基础概念知识

RAII(Resource Acquisition is Initialization) 资源获取即初始化

将资源或者状态与对象的生命周期绑定，实现资源和状态的安全管理，消除资源泄漏并保证异常安全，如（智能指针，分配的堆内存，执行线程，打开套接字，打开的文件，锁定的互斥锁，磁盘空间，数据库连接等）

1. 用户动态申请的**堆内存未释放**，无法被系统回收而不能再被重复利用：使用智能指针；
2. 智能指针中的**循环引用**导致指针无法正常释放的资源泄露情况：使用 weak_ptr 进行循环引用；
3. 打开套接字、文件、数据库连接等在程序执行结束之后未释放；
4. 如果 std::thread 类传入的参数含有引用或指针，则子线程中的数据依赖于主线程中的内存，使用 **detach()**时，可能存在主线程比子线程先结束的情况，主线程结束后会释放掉自身的内存空间，子线程会出现未定义的错误；

动态、静态链接库

1.静态库（.lib / .a）

在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其他模块组合起来创建最终的可执行文件（.EXE 文件）。当发布产品时，只需要发布这个可执行文件，并不需要发布被使用的静态库。

2.动态库（.dll / .so）

一种可执行代码模块，允许多个程序共享相同的库文件，从而减少内存使用和磁盘空间占用。在 Linux 系统中，这类文件通常以.so 为后缀名，而在 Windows 系统中则为.dll。使用动态库可以(1)节省内存和磁盘空间：多个程序可以同时使用同一个动态库实例。(2)简化更新：编译更新动态库后，所有使用该库的程序都自动获得更新后的功能。(3)动态加载：程序可以在运行时选择性地加载，从而实现模块化设计和延迟加载，提升启动性能。

条件变量

使多个线程根据条件变量 x 同步执行；多个线程共用同一把互斥锁 mutex，先加锁后查看是否满足条件（防止条件改变），不满足就通过调用 pthread_cond_wait 进入 x 的请求队列中等待，并解锁（使等待的时候 x 能够被其他线程所改变）；收到其他线程的 signal/broadcase 唤醒信号后多个请求队列中等待的线程竞态抢锁，先加锁，然后判断条件是否满足（不满足条件的会继续用 wait 等待，同时释放锁），满足则会执行该子线程的操作，最后修改条件变量，并唤醒其他所有等待线程，并解锁。

```
mutex mtx;
condition_variable cv;
string str = "A";
void process1() {
    while (true) {
        unique_lock<mutex> lck(mtx);
        while (str == "B") {
```

```
            cv.wait(lck);
        }
        this_thread::sleep_for(chrono::seconds(1));
        cout << "A" << endl;
        str = "B";
        lck.unlock();
        cv.notify_all();
    }
}
```

互斥锁

foo 中锁住 mtx1 再输出 foo，解锁 mtx2；bar 中锁住 mtx2 输出后再解锁 mtx1；循环上锁、解锁

```
class FooBar {
private:
	int n;
	mutex mtx1,mtx2;
public:
	FooBar(int n) {
		this->n = n;
		mtx2.lock();  //初始化的时候锁住 mtx2，让 foo 先输出
	}
	void foo(function<void()> printFoo) {
		for (int i = 0; i < n; i++) {
			mtx1.lock();
			printFoo();
			mtx2.unlock();
		}
	}
	void bar(function<void()> printBar) {
		for (int i = 0; i < n; i++) {
			mtx2.lock();
			printBar();
			mtx1.unlock();
		}
	}
};
```

信号量

```
#include<semaphore.h>
class FooBar {
private:
	int n;
	sem_t foo_done,bar_done;
```

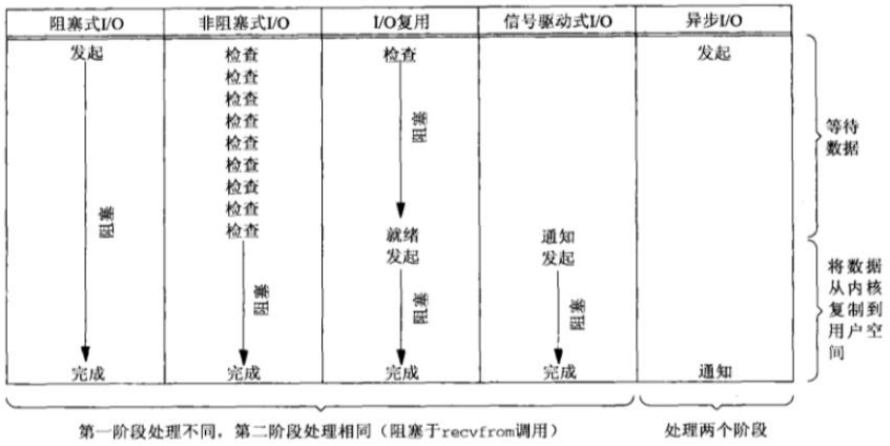
```
public:
	FooBar(int n) {
		this->n = n;
		sem_init(&foo_done,0,0);
		sem_init(&bar_done,0,1);
	}
	void foo(function<void()> printFoo) {
		for (int i = 0; i < n; i++) {
			sem_wait(&bar_done);
			printFoo();
			sem_post(&foo_done);
		}
	}
	void bar(function<void()> printBar) {
		for (int i = 0; i < n; i++) {
			sem_wait(&foo_done);
			printBar();
			sem_post(&bar_done);
		}
	}
};
```

