名默认切换为root
-l 同-，后面要添加使用者的账号
-p 仅切换用户身份，不切换工作环境
-m 和-p一样
-c 命令 仅切换用户执行一次命令，切换后自动切换回来

13.添加用户组 groupadd [选项] 组名
-g GID 指定组id
-r 创建系统群组

14.修改用户组 groupmod [选项] 组名
-g GID 修改组id
-n 新组名 修改组名

15.删除用户组 groupdel 组名

16.把用户添加进组或从组中删除 gpasswd [选项] 组名

选项为空时，表示给群组设置密码
-A user1，... 设置群组管理员，可设置多个
-M user1，... 将user1...加入到此群组
-r 删除群组的密码
-R 让群组密码失效
-a user 将user用户加入到群组中
-d user 将user用户从群组中移除

17.切换用户的有效组 newgrp 组名

######权限管理######

1.修改文件和目录的所属组 chgrp [-R] 所属组 文件名(目录名)
-R 更改目录的所属组，两桶子目录中的所有文件

2.修改文件和目录的所有者和所有组 chown [-R] 所有者 文件或目录
-R 两桶子目录中的所有文件，都更改所有者
chown [-R] 所有者:所有组 文件或目录

3.修改文件或目录的权限chmod(数字)
r-->4 读
w-->2 写
x-->1 执行
例: rwxrw-r-x 权限值为765
所有者=rwx=4+2+1=7
所属组=rw-=4+2=6
其他人=r-x=4+1=5
chmod [-R] 权限值 文件名

4.修改文件或目录的权限chmod(符号)
u 所有者user
g 所属组group
o 其他人other
a 所有人all
chmod u=rwx,go=rx filename
chmod a+w filename 所有用户都可做写操作的权限

5.文件特殊权限(SUID SGID SBIT)

6.修改文件系统的权限属性 chattr [+-=] [属性] 文件或目录名
+ 表示给文件或目录添加属性
- 表示移除文件或目录拥有的某些属性
= 表示给文件或目录设定一些属性
i 文件:不允许删除、改名，也不能添加和修改数据
目录:允许修改目录下文件中的数据，不允许建立和删除文件

a 文件:只能在文件中增加数据，不能删除和修改数据
目录:只允许在目录中建立和修改文件，不允许删除文件
u 删除时内容会被保存，防止意外删除文件或目录
s 和u相反，彻底删除，不可恢复

7.查看文件系统属性 laattr [选项] 文件名或目录
-a 后面啥也不带，显示所有文件和目录(包括隐藏文件)
-d 只显示目录本身，不列出所含文件或子目录
-R 和-d相反，子目录的隐藏信息数据一并显示

8.系统权限管理 sudo [-b] [-u新使用者账号] 要执行的命令
-b 将命令放到背景中让系统运行
-u 想要切换的用户名，默认为root
-l sudo -l 显示当前用户可以用sudo执行哪些命令

######文件系统管理######

1.查看文件系统硬盘使用情况 df [选项] [目录或文件名]
-a 显示所有文件系统信息
-m 以MB为单位显示容量
-k 以KB为单位显示容量，默认以KB为单位
-h 使用习惯的KB,MB,GB等单位自行显示容量
-T 显示该分区的文件系统名称
-i 不用硬盘容量显示，而是以含有inode的数量显示

2.统计目录或文件所占磁盘空间大小 du [选项] [目录或文件名]
-a 显示每个子文件的磁盘占用量，默认只统计子目录的磁盘占用量
-h 使用习惯单位显示磁盘占用量
-s 统计总磁盘占用量，不显示子目录和子文件的占用量

3.挂在Linux系统外的文件
mount [-l] 单纯使用mount命令，会显示出系统中已挂载的设备信息，使用-l选项会额外显示出卷标名称
mount -a 检查/etc/fstab文件中有无疏漏被挂载的设备文件，如果有则进行自动挂载操作

mount [-t系统类型] [-L卷标名] [-o特殊选项] [-n] 设备文件名挂载点
-t 系统类型，不指定则自动检测
-L 卷标名，除了使用设备文件名之外，还可以利用文件系统的卷标名挂载
-n 单人维护模式，刻意不写入
-o 特殊选项 读写权限，同步异步，文件是否可执行

