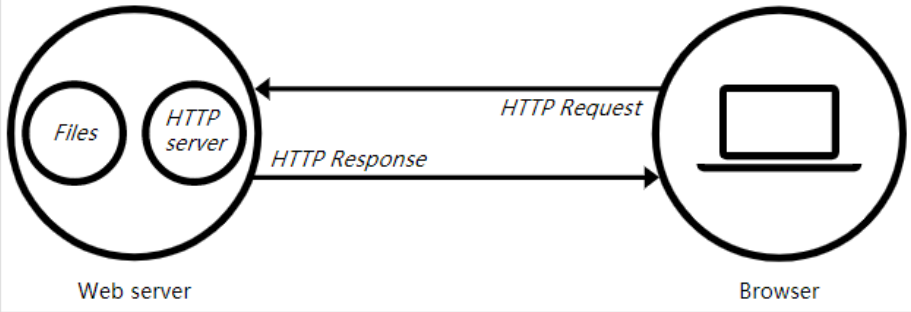
小白视角：一文读懂社长的TinyWebServer

1.什么是Web Server （网络服务器）

一个Web Server就是一个服务器软件（程序），或者是运行这个服务器软件的硬件（计算机）。其主要功能是通过HTTP协议与客户端（通常是浏览器（Browser））进行通信，来接收，存储，处理来自客户端的HTTP请求，并对其请求做出HTTP响应，返回给客户端其请求的内容（文件、网页等）或返回一个Error信息。



https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server

2.用户如何与你的Web服务器进行通信

通常用户使用web浏览器鱼相应服务器进行通信 在浏览器中键入域名 或者 ip地址：端口号 浏览器则先将你的域名解析成相应的IP地址或者直接根据你的IP地址向对应的web服务器发送一个HTTP请求 这一过程首先要通过TCP协议的三次握手建立与目标web服务器的连接 然后HTTP协议生成针对目标web服务器的http请求报文 通过TCP IP等协议发送到目标Web服务器

3.Web服务器如何接收客户端发来的HTTP请求报文呢

Web服务器通过 套接字 socket监听来自用户的请求

```
#include <sys/socket.h> //C
#include <netinet/in.h>
/* 创建监听socket文件描述符 PF_INET代表网络地址范围为IPV4协议簇 SOCK_STREAM（TCP网络）、SOCK_DGRAM（UDP）、SOCK_SEQPACKET 最后的0为指定默认的协议 */
int listenfd = socket(PF_INET, SOCK_STREAM, 0);
/* 创建监听socket的TCP/IP的IPV4 socket地址 sockaddr_in是IPv4结构体 */
struct sockaddr_in address;
bzero(&address, sizeof(address));//将其清零
address.sin_family = AF_INET;//地址族使用IPv4网络协议中使用的地址族
address.sin_addr.s_addr = htonl(INADDR_ANY);  /* INADDR_ANY：将套接字绑定到所有可用的端口 转换过来就是 0.0.0.0 也就是所有网卡ip地址的意思 所有人都可以连接 sin_addr.s_addr保存32位IP地址信息 以网络字节序保存 htonl函数将32位的主机子节顺序转换为网络字节顺序*/
address.sin_port = htons(port);//sin_port保存16位端口号 以网络子节顺序保存 因此需要htons转换 s是short 上面的l是long

int flag = 1;
/* SO_REUSEADDR 允许端口被重复使用 */
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &flag, sizeof(flag));//
setsockopt 设置socket选项
```

```
/* 绑定socket和它的地址 最后为地址的长度*/
ret = bind(listenfd, (struct sockaddr*)&address, sizeof(address));
/* 创建监听队列以存放待处理的客户连接，在这些客户连接被accept()之前 */
ret = listen(listenfd, 5);
```

远端的很多用户会尝试去`connect()`这个Web Server上正在`listen`的这个`port`，而监听到的这些连接会排队等待被`accept()`。由于用户连接请求是随机到达的异步事件，每当监听socket（`listenfd`）`listen`到新的客户连接并且放入监听队列，我们都需要告诉我们的Web服务器有连接来了，`accept`这个连接，并分配一个逻辑单元来处理这个用户请求。而且，我们在处理这个请求的同时，还需要继续监听其他客户的请求并分配其另一逻辑单元来处理（并发，同时处理多个事件，后面会提到使用线程池实现并发）。这里，服务器通过**epoll**这种I/O复用技术（还有select和poll）来实现对监听socket（`listenfd`）和连接socket（客户请求）的同时监听。注意I/O复用虽然可以同时监听多个文件描述符，但是它本身是阻塞的，并且当有多个文件描述符同时就绪的时候，如果不采取额外措施，程序则只能按顺序处理其中就绪的每一个文件描述符，所以为提高效率，我们将在这部分通过线程池来实现并发（多线程并发），为每个就绪的文件描述符分配一个逻辑单元（线程）来处理。

```
#include <sys/epoll.h>
/* 将fd上的EPOLLIN和EPOLLET事件注册到epollfd指示的epoll内核事件中 */
void addfd(int epollfd, int fd, bool one_shot) {
    epoll_event event;//epoll_event为一个epoll机制中的结构体 分伪event和data
    event.data.fd = fd; //fd传递了socket句柄的作用
    event.events = EPOLLIN | EPOLLET | EPOLLRDHUP;//event可以为这几个宏的集合
    /*EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
EPOLLOUT：表示对应的文件描述符可以写；
EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；
EPOLLERR：表示对应的文件描述符发生错误；
EPOLLHUP：表示对应的文件描述符被挂断；
EPOLLET：  将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。
EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里
*/
    /* 针对connfd，开启EPOLLONESHOT，因为我们希望每个socket在任意时刻都只被一个线程处理 */
    if(one_shot)
        event.events |= EPOLLONESHOT;
/*int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。第一个参数是epoll_create()的返回值，第二个参数表示动作，用三个宏来表示：
EPOLL_CTL_ADD：注册新的fd到epfd中；
EPOLL_CTL_MOD：修改已经注册的fd的监听事件；
EPOLL_CTL_DEL：从epfd中删除一个fd；
第三个参数是需要监听的fd，第四个参数是告诉内核需要监听什么事*/
    epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &event);//epollfd = int epoll_create(int size)创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大
    setnonblocking(fd);//将需要监听的文件描述符fd设为非阻塞状态
}
/* 创建一个额外的文件描述符来唯一标识内核中的epoll事件表 */
int epollfd = epoll_create(5);
/* 用于存储epoll事件表中就绪事件的event数组 */
epoll_event events[MAX_EVENT_NUMBER];
/* 主线程往epoll内核事件表中注册监听socket事件，当listen到新的客户连接时，listenfd变为就绪事件 */
addfd(epollfd, listenfd, false);
/* 主线程调用epoll_wait等待一组文件描述符上的事件，并将当前所有就绪的epoll_event复制到events数组中 */
```

```
int number = epoll_wait(epollfd, events, MAX_EVENT_NUMBER, -1);//int
epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout) 等
待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个
events有多大，这个 maxevents的值不能大于创建epoll_create()时的size，参数timeout是超时时间
（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表
示已超时
/* 然后我们遍历这一数组以处理这些已经就绪的事件 */
for(int i = 0; i < number; ++i) {
    int sockfd = events[i].data.fd;  // 事件表中就绪的socket文件描述符
    if(sockfd == listenfd) {  // 当listen到新的用户连接，listenfd上则产生就绪事件
        struct sockaddr_in client_address;
        socklen_t client_addrlength = sizeof(client_address);
        /* ET模式  边沿触发 */
        while(1) {
            /* accept()返回一个新的socket文件描述符用于send()和recv() */
            int connfd = accept(listenfd, (struct sockaddr *) &client_address,
&client_addrlength);
            /* 并将connfd注册到内核事件表中 */
            users[connfd].init(connfd, client_address);
            /* ... */
        }
    }
    else if(events[i].events & (EPOLLRDHUP | EPOLLHUP | EPOLLERR)) {
        // 如有异常，则直接关闭客户连接，并删除该用户的timer
        /* ... */
    }
    else if(events[i].events & EPOLLIN) {
        /* 当这一sockfd上有可读事件时，epoll_wait通知主线程。*/
        if(users[sockfd].read()) { /* 主线程从这一sockfd循环读取数据，直到没有更多数据可
读 */
            pool->append(users + sockfd);  /* 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 */
            /* ... */
        }
        else
            /* ... */
    }
    else if(events[i].events & EPOLLOUT) {
        /* 当这一sockfd上有可写事件时，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 */
        if(users[sockfd].write()) {
            /* ... */
        }
        else
            /* ... */
    }
}
```

服务器程序通常需要处理三类事件 I/O事件 信号和定时事件 有**两种事件处理模式**

- Reactor模式：要求主线程（I/O处理单元）只负责监听文件描述符上是否有事件发生（可读、可写），若有，则立即通知工作线程（逻辑单元），将socket可读可写事件放入请求队列，交给工作线程处理。
- Proactor模式：将所有的I/O操作都交给主线程和内核来处理（进行读、写），工作线程仅负责处理逻辑，如主线程读完成后`users[sockfd].read()`，选择一个工作线程来处理客户请求`pool->append(users + sockfd)`。

通常使用同步I/O模型（如`epoll_wait`）实现Reactor，使用异步I/O（如`aio_read`和`aio_write`）实现Proactor。但在此项目中，我们使用的是**同步I/O模拟的Proactor**事件处理模式。那么什么是同步I/O，什么是异步I/O呢?

- 同步（阻塞）I/O：在一个线程中，CPU执行代码的速度极快，然而，一旦遇到IO操作，如读写文件、发送网络数据时，就需要等待IO操作完成，才能继续进行下一步操作。这种情况称为同步IO。
- 异步（非阻塞）I/O：当代码需要执行一个耗时的IO操作时，它只发出IO指令，并不等待IO结果，然后就去执行其他代码了。一段时间后，当IO返回结果时，再通知CPU进行处理。

Linux下有三种IO复用方式：epoll，select和poll，为什么用epoll，它和其他两个有什么区别呢？（参考StackOverflow上的一个问题：Why is epoll faster than select?）

- 对于select和poll来说 所有的文件描述符fd都是在用户态被加入其文件描述符集合的 每次调用都需要将整个集合拷贝到内核态 epoll则将整个文件描述符集合维护在内核态 每次添加文件描述符的时候都需要执行一个系统调用 系统的开销是很大的 而且在有很多短期活跃连接的情况下 epoll可能会慢于select和poll由于这些大量的系统调用开销
- select使用线性表来描述文件描述符集合 文件描述有上限 poll使用链表来描述 epoll底层通过红黑树来描述 并且维护一个ready list 将事件表中已经就绪的事件添加到这里 在使用epoll_wait调用时仅仅观察这个list中有没有数据即可
- select和poll最大的开销来自于内核判断是否有文件描述符就绪这一过程：每次执行select或者poll调用时 他们会采取遍历的方式 遍历整个文件描述符集合去判断各个文件描述符是否有活动 epoll则不需要去以这种方式检查 当有活动产生时 会自动触发epoll回调函数通知epoll文件描述符 然后内核将这些就绪的文件描述符放到之前提到的ready_list中等待epoll_wait调用后被处理
- select和poll都只能工作在相对低效的LT模式下 而epoll同时支持LT和ET模式
- 综上 当监测的fd数量较小 且各个fd都很活跃的情况下 建议使用select和poll 当监听的fd数量较多且单位时间仅部分fd活跃的情况下 使用epoll会明显提升性能

`Epoll`对文件操作符的操作有两种模式：LT（电平触发）和ET（边缘触发），二者的区别在于当你调用`epoll_wait`的时候内核里面发生了什么：

- LT 电平触发：类似select LT会去遍历在epoll事件表中每个文件描述符 来观察是否有我们感兴趣的事件发生 如果有（触发了该文件描述符上的回调函数）epoll_wait就会以非阻塞的方式返回 若该epoll事件没有处理完 （没有返回EWOULDBLOCK）该事件还会被后续的epoll_wait再次触发
- ET 边缘触发：ET在发现有我们感兴趣的事件发生后 立即返回 并且sleep这一事件的epoll_wait 不管该事件有没有结束

在使用ET模式时 必须要保证该文件描述符是非阻塞的（确保没有数据可读时 该描述符不会一直阻塞）并且每次调用read 和 write的时候必须等到他们返回EWOULDBLOCK 确保所有数据都已经读写完

4.Web服务器如何处理及响应接收到的HTTP请求报文呢

该项目使用线程池（半同步半反应堆模式）并发处理用户请求，主线程负责读写，工作线程（线程池中的线程）负责处理逻辑（HTTP请求报文的解析等等）。通过之前的代码，我们将`listenfd`上到达的`connection`通过`accept()`接收，并返回一个新的socket文件描述符`connfd`用于和用户通信，并对用户请求返回响应，同时将这个`connfd`注册到内核事件表中，等用户发来请求报文。这个过程是：通过`epoll_wait`发现这个`connfd`上有可读事件了（`EPOLLIN`），主线程就将这个HTTP的请求报文读进这个连接socket的读缓存中`users[sockfd].read()`，然后将该任务对象（指针）插入线程池的请求队列中`pool->append(users + sockfd);`，线程池的实现还需要依靠**锁机制**以及**信号量**机制来实现线程同步，保证操作的原子性。

在线程池部分做几点解释，然后大家去看代码的时候就更容易看懂了：

- 所谓线程池 就是一个pthread_t类型的普通数组 通过`pthread_create()`函数创建`m_thread_number`个**线程** 用来执行worker()函数以执行每个请求处理函数（HTTP请求的process

函数） 通过`pthread_detach()`将线程设置成脱离态（detached）后 当这一线程运行结束时 它的资源会被系统自动回收 而不再需要在其他线程中对其进行pthread_join()操作
- 操作工作队列一定要加锁(locker) 因为它被所有线程共享
- 我们用信号量来标识请求队列中的请求数 通过m_queuestat.wait()来等待一个请求队列中待处理的HTTP请求 然后交给线程池中的空闲线程来处理

为什么要使用线程池?

当你需要限制你应用中同时运行的线程数时 线程池非常有用 因为启动一个新线程会带来性能开销 每个线程也会为其堆栈分配一些内存等 为了任务的并发执行 我们可以将这些任务传递到线程池 而不是为每个任务动态开启一个新的线程

☆线程池中的线程数量是依据什么确定的?

在StackOverflow上面发现了一个还不错的回答，意思是：
线程池中的线程数量最直接的限制因素是中央处理器(CPU)的处理器(processors/cores)的数量`N`：如果你的CPU是4-cores的，对于CPU密集型的任务(如视频剪辑等消耗CPU计算资源的任务)来说，那线程池中的线程数量最好也设置为4（或者+1防止其他因素造成的线程阻塞）；对于IO密集型的任务，一般要多于CPU的核数，因为线程间竞争的不是CPU的计算资源而是IO，IO的处理一般较慢，多于cores数的线程将为CPU争取更多的任务，不至在线程处理IO的过程造成CPU空闲导致资源浪费，公式：`最佳线程数 = CPU当前可使用的Cores数 * 当前CPU的利用率 * (1 + CPU等待时间 / CPU处理时间)`（还有回答里面提到的Amdahl准则可以了解一下）

OK，接下来说说每个`read()`后的HTTP请求是如何被处理的，我们直接看这个处理HTTP请求的入口函数：

```
void http_conn::process() {
    HTTP_CODE read_ret = process_read();
    if(read_ret == NO_REQUEST) {
        modfd(m_epollfd, m_sockfd, EPOLLIN);
        return;
    }
    bool write_ret = process_write(read_ret);
    if(!write_ret)
        close_conn();
    modfd(m_epollfd, m_sockfd, EPOLLOUT);
}
```

首先 process_read() 也就是对我们读入该connfd读缓冲区的请求报文进行解析

HTTP请求报文由请求行 请求头部 空行和请求数据四个部分组成

有两种报文

GET（Example）

```
GET /562f25980001b1b106000338.jpg HTTP/1.1
Host:img.mukewang.com
User-Agent:Mozilla/5.0 (Windows NT 10.0; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.106 Safari/537.36
Accept:image/webp,image/*,*/*;q=0.8
Referer:http://www.imooc.com/
Accept-Encoding:gzip, deflate, sdch
Accept-Language:zh-CN,zh;q=0.8
空行
请求数据为空
```

POST（Example，注意POST的请求内容不为空）

```
POST / HTTP1.1
Host:www.wrox.com
User-Agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
Content-Type:application/x-www-form-urlencoded
Content-Length:40
Connection: Keep-Alive
空行
name=Professional%20Ajax&publisher=Wiley
```

GET和POST的区别

- 最直观的区别是GET把参数包含在URL中 POST通过request body传递参数
- GET请求参数会被完整的保存在浏览器历史记录里 而POST中的参数不会被保留
- GET请求在URL中传送的参数是有长度限制 大多数浏览器都会限制URL长度在2k子节 而大多数服务器最多处理64K大小的url
- GET产生一个TCP数据包 POST产生两个TCP数据报 对于GET方式的请求 浏览器会把http header和data一并发送出去 服务器响应200（返回数据） 而对于POST 浏览器会先发送header 服务器响应100 （提示信息 表示请求已接收 继续处理） continue 浏览器再发送data 服务器响应200 0k（返回数据）

`process_read()` 函数的作用就是将类似上述例子的请求报文进行解析 因为用户请求内容包含在这个请求报文里面 只有通过解析 知道用户请求的内容是什么 是请求图片 还是视频 还是其他请求 我们根据这些请求返回相应的HTML页面等 项目中使用主从状态机的模式进行解析 从状态机（parse_line)负责读取报文的一行 主状态机负责对该数据进行解析 主状态机内部调用从状态机 从状态机驱动主状态机 每解析一部分都会将整个请求的 m_check_state状态改变 状态机也就是根据这个状态来进行不同部分的解析跳转的

- `parse_request_line(text)`，解析请求行，也就是GET中的`GET /562f25980001b1b106000338.jpg HTTP/1.1`这一行，或者POST中的`POST / HTTP1.1`这一行。通过请求行的解析我们可以判断该HTTP请求的类型（GET/POST），而请求行中最重要的部分就是`URL`部分，我们会将这部分保存下来用于后面的生成HTTP响应。
- `parse_headers(text);`，解析请求头部，GET和POST中`空行`以上，请求行以下的部分。
- `parse_content(text);`，解析请求数据，对于GET来说这部分是空的，因为这部分内容已经以明文的方式包含在了请求行中的`URL`部分了；只有POST的这部分是有数据的，项目中的这部分数据为用户名和密码，我们会根据这部分内容做登录和校验，并涉及到与数据库的连接。

经过上述的解析 当得到一个完整的 正确的HTTP请求时 就到了do_request代码部分 我们首先对GET请求和不同POST请求（登陆 注册 请求图片 视频等等）做不同的预处理 然后分析目标文件的属性 若目标文件存在 对所有用户可读且不失目录时 则使用mmap将其映射到内存地址 m_file_address处 并告诉调用者获取文件成功