与互斥锁同理，初始化两个信号量，一个为初始值为 1 用以先输出 foo，再释放另一个信号量

五种 I/O 模型

同步 IO：阻塞、非阻塞、信号驱动、IO 复用

异步 IO



阻塞式 IO：从进程调用 recvfrom，内核接收数据、拷贝数据到用户空间整个过程进程阻塞。

非阻塞式 IO：

1. 进程调用 recvfrom 后，轮询内核是否完成数据接收；

2. 2. 成功后进入阻塞，等待内核将数据拷贝到用户空间。

IO 复用：

1. 通过 select 或 epoll 系统调用，使用单个线程同时监听多个输入流（网络套接字、文件描述符等），等待多个套接字（优势）中的任一个变为可读；

2. 通过 recvfrom 系统调用将数据拷贝到用户空间。

信号驱动式 IO：

1. 开启套接字信号驱动式 IO 功能，sigaction 系统调用告知内核在某个文件描述符就绪时通知进程，进程不会阻塞而是继续执行；
2. 主进程接受到 SIGIO 信号，同步将数据拷贝到用户空间。

异步 IO：

告知内核自动执行操作，并让内核在整个操作完成后通知我们

事件处理模式

Reactor：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：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。

Webserver 中采用 Reactor+同步 IO 模拟实现 Proactor 模式

1. 主线程往 epoll 内核事件表注册 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，epoll_wait 通知主线程,主线程从 socket 循环读取数据，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
5. 主线程调用 epoll_wait 等待 socket 可写。
6. 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。

epoll 系统调用---Linux 系统下的多路 I/O 就绪通知方法

epoll 把用户关心的文件描述符上面的事件放在内核里的一个事件表中（用一个额外的文件描述符来唯一标识这个事件表），而无须像 select（用数组维护）和 poll（用链表维护）那样每次调用都要重复传入所有的文件描述符或事件集。

三个方法：

1. int epoll_create(int size)

返回事件表的文件描述符，作为其他方法的参数以指定要访问的内核事件表

2. int epoll_ctl(int epfd, int op, int fd ,struct epoll_event *event)

用于对内核事件表执行指定的操作：

EOPLL_CTL_ADD, 往事件表中注册 fd 上的事件；

EOPLL_CTL_MOD ， 修改 fd 上的注册事件；

EOPLL_CTL_DEL ， 删除 fd 上的注册事件。

3. int epoll_wait(int epfd ,struct epoll_event *events,int maxevents, int timeout)

它在一段超时时间内等待一组文件描述符上面的事件，成功时返回就绪的文件描述符的个数，失败时返回-1 并设置相关的 errno；

epoll_wait 函数如果检测到事件，就将所有就绪的事件从内核事件表中(由 epfd 参数指定) 中复制到它的第二个参数 events 指向的数组中。这个数组只用于输出 epoll_wait 检测到的就绪事件, 而不像 select 和 poll 的数组参数那样传入用户注册的所有事件，由用户输出内核检测到的就绪事件。这就极大地提高了应用程序索引就绪文件描述的效率。

LT 和 ET 模式、EPOLLONESHOT

LT（Level Trigger，水平触发）

当 epoll_wait 检测到有事件发生并将此事件通知给应用程序后，应用程序可以不立刻处理该事件，当应用程序下一次调用 epoll_wait 时，epoll 函数在仍会告知此事件，直到该事件被处理为止。

ET（Edge Trigger，边缘触发）

当 epoll_wait 检测到有事件发生并将此事件通知给应用程序后，应用程序必须立刻处理该事件 ，因为后续的 epoll_wait 调用将不再重复通知此事件。

EPOLLONESHOT

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

线程基础知识

引入头文件#include<thread>，该头文件中包括 std::thread 类，可以用了实例化线程。

1. 语句"std::thread th1(proc1);"创建了一个名为 th1 的线程，并且**程序执行此语句后线程 th1 就已经开始执行**。

2. **join 方法**用于阻塞主线程（等待），让其等待子线程执行完成，所以一般不应该在创建了线程后马上 join；**detach 方法**可以使主线程不阻塞而继续执行后续代码，但会导致子线程还未完成主线程就结束的情况，这时候主线程应添加等待条件，防止上述情况。

3.