4.卸载文件系统 umount 设备文件名或挂载点

5.检测和修复文件系统 fsck [选项] 分区设备文件名
注意:修改前文件系统对应的磁盘分区一定要处于卸载状态

-a 自动修复文件系统
-r 交互修复，进行询问
-A 安装/etc/fstab配置文件的内容，检测文件内罗列的全部文件系统
-t 文件系统类型 指定要检查的文件系统类型
-C 显示检查分区的进度条
-f 强制检测
-y 自动修复，和-a作用一样

6.查看文件系统信息 dumoe2fs [-h] 文件名
-h 仅列出superblock的数据信息

7.给硬盘分区
fdisk ~l 列出系统分区
fdisk 设备文件名 给硬盘分区

8.格式化分区(为分区写入文件系统) mkfs [选项] 分区设备文件名
-t 文件系统格式:用于指定格式化的文件系统，如ext3,ext4

9.格式化硬盘(给硬盘写入文件系统) mk2fs [选项] 分区设备文件名
-t 文件系统:指定格式化成哪个文件系统，如ext3,ext4
-b 字节 指定block的大小
-i 字节 多少字节分配一个inode
-j 建立带有ext3日志功能的文件系统
-L 卷标名:给文件系统设置卷标名

######系统管理######

1.查看正在运行的进程
ps aux 查看系统中的所有进程
ps -le 查看系统中的所有进程，而且还能看到进程的父进程的PID和进程优先级
ps -l 只能看到当前Shell产生的进程

2.持续监听进程运行状态 top [选项]
-d 秒数:指定top命令每隔几秒更新，默认3s
-b 模式输出，和-n连用，用于把top命令重定向到文件中
-n 次数:指定top命令执行的次数
-p 进程PID:仅查看指定ID的进程
-s 使top命令在安全模式中运行，避免在交互模式中出现错误
-u 用户名:只监听某个用户的进程

3.查看进程树 pstree [选项] [PID或用户名]
-a 显示启动每个进程对应的完整指令
-c 显示的进程中包含子进程和父进程
-n 按进程PID号来排序输出，默认是以程序名排序输出
-p 显示进程PID

-u 显示进程对应的用户名称

4.列出进程调用或打开的文件信息 lsof [选项]
-c 字符串 只列出以字符串开头的进程打开的文件
+d 目录名 列出某个目录中所有被进程调用的文件
-u 用户名 只列出某个用户的进程打开的文件
-p PID 列出某个PID进程打开的文件

5.进程优先级 PRI代表Priority NI代表Nice(数值越小优先级越高)
PRI(最终值) = PRI(原始值) + NI
NI范围是-20~19
普通用户调整NI值的范围是0~19,而且只能调整自己的进程
普通用户只能调整NI，而不能降低
只有root用户才能设定进程NI值为负值，而且可以调整任何用户的进程

6.改变进程优先级 nice和renice
nice [-n NI值] 命令 给要启动的进程赋NI值(-20~19)，但是不能修改已运行进程的NI值
renice [优先级] PID 与nice命令相反，可以在进程运行时修改NI值，从而调整优先级(常与ps命令配合使用)

7.终止进程 kill [信号] PID
0 EXIT 程序退出时收到该信息
1 HUP 挂掉电话线或终端链接的挂起信号
2 INT 表示结束进程，但不是强制的，"Ctrl+C"
3 QUIT 退出
9 KILL 强制结束进程
11 SEGV 段错误
15 TERM 正常结束进程，是kill命令的默认信号

8.终止特定的一类进程 killall [选项] [信号] 进程名
-i 交互式询问是否要杀死某个进程
-l 忽略进程名的大小写

9.命令放入后台运行方法
命令 &
Ctrl+Z

10.查看当前终端放入后台的工作 jobs [选项]
-l 列出进程的PID号
-n 只列出上次发出通知后改变了状态的进程
-p 只列出进程的PID号
-r 只列出运行中的进程
-s 只列出已停止的进程

11.把后台命令恢复在前台执行 fg [%] 工作号
把后台啊暂停的工作恢复到后台执行 bg [%] 工作号

12.循环执行定时任务 crontab [选项] [file]