抛开`mmap`这部分，先来看看这些不同请求是怎么来的：
假设你已经搭好了你的HTTP服务器，然后你在本地浏览器中键入`localhost:9000`，然后回车，这时候你就给你的服务器发送了一个GET请求，什么都没做，然后服务器端就会解析你的这个HTTP请求，然后发现是个GET请求，然后返回给你一个静态HTML页面，也就是项目中的`judge.html`页面，那POST请求怎么来的呢？这时你会发现，返回的这个`judge`页面中包含着一些`新用户`和`已有账号`这两个`button`元素，当你用鼠标点击这个`button`时，你的浏览器就会向你的服务器发送一个POST请求，服务器段通过检查`action`来判断你的POST请求类型是什么，进而做出不同的响应。

```
/* judge.html */
<!DOCTYPE html>
```

```
<html>
    <head>
        <meta charset="UTF-8">
        <title>WebServer</title>
    </head>
    <body>
    <br/>
    <br/>
    <div align="center"><font size="5"> <strong>欢迎访问</strong></font></div>
    <br/>
        <br/>
        <form action="0" method="post">
            <div align="center"><button type="submit">新用户</button></div>
        </form>
        <br/>
        <form action="1" method="post">
            <div align="center"><button type="submit" >已有账号</button></div>
        </form>

        </div>
    </body>
</html>
```

5.数据库连接池是如何运行的

在用户注册 登陆请求的时候 我们需要将这些用户的用户名和密码保存下来用于新用户的注册及老用户的登录校验 相信每个人都体验过，当你在一个网站上注册一个用户时，应该经常会遇到“您的用户名已被使用”，或者在登录的时候输错密码了网页会提示你“您输入的用户名或密码有误”等等类似情况，这种功能是服务器端通过用户键入的用户密码和数据库中已记录下来的用户名密码数据进行校验实现的 若每次用户请求我们都需要新建一个数据库连接 请求结束后我们释放该数据库连接 当用户请求连接过多时 这种做法过于低效 所以类似线程池的做法 我们构建一个数据库连接池 预先生成一些数据库连接放在那里供用户请求使用

我们首先看单个数据库连接是如何生成的

1.使用mysql_init（）初始化连接

2.使用mysql_real_connect（）建立一个到mysql数据库的连接

3.使用mysql_query（）执行查询语句

4.使用result = mysql_store_result（mysql）获取结果集

5.使用mysql_num_fields（result）获取查询的列数 mysql_num_rows（result）获取结果集的行数

6.通过mysql_fetch_row（result）不断获取下一行 然后循环输出

7.使用mysql_free_result（result）释放结果集所占内存

8.使用mysql_close（conn）关闭连接

对于一个数据库连接池来讲 就是预先生成多个这样的数据库连接 然后放在一个链表中 同时维护最大连接数 MAX_CONN 当前可用连接数 FREE_CONN和当前已用连接数CUR_CONN三个变量 同样注意在对连接池操作时 （获取 释放）要用到锁机制 因为它被所有线程共享

6.什么是CGI校验

OK，弄清楚了数据库连接池的概念及实现方式，我们继续回到第4部分，对用户的登录及注册等POST请求，服务器是如何做校验的。

当点击 新用户 按钮时 服务器对这个POST请求的响应是 返回用户一个登录界面 当你在用户名和密码框中输入后 你的POST请求报文中会连同你的用户名密码一起发给服务器 然后我们拿着你的用户名和密码在数据库连接池中取出一个链接用于 mysql_query（）进行查询 逻辑很简单，同步线程校验`SYNSQL`方式相信大家都能明白，但是这里社长又给出了其他两种校验方式，CGI什么的，就很容易让小白一时摸不到头脑，接下来就简单解释一下CGI是什么

CGI（通用网关接口） 它是一个运行在Web服务器上的程序 在编译的时候将响应的.cpp文件编程成.cgi文件并在主程序中调用即可 （通过社长的`makefile`文件内容也可以看出）这些CGI程序通常通过客户在其浏览器上点击一个button时运行 这些程序通常用来执行一些信息搜索 存储等任务 而且通常会生成一个动态的HTML网页来响应客户的HTTP请求 我么可以发现项目中的`sign.cpp`文件就是我们的CGI程序，将用户请求中的用户名和密码保存在一个`id_passwd.txt`文件中，通过将数据库中的用户名和密码存到一个`map`中用于校验。在主程序中通过`execl(m_real_file, &flag, name, password, NULL);`这句命令来执行这个CGI文件，这里CGI程序仅用于校验，并未直接返回给用户响应。这个CGI程序的运行通过多进程来实现，根据其返回结果判断校验结果（使用`pipe`进行父子进程的通信，子进程将校验结果写到pipe的写端，父进程在读端读取）。

7.生成HTTP响应并返回给用户

通过以上操作，我们已经对读到的请求做好了处理，然后也对目标文件的属性作了分析，若目标文件存在、对所有用户可读且不是目录时 则使用mmap将其映射到内存地址 m_file_address处 并告诉调用者获取文件成功 FILE_REQUEST 接下来要做的就是根据读取结果对用户做出响应了 也就是到了process_write(read_ret)这一步 这一步 该函数根据 process_read（）的返回结果来判断应该返回给用户什么响应 我们最常见的就是404错误了 说明客户请求的文件不存在 除此之外还有其他类型的请求出错的响应 然后呢 假设用户请求的文件存在 而且已经被mmap到m_file_address这里了 那我们就将做如下写操作 将响应写到这个connfd的写缓存m_write_buf中去

```
case FILE_REQUEST: {
    add_status_line(200, ok_200_title);//状态行写入写缓存
    if(m_file_stat.st_size != 0) {
        add_headers(m_file_stat.st_size);//响应头写进connfd的写缓存
        m_iv[0].iov_base = m_write_buf;
        m_iv[0].iov_len = m_write_idx;
        m_iv[1].iov_base = m_file_address;
        m_iv[1].iov_len = m_file_stat.st_size;
        m_iv_count = 2;
        bytes_to_send = m_write_idx + m_file_stat.st_size;
        return true;
    }
    else {
        const char* ok_string = "<html><body></body></html>";
        add_headers(strlen(ok_string));
        if(!add_content(ok_string))
            return false;
    }
}
```

首先将状态行写入写缓存 响应头也是要写进connfd的写缓存（HTTP类自己定义的 与socket无关）中的对于请求的文件 我们已经直接将其映射到m_file_address里面 然后将该connfd文件描述符上修改为EPOLLOUT（可写）事件 然后epoll_wait监测到这一事件后 使用writev来将相应信息和请求文件聚集写道TCP socket本身定义的发送缓冲区（这个缓冲区大小一般是默认的 但我们也可以通过setsockeopt来修改）中 交由内核发送给用户 over

8.服务器优化：定时器处理非活动连接

项目中 我们预先分配了MAX_FD个http连接对象

```
// 预先为每个可能的客户连接分配一个http_conn对象
http_conn* users = new http_conn[MAX_FD];
```

如果某一用户`connect()`到服务器之后，长时间不交换数据，一直占用服务器端的文件描述符，导致连接资源的浪费。这时候就应该利用定时器把这些超时的非活动连接释放掉，关闭其占用的文件描述符。这种情况也很常见，当你登录一个网站后长时间没有操作该网站的网页，再次访问的时候你会发现需要重新登录。

项目中使用的是SIGNAL RM信号来实现定时器 利用alarm函数周期性的触发SIGALRM信号 信号处理函数利用管道通知主循环 主循环接收到该信号后对升序链表上所有定时器进行处理 若该段时间内没有交换数据 则将该连接关闭 释放所占用的资源 接下来看项目中的具体实现

```
/* 定时器相关参数 */
static int pipefd[2];
static sort_timer_lst timer_lst//利用升序链表来管理定时器

/* 每个user（http请求）对应的timer */
client_data* user_timer = new client_data[MAX_FD];
/* 每隔TIMESLOT时间触发SIGALRM信号 */
alarm(TIMESLOT);
/* 创建管道，注册pipefd[0]上的可读事件 */
/*int socketpair(int d, int type, int protocol, int sv[2])   socketpair()函数用于创建一对无名的、相互连接的套接子。
如果函数成功，则返回0，创建好的套接字分别是sv[0]和sv[1]；否则返回-1，错误码保存于errno中   这对套接字可以用于全双工通信 */
int ret = socketpair(PF_UNIX, SOCK_STREAM, 0, pipefd);
/* 设置管道写端为非阻塞 */
setnonblocking(pipefd[1]);
/* 设置管道读端为ET非阻塞，并添加到epoll内核事件表 */
addfd(epollfd, pipefd[0], false);

addsig(SIGALRM, sig_handler, false);
addsig(SIGTERM, sig_handler, false);
```

alarm函数会定期触发SIGALRM信号 这个信号交由sig_handler来处理 每当监测到有这个信号的时候 都会将这个信号写到pipefd[1]里面 传递给主循环

```
/* 处理信号 */
else if(sockfd == pipefd[0] && (events[i].events & EPOLLIN)) {
    int sig;
    char signals[1024];//用来存放rcv函数接收到的数据的缓冲区
    /*用rcv函数从TCP连接的另一端接收数据 第一个参数指定接收端套接字描述符 第二个参数指明一个缓冲区 第三个参数为buf的长度 第四个参数一般置0 */
    ret = recv(pipefd[0], signals, sizeof(signals), 0);
    /*recv函数返回其实际copy的字节数。如果recv在copy时出错，那么它返回  SOCKET_ERROR；如果recv函数在等待协议接收数据时网络中断了，那么它返回0*/
    if(ret == -1) {
        continue;  // handle the error
    }
    else if(ret == 0) {
        continue;
    }
```

```
    else {
        for(int j = 0; j < ret; ++j) {
            switch (signals[j]) {
                case SIGALRM: {
                    timeout = true;
                    break;
                }
                case SIGTERM: {
                    stop_server = true;
                }
            }
        }
    }
}
```

当我们在读端pipefd[0]读到这个信号的时候 就会将timeout置为true 并且跳出循环 让timer_handler（）函数取出来定时器上的到期任务 该定时器是通过升序链表来实现的 从头到尾对检查任务是否超时 若超时则调用定时器的回调函数cb_func（） 关闭该socket连接 并删除其对应的定时器 del_timer

```
void timer_handler() {
    /* 定时处理任务 */
    timer_lst.tick();
    /* 重新定时以不断触发SIGALRM信号 */
    alarm(TIMESLOT);
}
```

定时器优化

这个基于升序双向链表实现的定时器存在着其固有缺点：

- 每次遍历添加和修改定时器的效率偏低(O(n))，使用最小堆结构可以降低时间复杂度降至(O(logn))。
- 每次以固定的时间间隔触发`SIGALRM`信号，调用`tick`函数处理超时连接会造成一定的触发浪费，举个例子，若当前的`TIMESLOT=5`，即每隔5ms触发一次`SIGALRM`，跳出循环执行`tick`函数，这时如果当前即将超时的任务距离现在还有`20ms`，那么在这个期间，`SIGALRM`信号被触发了4次，`tick`函数也被执行了4次，可是在这4次中，前三次触发都是无意义的。对此，我们可以动态的设置`TIMESLOT`的值，每次将其值设置为**当前最先超时的定时器与当前时间的时间差**，这样每次调用`tick`函数，超时时间最小的定时器必然到期，并被处理，然后在从时间堆中取一个最先超时的定时器的时间与当前时间做时间差，更新`TIMESLOT`的值。

服务器优化：日志

日志 由服务器自动创建 并记录运行状态 错误信息 访问数据的文件

这部分内容个人感觉相对抽象一点，涉及单例模式以及单例模式的两种实现方式：懒汉模式和恶汉模式，以及条件变量机制和生产者消费者模型。这里大概就上述提到的几点做下简单解释，具体的还是去看参考中社长的笔记。

- 单例模式 最常用的设计模式之一 保证一个类仅有一个实例 并提供一个访问它的全局访问点 该实例被所有程序模块共享 实现思路：私有化它的构造函数 以防止外界创建单例类的对象 实用类的私有静态指针变量指向类的唯一实例 并用一个共有的静态方法获取该实例
    - 懒汉模式 即非常懒 不用的时候不去初始化 所以在第一次被使用时才进行初始化 （实例的初始化放在getinstance函数内部）
        - 经典的线程安全懒汉模式 使用双检测锁机制 （p == NULL 检测了两次）

        - 利用局部静态变量实现线程安全懒汉模式
    - 饿汉模式：即迫不及待 在程序运行时立即初始化 （实例的初始化放在getinstance函数外部 getinstance函数仅返回该唯一实例的指针）

日志系统的运行机制

- 日志文件
    - 局部变量的懒汉模式获取实例
    - 生成日志文件，并判断同步和异步写入方式
- 同步
    - 判断是否分文件
    - 直接格式化输出内容，将信息写入日志文件
- 异步
    - 判断是否分文件
    - 格式化输出内容，将内容写入阻塞队列，创建一个写线程，从阻塞队列取出内容写入日志文件

压测 （非常关键）

一个服务器项目，你在本地浏览器键入`localhost:9000`发现可以运行无异常还不够，你需要对他进行压测（即服务器并发量测试），压测过了，才说明你的服务器比较稳定了。社长的项目是如何压测的呢?

用到了一个压测软件叫做Webbench，可以直接在社长的Gtihub里面下载，解压，然后在解压目录打开终端运行命令（`-c`表示客户端数，`-t`表示时间）：

```
./webbench -c 10001 -t 5 http://127.0.0.1:9006/
```

直接解压的`webbench-1.5`文件夹下的`webbench`文件可能会因为权限问题找不到命令或者无法执行，这时你需要重新编译一下该文件即可：

```
gcc webbench.c -o webbench
```

然后我们就可以压测得到结果了（我本人电脑的用户数量`-c`设置为`10500`会造成资源不足的错误）：

```
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.

Benchmarking: GET http://127.0.0.1:9006/
10001 clients, running 5 sec.

Speed=1044336 pages/min, 2349459 bytes/sec.
Requests: 87028 susceed, 0 failed.
```

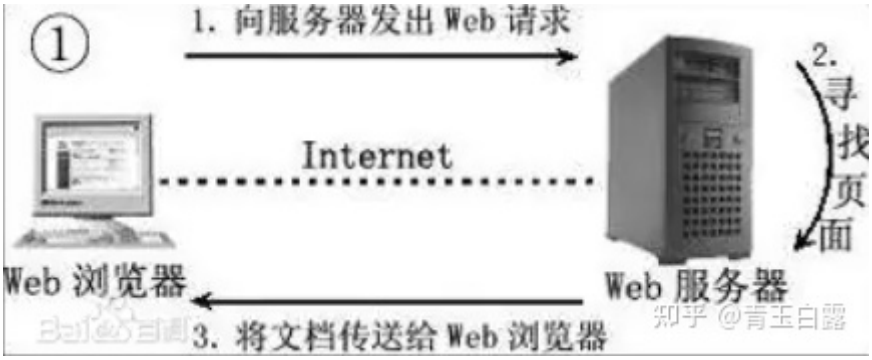
Webbench是什么，介绍一下原理
父进程fork若干个子进程，每个子进程在用户要求时间或默认的时间内对目标web循环发出实际访问请求，父子进程通过管道进行通信，子进程通过管道写端向父进程传递在若干次请求访问完毕后记录到的总信息，父进程通过管道读端读取子进程发来的相关信息，子进程在时间到后结束，父进程在所有子进程退出后统计并给用户显示最后的测试结果，然后退出。

庖丁解牛

00.基础知识

什么是web sever?

Web服务器一般指网站服务器，是指驻留于因特网上某种类型计算机的**程序**，可以处理**浏览器等Web客户端的请求并返回相应响应——**可以放置网站文件，让全世界浏览；可以放置数据文件，让全世界下载。目前最主流的三个Web服务器是**Apache、 Nginx 、IIS。**服务器与客户端的关系如下：



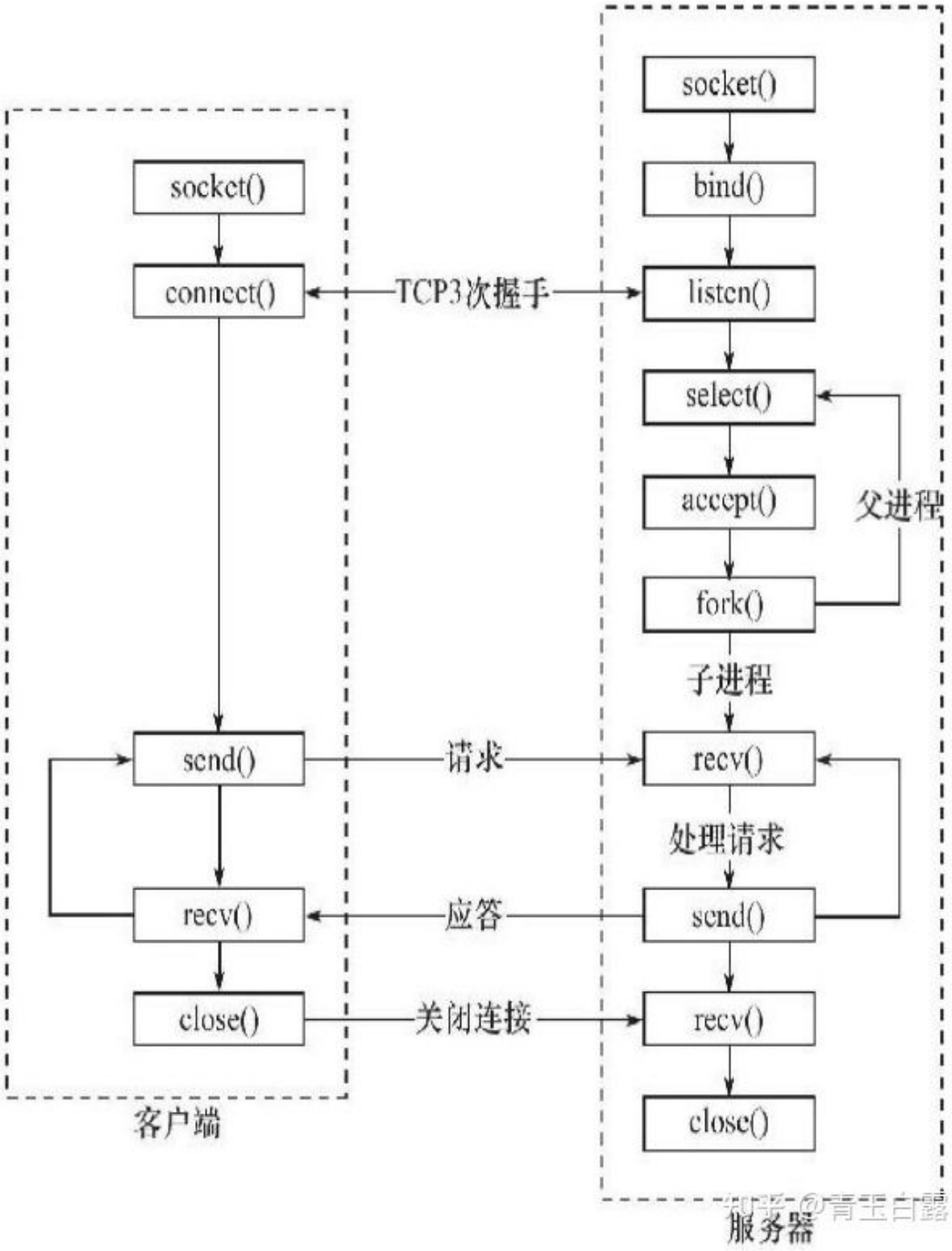
在本项目中，web请求主要是指HTTP协议，有关HTTP协议知识可以参考介绍，HTTP基于TCP/IP，进一步了解请百度。

什么是socket?