1）若函数为有参函数,如"void proc2(int a,int b)",那么实例化 std::thread 类对象时，则需要传递更多参数，参数顺序依次为函数名、该函数的第一个参数、该函数的第二个参数，…，如"std::thread th2(proc2,a,b);　2）如果函数为**成员函数**如"void Utils::proc3(int a,int b)",那么创建线程的语句为"std::thread th3(&Utils::proc3,a,b);　3)当传入的参数为引用时，实参必须用 ref()函数处理后传递给形参，否则编译不通过，此时不存在“拷贝”行为。引用只是变量的别名 如：thread t1(proc, ref(a));

4. 异步线程

#include<future>

std::async 是一个函数模板，用来启动一个**异步**任务，它返回一个 std::future 类模板对，future 对象起到了**占位**的作用;通过调用 std::future 对象的 get()成员函数，主线程会被**阻塞**直到异步线程执行结束，并把返回结果传递给 std::future

```
double a = 2.3; double b= 6.7;
future<double> fu = async(t1, a, b);   //创建异步线程线程，并将线程的执行结果用 fu 占位；
cout << "计算结果：" << fu.get() << endl;  //阻塞主线程，直至异步线程 return
```

5. 原子变量是一种特殊的数据类型，用于执行原子操作。原子操作是不可分割的操作，可以确保在多线程环境中线程安全地执行。C++中的 std::atomic 提供了对原子操作的支持。（互斥量的加锁支持一个代码段的原子操作）

std::atomic<int> atomicInt(0); //声明+初始化

int value = atomicInt.load(); //读取

atomicInt.store(42); //修改

int increment = 5; //执行加法

int result = atomicValue.fetch_add(increment);

应用场景：多线程中的计数器（**智能指针**）、控制子线程的启动或停止

主线程接收客户端的请求

1. 通过 int epollfd = epoll_create(num)创建内核中的 epoll 事件表
2. 主线程往 epoll 内核事件表中注册监听的 socket 事件；并循环通过 epoll_wait 获取就绪的事件，再逐个进行处理。
3. 处理不同的事件

新用户连接事件：根据 ET/LT 模式接受连接，将用于接收 recv()、发送 send()信息的文件描述符放入 http 处理对象中进行初始化。

用户读事件 EPOLLIN：主线程从当前 socket 中循环读取数据，读入对应 http 对象的缓冲区；直到没数据可读再将该对象插入请求队列，由子线程异步执行请求的解析、处理操作。

用户写事件 EPOLLOUT：主线程往 socket 上写入服务器处理用户请求的结果。

异常、错误事件：直接关闭该连接；

线程池的创建

线程池的类 threadpool 中：

```
//thread_number 是线程池中线程的数量
    //max_requests 是请求队列中最多允许的、等待处理的请求的数量
    //connPool 是数据库连接池指针
threadpool(connection_pool *connPool, int thread_number = 8, int max_request = 10000);
m_threads=new pthread_t[m_thread_number];  //线程池中的线程 id 初始化
for(int i=0;i<thread_number;++i){
```

```
            //循环创建线程，并将工作线程按要求进行运行
            if(pthread_create(m_threads+i,NULL,worker,this)!=0){
                delete  []  m_threads;
                throw  std::exception();
            }
            //将线程进行分离后，不用单独对工作线程进行回收
            if(pthread_detach(m_threads[i])){
                delete[]  m_threads;
                throw  std::exception();
            }
        }
```

子线程循环执行静态私有成员函数 worker，worker 执行私有成员函数 run（）

```
void  threadpool<T>::run()
```

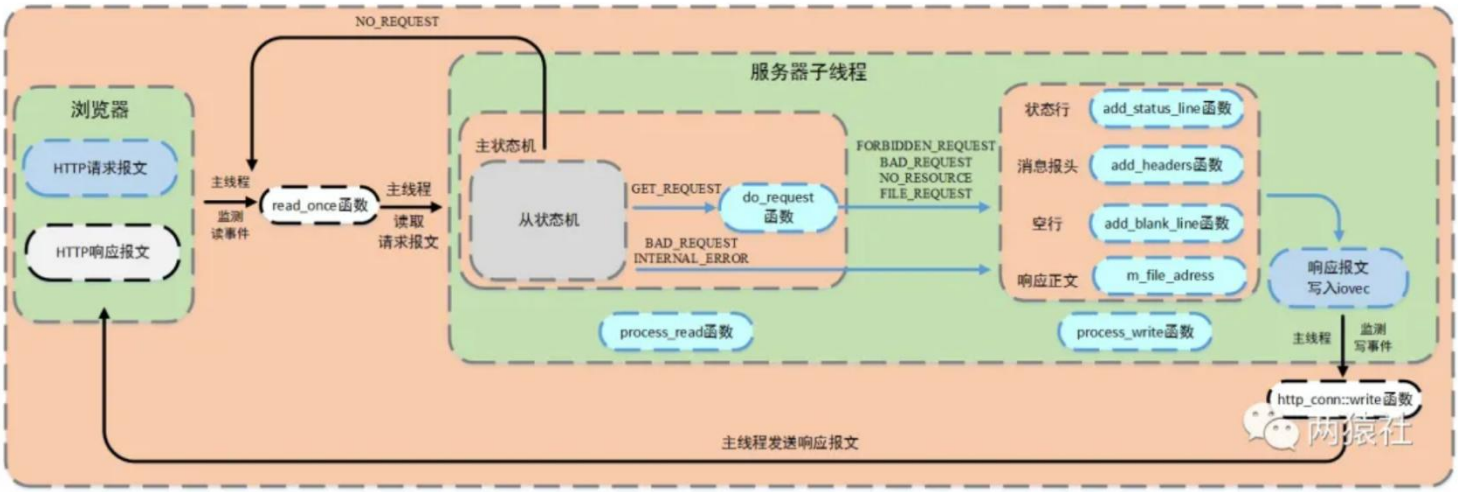
信号量等待、唤醒后添加互斥锁；
从请求队列中取出第一个任务，并将任务从请求队列删除；
解开互斥锁、信号量唤醒等待线程；
从数据库取出一个连接；
调用请求队列取出对象的处理方法 process 进行处理；
将数据库连接放回连接池