13.查看内存使用状态 free[选项] (与top相似)

14.查看登录用户信息 w [选项] [用户名]
        -h 不显示输出信息的标题
        -l 用长格式输出
        -s 短格式输出，不显示登录时间
        -V 显示版本信息

15.查看过去登录的用户信息 last [选项]
        -a 把从何处登录系统的主机名或IP地址显示在最后一行
        -R 不显示登录系统的主机名或IP地址
        -x 显示系统关机、重新开机以及执行等级的改变信息
        -n 显示列数
        -d 将显示的IP地址转换成主机名称

######Linux数据备份与恢复######

######Linux系统服务管理######

1.端口及查询方法 netstat [选项]
        -a 列出系统中所有网络链接
        -t 列出TCP数据
        -u 列出UDP数据
        -l 列出正在监听的网络服务
        -n 用端口号来显示而不用服务名
        -p 列出该服务的进程IP(PID)

vim多窗口使用技巧

1、打开多个窗口打开
    多个窗口的命令以下几个:
    横向切割窗口:new+窗口名(保存后就是文件名)
    :split+窗口名，也可以简写为:sp+窗口名纵向切割窗口名
    :vsplit+窗口名，也可以简写为: vsp+窗口名
2、关闭多窗口
    可以用: q!，也可以使用: close，最后一个窗口不能使用close关闭。
    使用close只是暂时关闭窗口，其内容还在缓存中，只有使用q!、w!或x才能真能退出。
    :tabc 关闭当前窗口
    :tabo 关闭所有窗口
3、窗口切换

:ctrl+w+j/k，通过j/k可以上下切换，或者:ctrl+w加上下左右键，还可以通过快速双击ctrl+w依次切换窗口。

4、窗口大小调整

纵向调整

:ctrl+w + 纵向扩大（行数增加）

:ctrl+w - 纵向缩小 （行数减少）

:res(ize) num 例如：:res 5，显示行数调整为5行

:res(ize)+num 把当前窗口高度增加num行

:res(ize)-num 把当前窗口高度减少num行

横向调整

:vertical res(ize) num 指定当前窗口为num列

:vertical res(ize)+num 把当前窗口增加num列

:vertical res(ize)-num 把当前窗口减少num列

5、给窗口重命名

:f file

6、vi打开多文件vi a b c

:n 跳至下一个文件，也可以直接指定要跳的文件，如:n c，可以直接跳到c文件:e# 回到刚才编辑的文件

7、文件浏览

:Ex 开启目录浏览器，可以浏览当前目录下的所有文件，并可以选择

:Sex 水平分割当前窗口，并在一个窗口中开启目录浏览器

:ls 显示当前buffer情况

8、vi与shell切换

:shell 可以在不关闭vi的情况下切换到shell命令行

:exit 从shell回到vi

github

上传代码到 github

1. 在github上创建一个空仓库，复制创建好后的仓库网址
2. 终端输入：git clone https://github.com/your_name/your_repository_name.git
3. 会生成一个以仓库名命名的文件夹，复制你所要的文件或文件夹到仓库文件夹里面。接着终端操作，进入仓库文件夹。
4. cd your_repository_name 进入到仓库文件夹
5. git add file_name 要上传的文件名
   git commit -m “注释” 添加注释
6. git config --global user.email “zhanghy7447@126.com”
   git config --global user.name “guiguaidashu”
7. git push 开始上传