客户端与主机之间是如何通信的？——Socket

socket起源于Unix，**而Unix/Linux基本哲学之一就是“一切皆文件”，都可以用“打开open –> 读写write/read –> 关闭close”模式来操作。**Socket就是该模式的一个实现，socket即是一种特殊的文件，一些socket函数就是对其进行的操作（读/写IO、打开、关闭）

TCP服务器与TCP客户端的工作流程见下：

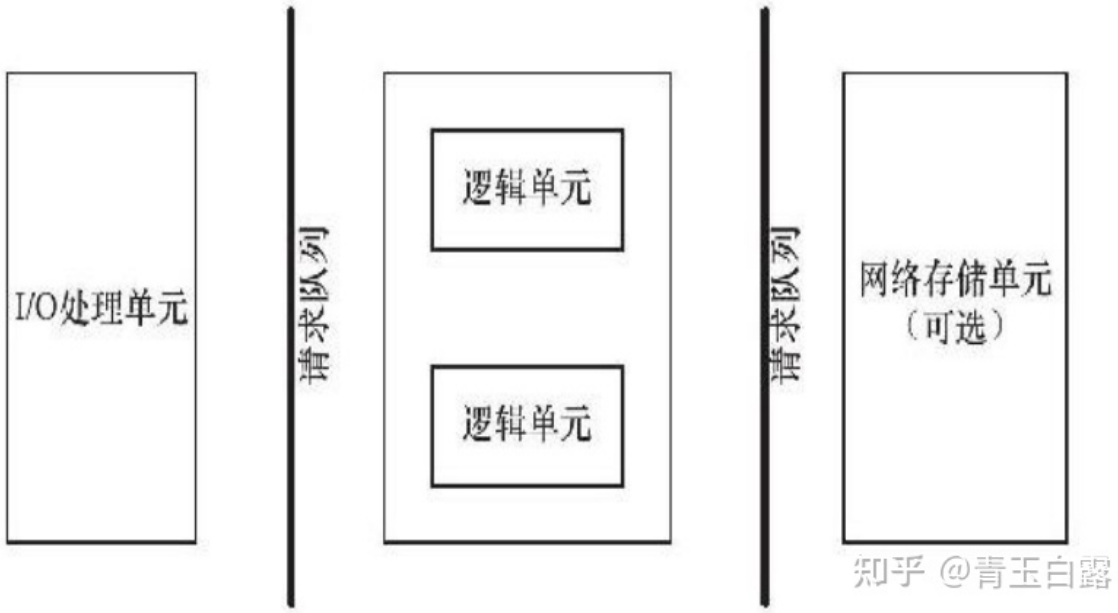


试想，如果有多个客户端都想connect服务器，那么服务器如何对这些客户端进行处理？这就需要介绍一下IO复用。

IO复用是什么

IO复用指的是在单个进程中通过记录跟踪每一个socket（i./o流）的状态来同时管理多个I/O流 发明它的原因是尽量多的提高服务器的吞吐能力

如上文所说，当多个客户端与服务器连接时，这就涉及如何"同时"给每个客户端提供服务的问题。服务器的基本框架如下：



本项目是利用epoll IO复用技术实现对监听socket(listenfd)和连接socket（客户请求连接之后的)的同时监听 注意I/O复用虽然可以同时监听多个文件描述符 但是它本身是阻塞的 所以为提高效率 这部分通过线程池来实现并发 为每个就绪的文件描述符分配一个逻辑单元（线程）来处理

Unix有**五种基本的IO模型**：

- 阻塞式IO（守株待兔）
- 非阻塞式IO（没有就返回，直到有，其实是一种轮询（polling）操作）
- IO复用（select、poll等，使系统阻塞在select或poll调用上，而不是真正的IO系统调用（如recvfrom），等待select返回可读才调用IO系统，其优势就在于可以等待多个描述符就位）
- 信号驱动式IO（sigio，即利用信号处理函数来通知数据已完备且不阻塞主进程）
- 异步IO（posix的aio_系列函数，与信号驱动的区别在于，信号驱动是内核告诉我们何时可以进行IO，而后者是内核通知何时IO操作已完成）

对于到来的IO事件（或是其他的信号/定时事件），又有**两种事件处理模式**：

- **Reactor模式：**要求**主线程（I/O处理单元）**只负责监听文件描述符上是否有事件发生（可读、可写），若有，**则立即通知工作线程**，将socket**可读可写事件放入请求队列**，读写数据、接受新连接及处理客户请求**均在工作线程中完成。**(需要区别读和写事件)
- **Proactor模式：**主线程和内核负责处理读写数据、接受新连接**等I/O操作**，**工作线程仅负责业务逻辑（给予相应的返回url），如处理客户请求。**

通常使用**同步I/O模型（如epoll_wait）实现Reactor**，使用**异步I/O**（如aio_read和aio_write）实现Proactor，**但是异步IO并不成熟，本项目中使用同步IO模拟proactor模式。**有关这一部分的进一步介绍请参考第四章、线程池。

PS：什么是同步I/O，什么是异步I/O呢?

- 同步（阻塞）I/O：**等待IO操作完成，才能继续进行下一步操作。**这种情况称为同步IO。
- 异步（非阻塞）I/O：当代码执行IO操作时，它只发出IO指令，并不等待IO结果，然后就去执行其他代码了。一段时间后，当IO返回结果时（内核已经完成数据拷贝），再通知CPU进行处理。（异步操作的潜台词就是**你先做，我去忙其他的，你好了再叫我**）

- 对于select和poll来说，所有文件描述符都是在用户态被加入其文件描述符集合的，**每次调用都需要将整个集合拷贝到内核态**；epoll则将整个文件描述符集合维护在内核态，每次添加文件描述符的

时候都需要**执行一个系统调用。**系统调用的开销是很大的，而且在有很多短期活跃连接的情况下，epoll可能会慢于select和poll由于这些大量的系统调用开销。

- select使用线性表描述文件描述符集合，**文件描述符有上限**；poll使用**链表来描述**；epoll底层通过红黑树来描述，并且维护一个ready list，将事件表中已经就绪的事件添加到这里，在使用epoll_wait调用时，仅观察这个list中有没有数据即可。
- select和poll的最大开销来自内核判断是否有文件描述符就绪这一过程：每次执行select或poll调用时，**它们会采用遍历的方式**，遍历整个文件描述符集合去判断各个文件描述符是否有活动；epoll则不需要去以这种方式检查，当有活动产生时，**会自动触发epoll回调函数通知epoll文件描述符**，然后内核将这些就绪的文件描述符放到之前提到的**ready list中等待epoll_wait调用后被处理。**
- select和poll都只能工作在**相对低效的LT模式下**，而epoll同时支持LT和ET模式。
- 综上，**当监测的fd数量较小**，且各个fd都很活跃的情况下，建议使用select和poll；**当监听的fd数量较多**，且单位时间仅部分fd活跃的情况下，使用epoll会明显提升性能。

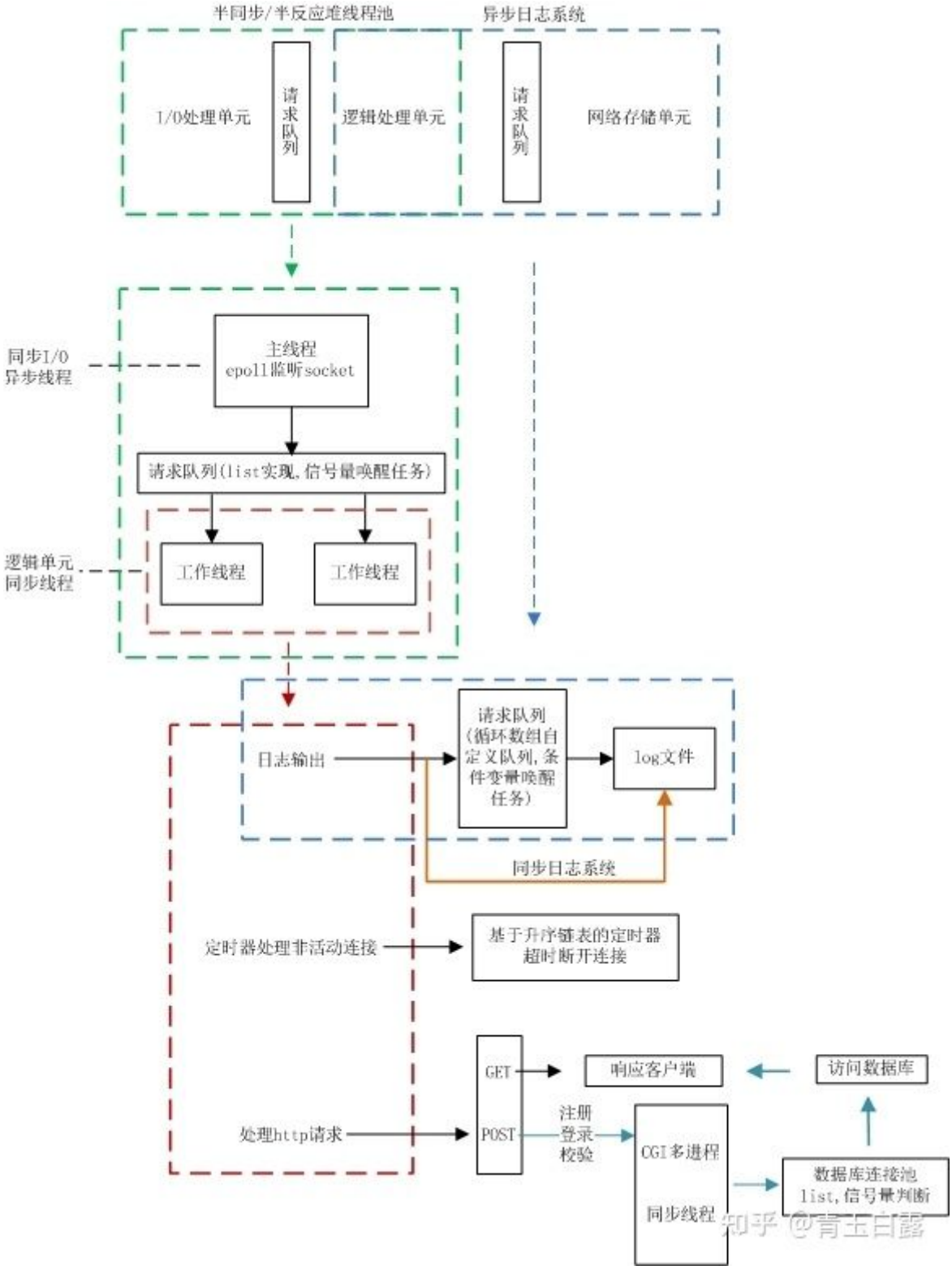
其中提到的LT与ET是什么意思?

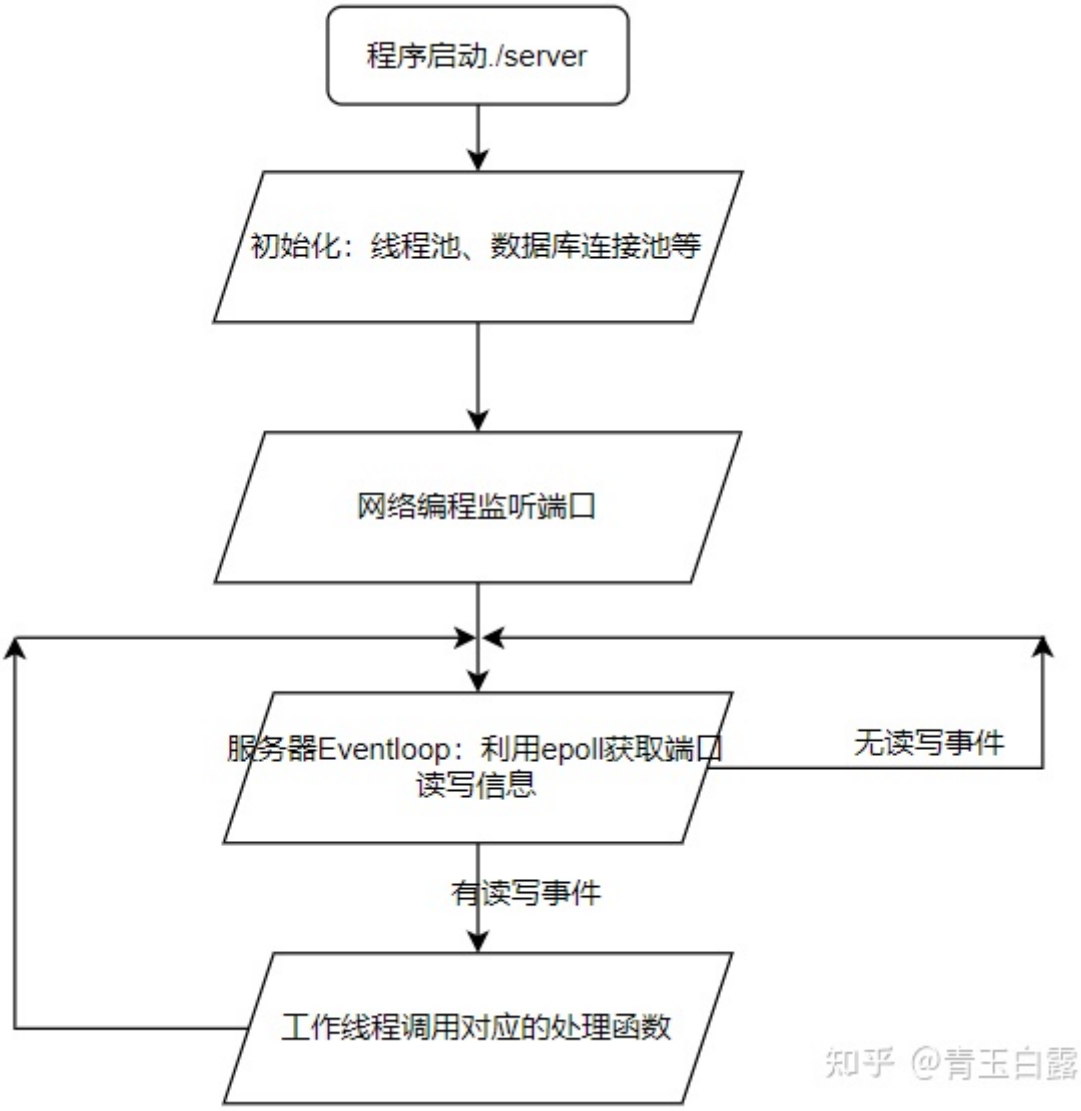
- LT是指电平触发（level trigger），当IO事件就绪时，内核会一直通知，直到该IO事件被处理；
- ET是指边沿触发（Edge trigger），当IO事件就绪时，内核只会通知一次，如果在这次没有及时处理，该IO事件就丢失了。

为什么是多线程

上文提到了并发多线程，在计算机中程序是作为一个进程存在的，**线程是对进程的进一步划分**，即在一个进程中可以有多个不同的代码执行路径。相对于进程而言，线程不需要操作系统为其分配资源，因为它的资源就在进程中，并且线程的创建和销毁相比于进程小得多，所以多线程程序效率较高。

但是在服务器项目中，**如果频繁地创建/销毁线程也是不可取的**，这就引入了线程池技术，即提前创建一批线程，当有任务需要执行时，就从线程池中选一个线程来进行任务的执行，任务执行完毕之后，再将该线程丢进线程池中，以等待后续的任务。


半同步/半反应堆线程池
异步日志系统
I/O处理单元
请求队列
逻辑处理单元
请求队列
网络存储单元
同步I/O
异步线程
主线程
epoll监听socket
请求队列(list实现,信号量唤醒任务)
逻辑单元
同步线程
工作线程
工作线程
请求队列
(循环数组自定义队列,条件变量唤醒任务)
日志输出
log文件
同步日志系统
定时器处理非活动连接
基于升序链表的定时器
超时断开连接
GET
响应客户端
访问数据库
处理http请求
POST
注册
登录
校验
CGI多进程
同步线程
数据库连接池
list,信号量判断
知乎 @青玉白露



远端的很多用户会尝试去connect（）这个webserver上正在listen的这个port 而监听到的这些连接会排队等待被accept（） 由于用户连接请求是随即到达的异步事件

每当监听到新的客户连接listenfd并且放入监听队列 我们都需要告诉我们的web服务器有连接来了accept这个连接 并分配一个逻辑单元来处理这个用户请求

而且我们在处理这个请求的同时 还需要继续监听其他客户的请求并分配其零一逻辑单元来处理 （并发利用线程池实现）这里，服务器通过**epoll**这种I/O复用技术（还有select和poll）来实现对监听socket（`listenfd`）和连接socket（connfd 客户请求）的同时监听。注意I/O复用虽然可以同时监听多个文件描述符，但是它本身是阻塞的，并且当有多个文件描述符同时就绪的时候，如果不采取额外措施，程序则只能按顺序处理其中就绪的每一个文件描述符，所以为提高效率，我们将在这部分通过线程池来实现并发（多线程并发），为每个就绪的文件描述符分配一个逻辑单元（线程）来处理。

该项目使用线程池并发处理用户的请求 **主线程负责读写 工作线程（线程池中的线程）负责处理逻辑（HTTP请求报文的解析等等）**通过前一步的工作 我们将listenfd上到达的connection通过accept（）接收 并返回一个新的socket文件描述符connfd用于和用户通信 并对用户请求返回响应 同时将这个connfd注册到内核事件表中 等待用户发来请求报文 这个过程是：

通过epoll_wait发现这个connfd上有可读事件了EPOLLIN 主线程就把这个HTTP请求报文读进这个连接socket的读缓存中 users[sockfd].read() 然后将该任务对象（指针）插入线程池的请求队列中 pool->append(users + sockfd)

线程池的实现还需要依靠锁机制以及信号量机制来实现线程同步 保证操作的原子性

01.线程同步机制封装类

基础知识

RAII

- RAII全称是“Resource Acquisition is Initialization”，直译过来是“资源获取即初始化”.
- 在构造函数中申请分配资源，在析构函数中释放资源。因为C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数。所以，在RAII的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定
- RAII的核心思想是将资源或者状态与对象的生命周期绑定，通过C++的语言机制，实现资源和状态的安全管理,智能指针是RAII最好的例子

信号量

信号量是一种特殊的变量，它只能取自然数值并且只支持两种操作：等待(P)和信号(V).假设有信号量SV，对其的P、V操作如下：

> - P，如果SV的值大于0，则将其减一；若SV的值为0，则挂起执行
> - V，如果有其他进行因为等待SV而挂起，则唤醒；若没有，则将SV值加一

信号量的取值可以是任何自然数，最常用的，最简单的信号量是二进制信号量，只有0和1两个值.