线程池所解决的问题：

需要频繁创建与销毁大量线程的情况下，由于线程预先就创建好了，接到任务就能马上从线程池中调用线程来处理任务，减少了创建与销毁线程带来的时间开销和 CPU 资源占用。

需要并发的任务很多时候，无法为每个任务指定一个线程（线程不够分），使用线程池可以将提交的任务挂在任务队列上，等到池中有空闲线程时就可以为该任务指定线程。

HTTP 请求报文接受、解析、响应方法：



1.请求接收：

主线程创建 http 对象数组 users=new http_conn[MAX_FD]、内核事件表 epollfd=epoll_create(5)，等待客户端的 http 请求。

浏览器端发出 http 连接请求，主线程通过 epoll 监听是否有事件发生，同步对所有就绪事件（连接请求事件、读事件、写事件、异常重置事件）进行处理：

连接请求事件：根据模式（LT、ET）执行 accept 接受连接请求、init 初始化 http 对象；然后往 epoll 内核事件表中注册该 socket 上的读写事件

读事件：连接 socket 上有读写事件发生，主线程利用 read_once 从 socket 上接收数据，直到无数据可读或对方关闭连接，读取到 m_read_buffer 中，并更新 m_read_idx，然后将该对象插入**请求队列**；此后工作**子线程异步**（会一直循环持续工作）从任务队列中加锁并取出一个任务进行处理。

写事件：当 socket 上有数据可写，epoll_wait 通知主线程，主线程调用 write 将数据写入 m_sockfd 往 socket 上写入服务器处理客户请求的结果。

各子线程通过对象中的 process 函数对任务进行处理，分别调用 process_read 函数和 process_write 函数分别完成**报文解析**与**报文响应**两个任务，随后注册 epollout 写事件。最后主线程执行写事件，将数据发送给浏览器端。

2.报文解析：

主状态机：三个状态分别用以标识解析**请求行**、**请求头**、**消息体**；

从状态机：循环读取对象中 m_read_buffer 所保存的数据，以供主状态机进行解析

process_read()：

初始化从状态机状态、HTTP 请求解析结果；

while 循环持续读取 m_read_buffer

(通过 switch-case，根据主状态机状态，跳转到该状态的解析方法中)

text = get_line()读取数据

switch(m_check_state)

case 请求行：

调用 http_conn::parse_request_line(text)
对读取的 text 进行处理，获得**请求方法** m_method、**协议版本号** m_version 和**请求的资源** m_url，
将主状态机状态转为请求头；

case 请求头：
调用 http_conn::parse_headers(text)
对读取的 text 进行处理，获得**连接方法** m_linger、**请求头部长度** m_content_length、host **字段** m_host，
判断为 pose 请求则将主状态机状态改为请求体；
为 get 请求则 do_request()执行报文响应函数

case 请求体：
http_conn::parse_content(text)
默认为**用户名和密码**，直接保存在 m_string 中
do_request()执行报文响应函数

返回解析后的状态 ret，以标识是否解析成功。

do_request()利用 stat 判断文件是否可以访问，再用 mmap 进行映射，通过 url 和本地目录，将请求的目标文件映射到内存逻辑地址，保存在对象的 m_file_address 变量中。

3.报文响应：

现已保存了解析 http 请求报文后的各类信息和**目标文件** m_file_address，要构建**响应报文** m_write_buf 并将其连同客户请求的资源一起回传：

```
process_write(HTTP_CODE  ret):
    switch(ret)
        case 500:
            //依次添加状态行、消息报头、空行和文本 content，退出 switch
        case 404:
            //依次添加状态行、消息报头、空行和文本 content，退出 switch
        case 403:
            //依次添加状态行、消息报头、空行和文本 content，退出 switch
        case 200:      //请求报文无误，文件存在
            //资源存在可访问时
                add_status_line(200,ok_200_title);     //添加状态行
                add_headers(m_file_stat.st_size); //添加消息报头（包括资源长度、是否长连接、空行）；
                将 io 向量机制 iovec 中第一个指针指向报文 m_write_buf，第二个指向目标文件 mmap 的地址 m_file_address;
                return true
            //资源为空或不可访问时
                //返回空白 html 文件
                const  char*  ok_string="<html><body></body></html>";
                add_headers(strlen(ok_string));
                return true
    //除 FILE_REQUEST 状态外，其余状态只申请一个 iovec，指向响应报文缓冲区
```

```
m_iv[0].iov_base=m_write_buf;
    m_iv[0].iov_len=m_write_idx;
    m_iv_count=1;
    return  true;
```

服务器子线程调用 **http_conn::process_write()** 和 **do_request()** 完成**响应报文（包括报文和数据）**，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。

http_conn::write()

