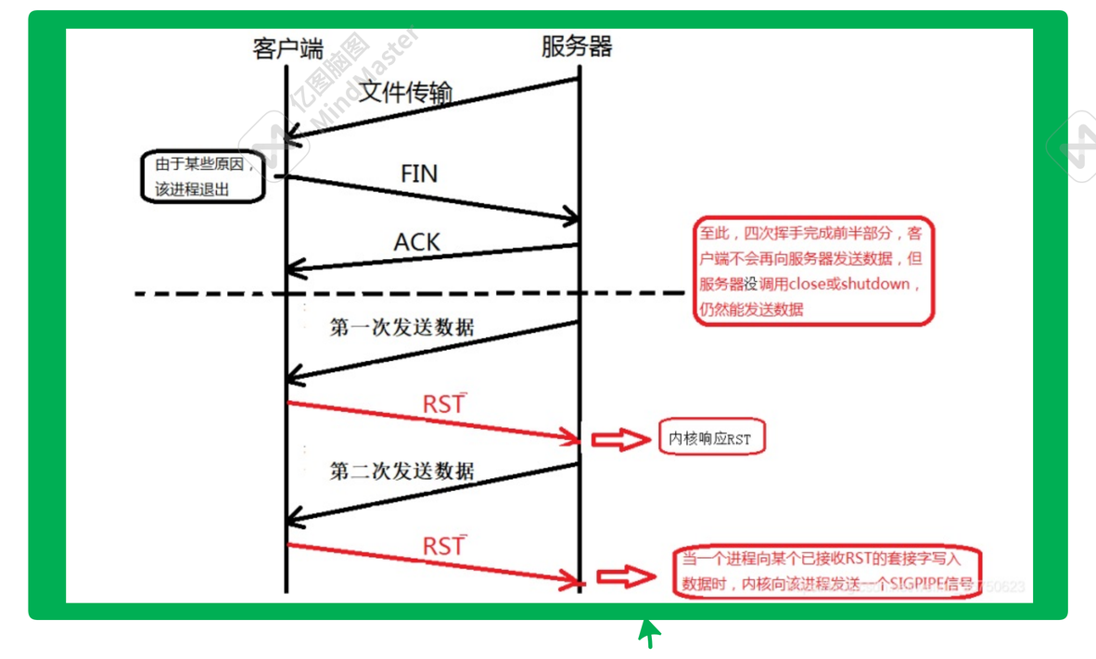
Linux服务器

主线程 main.cpp

获取端口号

添加信号捕捉
addsig(SIGPIPE,SIG_IGN)
对信号SIGPIPE进行处理

```
//添加信号捕捉
void addsig(int sig,void(handler)(int)){
    //注册信号
    struct sigaction sa;
    memset(&sa,'\0',sizeof(sa));
    sa.sa_handler= handler;
    sigfillset(&sa.sa_mask);// 设置临时信号阻塞集
    sigaction(sig,&sa,NULL);
}
```



创建线程池，里面存放http_conn连接类对象
threadpool<http_conn> *pool = NULL;

//创建一个数组用于保存所有的客户端信息。
http_conn *users = new http_conn[MAX_FD];

int listenfd= socket(PF_INET,SOCK_STREAM,0);
利用socket创建监听套接字listenfd

//绑定服务器地址之前设置端口复用
int reuse = 1;
setsockopt(listenfd,SOL_SOCKET,SO_REUSEADDR,&reuse,sizeof(reuse));

//绑定服务器地址结构 IP+port
struct sockaddr_in address;
address.sin_family=AF_INET;
address.sin_addr.s_addr=INADDR_ANY;
address.sin_port = htons(port);
bind(listenfd,(struct sockaddr *)&address,sizeof(address));

//设置监听上限
listen(listenfd,5);

//创建epoll对象,事件数组
epoll_event events[MAX_EVENT_NUM];
int epollfd = epoll_create(5);

//将监听的文件描述符添加到epoll对象中
addfd(epollfd,listenfd,false);

epoll_event结构数组

```
//向epoll中添加需要监听的文件描述符
void addfd(int epollfd,int fd,bool one_shot){
    epoll_event event;
    event.data.fd = fd;
    //event.events = EPOLLIN|EPOLLRDHUP; // REVEC ,READ,直接通过信号
    event.events = EPOLLIN|EPOLLET |EPOLLRDHUP; // 边缘触发
    if(one_shot){
        event.events |= EPOLLONESHOT;
    }
    epoll_ctl(epollfd,EPOLL_CTL_ADD,fd,&event);
    //设置文件描述符非阻塞•;
    setnoblocking(fd);
}
```

```
int setnoblocking(int fd){
    int old_flag = fcntl(fd,F_GETFL);
    int new_flag = old_flag |O_NONBLOCK;
    fcntl(fd,F_SETFL,new_flag); // 设置 cfd 非阻塞  ET+非阻塞模式
    return old_flag;
}
```