> - sem_init函数用于初始化一个未命名的信号量
> - sem_destory函数用于销毁信号量
> - sem_wait函数将以原子操作方式将信号量减一,信号量为0时,sem_wait阻塞
> - sem_post函数以原子操作方式将信号量加一,信号量大于0时,唤醒调用sem_post的线程

以上，成功返回0，失败返回errno

互斥量

互斥锁,也成互斥量,可以保护关键代码段,以确保独占式访问.当进入关键代码段,获得互斥锁将其加锁;离开关键代码段,唤醒等待该互斥锁的线程.

> - pthread_mutex_init函数用于初始化互斥锁
> - pthread_mutex_destory函数用于销毁互斥锁
> - pthread_mutex_lock函数以原子操作方式给互斥锁加锁
> - pthread_mutex_unlock函数以原子操作方式给互斥锁解锁

以上，成功返回0，失败返回errno

条件变量

条件变量提供了一种线程间的通知机制,当某个共享数据达到某个值时,唤醒等待这个共享数据的线程.

> - pthread_cond_init函数用于初始化条件变量
> - pthread_cond_destory函数销毁条件变量
> - pthread_cond_broadcast函数以广播的方式唤醒**所有**等待目标条件变量的线程
> - pthread_cond_wait函数用于等待目标条件变量.该函数调用时需要传入 **mutex参数(加锁的互斥锁)** ,函数执行时,先把调用线程放入条件变量的请求队列,然后将互斥锁mutex解锁,当函数成功返回为0时,互斥锁会再次被锁上. **也就是说函数内部会有一次解锁和加锁操作**.

功能

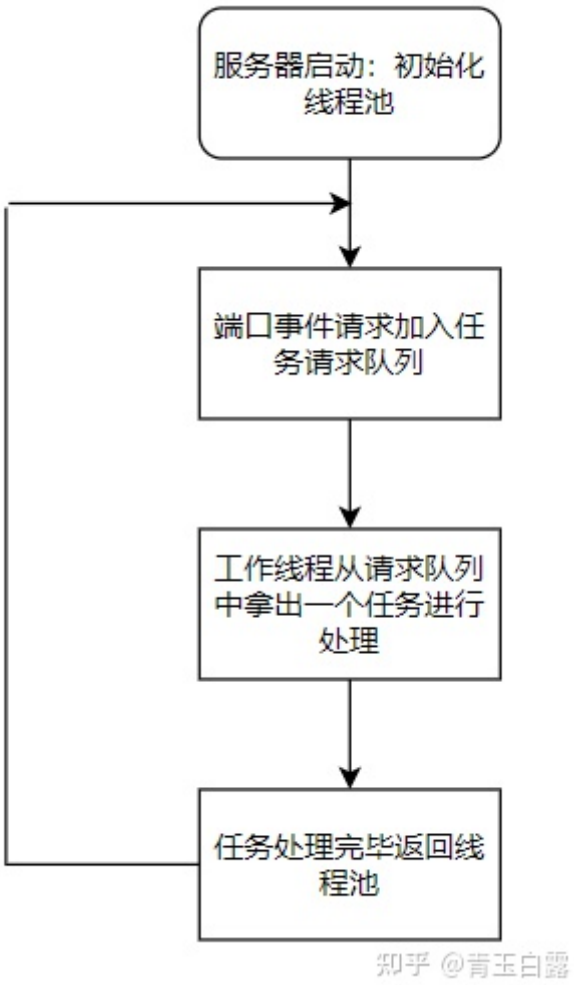
锁机制的功能

- 实现多线程同步 通过锁机制 **确保任一时刻只能有一个线程能进入关键代码段**

封装的功能

- 类中主要是Linux下三种锁进行封装，将锁的创建与销毁函数封装在类的构造与析构函数中，实现RAII机制

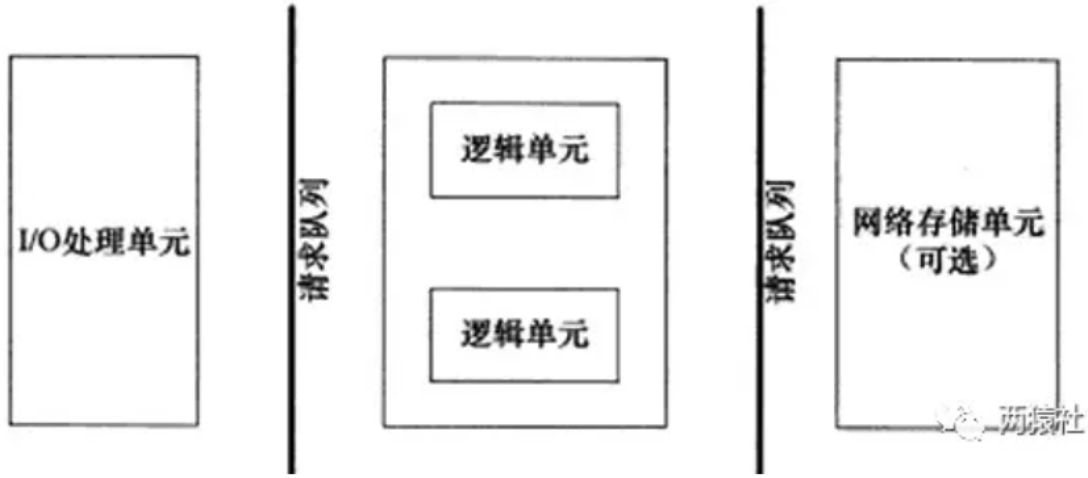
02.半同步半反应堆线程池（上）



服务器编程基本框架

主要由I/O单元，逻辑单元和网络存储单元组成，其中每个单元之间通过请求队列进行通信，从而协同完成任务。

其中I/O单元用于处理客户端连接，读写网络数据；**逻辑单元用于处理业务逻辑的线程**；**网络存储单元指本地数据库和文件等。**



五种I/O模型

- **阻塞IO**:调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的去检查这个函数有没有返回，必须等这个函数返回才能进行下一步动作
- **非阻塞IO**:非阻塞等待，每隔一段时间就去检测IO事件是否就绪。没有就绪就可以做其他事。非阻塞I/O执行系统调用总是立即返回，不管时间是否已经发生，若时间没有发生，则返回-1，此时可以根据errno区分这两种情况，对于accept，recv和send，事件未发生时，errno通常被设置成eagain
- **信号驱动IO**:linux用套接口进行信号驱动IO，安装一个信号处理函数，进程继续运行并不阻塞，当IO时间就绪，进程收到SIGIO信号。然后处理IO事件。
- **IO复用**:linux用select/poll函数实现IO复用模型，这两个函数也会使进程阻塞，但是和阻塞IO所不同的是这两个函数可以同时阻塞多个IO操作。而且可以同时对多个读操作、写操作的IO函数进行检测。知道有数据可读或可写时，才真正调用IO操作函数
- **异步IO**:linux中，可以调用aio_read函数告诉内核描述字缓冲区指针和缓冲区的大小、文件偏移及通知的方式，然后立即返回，当内核将数据拷贝到缓冲区后，再通知应用程序。

注意：阻塞I/O，非阻塞I/O，信号驱动I/O和I/O复用都是同步I/O。同步I/O指内核向应用程序通知的是就绪事件，比如只通知有客户端连接，要求用户代码自行执行I/O操作，异步I/O是指内核向应用程序通知的是完成事件，比如读取客户端的数据后才通知应用程序，由内核完成I/O操作。

事件处理模式

- reactor模式中，主线程(**I/O处理单元**)只负责监听文件描述符上是否有事件发生，有的话立即通知工作线程(**逻辑单元**)，读写数据、接受新连接及处理客户请求均在工作线程中完成。通常由**同步I/O**实现。
- proactor模式中，主线程和内核负责处理读写数据、接受新连接等I/O操作，工作线程仅负责业务逻辑，如处理客户请求。通常由**异步I/O**实现。

同步I/O模拟proactor模式

由于异步I/O并不成熟，实际中使用较少，这里将使用同步I/O模拟实现proactor模式。

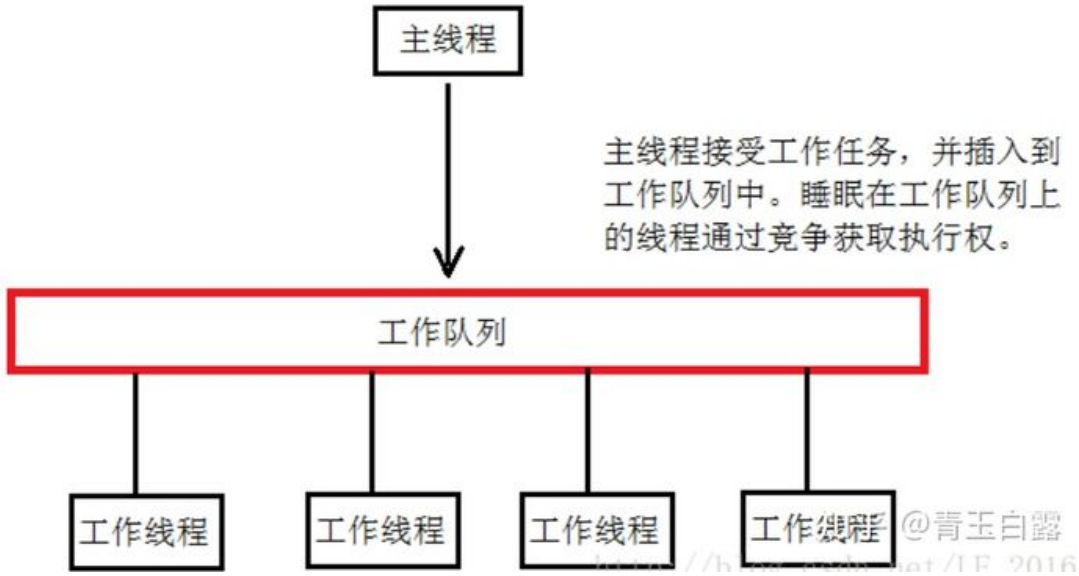
同步I/O模型的工作流程如下（epoll_wait为例）：

> - 主线程往epoll内核事件表注册socket上的读就绪事件。
> - 主线程调用epoll_wait等待socket上有数据可读
> - 当socket上有数据可读，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。
> - 睡眠在请求队列上某个**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内核事件表中注册该socket上的写就绪事件

> - 主线程调用epoll_wait等待socket可写。
> - 当socket上有数据可写，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。

以Proactor模式为例的工作流程如下：

- 主线程充当异步线程 负责监听所有socket上的事件
- 若有请求到来 主线程接受之以得到新的 连接socketconnfd 然后往内核事件表中注册该socket上的读写事件
- 如果connfd有读写事件发生 主线程从socket上接收数据 并将数据封装成请求对象插入到请求队列中
- 所有工作线程睡眠在请求队列上 当有任务来 通过竞争 如互斥锁获得任务的接管权力



并发编程模式

并发编程方法的实现有多线程和多进程两种，但这里涉及的并发模式指I/O处理单元与逻辑单元的协同完成任务的方法。

- 半同步/半异步模式
- 领导者/追随者模式

半同步/半反应堆

半同步/半反应堆并发模式是半同步/半异步的变体，将半异步具体化为某种事件处理模式.

并发模式中的同步和异步

> - 同步指的是程序完全按照代码序列的顺序执行
> - 异步指的是程序的执行需要由系统事件驱动

半同步/半异步模式工作流程

> - 同步线程用于处理客户逻辑
> - 异步线程用于处理I/O事件
> - 异步线程监听到客户请求后，就将其封装成请求对象并插入请求队列中
> - 请求队列将通知某个工作在**同步模式的工作线程**来读取并处理该请求对象

半同步/半反应堆工作流程（以Proactor模式为例）

> - 主线程充当异步线程，负责监听所有socket上的事件

> - 若有新请求到来，主线程接收之以得到新的连接socket，然后往epoll内核事件表中注册该socket上的读写事件
> - 如果连接socket上有读写事件发生，主线程从socket上接收数据，**并将数据封装成请求对象插入到请求队列中**
> - **所有工作线程睡眠在请求队列上**，当有任务到来时，通过竞争（如互斥锁）获得任务的接管权

线程池

- 空间换时间,浪费服务器的硬件资源,换取运行效率.
- 池是一组资源的集合,**这组资源在服务器启动之初就被完全创建好并初始化,这称为静态资源.**
- 当服务器进入正式运行阶段,开始处理客户请求的时候,如果它需要相关的资源,**可以直接从池中获取,无需动态分配.**
- 当服务器处理完一个客户连接后,**可以把相关的资源放回池中,无需执行系统调用释放资源.**
- 在建立线程池时 调用pthread_create指向了worker（）静态成员函数 而worker内部调用run（）
- run（）函数其实可以看做一个回环事件 一直等待m_reauestat（）信号变量post 即新任务进入请求队列 这时请求队列中取出一个任务进行处理
- 每调用一次pthread_craete就会调用一次run 因为每个线程是相互独立的 都睡眠在工作队列上 仅仅当信号量更新才会唤醒进行任务的竞争

03.半同步半反应堆线程池（下）

基础知识

静态成员变量

将类成员变量声明为static，则为静态成员变量，与一般的成员变量不同，无论建立多少对象，都只有一个静态成员变量的拷贝，静态成员变量属于一个类，所有对象共享。

静态变量在编译阶段就分配了空间，对象还没创建时就已经分配了空间，放到全局静态区。

- 静态成员变量
- - 最好是类内声明，**类外初始化**（以免类名访问静态成员访问不到）。
  - 无论公有，私有，静态成员都可以在类外定义，但私有成员仍有访问权限。
  - 非静态成员类外不能初始化。
  - 静态成员数据是共享的。

静态成员函数

将类成员函数声明为static，则为静态成员函数。

- 静态成员函数
- - 静态成员函数可以直接访问静态成员变量，不能直接访问普通成员变量，但可以通过参数传递的方式访问。
  - 普通成员函数可以访问普通成员变量，也可以访问静态成员变量。
  - 静态成员函数没有this指针。非静态数据成员为对象单独维护，**但静态成员函数为共享函数，无法区分是哪个对象，因此不能直接访问普通变量成员**，也没有this指针。

pthread_create陷阱

首先看一下该函数的函数原型。

```
#include <pthread.h>
int pthread_create (pthread_t *thread_tid,                  //返回新生成的线程的id
const pthread_attr_t *attr,          //指向线程属性的指针,通常设置为NULL
void * (*start_routine) (void *),    //处理线程函数的地址
void *arg);                          //start_routine()中的参数
```

函数原型中的第三个参数，为函数指针，指向处理线程函数的地址。该函数，要求为静态函数。如果处理线程函数为类成员函数时，需要将其设置为**静态成员函数。**

this指针的锅

pthread_create的函数原型中第三个参数的类型为函数指针，指向的线程处理函数参数类型为`(void *)`,若线程函数为类成员函数，则this指针会作为默认的参数被传进函数中，从而和线程函数参数`(void*)`不能匹配，不能通过编译。

静态成员函数就没有这个问题，里面没有this指针。

04.http连接处理（上）

epoll

epoll涉及的知识较多，这里仅对API和基础知识作介绍。更多资料请查阅资料，或查阅 游双的Linux高性能服务器编程 第9章 I/O复用

epoll_create函数

```
#include <sys/epoll.h>//创建一个指示epoll内核事件表的文件描述符
//该描述符将用作其他epoll系统调用的第一个参数
int epoll_create(int size)//size不起作用
```

创建一个指示epoll内核事件表的文件描述符，该描述符将用作其他epoll系统调用的第一个参数，size不起作用。

epoll_ctl函数

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
```

该函数用于操作内核事件表监控的文件描述符上的事件：注册、修改、删除

- epfd：为epoll_creat的句柄
- op：表示动作，用3个宏来表示：
- - EPOLL_CTL_ADD (注册新的fd到epfd)，
  - EPOLL_CTL_MOD (修改已经注册的fd的监听事件)，
  - EPOLL_CTL_DEL (从epfd删除一个fd)；
- event：告诉内核需要监听的事件

上述event是epoll_event结构体指针类型，表示内核所监听的事件，具体定义如下：

```
struct epoll_event {
__uint32_t events; /* Epoll events */
epoll_data_t data; /* User data variable */
};
```

- events描述事件类型，其中epoll事件类型有以下几种
- - EPOLLIN：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）
  - EPOLLOUT：表示对应的文件描述符可以写
  - EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）
  - EPOLLERR：表示对应的文件描述符发生错误
  - EPOLLHUP：表示对应的文件描述符被挂断；
  - EPOLLET：将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)而言的
  - EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

epoll_wait函数

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)
```

该函数用于等待所监控文件描述符上有事件的产生，返回就绪的文件描述符个数

- events：用来存内核得到事件的集合，
- maxevents：告之内核这个events有多大，这个maxevents的值不能大于创建epoll_create()时的size，
- timeout：是超时时间
- - -1：阻塞
  - 0：立即返回，非阻塞
  - >0：指定毫秒
- 返回值：成功返回有多少文件描述符就绪，时间到时返回0，出错返回-1

select/poll/epoll

- 调用函数
- - select和poll都是一个函数，epoll是一组函数
- 文件描述符数量
- - select通过线性表描述文件描述符集合，文件描述符有上限，一般是1024，但可以修改源码，重新编译内核，不推荐
  - poll是链表描述，突破了文件描述符上限，最大可以打开文件的数目
  - epoll通过红黑树描述，最大可以打开文件的数目，可以通过命令ulimit -n number修改，仅对当前终端有效
- 将文件描述符从用户传给内核
- - select和poll通过将所有文件描述符拷贝到内核态，每次调用都需要拷贝
  - epoll的文件描述符直接维护在内核 省去了内核到用户态的拷贝 但是每次创建描述符都需要一次系统调用 epoll通过epoll_create建立一棵红黑树，通过epoll_ctl将要监听的文件描述符注册到红黑树上
- 内核判断就绪的文件描述符
- - select和poll通过遍历文件描述符集合，判断哪个文件描述符上有事件发生 很慢

    - epoll_create时，内核除了帮我们在epoll文件系统里建了个红黑树用于存储以后epoll_ctl传来的fd外，还会再建立一个list链表，用于存储准备就绪的事件，当epoll_wait调用时，仅仅观察这个list链表里有没有数据即可。
    - epoll是根据每个fd上面的回调函数(中断函数)判断，只有发生了事件的socket才会主动的去调用 callback函数，其他空闲状态socket则不会，若是就绪事件，插入list
- 应用程序索引就绪文件描述符
-
    - select/poll只返回发生了事件的文件描述符的个数，若知道是哪个发生了事件，同样需要遍历
    - epoll返回的发生了事件的个数和结构体数组，结构体包含socket的信息，因此直接处理返回的数组即可
- 工作模式
-
    - select和poll都只能工作在相对低效的LT模式下
    - epoll则可以工作在ET高效模式，**并且epoll还支持EPOLLONESHOT事件，该事件能进一步减少可读、可写和异常事件被触发的次数。**
- 应用场景
-
    - 当所有的fd都是活跃连接，使用epoll，需要建立文件系统，红黑书和链表对于此来说，效率反而不高，不如selece和poll
    - 当监测的fd数目较小，且各个fd都比较活跃，建议使用select或者poll
    - **当监测的fd数目非常大，成千上万，且单位时间只有其中的一部分fd处于就绪状态，这个时候使用epoll能够明显提升性能**