线程进程

线程池

```
struct threadpool_t {

    pthread_mutex_t lock;               /* 用于锁住本结构体 */
    pthread_mutex_t thread_counter;     /* 记录忙状态线程个数de琐 -- busy_thr_num */

    pthread_cond_t queue_not_full;      /* 当任务队列满时，添加任务的线程阻塞，等待此条件变量 */
    pthread_cond_t queue_not_empty;     /* 任务队列里不为空时，通知等待任务的线程 */

    pthread_t *threads;                 /* 存放线程池中每个线程的tid。数组 */
    pthread_t adjust_tid;               /* 存管理线程tid */
    threadpool_task_t *task_queue;      /* 任务队列(数组首地址) */

    int min_thr_num;                    /* 线程池最小线程数 */
    int max_thr_num;                    /* 线程池最大线程数 */
    int live_thr_num;                   /* 当前存活线程个数 */
    int busy_thr_num;                   /* 忙状态线程个数 */
    int wait_exit_thr_num;              /* 要销毁的线程个数 */

    int queue_front;                    /* task_queue队头下标 */
    int queue_rear;                     /* task_queue队尾下标 */
    int queue_size;                     /* task_queue队中实际任务数 */
    int queue_max_size;                 /* task_queue队列可容纳任务数上限 */

    int shutdown;                       /* 标志位，线程池使用状态，true或false */
};


typedef struct {

    void *(*function)(void *);          /* 函数指针，回调函数 */
    void *arg;                          /* 上面函数的参数 */

} threadpool_task_t;                    /* 各子线程任务结构体 */
```

线程池模块分析：

1.main();

创建线程池。

向线程池中添加任务，借助回调处理任务。

销毁线程池。

2.pthreadpool_create();

创建线程池结构体 指针。

初始化线程池结构体{ N个成员变量 }

创建 N 个任务线程。

创建 1 个管理者线程。

失败时，销毁开辟的所有空间。（释放）

3.threadpool_thread();

进入子线程回调函数。

接收参数 void *arg -- pool 结构体

加锁 -- lock -- 整个结构体加锁

判断条件变量 -- wait -----------170

4.adjust_thread();

循环 10s 执行一次。

进入管理者线程回调函数。
接收参数 void *arg -- pool 结构体
加锁 -- lock -- 整个结构体加锁
获取管理线程池要用到的 变量。 task_num, live_num, busy_num
根据既定算法，使用上述3个变量，判断是否应该创建、销毁线程池中指定步长的线程。

5.threadpool_add();
总功能：
模拟产生任务。 num[20]
设置回调函数，处理任务。 -- sleep(1) 代表处理完成
内部实现：
加锁
初始化任务队列结构体成员。
利用环形队列机制，实现添加任务。借助队尾指针挪移 % 实现。
唤醒阻塞在条件变量上的进程
解锁

6.从 3. 中的wait之后继续执行，处理任务
加锁
获取任务处理回调函数 及参数。
利用环形队列机制，实现添加任务。借助队尾指针挪移 % 实现。
唤醒阻塞在条件变量上的 server。
解锁
加锁
改忙线程数++
解锁
执行处理任务的线程
加锁
改忙线程数--
解锁

7.创建、销毁线程
管理者线程根据 task_num, live_num, busy_num
根据既定算法，使用上述3个变量，判断是否应该创建、销毁线程池中指定步长的线程。
如果满足 创建条件：
pthread_create(); 回调任务线程函数。 live_num++
如果满足 销毁条件：
wait_exit_thr_num = 10;
signal 给阻塞在条件变量上的线程 发送 假条件满足信号。
跳转至 -- 170行 wait阻塞线程会被 假信号 唤醒。判断：wait_exit_thr_num > 0
pthread_exit();

TCP通信和UDP通信各自的优缺点：

TCP：面向连接的，可靠数据包传输。对于不稳定的网络层，采取完全弥补的通信方式。丢包重传
优点：数据流量稳定、速度稳定、顺序稳定。
缺点：传输速度慢、效率低、开销大。

使用场景：数据的完整性要去较高，不追求效率。
大数据传输、文件传输

UCP：无连接的，不可靠的数据报传递。对于不稳定的网络层，采取完全不弥补的通信方式。默认还原网络状况。
优点：传输速度快、效率高、开销小。
缺点：数据流量不稳定、速度不稳定、顺序不稳定。
使用场景：对时效性要求较高的场合。稳定性其次
游戏、视频会议、视频通话
腾讯、华为、阿里 -- 应用层数据校验协议，弥补UDP的不足。