//初始化http连接的文件描述符
http_conn::m_epollfd = epollfd;

while(true)

- //epoll_wait返回就绪的文件描述符个数
  int num = epoll_wait(epollfd,events,MAX_EVENT_NUM,-1);
  - 遍历就绪的描述符集合
    for(int i = 0 ;i < num;i++)
    - sockfd = events[i].data.fd;
    - //如果是连接事件
      if(sockfd == listenfd)
      - //有客户端连接进来 accept连接事件:
        struct sockaddr_in client_address;
        socklen_t client_addrlen = sizeof(client_address);
        int connfd = accept(listenfd,(struct sockaddr *)&client_address,&client_addrlen);
      - //初始化connfd
        users[connfd].init(connfd,client_address);

EPOLLONESHOT
(1)如果一个线程读完某个socket上的数据，此时该socket上又有新的数据可读，此时另外一个线程被唤醒读取数据，此时两个线程同时操作一个socket,我们希望socket在任意时刻都只有一个线程处理，所以可以设置epolloneshot.

```
//初始化连接
void http_conn::init(int sockfd,const sockaddr_in& addr){
    m_sockfd = sockfd;
    m_address = addr;
    //设置端口复用
    int reuse = 1;
    setsockopt(m_sockfd,SOL_SOCKET,SO_REUSEADDR,&reuse,sizeof(reuse));

    //添加到epoll对象
    addfd(m_epollfd,sockfd,true); //true ， 为客户端设置EPOLLONESHOT
    m_user_count++; //连接的数量加1

    init();
}
```

```
//关闭连接
void http_conn::close_conn(){
    if(m_sockfd != -1){
        removefd(m_epollfd,m_sockfd);
        m_sockfd = -1;
        m_user_count--; //关闭连接，客户减
—
    }
}
```

```
//对方异常断开或者错误发生
else if(events[i].events& (EPOLLRDHUP|EPOLLHUP|EPOLLERR)){
        users[sockfd].close_conn();
```

```
//如果是读事件 一次性读完
else if(events[i].events&EPOLLIN){
        //判断该文件描述符有无数据可读
        if(users[sockfd].read()){
            //如果有数据可读，加入请求队列中，同时唤醒在等待的线程
            pool->append(users+sockfd);
        }else{
         //否则关闭连接
            users[sockfd].close_conn();
        }
}
```

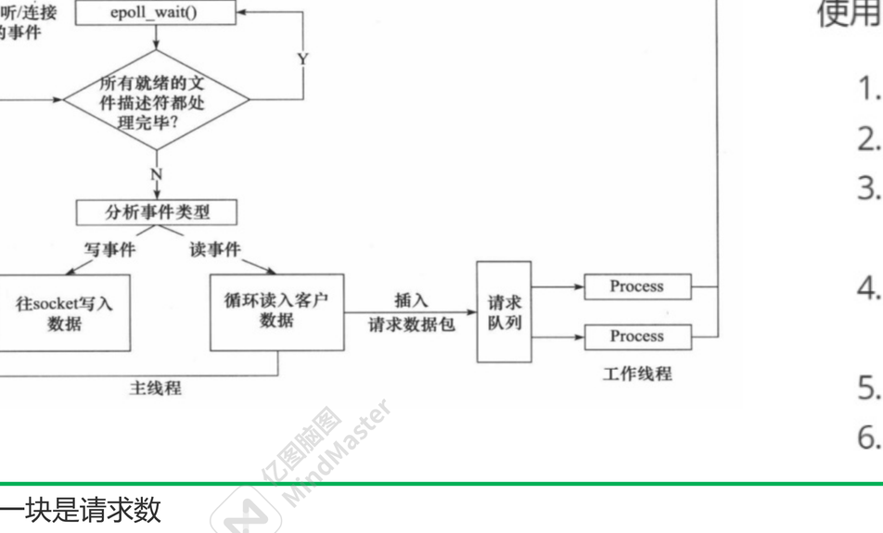
```
//如果是写事件,
else if(events[i].events&EPOLLOUT){
        //一次性写完所有数据，写失败返回false关闭连接
        if(!users[sockfd].write()){
            users[sockfd].close_conn();
        }
}
```