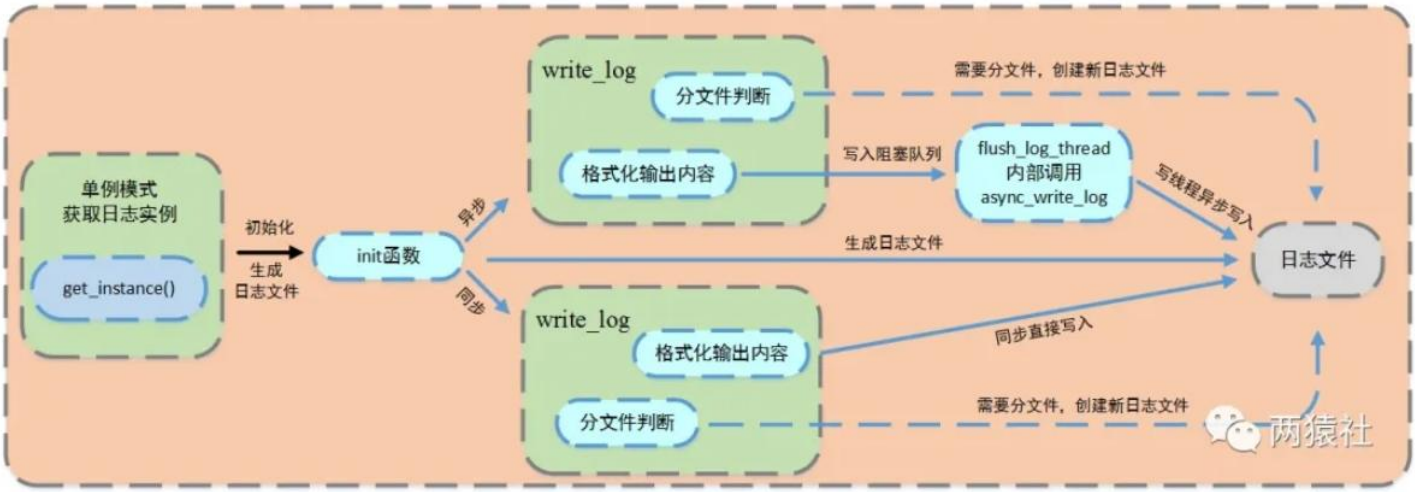
```
temp=writev(m_sockfd,m_iv,m_iv_count); //通过 writev()循环将 io 向量机制
                                        m_iv 存储的所有数据写入 m_sockfd 中；
modfd(m_epollfd,  m_sockfd,  EPOLLOUT);     //然后重新注册写事件，让主线
                                            程处理并发送该 socket 中的数据
```

通过 bytes_to_send 判断数据是否全部发送完毕

关闭 mmap 并重置 m_sockfd

若为长连接则重新初始化 HTTP 对象

日志系统



日志类 Log: 采用单例模式在服务器端创建唯一的一个日志系统，对服务器运行状态、错误信息和访问数据通过异步的方式记录在文件中。需要实现的具体方法有：

1）公有的日志实例获取方法 `static Log* get_instance()`

2）初始化日志文件方法 `bool init`(文件名，日志缓冲区大小，最大行数)

初始化写入的目标文件、阻塞队列、异步线程

3）公有的静态异步日志写入方法 `static void* flush_log_thread(void *args)`

函数内部调用私有方法 async_write_log() 用于**创建一个子线程**异步执行“从阻塞队列 pop 取一个日志 string，将其写入目标文件”操作

4）公有的标准日志内容生成方法 `Log::write_log(int level, const char *format, ...)`

函数传入需要记录的日志内容，执行时**判断**当前时间是否为创建日志的时间，行数是否超过最大限制，否则需要重新创建写入的文件，更新文件描述符。

其次按照**时间+内容**的格式在日志内容前添加时间戳，并将其通过 push 添加到阻塞队列中，让子线程**异步**完成写入操作。

自定义阻塞队列 block_queue： 用循环数组实现阻塞队列，根据生产者-消费者模型实现阻塞队列的 push()、pop()方法。

数据库连接池

数据库连接池类 connection_pool： 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

1）单例模式 `static connection_pool *GetInstance();` 获取唯一的数据库池实例

2）初始化方法 `void connection_pool::init(string url, string User, string PassWord, string DBName, int Port, unsigned int MaxConn)`

根据输入的主机地址 Url、数据库端口号 Port、登录数据库用户名 User、登录密码 Password、和使用的数据库名 DatabaseName 通过 **mysql_real_connect()方法**循环 MaxConn 次，将初始化数据库指针 **Mysql* con** 加入到链表连接池 **connList** 中。更新当前空闲连接数 FreeConn，同时初始化管理连接的信号量 `sem reserve;`

每一个处理 http 请求的**工作子线程**都会同时获取一个数据库连接，需要使用**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

3）从数据库连接池获取 mysql 连接

```
MYSQL* connection_pool::GetConnection()
    reserve.wait();//原子减 1
    lock.lock();  con = connList.front();  lock.unlock();  return con;
```