ET、LT、EPOLLONESHOT

- LT水平触发模式
-
    - **epoll_wait检测到文件描述符有事件发生**，则将其通知给应用程序，应用程序可以不立即处理该事件。
    - 当下一次调用epoll_wait时，epoll_wait还会再次向应用程序报告此事件，直至被处理
- ET边缘触发模式
-
    - epoll_wait检测到文件描述符有事件发生，则将其通知给应用程序，应用程序必须立即处理该事件
    - 必须要一次性将数据读取完，因此使用非阻塞I/O，读取到出现eagain
- EPOLLONESHOT
-
    - 一个线程读取某个socket上的数据后开始处理数据，在处理过程中该socket上又有新数据可读，此时另一个线程被唤醒读取，**此时出现两个线程处理同一个socket**
    - 我们期望的是一个socket连接在任一时刻都只被一个线程处理，通过epoll_ctl对该文件描述符注册epolloneshot事件，**一个线程处理socket时，其他线程将无法处理，当该线程处理完后，需要通过epoll_ctl重置epolloneshot事件**

HTTP报文格式

HTTP报文分为请求报文和响应报文两种，每种报文必须按照特有格式生成，才能被浏览器端识别。

其中，浏览器端向服务器发送的为请求报文，服务器处理后返回给浏览器端的为响应报文。

请求报文

HTTP请求报文由**请求行（request line）、请求头部（header）、空行和请求数据四个部分组成。**

其中，请求分为两种，**GET和POST**，具体的：

- **GET**

```
1    GET /562f25980001b1b106000338.jpg HTTP/1.1
2    Host:img.mukewang.com
3    User-Agent:Mozilla/5.0 (Windows NT 10.0; WOW64)
4    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.106 Safari/537.36
5    Accept:image/webp,image/*,*/*;q=0.8
6    Referer:http://www.imooc.com/
7    Accept-Encoding:gzip, deflate, sdch
8    Accept-Language:zh-CN,zh;q=0.8
9    空行
10   请求数据为空
```

- **POST**

```
1    POST / HTTP1.1
2    Host:www.wrox.com
3    User-Agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
4    Content-Type:application/x-www-form-urlencoded
5    Content-Length:40
6    Connection: Keep-Alive
7    空行
8    name=Professional%20Ajax&publisher=Wiley
```

> - **请求行**，用来说明请求类型,要访问的资源以及所使用的HTTP版本。
> GET说明请求类型为GET，/562f25980001b1b106000338.jpg(URL)为要访问的资源，该行的最后一部分说明使用的是HTTP1.1版本。
> - **请求头部**，紧接着请求行（即第一行）之后的部分，用来说明服务器要使用的附加信息。
> -
> - HOST，给出请求资源所在服务器的域名。
> - User-Agent，HTTP客户端程序的信息，该信息由你发出请求使用的浏览器来定义,并且在每个请求中自动发送等。
> - Accept，说明用户代理可处理的媒体类型。
> - Accept-Encoding，说明用户代理支持的内容编码。
> - Accept-Language，说明用户代理能够处理的自然语言集。
> - Content-Type，说明实现主体的媒体类型。
> - Content-Length，说明实现主体的大小。
> - Connection，连接管理，可以是Keep-Alive或close。
> - **空行**，请求头部后面的空行是必须的即使第四部分的请求数据为空，也必须有空行。
> - **请求数据**也叫主体，可以添加任意的其他数据。

响应报文

HTTP响应也由四个部分组成，分别是：**状态行、消息报头、空行和响应正文。**

```
HTTP/1.1 200 OK
Date: Fri, 22 May 2009 06:07:21 GMT
Content-Type: text/html; charset=UTF-8
空行
<html>
      <head></head>
      <body>
            <!--body goes here-->
      </body>
</html>
```

> - 状态行，由HTTP协议版本号， 状态码， 状态消息 三部分组成。
>   第一行为状态行，（HTTP/1.1）表明HTTP版本为1.1版本，状态码为200，状态消息为OK。
> - 消息报头，用来说明客户端要使用的一些附加信息。
>   第二行和第三行为消息报头，Date:生成响应的日期和时间；Content-Type:指定了MIME类型的HTML(text/html),编码类型是UTF-8。
> - 空行，消息报头后面的空行是必须的。
> - 响应正文，服务器返回给客户端的文本信息。空行后面的html部分为响应正文。

HTTP状态码

HTTP有5种类型的状态码，具体的：

- 1xx：指示信息--表示请求已接收，继续处理。
- 2xx：成功--表示请求正常处理完毕。
  - 200 OK：客户端请求被正常处理。
  - 206 Partial content：客户端进行了范围请求。
- 3xx：重定向--要完成请求必须进行更进一步的操作。
  - 301 Moved Permanently：永久重定向，该资源已被永久移动到新位置，将来任何对该资源的访问都要使用本响应返回的若干个URI之一。
  - 302 Found：临时重定向，请求的资源现在临时从不同的URI中获得。
- 4xx：客户端错误--请求有语法错误，服务器无法处理请求。
  - 400 Bad Request：请求报文存在语法错误。
  - 403 Forbidden：请求被服务器拒绝。
  - 404 Not Found：请求不存在，服务器上找不到请求的资源。
- 5xx：服务器端错误--服务器处理请求出错。
  - 500 Internal Server Error：服务器在执行请求时出现错误。

有限状态机

有限状态机，是一种抽象的理论模型，它能够把有限个变量描述的状态变化过程，以可构造可验证的方式呈现出来。比如，封闭的有向图。

有限状态机可以通过if-else,switch-case和函数指针来实现，从软件工程的角度看，**主要是为了封装逻辑。**

带有状态转移的有限状态机示例代码。

```
STATE_MACHINE(){
    State cur_State = type_A;
    while(cur_State != type_C){
        Package _pack = getNewPackage();
        switch(){
```

```
 6            case type_A:
 7                process_pkg_state_A(_pack);
 8                cur_State = type_B;
 9                break;
10            case type_B:
11                process_pkg_state_B(_pack);
12                cur_State = type_C;
13                break;
14        }
15    }
16}
```

该状态机包含三种状态：type_A，type_B和type_C。其中，type_A是初始状态，type_C是结束状态。

状态机的当前状态记录在cur_State变量中，逻辑处理时，状态机先通过getNewPackage获取数据包，然后根据当前状态对数据进行处理，处理完后，状态机通过改变cur_State完成状态转移。

有限状态机一种逻辑单元内部的一种高效编程方法，在服务器编程中，服务器可以根据不同状态或者消息类型进行相应的处理逻辑，使得程序逻辑清晰易懂。

http处理流程

首先对http报文处理的流程进行简要介绍，然后具体介绍http类的定义和服务器接收http请求的具体过程。

http报文处理流程

- 浏览器端发出http连接请求，主线程创建**http对象**接收请求并将所有数据**读入对应buffer**，将该对象插入任务队列，工作线程从任务队列中取出一个任务进行处理。(**本篇讲**)——**连接处理**
- 工作线程取出任务后，调用process_read函数，通过**主、从状态机**对请求报文进行解析。(**中篇讲**)——**处理报文请求**
- 解析完之后，**跳转do_request函数生成响应报文**，通过process_write**写入buffer**，返回给浏览器端。(**下篇讲**)——**返回响应报文**

epoll相关代码

项目中epoll相关代码部分包括**非阻塞模式**、**内核事件表注册事件**、**删除事件**、**重置EPOLLONESHOT**事件四种。

连接处理

在连接阶段，最重要的是**tcp连接过程和读取http的请求报文**（其实读取请求报文就是读取客户端发送的数据而已）。tcp连接过程涉及epoll内核事件创建等，详见**后续的epoll部分。**

服务器是如何实现读取http的报文的呢?

首先 服务器需要**对每一个已建立连接http**建立一个**http的类对象** 服务器一直在运行evenloop即回旋事件 因为整个服务器其实是**事件驱动**

22行的dealclientdata（） 调用timer（）**创建新的client客户user 同时新增一个定时事件**

完成这一系列步骤后 服务器中就维护着一系列的客户端client连接 当其中一个客户点击网页某一按钮 生成一个请求报文并传输到服务器时 在上述事件回环代码中调用给dealwithread() 该函数中将该端口事件**append加入任务请求队列 等待线程池中的线程执行该任务** 根据reactor/proactor模式 工作线程堆http请求报文数据的读取由read_once函数完成 read_once()函数将浏览器(客户端)的数据读入到缓存数组以待后续工作线程进行处理

05.http连接处理（中）

本文内容

在webserver的线程池有空闲线程时 某一线程调用**process（）**来完成请求报文的解析及报文相应任务

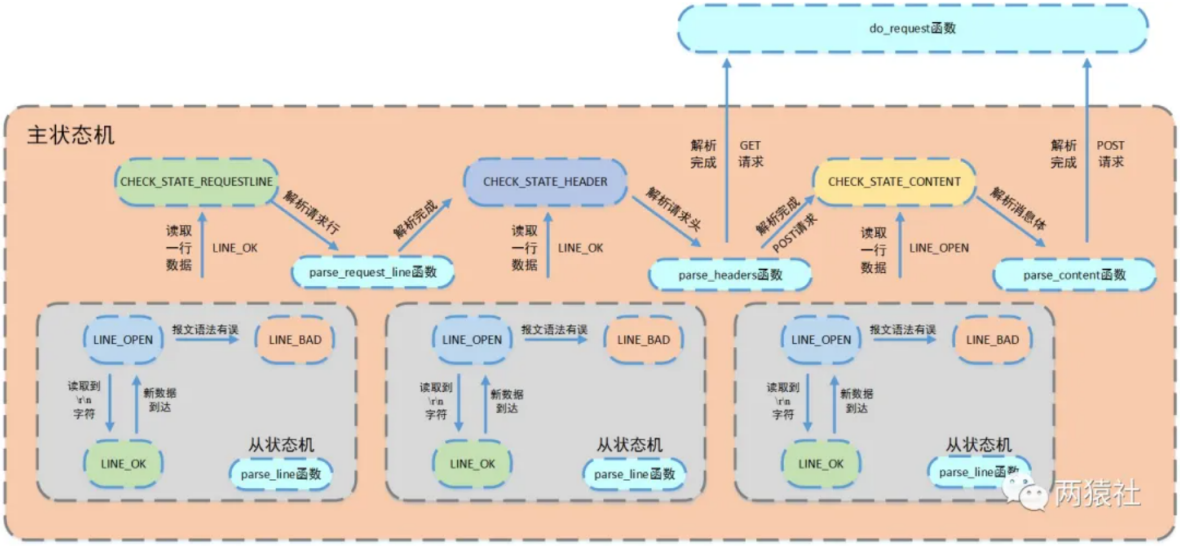
上篇，我们对http连接的基础知识、服务器接收请求的处理流程进行了介绍，本篇将结合流程图和代码分别对**状态机和服务器解析请求报文**进行详解。

流程图部分，描述主、从状态机调用关系与状态转移过程。

代码部分，结合代码对http请求报文的解析进行详解。

流程图与状态机

从状态机负责读取报文的一行，主状态机负责对该行数据进行解析，主状态机内部调用从状态机，从状态机驱动主状态机。



主状态机

三种状态，标识解析位置。

- CHECK_STATE_REQUESTLINE，解析请求行
- CHECK_STATE_HEADER，解析请求头
- CHECK_STATE_CONTENT，解析消息体，**仅用于解析POST请求**

从状态机

三种状态，标识解析一行的读取状态。

- LINE_OK，**完整读取一行**
- LINE_BAD，**报文语法有误**
- LINE_OPEN，**读取的行不完整**

HTTP_CODE含义

表示HTTP请求的处理结果，在头文件中初始化了八种情形，在报文解析时只涉及到四种。

- NO_REQUEST
- - **请求不完整，需要继续读取请求报文数据**
- GET_REQUEST
- - **获得了完整的HTTP请求**
- BAD_REQUEST

- **HTTP请求报文有语法错误**
- INTERNAL_ERROR
- **服务器内部错误**，该结果在主状态机逻辑switch的default下，一般不会触发

解析报文整体流程

process_read通过**while循环，将主从状态机进行封装，对报文的每一行进行循环处理。**这里的**主状态机**，指的是**process_read()**函数，**从状态机**是指**parse_line()**函数。主状态机是使用switch...case来实现主状态机的选择 而主状态机状态是由CHECK_STATE_REQUESTLINE/HEADER/CONTENT，这三个标志来表示的： **正在解析请求行 解析请求头 解析消息体**

- 判断条件
- 主状态机转移到CHECK_STATE_CONTENT，该条件涉及解析消息体
  - 从状态机转移到LINE_OK，该条件涉及解析请求行和请求头部
  - 两者为或关系，当条件为真则继续循环，否则退出
- 循环体
- 从状态机读取数据
  - 调用**get_line****函数**，通过m_start_line将从状态机读取数据间接赋给text
  - 主状态机解析**text**

主状态机初始状态是**CHECK_STATE_REQUESTLINE**，而后调用parse_request_line()解析请求行，获得HTTP的请求方法、目标URL以及HTTP版本号，**状态变为CHECK_STATE_HEADER。**

此时进入循环体之后，调用parse_headers()解析请求头部信息。先要判断是空行还是请求头，**空行进一步区分POST还是GET。**若是请求头，则更新长短连接状态、host等等。

注：GET和POST请求报文的区别之一是有无消息体部分。

当使用POST请求时，**需要进行CHECK_STATE_CONTENT的解析，取出POST消息体中的信息（用户名、密码）。**

从状态机逻辑

上一篇的基础知识讲解中，对于HTTP报文的讲解遗漏了一点细节，在这里作为补充。

在HTTP报文中，每一行的数据由**\r\n**作为结束字符 可以跳到下一行的行首，空行则是仅仅是字符\r\n。**因此，可以通过查找\r\n将报文拆解成单独的行进行解析，项目中便是利用了这一点。**

从状态机负责读取buffer中的数据，将每行数据末尾的\r\n置为\0\0，并更新从状态机在buffer中读取的位置m_checked_idx，以此来驱动主状态机解析。

- 从状态机从m_read_buf中逐字节读取，判断当前字节是否为\r
- **接下来的字符是\n，将\r\n修改成\0\0，将m_checked_idx指向下一行的开头，则返回LINE_OK**
  - **接下来达到了buffer末尾，表示buffer还需要继续接收，返回LINE_OPEN**
  - **否则，表示语法错误，返回LINE_BAD**
- 当前字节不是\r，判断是否是\n（**一般是上次读取到\r就到了buffer末尾，没有接收完整，再次接收时会出现这种情况**）
- 如果前一个字符是\r，则将\r\n修改成\0\0，将m_checked_idx指向下一行的开头，则返回LINE_OK
- 当前字节既不是\r，也不是\n
- **表示接收不完整，需要继续接收，返回LINE_OPEN**

主状态机逻辑

主状态机初始状态是CHECK_STATE_REQUESTLINE 解析请求行，通过调用从状态机来驱动主状态机，在主状态机进行解析前，**从状态机已经将每一行的末尾\r\n符号改为\0\0，以便于主状态机直接取出对应字符串进行处理。**

- CHECK_STATE_REQUESTLINE
- - 主状态机的初始状态，**调用parse_request_line函数解析请求行**
  - 解析函数从m_read_buf中解析HTTP请求行，**获得请求方法、目标URL及HTTP版本号**
  - 解析完成后主状态机的状态变为**CHECK_STATE_HEADER**

解析完请求行后，**主状态机继续分析请求头。**在报文中，请求头和空行的处理使用的同一个函数，这里**通过判断当前的text首位是不是\0字符，若是，则表示当前处理的是空行，若不是，则表示当前处理的是请求头。**

- CHECK_STATE_HEADER
- - 调用parse_headers函数解析请求头部信息
  - 判断是空行还是请求头，若是空行，进而判断content-length是否为0，**如果不是0，表明是POST请求，则状态转移到CHECK_STATE_CONTENT，否则说明是GET请求，则报文解析结束。**
  - 若解析的是请求头部字段，则主要分析connection字段，**content-length**字段，其他字段可以直接跳过，各位也可以根据需求继续分析。
  - **connection字段判断是keep-alive还是close，决定是长连接还是短连接**
  - **content-length字段，这里用于读取post请求的消息体长度**

如果仅仅是GET请求，如项目中的欢迎界面，那么主状态机只设置之前的两个状态足矣。

因为在上篇推文中我们曾说道，GET和POST请求报文的区别之一是有无消息体部分，GET请求没有消息体，当解析完空行之后，便完成了报文的解析。

但后续的登录和注册功能，**为了避免将用户名和密码直接暴露在URL中**，我们在项目中改用了POST请求，**将用户名和密码添加在报文中作为消息体进行了封装。**

为此，我们需要在解析报文的部分添加解析消息体的模块。

为此，我们需要在解析报文的部分添加解析消息体的模块。

```
1while((m_check_state==CHECK_STATE_CONTENT && line_status==LINE_OK)||
((line_status=parse_line())==LINE_OK))
那么，这里的判断条件为什么要写成这样呢?
```

在GET请求报文中，每一行都是\r\n作为结束，所以对报文进行拆解时，仅用从状态机的状态line_status=parse_line())==LINE_OK语句即可。

但，在POST请求报文中，消息体的末尾没有任何字符，所以不能使用从状态机的状态，这里转而使用主状态机的状态作为循环入口条件。

```
那后面的&& line_status==LINE_OK又是为什么?
```

解析完消息体后，报文的完整解析就完成了，但此时主状态机的状态还是CHECK_STATE_CONTENT，也就是说，符合循环入口条件，还会再次进入循环，这并不是我们所希望的。

为此，增加了该语句，并在完成消息体解析后，将line_status变量更改为LINE_OPEN，此时可以跳出循环，完成报文解析任务。

- CHECK_STATE_CONTENT

- 仅用于解析POST请求，调用parse_content函数解析消息体
- **用于保存post请求消息体，为后面的登录和注册做准备**

06.http连接处理（下）

本文内容

上一篇详解中，我们对状态机和服务器解析请求报文进行了介绍。

本篇，我们将介绍服务器如何响应请求报文，并将该报文发送给浏览器端。首先介绍一些基础API，然后结合流程图和代码对服务器响应请求报文进行详解。

基础API部分，介绍`stat`、`mmap`、`iovec`、`writev`。

流程图部分，描述服务器端响应请求报文的逻辑，各模块间的关系。

代码部分，结合代码对服务器响应请求报文进行详解。

基础API

为了更好的源码阅读体验，这里提前对代码中使用的一些API进行简要介绍，更丰富的用法可以自行查阅资料。

stat

stat函数用于**取得指定文件的文件属性**，并将文件属性存储在结构体stat里，这里仅对其中用到的成员进行介绍。

```
 1#include <sys/types.h>
 2#include <sys/stat.h>
 3#include <unistd.h>
 4
 5//获取文件属性，存储在statbuf中
 6int stat(const char *pathname, struct stat *statbuf);
 7
 8struct stat
 9{
10   mode_t    st_mode;        /* 文件类型和权限 */
11   off_t     st_size;        /* 文件大小，字节数*/
12};
```

mmap

用于将一个文件或其他对象映射到内存，提高文件的访问速度。

```
1void* mmap(void* start,size_t length,int prot,int flags,int fd,off_t offset);
2int munmap(void* start,size_t length);
```