UDP实现 C/S 模型：

```
recv() / send() 只能用于 TCP 通信，替代read、write
accept(); --- Connect(); --- 被舍弃

server:
    lfd = socket(AF_INET, SOCK_DGRAM, 0);   SOCK_DGRAM -- 报式协议
    bind();
    listen(); -- 可有可无
    while(1){
        read(cfd, buf, sizeof) -- 被替换 -- recvfrom()
        ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                         struct sockaddr *src_addr, socklen_t *addrlen);
            sockfd:     套接字
            buf:        缓冲区地址
            len:        缓冲区大小
            flags:      0
            src_addr:   (strcut sockaddr *)&addr  传出  对端的地址结构
            addrlen:    传入传出
        返回值：成功接收数据的字节数， 失败 -1 errno。  = 0 对端关闭

        小 -- 大

        write() -- 被替换 -- sendto()
        ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
                       const struct sockaddr *dest_addr, socklen_t addrlen);
            sockfd:     套接字
            buf:        存储数据的缓冲区
            len:        数据长度
            flags:      0
            src_addr:   (strcut sockaddr *)&addr  传入  目标地址结构
            addrlen:    地址结构长度
        返回值：成功写出数据的字节数，失败 -1 errno。
    }
    close();

client:
    connfd = socket(AF_INET, SOCK_DGRAM, 0);
    sendto("服务器地址结构", "地址结构大小")；
    recvfrom()
```

写到屏幕
close();

本地套接字：

IPC： pipe、fifo、信号、本地套(domain) -- C/S模型

对比网络编程 TCP C/S模型，注意一下几点：

1. int socket(int domain, int type, int protocol);

参数domain： AF_INET -- AF_UNIX / AF_LOCAL

type: SOCK_STREAM / SOCK_DGRAM 都可以

2. 地址结构 sockaddr_in -- sockaddr_un

```
struct sockaddr_in srv_addr;
srv_addr.sin_family = AF_INET / AF_INET6    man 7 ip
srv_addr.sin_port = htons(9527);
srv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
bind(fd, (struct sockaddr *)&srv_addr, sizeof(srv_addr));

struct sockaddr_un srv_addr;
srv_addr.sun_family = AF_UNIX;
strcpy(srv_addr.sun_path, “srv.socket”);
len = offsetof(struct sockaddr_un, sun_path) + strlen("srv.socket");
bind(fd, (struct sockaddr *)&srv_addr, len);
```

3. bind()函数调用成功，会创建一个socket。

因此为保证 bind 成功，通常我们在 bind 之前，可以使用 unlink("srv.socket");

网络套接字和本地套接字对比：

网络套接字	本地套接字
lfd = socket(AF_INET, SOCK_STREAM, 0);	lfd = socket(AF_UNIX, SOCK_STREAM, 0);
bzero() ---- struct sockaddr_in serv_addr;	bzero() ---- struct sockaddr_un serv_addr, clie_addr;

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(8888);
	serv_addr.sun_family = AF_UNIX;
	strcpy (serv_addr.sun_path, "套接字文件名")
	len = offsetof(sockaddr_un, sun_path) + strlen();

bind(lfd, (struct sockaddr *)&serv_addr, sizeof());	unlink("套接字文件名"); bind(lfd, (struct sockaddr *)&serv_addr, len); 创建新文件
Listen(lfd, 128);	Listen(lfd, 128);
cfd = Accept(lfd, ()&clie_addr, &len);	cfd = Accept(lfd, ()&clie_addr, &len);

```
lfd = socket(AF_INET, SOCK_STREAM, 0);
" 隐式绑定 IP+port"



bzero() ---- struct sockaddr_in serv_addr;
serv_addr.sin_family = AF_INET;
inet_pton(AF_INT, "服务器IP", &sin_addr.s_addr)
serv_addr.sin_port = htons("服务器端口");


connect(lfd, &serv_addr, sizeof());
```

```
lfd = socket(AF_UNIX, SOCK_STREAM, 0);

bzero() ---- struct sockaddr_un clie_addr;
clie_addr.sun_family = AF_UNIX;
strcpy (clie_addr.sun_path, "client套接字文件名")
len = offsetof(sockaddr_un, sun_path) + strlen();
unlink( "client套接字文件名");
bind(lfd, (struct sockaddr *)&clie_addr, len);

bzero() ---- struct sockaddr_un serv_addr;
serv_addr.sun_family = AF_UNIX;

strcpy (serv_addr.sun_path, "server套接字文件名")

len = offsetof(sockaddr_un, sun_path) + strlen();
connect(lfd, &serv_addr, len);
```