4）将连接释放回连接池

```
Bool connection_pool::ReleaseConnection(MYSQL  *con)
    Lock.lock();  connList.push_back(conn);  lock.unlock();
```

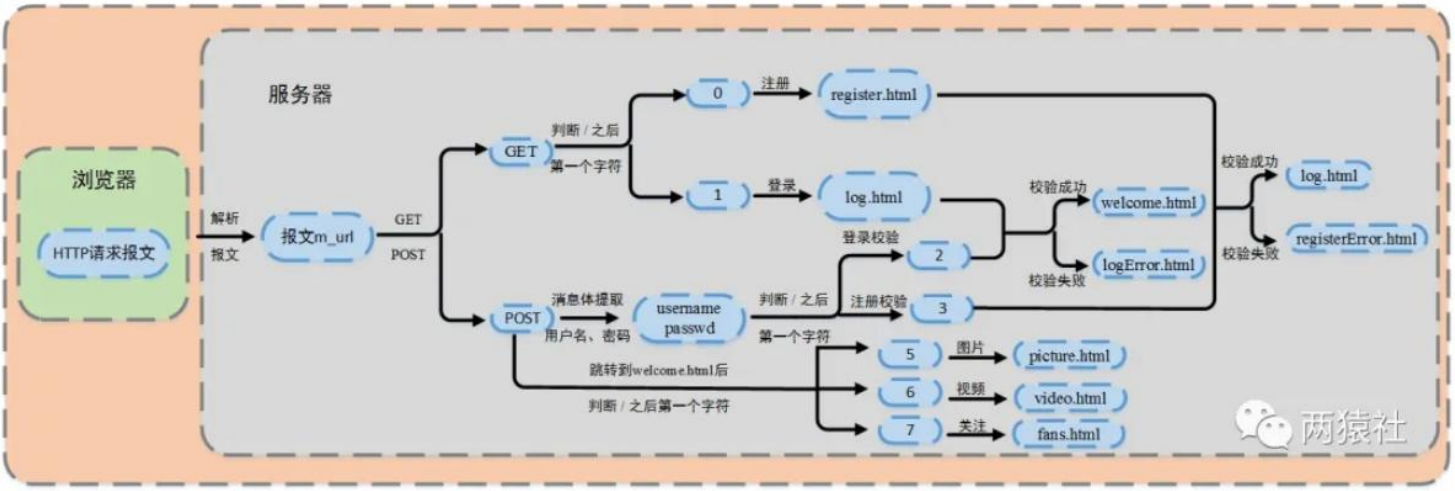
```
    reserve.post();//原子加 1
```

5）销毁数据库连接池：利用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。

```
void  connection_pool::DestroyPool()
    lock.lock(); //保证销毁操作的原子性
    for(auto it = connList.begin(); it != connList.end(); ++it){
        MYSQL* con = *it;
        mysql_close(con);
    }
    CurConn = 0; FreeConn = 0;  //将现有连接、现存连接全置 0
    connList.clear(); //清空 list
    lock.unlock();
```

最后，为了避免手动释放，而将数据库连接的获取与释放通过 RAII 机制封装为一个新的类 connectionRAII，类中通过创建好的**线程池**和**数据库连接指针**进行初始化，析构函数调用 DestroyPool 释放连接池。

注册与登录



子线程从阻塞队列中竞态获取 http 任务对象，并开始对每一个 **http 请求**进行解析：

如果是 get 请求，则直接返回 0：注册或 1：登录界面；

如果是 post 请求，则解析请求体中的用户和密码，再根据不同 url 返回相应的目标：

2：登录校验 执行“**返回登录成功/失败界面**”

3：注册校验 执行“**insert 成功调整到 login 界面/用户名已存在 insert 失败的** error 页面”。

5：返回图片页面

6：返回视频页面

7：返回关注页面

改进：

1）**功能方面：添加“修改密码”等界面**：post 请求中的 url 中指定代码为 8，并在请求体中附带用户名 username 和修改后的密码 password,解析 http 指令后对比数据库表中是否已存在该用户名，若存在则执行 **update password from user where username = username;**否则返回错误

2）**安全方面：http 协议可以修改为 https 协议**，加密请求中的用户及密码信息，提高可靠性。

3）异步日志通过单个子线程写入--可以修改为多子线程互斥写入

面试问答：

1.给你 40 亿的数据，再给你一个数 k，如何快速的确认数据中是否存在 k（百度）

采用分治的方法：利用多线程机制，将所有的数据分成 4000 份，每份有 106 个数据，将这些数据插入到等待队列之中，再根据硬件性能，创建多个工作线程，每个工作线程从等待队列中循环地取数据，逐个与 k 进行比较，当某个子线程找到时返回 true。

2.40 亿个数中如何高效地找到最大的一个数/最大的第 K 个数

同样采用分支的方法：利用多线程机制，将所有的数据分成 4000 份，每份有 106 个数据，将这些数据插入到等待队列之中，再根据硬件性能，创建多个工作线程，每个工作线程从等待队列中循环地取数据，直接遍历在 O(n)复杂度下找到最大值/建立小顶堆在 O(nlogm)复杂度下得到最大的前 k 个数；然后合并所有结果：比较所有最大值/合并所有子集的最大前 k 个数，同样建立小顶堆找全局的“最大的第 K 个数”