// 分散写,两块不同内存的数据一起写出去，一块是应答协议内容，一块是请求数据，其中请求文件被mmap到内存中
temp = writev(m_sockfd, m_iv, m_iv_count);

- num <0 break;

这里采用同步IO模拟Proactor模式

同步 I/O 模拟 Proactor 模式的工作流程：



使用同步 I/O 方式模拟出 Proactor 模式。原理是：主线程执行数据读写操作，读写完成之后，主线程向工作线程通知这一"完成事件"。那么从工作线程的角度来看，它们就直接获得了数据读写的结果，接下来要做的只是对读写的结果进行逻辑处理。

使用同步 I/O 模型（以 epoll_wait为例）模拟出的 Proactor 模式的工作流程如下：

1. 主线程往 epoll 内核事件表中注册 socket 上的读就绪事件。
2. 主线程调用 epoll_wait 等待 socket 上有数据可读。
3. 当 socket 上有数据可读时，epoll_wait 通知主线程。主线程从 socket 循环读取数据，直到没有更多数据可读，然后将读取到的数据封装成一个请求对象并插入请求队列。
4. 睡眠在请求队列上的某个工作线程被唤醒，它获得请求对象并处理客户请求，然后往 epoll 内核事件表中注册 socket 上的写就绪事件。
5. 主线程调用 epoll_wait 等待 socket 可写。
6. 当 socket 可写时，epoll_wait 通知主线程。主线程往 socket 上写入服务器处理客户请求的结果。

SIGPIPE信号产生的规则：当一个进程向某个已收到RST的套接字执行写操作时，内核向该进程发送SIGPIPE信号。

为了避免进程退出, 可以捕获SIGPIPE信号, 或者忽略它, 给它设置SIG_IGN信号处理函数,这样, 第二次调用write方法时, 会返回-1, 同时errno置为SIGPIPE. 程序便能知道对端已经关闭

线程池 threadpool.h

//创建线程池
pthread_create(m_threads + i,NULL,worker,this) !=0

```
void * threadpool<T>::worker(void* arg){
    threadpool* pool = (threadpool *)arg;
    pool->run();
    return pool;
}
```

```
bool threadpool<T>::append(T* request){
    // 操作工作队列时一定要加锁，因为它被所有线程共享。
    m_queuelocker.lock();
    if(m_workqueue.size() > m_max_requests){
        m_queuelocker.unlock();
        return false;
    }
    m_workqueue.push_back(request);
    m_queuelocker.unlock();
    m_queuestat.post(); //信号量加1，队列中取数据，根据信号量判断是阻塞还是运行
    return true;
}
```

```
//线程开始启动
从请求队列中取出头部请求
注意对请求队列加锁，因为它被所有线程共享
void threadpool<T>::run(){
    while(!m_stop){
        m_queuestat.wait();
        m_queuelocker.lock();
        if(m_workqueue.empty()){
            m_queuelocker.unlock();
            continue;
        }
        //有数据
        T* request = m_workqueue.front();
        m_workqueue.pop_front();
        m_queuelocker.unlock();

        if(!request){
            continue;
        }
        //做任务
        request->process();
    }
}
```

```
//线程开始工作
void http_conn::process(){
    //解析HTTP请求
    HTTP_CODE read_ret= process_read();
    if(read_ret == NO_REQUEST){
        modfd(m_epollfd,m_sockfd,EPOLLIN);
        return;
    }

    printf("parse requset,create response\n");

    //生成响应
    bool write_ret = process_write(read_ret);
    if(!read_ret){
        close_conn();
    }
    modfd(m_epollfd,m_sockfd,EPOLLOUT);
}
```

- process_read()通过主从状态机解析HTTP请求协议
  - 解析出HTTP版本号、协议类型 — 存在m_write_buf中
  - 解析出请求资源的文件名 — 将文件内容映射到mmap共享内存中，得到一个地址
- process_write通过解析协议的返回码给予不同的应答
  - 根据返回的状态响应码回复相应的内容write(m_sockfd,buf)
  - 这里由于有两块内容一个在写缓冲区，一块在mmap内存中
    写用writev----可以分散写，将两块内存写到m_sockfd中

http连接对象 http_conn.cpp