- start：映射区的开始地址，设置为0时表示由系统决定映射区的起始地址
- length：映射区的长度
- prot：期望的内存保护标志，不能与文件的打开模式冲突
- PROT_READ 表示页内容可以被读取
- flags：指定映射对象的类型，映射选项和映射页是否可以共享

- ◦ MAP_PRIVATE 建立一个写入时拷贝的私有映射，内存区域的写入不会影响到原文件
- fd：有效的文件描述符，一般是由open()函数返回
- off_toffset：被映射对象内容的起点

iovec

定义了一个向量元素，通常，这个结构用作一个多元素的数组。

```
1struct iovec {
2    void      *iov_base;      /* starting address of buffer */
3    size_t    iov_len;        /* size of buffer */
4};
```

- iov_base指向数据的地址
- iov_len表示数据的长度

writev

writev函数用于在一次函数调用中写多个非连续缓冲区，有时也将这该函数称为聚集写。

```
1#include <sys/uio.h>
2ssize_t writev(int filedes, const struct iovec *iov, int iovcnt);
```

- filedes表示文件描述符
- iov为前述io向量机制结构体iovec
- iovcnt为结构体的个数

若成功则返回已写的字节数，若出错则返回-1。`writev`**以顺序**`iov[0]`**，**`iov[1]`**至**`iov[iovcnt-1]`**从缓冲区中聚集输出数据。**`writev`**返回输出的字节总数，**通常，它应等于所有缓冲区长度之和。**

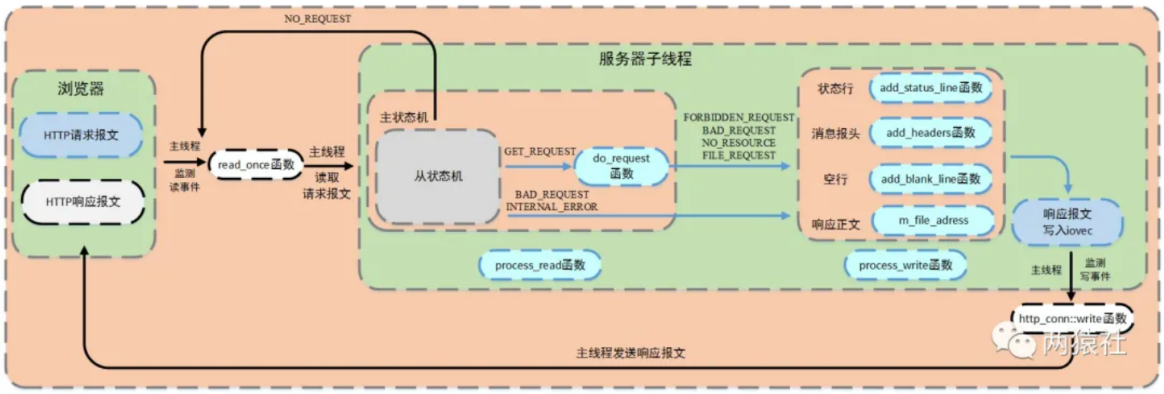
特别注意： 循环调用writev时，需要重新处理iovec中的指针和长度，该函数不会对这两个成员做任何处理。writev的返回值为已写的字节数，但这个返回值"实用性"并不高，因为参数传入的是iovec数组，计量单位是iovcnt，而不是字节数，我们仍然需要通过遍历iovec来计算新的基址，另外写入数据的"结束点"可能位于一个iovec的中间某个位置，因此需要调整临界iovec的io_base和io_len。

流程图

在完成请求报文的解析之后 明确用户想要登陆/注册 需要跳转到对应的界面 添加用户名 验证用户等 并将相应的数据写入相应报文 返回给浏览器

浏览器端发出HTTP请求报文，服务器端接收该报文并调用`process_read`对其进行解析，**根据解析结果**`HTTP_CODE`**，进入相应的逻辑和模块。**

其中，**服务器子线程完成报文的解析与响应；主线程监测读写事件**，调用`read_once`和`http_conn::write`完成数据的读取与发送。



这个在process_read()中完成请求报文的解析之后，状态机会调用do_request()函数，该函数是处理功能逻辑的。**该函数将网站根目录和url文件拼接**，然后通过stat判断该文件属性。url，可以将其抽象成ip:port/xxx，xxx通过html文件的action属性（即请求报文）进行设置。**m_url为请求报文中解析出的请求资源，以/开头，也就是x，项目中解析后的m_url有8种情况，见do_request()函数**

HTTP_CODE含义

表示HTTP请求的处理结果，在头文件中初始化了八种情形，在报文解析与响应中只用到了七种。

- NO_REQUEST
- - 请求不完整，需要继续读取请求报文数据
  - **跳转主线程继续监测读事件**
- GET_REQUEST
- - 获得了完整的HTTP请求
  - **调用do_request完成请求资源映射**
- NO_RESOURCE
- - 请求资源不存在
  - **跳转process_write完成响应报文**
- BAD_REQUEST
- - HTTP请求报文有语法错误或请求资源为目录
  - **跳转process_write完成响应报文**
- FORBIDDEN_REQUEST
- - 请求资源禁止访问，没有读取权限
  - **跳转process_write完成响应报文**
- FILE_REQUEST
- - 请求资源可以正常访问
  - **跳转process_write完成响应报文**
- INTERNAL_ERROR
- - 服务器内部错误，该结果在主状态机逻辑switch的default下，一般不会触发

执行do_request函数之后 子线程调用process_write进行响应报文 （**add_status_line add_headers等函数**）的生成 在生成响应报文的过程中主要调用**add_response（）函数更新m_write_idx和m_write_buf**

值得注意的是 响应报文分为两种 **一种是请求文件的存在** 通过io向量机制iovec 声明两个iovec 第一个指向**m_write_buf 第二个指向mmap的地址m_file_addresss** 另一种是请求出错 这时候只申请一个iovec指向m_write_buf

另外用户登录注册的验证逻辑代码在**do_request**中 **通过对mysql数据库进行查询或者插入 验证 添加用户**

代码分析

do_request

`process_read`函数的返回值是对请求的文件分析后的结果，一部分是语法错误导致的`BAD_REQUEST`，一部分是`do_request`的返回结果.**该函数将网站根目录和`url`文件拼接，然后通过stat判断该文件属性。**另外，为了提高访问速度，**通过mmap进行映射，将普通文件映射到内存逻辑地址。**

为了更好的理解请求资源的访问流程，这里对各种各页面跳转机制进行简要介绍。其中，浏览器网址栏中的字符，即`url`，可以将其抽象成`ip:port/xxx`，`xxx`通过`html`文件的`action`属性进行设置。

m_url为请求报文中解析出的请求资源，以/开头，也就是`/xxx`，项目中解析后的m_url有8种情况。

- /
    - **GET请求，跳转到judge.html，即欢迎访问页面**
- /0
    - **POST请求，跳转到register.html，即注册页面**
- /1
    - POST请求，跳转到log.html，即登录页面
- /2CGISQL.cgi
    - POST请求，进行登录校验
    - 验证成功跳转到welcome.html，即资源请求成功页面
    - 验证失败跳转到logError.html，即登录失败页面
- /3CGISQL.cgi
    - POST请求，进行注册校验
    - 注册成功跳转到log.html，即登录页面
    - 注册失败跳转到registerError.html，即注册失败页面
- /5
    - POST请求，跳转到picture.html，即图片请求页面
- /6
    - POST请求，跳转到video.html，即视频请求页面
- /7
    - POST请求，跳转到fans.html，即关注页面

如果大家对上述设置方式不理解，不用担心。具体的登录和注册校验功能会在第12节进行详解，到时候还会针对html进行介绍。

process_write

根据`do_request`的返回状态，服务器子线程调用`process_write`向`m_write_buf`中写入响应报文。

- **add_status_line函数，添加状态行**：http/1.1 状态码 状态消息
- add_headers函数添加消息报头，内部调用add_content_length和add_linger函数
    - **content-length记录响应报文长度，用于浏览器端判断服务器是否发送完数据**
    - **connection记录连接状态，用于告诉浏览器端保持长连接**
- add_blank_line添加空行

上述涉及的5个函数，**均是内部调用`add_response`函数更新`m_write_idx`指针和缓冲区`m_write_buf`中的内容。**

process_write

根据`do_request`的返回状态，服务器子线程调用`process_write`向`m_write_buf`中写入响应报文。

- add_status_line函数，添加状态行：http/1.1 状态码 状态消息
- add_headers函数添加消息报头，内部调用add_content_length和add_linger函数
- - content-length记录响应报文长度，用于浏览器端判断服务器是否发送完数据
  - connection记录连接状态，用于告诉浏览器端保持长连接
- add_blank_line添加空行

上述涉及的5个函数，均是内部调用`add_response`函数更新`m_write_idx`指针和缓冲区`m_write_buf`中的内容。

http_conn::write

服务器子线程调用`process_write`完成响应报文，**随后注册`epollout`事件。服务器主线程检测写事件，并调用`http_conn::write`函数将响应报文发送给浏览器端。**

该函数具体逻辑如下：

在生成响应报文时初始化byte_to_send，包括头部信息和文件数据大小。通过writev函数循环发送响应报文数据，根据返回值更新byte_have_send和iovec结构体的指针和长度，并判断响应报文整体是否发送成功。

- 若writev单次发送成功，更新**byte_to_send**和**byte_have_send**的大小，若响应报文整体发送成功,则取消mmap映射,并判断是否是长连接.
- - **长连接重置http类实例，注册读事件，不关闭连接，**
  - **短连接直接关闭连接**
- 若writev单次发送不成功，判断是否是写缓冲区满了。
- - **若不是因为缓冲区满了而失败，取消mmap映射，关闭连接**
  - **若eagain则满了，更新iovec结构体的指针和长度，并注册写事件，等待下一次写事件触发（当写缓冲区从不可写变为可写，触发epollout），因此在此期间无法立即接收到同一用户的下一请求，但可以保证连接的完整性。**

07.定时器处理非活动连接（上）

基础知识

`非活跃`，是指客户端（这里是浏览器）与服务器端建立连接后，长时间不交换数据，**一直占用服务器端的文件描述符，导致连接资源的浪费。**

`定时事件`，**是指固定一段时间之后触发某段代码，由该段代码处理一个事件，如从内核事件表删除事件，并关闭文件描述符，释放连接资源。**

`定时器`，是指**利用结构体或其他形式，将多种定时事件进行封装起来。**具体的，**这里只涉及一种定时事件，即定期检测非活跃连接，这里将该定时事件与连接资源封装为一个结构体定时器。**

`定时器容器`，是指使用某种容器类数据结构，将上述多个定时器组合起来，便于对定时事件统一管理。具体的，**项目中使用升序链表将所有定时器串联组织起来。**

除了处理非活跃的连接之外 **服务器还有一些定时事件 比如关闭文件描述符等**

整体概述

本项目中，服务器主循环为每一个连接创建一个定时器，并对每个连接进行定时。另外，利用升序时间链表容器将所有定时器串联起来，**若主循环接收到定时通知，则在链表中依次执行定时任务。**

`Linux`下提供了三种定时的方法:

- socket选项SO_RECVTIMEO和SO_SNDTIMEO
- **SIGALRM信号**
- I/O复用系统调用的超时参数

三种方法没有一劳永逸的应用场景，也没有绝对的优劣。由于项目中使用的是`SIGALRM`信号，这里仅对其进行介绍，另外两种方法可以查阅游双的`Linux高性能服务器编程 第11章 定时器`。

具体的，利用`alarm`函数周期性地触发`SIGALRM`信号，信号处理函数利用管道通知主循环，主循环接收到该信号后对升序链表上所有定时器进行处理，若该段时间内没有交换数据，则将该连接关闭，释放所占用的资源。

从上面的简要描述中，可以看出定时器处理非活动连接模块，主要分为两部分，其一为**定时方法与信号通知流程**，其二为**定时器及其容器设计与定时任务的处理。**

本篇将介绍定时方法与信号通知流程，具体的涉及到基础API、信号通知流程和代码实现。

基础API，描述`sigaction`结构体、`sigaction`函数、`sigfillset`函数、`SIGALRM`信号、`SIGTERM`信号、`alarm`函数、`socketpair`函数、`send`函数。

信号通知流程，介绍统一事件源和信号处理机制。

代码实现，**结合代码对信号处理函数的设计与使用进行详解。**

基础API

为了更好的源码阅读体验，这里提前对代码中使用的一些API进行简要介绍，更丰富的用法可以自行查阅资料。

sigaction结构体

```
1struct sigaction {
2    void (*sa_handler)(int);
3    void (*sa_sigaction)(int, siginfo_t *, void *);
4    sigset_t sa_mask;
5    int sa_flags;
6    void (*sa_restorer)(void);
7}
```

- sa_handler是一个函数指针，指向信号处理函数
- sa_sigaction同样是信号处理函数，有三个参数，可以获得关于信号更详细的信息
- sa_mask用来指定在信号处理函数执行期间需要被屏蔽的信号
- sa_flags用于指定信号处理的行为
- - SA_RESTART，使被信号打断的系统调用自动重新发起
  - SA_NOCLDSTOP，使父进程在它的子进程暂停或继续运行时不会收到 SIGCHLD 信号
  - SA_NOCLDWAIT，使父进程在它的子进程退出时不会收到 SIGCHLD 信号，这时子进程如果退出也不会成为僵尸进程
  - SA_NODEFER，使对信号的屏蔽无效，即在信号处理函数执行期间仍能发出这个信号

    - SA_RESETHAND，信号处理之后重新设置为默认的处理方式
    - SA_SIGINFO，使用 sa_sigaction 成员而不是 sa_handler 作为信号处理函数
- sa_restorer一般不使用

sigaction函数

```
1#include <signal.h>
2
3int sigaction(int signum, const struct sigaction *act, struct sigaction
*oldact);
```

- signum表示操作的信号。
- act表示对信号设置新的处理方式。
- oldact表示信号原来的处理方式。
- 返回值，0 表示成功，-1 表示有错误发生。

sigfillset函数

```
1#include <signal.h>
2
3int sigfillset(sigset_t *set);
```

用来将参数set信号集初始化，然后把所有的信号加入到此信号集里。

SIGALRM、SIGTERM信号

```
1#define SIGALRM  14     //由alarm系统调用产生timer时钟信号
2#define SIGTERM  15     //终端发送的终止信号
```

alarm函数

```
1#include <unistd.h>;
2
3unsigned int alarm(unsigned int seconds);
```

设置信号传送闹钟，即用来设置信号SIGALRM在经过参数seconds秒数后发送给目前的进程。如果未设置信号SIGALRM的处理函数，那么alarm()默认处理终止进程.

socketpair函数

在linux下，使用socketpair函数能够**创建一对套接字进行通信，项目中使用管道通信。**

```
1#include <sys/types.h>
2#include <sys/socket.h>
3
4int socketpair(int domain, int type, int protocol, int sv[2]);
```

- domain表示协议族，PF_UNIX或者AF_UNIX
- type表示协议，可以是SOCK_STREAM或者SOCK_DGRAM，SOCK_STREAM基于TCP，SOCK_DGRAM基于UDP

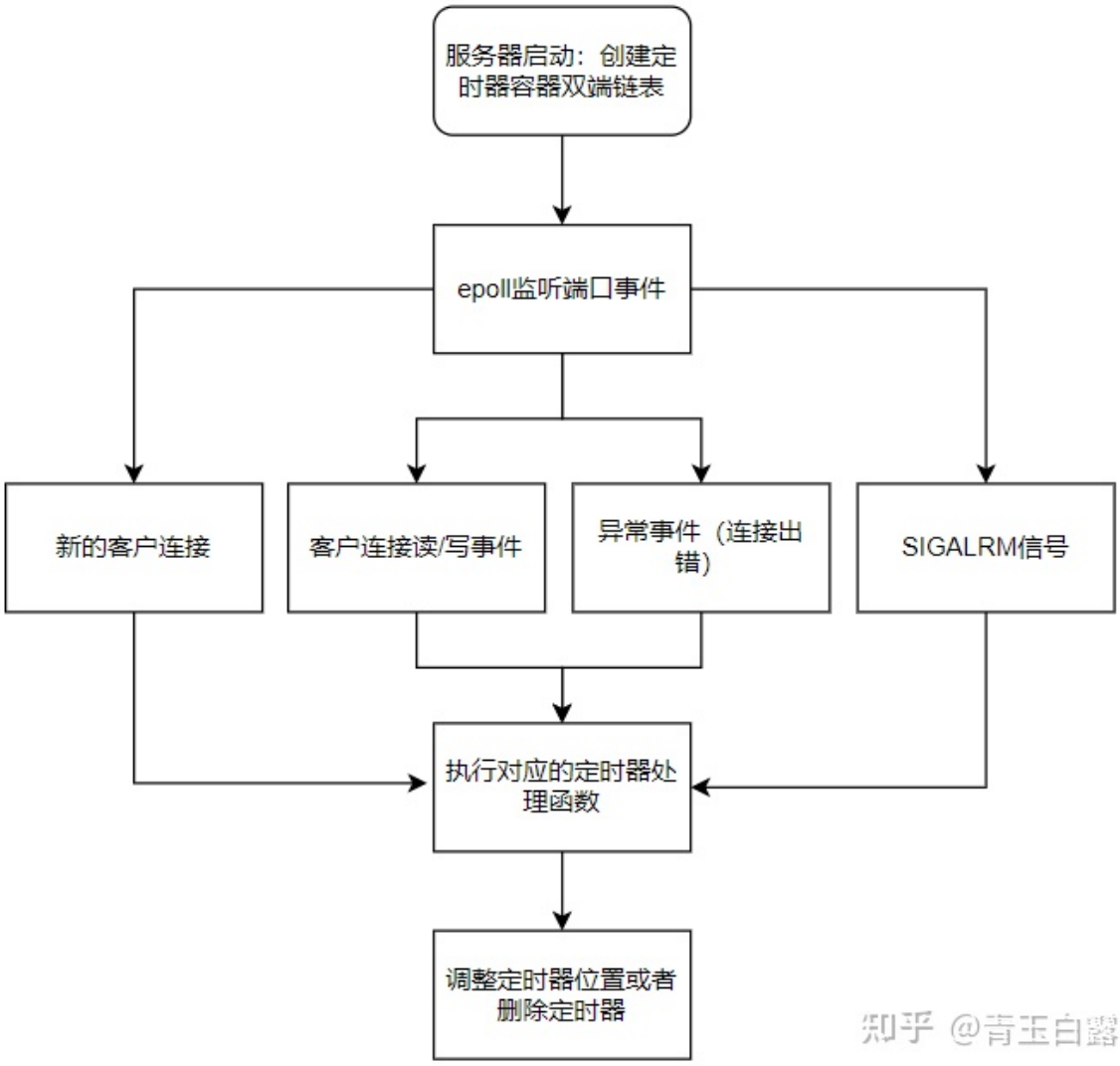
- protocol表示类型，只能为0
- sv[2]表示套节字柄对，该两个句柄作用相同，均能进行读写双向操作
- 返回结果， 0为创建成功，-1为创建失败