3.LCR 060 找数组中前 k 个高频元素：最高效的时间复杂度为 O(n)

通过快速排序的方法，每次都让 mid 与 k 比较位置，让“比 nums[k]都大的元素都在它的左边”

4.如何排查一条慢 SQL？可以从哪些方面入手？

引起慢 sql 的原因有：索引缺失或索引失效导致全表搜索，可以通过优化 sql 语句或使用 explain 指令去查看这个 sql 的执行计划，（排查不等查询、通配符、对索引使用函数导致索引失效的情况）

5.智能指针的原理、常用智能指针、实现

原理：智能指针是一个类，用于存储指向动态分配对象的指针，当类对象声明周期结束时，自动调用析构函数释放资源，防止堆内存泄露。

常用：1）share_ptr 共享指针，采用引用计数器的原理，允许多个智能指针指向同一个对象，每多一个时引用计数加 1，少一个时调用析构函数使引用计数减 1，计数为 0 时回自动释放动态分配的资源。2）unique_ptr 独占指针，一个非空的 unipue_ptr 拥有它所指向的资源。不支持普通的拷贝和赋值操作，如果强行拷贝，在释放资源时会造成对同一内存指针多次释放而导致程序崩溃。3）weak_ptr 弱引用指针，与 share_ptr 配套使用防止循环引用，它指向一个由 share_ptr 管理的对象而不影响所指对象的生命周期，即它只引用不计数。这种特性导致它不保证所指的内存一定是有效的。

6.内存泄露

一般来说内存泄漏是指堆内存泄漏，在程序运行时动态分配的内存块在运行结束之后没有显示释放而导致的泄露。要将基类的析构函数声明为虚函数（当基类指针指向派生类时，若积累析构函数不是虚函数，在析构时只会调用基类而不会调用派生类的析构函数，导致内存泄露），由 new 就有 delete，有 malloc 就有 free；使用智能指针；建立的 socket 连接、数据库连接也要在程序结束时手动释放。

7.数据库的备份和容灾

数据库**备份**是指定期将数据库的副本创建到另一个位置，一边在发生数据丢失或数据库崩溃时进行恢复。可以存储在远程服务器。

数据库**容灾**是指在数据库主服务器发生故障不可用时，确保数据库系统继续运行并提供服务的能力。通过准备**冗余硬件、多个数据库实例、数据复制**等方法，让**备用服务器可以接管**继续提供服务。

8.Linux 如何查看线程内存、cpu 占用情况

首先通过 top 命令找到进程对应的**进程 pid**

通过 **top -H -p 进程 pid** 查看线程　　cat /proc/进程 pid/status 查看进程

9.浮点数是怎么存储的

float 类型：

单精度浮点数 float(32bit)表示为： SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM

1）符号位(S)1 bit;

2）阶码(E)移码表示 8 bit，阶码的偏移量为 127;
3）尾数(M)原码表示 23 bit，用小数表示，小数点放在尾数域的最前面

$$\text{float} = (-1)^{S} * (1.M) * 2^{(E-127)}$$

双精度浮点数 double(64bit)
1)符号位(S)1 bit;
2)阶码(E)11 bit，阶码的偏移量为 1023(3FFH);
3)尾数(M)52 bit，用小数表示，小数点放在尾数域的最前面.

$$\text{double} = (-1)^{S} * (1.M) * 2^{(E-1023)}$$

10.并行和并发的区别

并发：指**一个处理器同时处理多个任务**，(通过进程调度算法),实现逻辑上的同时发生。

并行：**多个处理器或是多核处理器同时处理多个不同的任务**，是物理上同时发生。

11.讲一讲 lambda 函数的全部知识

Lambda 相当于编程中的一种语法糖，如果定义了一个 lambda 表达式，编译器会自动生成一个类，在该类中重载了 operator()。 对于 STL 的 sort，每次为了实现一个 algorithm 算法， 都要重新去写一个类，如果每次比较的逻辑不一样，还要去实现多个类，特别是相同类的命名，这些都给编程者带来了极大的不便。因此，在 C11 语法中出现了 Lambda 表达式。 sort(vc.begin(), vc.end(), [](Date& d1,Date& d2){ return d1 < d2; });

```
[capture] （parameters） mutable ->return-type {statement};
即 [捕获列表：值捕获、引用捕获](参数)mutable -&gt; 返回类型 {函数体}

auto plus = [] (int v1, int v2) -> int { return v1 + v2; }
int sum = plus(1, 2);
```

12.解释一下左值和右值

左值：可以出现在赋值运算符的左边、可以取地址、且有名字;
右值：出现在赋值运算符的右边、不可以取地址、没有名字;

13.解释一下浅拷贝和深拷贝

浅拷贝：复制的对象中引用了外部内容（如分配在堆上的数据），在复制这个对象的时候会让新旧两个对象指向同一个外部内容。

深拷贝：复制的对象会引用外部对象的独立复制。

14.push_back 和 emplace_back()的区别是什么？

当动态数组 vector<int> nums 中的空间足够时 **emplace_back 会比 push_back 少一次构造和析构操作：**

1. nums.push_back(A(1))传入右值或 int a = 1; nums.push_back(a);传入左值都会执行：

1）.调用构造函数构造出值 A；

2）.调用拷贝构造函数讲 A 拷贝到 nums 尾部数据空间

3）.调用 A 的析构函数

2. nums.emplace_back(1);传入右值时会执行：

1）.调用构造函数直接在 nums 的尾部的内存地址处构造对象。

还可以通过 std::move()函数将一个左值强制转化为右值引用，高效移动资源

```
int num = 2;
nums.push_back(move(num));
```

注意，在转移语义后 num 内不再存储任何东西。

15.如何设计一个高性能、高并发、高可用、高可靠、可扩展的系统

https://top.interviewguide.cn/issue/736

1).使用 CPP 提供的高性能封装的操作函数，如 **emplace_back()、move()**等

2).可以使用操作系统提供的 **mmap 内存映射**，提高数据的传输效率：内存映射将用户空间的一段内存区映射到内核空间，用户和内核都可以直接对空间里的数据进行修改，在减少了“磁盘数据->内核空间->用户空间”中最后一步

3).池化技术：创建线程池、数据库/TCP 连接池、对象池等，提高对象的复用，减少重复创建对象、销毁对象所需的系统开销。

4）.高效利用缓存：基于空间换时间的方法，解决高并发读情景。针对读密集型、热点数据进行缓存；还可以使用分布式缓存，将缓存数据通过 hash 算法分布在多台机器上，进一步缓解压力。

5）.异步化：在处理耗时的任务时，采用同步等待的方式会严重降低系统的吞吐量，可以通过异步化进行解决。

16.http 和 https 的区别

1）、http 协议传输的数据都是未加密的，也就是明文的，因此使用 HTTP 协议传输隐私信息非常不安全， HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全。

2）、https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。

3）、http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443

17.长连接和短连接，它们各有什么优缺点？

短连接

客户端和服务器每进行一次 http 连接就会建立一次 tcp 连接，任务结束就中断

优点：管理简单，不要额外的控制手段

缺点：客户端请求频繁时，将在 tcp 的连接和关闭操作上浪费时间和带宽。

长连接：

当一个网页打开完成后，客户端和服务器之间用于传输 http 数据的 TCP 连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接

优点：可以省去较多的 TCP 建立和关闭的操作，减少浪费，节约时间，适用于频繁请求资源的用户

缺点：存在存活功能的探测周期长的问题，遇到恶意连接时保活功能不够；随着客户端连接的增多，服务器可能扛不住，需要采取额外的策略来管理连接。

自我介绍

我叫杨礼铭，目前就读与华南理工大学计算机学院，研究生年级，研究方向**多模态融合的移动机器人导航方法**，主要解决机器人在自然语言指令下进行导航所产生的问题。
在校期间我参加了研电赛，并获得全国总决赛二等奖和安谋专项企业奖一等奖；
还参加了昇腾 AI 创新大赛，获得了广州赛区铜奖；
也参与过学院评奖评优，获得过研究生腾讯企业一等奖学金、优秀学生二等奖学金
我认为自己有一定的信息检索能力，遇到的技术问题基本能通过国内外论坛解决；
并且具有良好的学习能力，可以在短时间内学习项目需要的技术并实践；

hr 面提问：

问一下你的学校经历:除了在实验室跟的几个项目和自己找的开源项目以外，还参加了中国研究生电子设计竞赛，和其他成员从 0 到 1 搭建了整套系统，拿了全国技术赛

道二等和企业赛道一等两个奖，还参加了昇腾 ai 创新竞赛，拿了省三等奖，发明一项专利，发表一篇论文。

遇到的最困难的问题是什么，如何解决的：在确定参加研电赛后的那段时间是比较困难的，从最开始的选题、功能设计、系统设计，到具体的开发实现和后续的测试各个环节都充满了大大小小的问题，还记得在第一次系统测试的时候，发现了很严重的时延，导致测试的结果很不理想，这个问题又涉及了通信、识别模型、机器人导航多个模块，我们三个人一起花了整整 3 天时间去测试定位问题、并设计多个方案最终才解决问题。但回过头来看，我更认为这是一种很好的磨练与挑战，提升我的编程硬能力和团队协作软能力。

职业规划是什么

有没有一个自己独立着手的项目，以后想找哪些方面的工作等。

有没有实习：春招的时候是有想过要去实习，但后面跟导师深入沟通后，我个人觉得在校这段可以让我巩固基础知识的同时，继续深挖我的研究方向的大片时间是不会再有了，就没有随大流去参加实习的浪潮，也让我能有十足的把握去参加现在的秋招。

为什么选择北京工作：虽然我父母是在广东发展，但我有很多其他亲戚都是在北京生活，我也有很多本科同学在北京工作或读研，生活方面有人帮忙也不是问题。而且同样做为超一线城市，我个人认为北京有更多机会与可能，而且从人文环境、经济环境、生活习惯等多方面考虑来说都很吸引我，所以我会选择北京作为我的目标城市。

是不是独生子女：不是独生子女，父母健康，这也是我能够选择到北京就业的原因之一