send函数

```
1#include <sys/types.h>
2#include <sys/socket.h>
3
4ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

当套接字发送缓冲区变满时，**send通常会阻塞，除非套接字设置为非阻塞模式，当缓冲区变满时，返回EAGAIN或者EWOULDBLOCK错误**，此时可以调用select函数来监视何时可以发送数据。

信号通知流程



Linux下的信号采用的异步处理机制，**信号处理函数和当前进程是两条不同的执行路线。**具体的，当**进程收到信号时，操作系统会中断进程当前的正常流程，转而进入信号处理函数执行操作**，完成后再返回中断的地方继续执行。

为避免信号竞态现象发生，信号处理期间系统不会再次触发它。所以，为确保该信号不被屏蔽太久，**信号处理函数需要尽可能快地执行完毕。**

一般的信号处理函数需要处理该信号对应的逻辑，当该逻辑比较复杂时，信号处理函数执行时间过长，会导致信号屏蔽太久。

这里的解决方案是，**信号处理函数仅仅发送信号通知程序主循环，将信号对应的处理逻辑放在程序主循环中，由主循环执行信号对应的逻辑代码。**

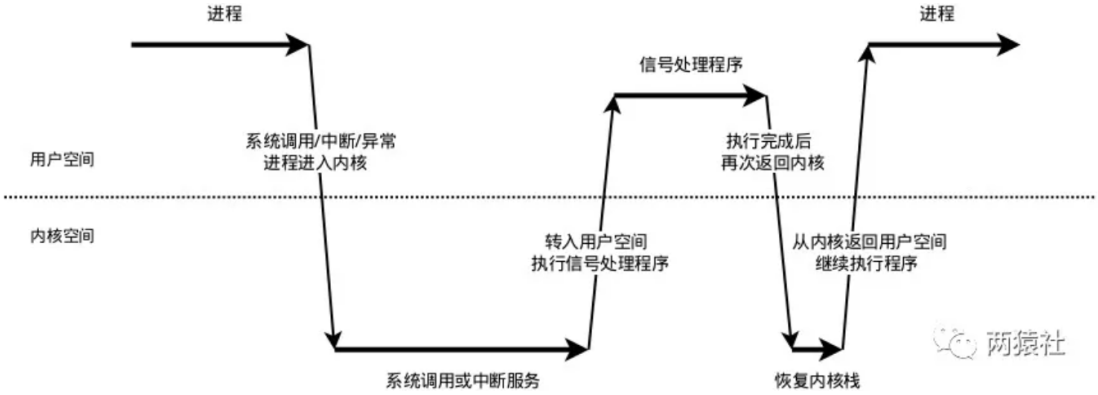
统一事件源

统一事件源，是指将信号事件与其他事件一样被处理。

具体的，**信号处理函数使用管道将信号传递给主循环**，信号处理函数往管道的写端写入信号值，主循环则从管道的读端读出信号值，**使用I/O复用系统调用来监听管道读端的可读事件，这样信号事件与其他文件描述符都可以通过epoll来监测，从而实现统一处理。**

信号处理机制

每个进程之中，都有存着一个表，里面存着每种信号所代表的含义，内核通过设置表项中每一个位来标识对应的信号类型。



- 信号的接收
  - **接收信号的任务是由内核代理的** 当内核接收到信号后 会将其放到对应进程的信号队列中 同时向进程发送一个中断 **使其陷入内核态** 注意 此时信号还只是在队列中 对进程来说暂时是不知道有信号到来的
- 信号的检测
  - 进程从内核态返回到用户态前进行信号检测
  - 进程在内核态中 从睡眠状态被唤醒的时候进行信号检测
  - 进程陷入内核态后 有两种场景会对信号进行检测
  - 当发现有新的信号时 便会进入下一步 信号的处理
- 信号的处理
  - 内核　　**信号处理函数是运行在用户态的** 调用处理函数之前 内核会将当前内核栈的内容备份拷贝到用户栈上 并且修改指令寄存器 eip将其指向信号处理函数
  - 用户　　接下来进程返回到用户台中 **执行相应的信号处理函数**
  - 内核　　信号处理函数执行完成后 还需要返回内核态 检查是否还有其他信号未处理
  - 用户　　**如果所有信号都处理完成 就会将内核栈恢复 （从用户站的备份拷贝回来） 同时恢复指令寄存器 eip将其指向中断前的运行位置 最后回到用户态继续执行进程**

至此 一个完整的信号处理流程便结束了 如果有同时多个信号到达 上面的处理流程会在第二步和第三步骤之间重复进行

信号通知逻辑

- 创建管道，其中**管道写端写入信号值，管道读端通过I/O复用系统监测读事件**
- 设置信号处理函数SIGALRM（时间到了触发）和SIGTERM（kill会触发，Ctrl+C）
-
  - 通过struct sigaction结构体和sigaction函数注册信号捕捉函数
  - 在结构体的handler参数设置信号处理函数，具体的，从管道写端写入信号的名字

- 利用I/O复用系统监听管道读端文件描述符的可读事件
- **信息值传递给主循环，主循环再根据接收到的信号值执行目标信号对应的逻辑代码**

为什么管道写端要非阻塞?

send是将信息发送给套接字缓冲区，如果缓冲区满了，**则会阻塞，这时候会进一步增加信号处理函数的执行时间，为此，将其修改为非阻塞。**

没有对非阻塞返回值处理，如果阻塞是不是意味着这一次定时事件失效了?

是的，但定时事件是非必须立即处理的事件，可以允许这样的情况发生。

管道传递的是什么类型? switch-case的变量冲突?

信号本身是整型数值，管道中传递的是ASCII码表中整型数值对应的字符。

switch的变量一般为字符或整型，当switch的变量为字符时，case中可以是字符，也可以是字符对应的ASCII码。

08.定时器处理非活动连接（下）

本文内容

定时器处理非活动连接模块，主要分为两部分，其一为定时方法与信号通知流程，其二为定时器及其容器设计、定时任务的处理。

本篇对第二部分进行介绍，**具体的涉及到定时器设计、容器设计、定时任务处理函数和使用定时器。**

`定时器设计`，将连接资源和定时事件等封装起来，**具体包括连接资源、超时时间和回调函数，这里的回调函数指向定时事件。**

`定时器容器设计`，将多个定时器串联组织起来统一处理，**具体包括升序链表设计。**

`定时任务处理函数`，**该函数封装在容器类中，具体的，函数遍历升序链表容器，根据超时时间，处理对应的定时器。**

`代码分析-使用定时器`，通过代码分析，如何在项目中使用定时器。

定时器设计

项目中将**连接资源、定时事件和超时时间**封装为定时器类，具体的，

- **连接资源包括客户端套接字地址、文件描述符和定时器**
- **定时事件为回调函数**，将其封装起来由用户自定义，这里是删除非活动socket上的注册事件，并关闭
- 定时器超时时间 = 浏览器和服务器连接时刻 + 固定时间(TIMESLOT)，可以看出，定时器使用绝对时间作为超时值，**这里alarm设置为5秒，连接超时为15秒。**

定时任务处理函数

使用统一事件源，SIGALRM信号每次被触发 主循环利用epoll复用监听信号，然后**主循环中调用一次定时任务处理函数**，处理链表容器中到期的定时器。

具体的逻辑如下，

- 遍历定时器升序链表容器，**从头结点开始依次处理每个定时器，直到遇到尚未到期的定时器**
- **若当前时间小于定时器超时时间，跳出循环，即未找到到期的定时器**
- **若当前时间大于定时器超时时间，即找到了到期的定时器，执行回调函数，然后将它从链表中删除，然后继续遍历**

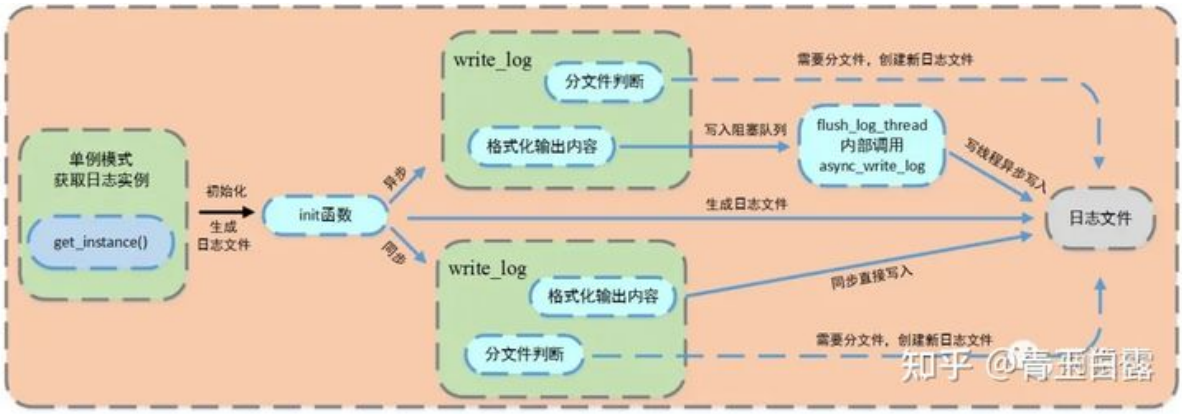
代码分析-如何使用定时器

服务器**首先创建定时器容器链表，然后用统一事件源将异常事件，读写事件和信号事件统一处理**，根据不同事件的对应逻辑使用定时器。

具体的，

- **浏览器与服务器连接时，创建该连接对应的定时器，并将该定时器添加到链表上**
- **处理异常事件时，执行定时事件，服务器关闭连接，从链表上移除对应定时器**
- 处理定时信号时，将定时标志设置为true
- **处理读事件时，若某连接上发生读事件，将对应定时器向后移动，否则，执行定时事件**
- **处理写事件时，若服务器通过某连接给浏览器发送数据，将对应定时器向后移动，否则，执行定时事件**

09.日志系统（上）



由上图可知 该系统有同步和异步两种写入方式

其中异步写入方式 **将生产者-消费者模型封装为阻塞队列** 创建一个写线程 工作线程将要写的内容push进队列 写线程从队列中取出内容 写入日志文件 对于同步写入方式 直接格式化输出内容 将信息写入日志文件

该系统可以实现按天分类 超行分类功能

基础知识

** `日志` **，由服务器自动创建，并记录运行状态，错误信息，访问数据的文件。

** `同步日志` **，日志写入函数与工作线程串行执行，由于涉及到I/O操作，当单条日志比较大的时候，同步模式会阻塞整个处理流程，服务器所能处理的并发能力将有所下降，尤其是在峰值的时候，写日志可能成为系统的瓶颈。**

** `生产者-消费者模型` **，并发编程中的经典模型。以多线程为例，**为了实现线程间数据同步，生产者线程与消费者线程共享一个缓冲区，其中生产者线程往缓冲区中push消息，消费者线程从缓冲区中pop消息。**

** `阻塞队列` **，将生产者-消费者模型进行封装，使用循环数组实现队列，作为两者共享的缓冲区。**

** `异步日志` **，将所写的日志内容先存入阻塞队列，写线程从阻塞队列中取出内容，写入日志。**

** `单例模式` **，最简单也是被问到最多的设计模式之一，**保证一个类只创建一个实例，同时提供全局访问的方法。**

整体概述

本项目中，**使用单例模式创建日志系统，对服务器运行状态、错误信息和访问数据进行记录**，该系统可以实现按天分类，超行分类功能，**可以根据实际情况分别使用同步和异步写入两种方式。**

其中异步写入方式，将生产者-消费者模型封装为阻塞队列，创建一个写线程，工作线程将要写的内容push进队列，写线程从队列中取出内容，写入日志文件。

日志系统大致可以分成两部分，**其一是单例模式与阻塞队列的定义，其二是日志类的定义与使用。**

本文内容

本篇将介绍单例模式与阻塞队列的定义，具体的涉及到单例模式、生产者-消费者模型，阻塞队列的代码实现。

单例模式，描述懒汉与饿汉两种单例模式，并结合线程安全进行讨论。

生产者-消费者模型，**描述条件变量**，基于该同步机制实现简单的生产者-消费者模型。

代码实现，结合代码对阻塞队列的设计进行详解。

单例模式

单例模式作为最常用的设计模式之一，**保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。**

实现思路：**私有化它的构造函数，以防止外界创建单例类的对象**；**使用类的私有静态指针变量指向类的唯一实例，并用一个公有的静态方法获取该实例。**

单例模式有两种实现方法，分别是懒汉和饿汉模式。顾名思义，懒汉模式，即非常懒，不用的时候不去初始化，**所以在第一次被使用时才进行初始化**；饿汉模式，即迫不及待，**在程序运行时立即初始化。**

经典的线程安全懒汉模式

单例模式的实现思路如前述所示，其中，经典的线程安全懒汉模式，使用双检测锁模式。

```
1class single{
2private:
3    //私有静态指针变量指向唯一实例
4    static single *p;
5
6    //静态锁，是由于静态函数只能访问静态成员
7    static pthread_mutex_t lock;
8
9    //私有化构造函数
10    single(){
11        pthread_mutex_init(&lock, NULL);
12    }
13    ~single(){}
14
15public:
16    //公有静态方法获取实例
17    static single* getinstance();
18
19};
20
21pthread_mutex_t single::lock;
22
23single* single::p = NULL;
```

```
24single* single::getinstance(){
25    if (NULL == p){
26        pthread_mutex_lock(&lock);
27        if (NULL == p){
28            p = new single;
29        }
30        pthread_mutex_unlock(&lock);
31    }
32    return p;
33}
```

`为什么要用双检测，只检测一次不行吗？`

如果只检测一次，在每次调用获取实例的方法时，都需要加锁，这将严重影响程序性能。双层检测可以有效避免这种情况，**仅在第一次创建单例的时候加锁，其他时候都不再符合NULL == p的情况**，直接返回已创建好的实例。

局部静态变量之线程安全懒汉模式

前面的双检测锁模式，写起来不太优雅，《Effective C++》（Item 04）中的提出另一种更优雅的单例模式实现，使用函数内的局部静态对象，这种方法不用加锁和解锁操作。

```
 1class single{
 2private:
 3    single(){}
 4    ~single(){}
 5
 6public:
 7    static single* getinstance();
 8
 9};
10
11single* single::getinstance(){
12    static single obj;
13    return &obj;
14}
```

`这时候有人说了，这种方法不加锁会不会造成线程安全问题？`

其实，C++0X以后，要求编译器保证内部静态变量的线程安全性，故C++0x之后该实现是线程安全的，C++0x之前仍需加锁，其中C++0x是C++11标准成为正式标准之前的草案临时名字。

所以，如果使用C++11之前的标准，还是需要加锁，这里同样给出加锁的版本。

```
 1class single{
 2private:
 3    static pthread_mutex_t lock;
 4    single(){
 5        pthread_mutex_init(&lock, NULL);
 6    }
 7    ~single(){}
 8
 9public:
10    static single* getinstance();
11
12};
13pthread_mutex_t single::lock;
```

```
14single* single::getinstance(){
15    pthread_mutex_lock(&lock);
16    static single obj;
17    pthread_mutex_unlock(&lock);
18    return &obj;
19}
```

饿汉模式

饿汉模式不需要用锁，就可以实现线程安全。原因在于，**在程序运行时就定义了对象，并对其初始化。**之后，**不管哪个线程调用成员函数getinstance()，都只不过是返回一个对象的指针而已。所以是线程安全的，不需要在获取实例的成员函数中加锁。**

```
 1class single{
 2private:
 3    static single* p;
 4    single(){}
 5    ~single(){}
 6
 7public:
 8    static single* getinstance();
 9
10};
11single* single::p = new single();
12single* single::getinstance(){
13    return p;
14}
15
16//测试方法
17int main(){
18
19    single *p1 = single::getinstance();
20    single *p2 = single::getinstance();
21
22    if (p1 == p2)
23        cout << "same" << endl;
24
25    system("pause");
26    return 0;
27}
```

饿汉模式虽好，但其存在隐藏的问题，**在于非静态对象（函数外的static对象）在不同编译单元中的初始化顺序是未定义的。如果在初始化完成之前调用 getInstance() 方法会返回一个未定义的实例。**

条件变量与生产者-消费者模型

条件变量API与陷阱

条件变量提供了一种线程间的通知机制，当某个共享数据达到某个值时,唤醒等待这个共享数据的线程。

基础API

- pthread_cond_init函数，用于初始化条件变量
- pthread_cond_destory函数，销毁条件变量
- pthread_cond_broadcast函数，以广播的方式唤醒**所有**等待目标条件变量的线程

- pthread_cond_wait函数，用于等待目标条件变量。该函数调用时需要传入 **mutex参数(加锁的互斥锁)**，函数执行时，先把调用线程放入条件变量的请求队列，然后将互斥锁mutex解锁，当函数成功返回为0时，表示重新抢到了互斥锁，互斥锁会再次被锁上， **也就是说函数内部会有一次解锁和加锁操作**.

使用pthread_cond_wait方式如下：

```
1pthread _mutex_lock(&mutex)
2
3while(线程执行的条件是否成立){
4    pthread_cond_wait(&cond, &mutex);
5}
6
7pthread_mutex_unlock(&mutex);
```

pthread_cond_wait执行后的内部操作分为以下几步：

- 将线程放在条件变量的请求队列后，内部解锁
- 线程等待被pthread_cond_broadcast信号唤醒或者pthread_cond_signal信号唤醒，唤醒后去竞争锁
- 若竞争到互斥锁，内部再次加锁

陷阱一

** `使用前要加锁，为什么要加锁？` **

多线程访问，为了避免资源竞争，所以要加锁，使得每个线程互斥的访问公有资源。

** `pthread_cond_wait内部为什么要解锁？` **

如果while或者if判断的时候，满足执行条件，线程便会调用pthread_cond_wait阻塞自己，此时它还在持有锁，如果他不解锁，那么其他线程将会无法访问公有资源。

具体到pthread_cond_wait的内部实现，当pthread_cond_wait被调用线程阻塞的时候，pthread_cond_wait会自动释放互斥锁。

** `为什么要把调用线程放入条件变量的请求队列后再解锁？` **

线程是并发执行的，如果在把调用线程A放在等待队列之前，就释放了互斥锁，这就意味着其他线程比如线程B可以获得互斥锁去访问公有资源，这时候线程A所等待的条件改变了，但是它没有被放在等待队列上，**导致A忽略了等待条件被满足的信号。**

倘若在线程A调用pthread_cond_wait开始，到把A放在等待队列的过程中，都持有互斥锁，其他线程无法得到互斥锁，就不能改变公有资源。

** `为什么最后还要加锁？` **

将线程放在条件变量的请求队列后，将其解锁，此时等待被唤醒，**若成功竞争到互斥锁，再次加锁。**

陷阱二

** `为什么判断线程执行的条件用while而不是if？` **

一般来说，在多线程资源竞争的时候，在一个使用资源的线程里面（消费者）判断资源是否可用，不可用，便调用pthread_cond_wait，在另一个线程里面（生产者）如果判断资源可用的话，则调用pthread_cond_signal发送一个资源可用信号。

在wait成功之后，资源就一定可以被使用么？答案是否定的，如果同时有两个或者两个以上的线程正在等待此资源，wait返回后，资源可能已经被使用了。

再具体点，有可能多个线程都在等待这个资源可用的信号，信号发出后只有一个资源可用，但是有A，B两个线程都在等待，B比较速度快，获得互斥锁，然后加锁，消耗资源，然后解锁，之后A获得互斥锁，但A回去发现资源已经被使用了，**它便有两个选择，一个是去访问不存在的资源，另一个就是继续等待，那么继续等待下去的条件就是使用while**，要不然使用if的话pthread_cond_wait返回后，就会顺序执行下去。

所以，在这种情况下，应该使用while而不是if:

```
1while(resource == FALSE)
2    pthread_cond_wait(&cond, &mutex);
```

如果只有一个消费者，那么使用if是可以的。

生产者-消费者模型

这里摘抄《Unix 环境高级编程》中第11章线程关于pthread_cond_wait的介绍中有一个生产者-消费者的例子P311，其中，process_msg相当于消费者，enqueue_msg相当于生产者，struct msg* workq作为缓冲队列。

生产者和消费者是互斥关系，两者对缓冲区访问互斥，**同时生产者和消费者又是一个相互协作与同步的关系，只有生产者生产之后，消费者才能消费。**

```
 1#include <pthread.h>
 2struct msg {
 3  struct msg *m_next;
 4  /* value...*/
 5};
 6
 7struct msg* workq;
 8pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
 9pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;
10
11void
12process_msg() {
13  struct msg* mp;
14  for (;;) {
15    pthread_mutex_lock(&qlock);
16    //这里需要用while，而不是if
17    while (workq == NULL) {
18      pthread_cond_wait(&qread, &qlock);
19    }
20    mq = workq;
21    workq = mp->m_next;
22    pthread_mutex_unlock(&qlock);
23    /* now process the message mp */
24  }
25}
26
27void
28enqueue_msg(struct msg* mp) {
29    pthread_mutex_lock(&qlock);
30    mp->m_next = workq;
31    workq = mp;
32    pthread_mutex_unlock(&qlock);
33    /** 此时另外一个线程在signal之前，执行了process_msg，刚好把mp元素拿走*/
34    pthread_cond_signal(&qready);
35    /** 此时执行signal，在pthread_cond_wait等待的线程被唤醒，
```

```
36          但是mp元素已经被另外一个线程拿走，所以，workq还是NULL ,因此需要继续等待*/
37}
```

阻塞队列代码分析

阻塞队列类中封装了生产者-消费者模型，**其中push成员是生产者，pop成员是消费者。**

阻塞队列中，使用了循环数组实现了队列，作为两者共享缓冲区，当然了，队列也可以使用STL中的queue。

自定义队列

当队列为空时，从队列中获取元素的线程将会被挂起；当队列是满时，往队列里添加元素的线程将会挂起。

阻塞队列类中，有些代码比较简单，这里仅对push和pop成员进行详解。

10.日志系统（下）

本文内容

日志系统分为两部分，其一是单例模式与阻塞队列的定义，其二是日志类的定义与使用。

本篇将介绍日志类的定义与使用，具体的涉及到基础API，流程图与日志类定义，功能实现。

基础API，描述fputs，可变参数宏**VA_ARGS**，fflush

流程图与日志类定义，描述日志系统整体运行流程，介绍日志类的具体定义

功能实现，结合代码分析同步、异步写文件逻辑，分析超行、按天分文件和日志分级的具体实现

基础API

为了更好的源码阅读体验，这里对一些API用法进行介绍。

fputs

```
1#include <stdio.h>
2int fputs(const char *str, FILE *stream);
```

- str，一个数组，包含了要写入的以空字符终止的字符序列。
- stream，指向FILE对象的指针，该FILE对象标识了要被写入字符串的流。

可变参数宏VA_ARGS

VA_ARGS是一个可变参数的宏，定义时宏定义中参数列表的最后一个参数为省略号，在实际使用时会发现有时会加##，有时又不加。

```
1//最简单的定义
2#define my_print1(...)  printf(__VA_ARGS__)
3
4//搭配va_list的format使用
5#define my_print2(format, ...) printf(format, __VA_ARGS__)
6#define my_print3(format, ...) printf(format, ##__VA_ARGS__)
```

VA_ARGS宏前面加上##的作用在于，当可变参数的个数为0时，这里printf参数列表中的的##会把前面多余的","去掉，否则会编译出错，建议使用后面这种，使得程序更加健壮。

fflush

```
1#include <stdio.h>
2int fflush(FILE *stream);
```

fflush()会强迫将缓冲区内的数据写回参数stream 指定的文件中，如果参数stream 为NULL，fflush()会将所有打开的文件数据更新。

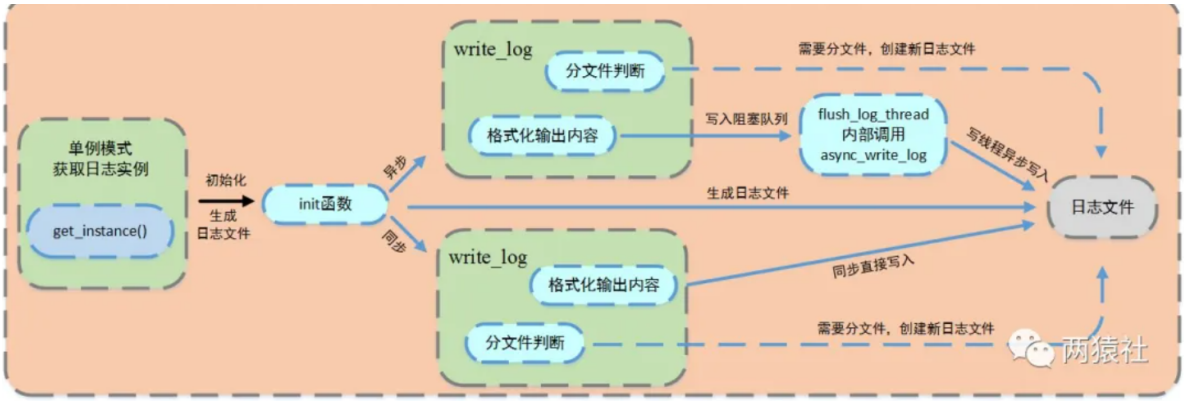
在使用多个输出函数连续进行多次输出到控制台时，有可能下一个数据再上一个数据还没输出完毕，还在输出缓冲区中时，下一个printf就把另一个数据加入输出缓冲区，结果冲掉了原来的数据，出现输出错误。

在prinf()后加上fflush(stdout); 强制马上输出到控制台，可以避免出现上述错误。

流程图与日志类定义

流程图

- 日志文件
- - **局部变量的懒汉模式获取实例**
  - 生成日志文件，并判断同步和异步写入方式
- 同步
- - 判断是否分文件
  - **直接格式化输出内容，将信息写入日志文件**
- 异步
- - 判断是否分文件
  - 格式化输出内容，**将内容写入阻塞队列，创建一个写线程，从阻塞队列取出内容写入日志文件**



日志类定义

通过局部变量的懒汉单例模式创建日志实例，对其进行初始化生成日志文件后，格式化输出内容，并根据不同的写入方式，完成对应逻辑，写入日志文件。

日志类包括但不限于如下方法，

- 公有的实例获取方法
- 初始化日志文件方法
- 异步日志写入方法，内部调用私有异步方法
- 内容格式化方法
- 刷新缓冲区
- ...

日志类中的方法都不会被其他程序直接调用，末尾的四个可变参数宏提供了其他程序的调用方法。

前述方法对日志等级进行分类，包括DEBUG，INFO，WARN和ERROR四种级别的日志。

功能实现

init函数实现日志创建、写入方式的判断。

write_log函数完成写入日志文件中的具体内容，主要实现日志分级、分文件、格式化输出内容。

生成日志文件 && 判断写入方式

通过单例模式获取唯一的日志类，调用init方法，**初始化生成日志文件，服务器启动按当前时刻创建日志，前缀为时间，后缀为自定义log文件名，并记录创建日志的时间day和行数count。**

写入方式通过初始化时**是否设置队列大小**（表示在队列中可以放几条数据）来判断，**若队列大小为0，则为同步，否则为异步。**

日志分级与分文件

日志分级的实现大同小异，一般的会提供五种级别，具体的，

- Debug，调试代码时的输出，在系统实际运行时，一般不使用。
- Warn，这种警告与调试时终端的warning类似，同样是调试代码时使用。
- Info，报告系统当前的状态，当前执行的流程或接收的信息等。
- Error和Fatal，输出系统的错误信息。

上述的使用方法仅仅是个人理解，在开发中具体如何选择等级因人而异。项目中给出了除Fatal外的四种分级，实际使用了Debug，Info和Error三种。

超行、按天分文件逻辑，具体的，

- 日志写入前会判断当前day是否为创建日志的时间，行数是否超过最大行限制
-
  - **若为创建日志时间，写入日志，否则按当前时间创建新log，更新创建时间和行数**
  - **若行数超过最大行限制，在当前日志的末尾加count/max_lines为后缀创建新log**

将系统信息格式化后输出，具体为：**格式化时间 + 格式化内容**

11.数据库连接池

基础知识

> 什么是数据库连接池?

池是一组资源的集合，这组资源在**服务器启动之初就被完全创建好并初始化。**通俗来说，池是资源的容器，**本质上是对资源的复用。**

顾名思义，连接池中的资源为一组数据库连接，由程序动态地对池中的连接进行使用，释放。

当系统开始处理客户请求的时候，如果它需要相关的资源，**可以直接从池中获取，无需动态分配**；当服务器处理完一个客户连接后,可以把相关的资源放回池中，**无需执行系统调用释放资源。**

> 数据库访问的一般流程是什么?

当系统需要访问数据库时，先系统创建数据库连接，完成数据库操作，然后系统断开数据库连接。

> 为什么要创建连接池?

从一般流程中可以看出，若系统需要频繁访问数据库，**则需要频繁创建和断开数据库连接，而创建数据库连接是一个很耗时的操作，也容易对数据库造成安全隐患。**

在程序初始化的时候，**集中创建多个数据库连接，并把他们集中管理，供程序使用，可以保证较快的数据库读写速度，更加安全可靠。**

整体概述

池可以看做资源的容器，所以多种实现方法，比如数组、链表、队列等。**这里，使用单例模式和链表创建数据库连接池，实现对数据库连接资源的复用。**

项目中的数据库模块分为两部分，其一是数据库连接池的定义，其二是利用连接池完成登录和注册的校验功能。具体的，工作线程从数据库连接池取得一个连接，访问数据库中的数据，访问完毕后将连接交还连接池。

本文内容

本篇将介绍数据库连接池的定义，具体的涉及到单例模式创建、连接池代码实现、RAII机制释放数据库连接。

单例模式创建，结合代码描述连接池的单例实现。

连接池代码实现，结合代码对连接池的外部访问接口进行详解。

RAII机制释放数据库连接，描述连接释放的封装逻辑。

连接池代码实现

连接池的定义中注释比较详细，这里仅对其实现进行解析。

连接池的功能主要有：**初始化，获取连接、释放连接，销毁连接池。**

初始化

值得注意的是，销毁连接池没有直接被外部调用，而是通过RAII机制来完成自动释放；使用信号量实现多线程争夺连接的同步机制，这里将信号量初始化为数据库的连接总数。

获取、释放连接

当线程数量大于数据库连接数量时，使用信号量进行同步，每次取出连接，信**号量原子减1，释放连接原子加1，若连接池内没有连接了，则阻塞等待。**

另外，由于多线程操作连接池，会造成竞争，**这里使用互斥锁完成同步，具体的同步机制均使用lock.h中封装好的类。**

销毁连接池

通过迭代器遍历连接池链表，关闭对应数据库连接，清空链表并重置空闲连接和现有连接数量。

RAII机制释放数据库连接

将数据库连接的获取与释放通过RAII机制封装，**避免手动释放。**

定义

这里需要注意的是，在获取连接时，通过有参构造对传入的参数进行修改。其中数据库连接本身是指针类型，所以参数需要通过双指针才能对其进行修改。

实现

不直接调用获取和释放连接的接口，**将其封装起来，通过RAII机制进行获取和释放。**

12.注册登陆

整体概述

本项目中，使用数据库连接池实现服务器访问数据库的功能，**使用POST请求完成注册和登录的校验工作。**

本文内容

本篇将介绍同步实现注册登录功能，具体的涉及到流程图，载入数据库表，提取用户名和密码，注册登录流程与页面跳转的的代码实现。

流程图，描述服务器从报文中提取出用户名密码，并完成注册和登录校验后，实现页面跳转的逻辑。

载入数据库表，结合代码将数据库中的数据载入到服务器中。

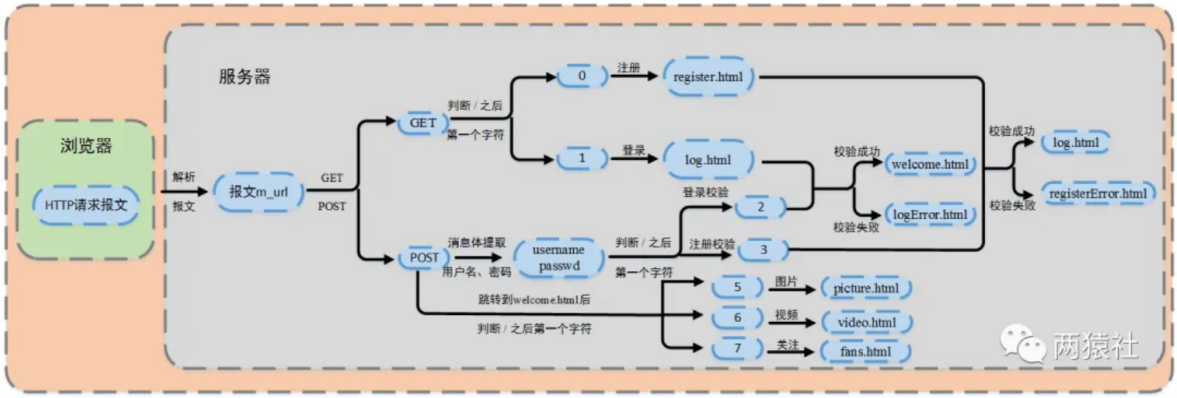
提取用户名和密码，结合代码对报文进行解析，提取用户名和密码。

注册登录流程，结合代码对描述服务器进行注册和登录校验的流程。

页面跳转，结合代码对页面跳转机制进行详解。

流程图

具体的，描述了GET和POST请求下的页面跳转流程。



载入数据库表

将数据库中的用户名和密码载入到服务器的map中来，map中的key为用户名，value为密码。

提取用户名和密码

服务器端解析浏览器的请求报文，**当解析为POST请求时，cgi标志位设置为1，并将请求报文的消息体赋值给m_string，进而提取出用户名和密码。**

同步线程登录注册

通过m_url定位/所在位置，根据/后的第一个字符判断是登录还是注册校验。

- 2
-
  - 登录校验
- 3

- 注册校验

根据校验结果，跳转对应页面。另外，对数据库进行操作时，需要通过锁来同步。

页面跳转

通过**m_url**定位/所在位置，根据/后的第一个字符，使用分支语句实现页面跳转。具体的，

- 0
- 跳转注册页面，GET
- 1
- 跳转登录页面，GET
- 5
- 显示图片页面，POST
- 6
- 显示视频页面，POST
- 7
- 显示关注页面，POST

13.踩坑和面试题

本文内容

本篇是项目的最终篇，将介绍踩坑与面试题两部分。

踩坑，描述做项目过程中遇到的问题与解决方案。

面试题，介绍项目相关的知识点变种和真实面试题，**这里不会给出答案**，具体的，可以在项目微信群中讨论。

踩坑

做项目过程中，肯定会遇到形形色色、大大小小的问题，但并不是所有问题都值得列出来探讨，这里仅列出个人认为有意义的问题。

具体的，包括大文件传输。

大文件传输

先看下之前的大文件传输，也就是游双书上的代码，发送数据只调用了writev函数，并对其返回值是否异常做了处理。

在实际测试中发现，当请求小文件，也就是调用一次writev函数就可以将数据全部发送出去的时候，不会报错，此时不会再次进入while循环。

一旦请求服务器文件较大文件时，需要多次调用writev函数，便会出现问题，不是文件显示不全，就是无法显示。

对数据传输过程分析后，定位到writev的m_iv结构体成员有问题，每次传输后不会自动偏移文件指针和传输长度，还会按照原有指针和原有长度发送数据。

根据前面的基础API分析，我们知道writev以顺序iov[0]，iov[1]至iov[iovcnt-1]从缓冲区中聚集输出数据。项目中，申请了2个iov，其中iov[0]为存储报文状态行的缓冲区，iov[1]指向资源文件指针。

对上述代码做了修改如下：

- 由于报文消息报头较小，第一次传输后，需要更新m_iv[1].iov_base和iov_len，m_iv[0].iov_len置成0，只传输文件，不用传输响应消息头
- 每次传输后都要更新下次传输的文件起始位置和长度

更新后，大文件传输得到了解决。

```
bool http_conn::write()
{
    int temp = 0;

    int newadd = 0;

    if (bytes_to_send == 0)
    {
        modfd(m_epollfd, m_sockfd, EPOLLIN, m_TRIGMode);
        init();
        return true;
    }

    while (1)
    {
        temp = writev(m_sockfd, m_iv, m_iv_count);

        if (temp >= 0)
        {
            bytes_have_send += temp;
            newadd = bytes_have_send - m_write_idx;
        }
        else
        {
            if (errno == EAGAIN)
            {
                if (bytes_have_send >= m_iv[0].iov_len)
                {
                    m_iv[0].iov_len = 0;
                    m_iv[1].iov_base = m_file_address + newadd;
                    m_iv[1].iov_len = bytes_to_send;
                }
                else
                {
                    m_iv[0].iov_base = m_write_buf + bytes_have_send;
                    m_iv[0].iov_len = m_iv[0].iov_len - bytes_have_send;
                }
                modfd(m_epollfd, m_sockfd, EPOLLOUT, m_TRIGMode);
                return true;
            }
            unmap();
            return false;
        }
        bytes_to_send -= temp;
        if (bytes_to_send <= 0)

        {
            unmap();
            modfd(m_epollfd, m_sockfd, EPOLLIN, m_TRIGMode);

            if (m_linger)
```

```
52            {
53                init();
54                return true;
55            }
56            else
57            {
58                return false;
59            }
60        }
61    }
62}
```