buffer

即每次send()不一定会发送完，没发完的数据要用一个容器进行接收，所以必须要实现应用层缓冲区。

vectorh缓冲区，设置原子类型的读写索引，对外借口为读取或写入某个fd。

从某个连接接受数据的时候，有可能会超过vector的容量，所以我们用readv()来分散接受来的数据。

在非阻塞网络编程中，如何设计并使用缓冲区？一方面我们希望减少系统调用，一次读的数据越多越划算，那么似乎应该准备一个大的缓冲区。另一方面，我们系统减少内存占用。如果有 10k 个连接，每个连接一建立就分配 64k 的读缓冲的话，将占用 640M 内存，而大多数时候这些缓冲区的使用率很低。muduo 用 readv 结合栈上空间巧妙地解决了这个问题。

在栈上申请一个65536字节的stackbuf，利用readv读取数据，iov有两块，第一块指向 muduo Buffer 中的 writable 字节，另一块指向栈上的 stackbuf。这样如果读入的数据不多，那么全部都读到 Buffer 中去了；如果长度超过 Buffer 的 writable 字节数，就会读到栈上的 stackbuf 里，然后程序再把 stackbuf 里的数据 append 到 Buffer 中。

利用临时的栈上空间，避免每个连接开辟巨大buffere造成的内存浪费，也避免u反复调用read()的系统调用开销

log

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。

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

阻塞队列使用锁和条件变量，push时如果没有空余空间，就等待在生产者条件变量，pop时如果没有元素就等待在消费者条件变量

日志采用单例模式，分为懒汉模式和饿汉模式。 懒汉模式： 顾名思义，非常的懒，只有当调用getInstance的时候，才会去初始化这个单例。其中在C++11后，不需要加锁，直接使用函数内局部静态对象即可。

```
class TaskQueue {
public:
    static TaskQueue* getInstance() {
        if (m_taskQ == nullptr) {
            lock();
            if (m_taskQ == nullptr) {
                m_taskQ = new TaskQueue();
            }
            unlock();
        }
        return m_taskQ;
    }
private:
```

```
    TaskQueue() = default;
    TaskQueue(const TaskQueue&);
    TaskQueue& operator=(const TaskQueue&);
    static TaskQueue* m_taskQ;
};
```

静态局部变量实现的懒汉模式:

```
class TaskQueue {
public:
    static TaskQueue* getInstance() {
        lock();
        static TaskQueue instance;
        unlock();
        return &instance;
    }
private:
    TaskQueue() = default;
    TaskQueue(const TaskQueue&);
    TaskQueue& operator=(const TaskQueue&);
};
```

饿汉模式： 即迫不及待，在程序运行时立即初始化。饿汉模式不需要加锁，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。

```
class TaskQueue {
public:
    static TaskQueue*(){
        return taskQ_;
    }
private:
    TaskQueue() = default;//私有化构造函数\拷贝构造函数\赋值运算符
    TaskQueue(const TaskQueue&);
    TaskQueue& operator=(const TaskQueue&);
    static TaskQueue* taskQ_;
};
```

TaskQueue* TaskQueue::taskQ_= new TaskQueue//类外定义并初始化静态成员变量，这句代码在程序启动时就执行，立即创建了 TaskQueue 的唯一实例

日志函数使用宏和可变参数列表实现

```
log->write(level, format, ##__VA_ARGS__);
va_list vaList;
va_start(vaList, format);
int m = vsnprintf(buff_.BeginWrite(), buff_.WritableBytes(), format,
```

```
vaList);
va_end(vaList);
```

ConnPool

连接池采用生产者消费者模型，采用锁+信号量，预先产生多条连接std::queue<MYSQL *>。

http释放时把连接释放回连接池，先加锁锁定内存空间，然后push，然后sem_post增加信号量。

http取连接时从连接池消费连接。等待在信号量上，当信号量大于0时加锁继续取连接。

连接池采用RAII思想，获取资源即初始化，避免资源泄露。智能指针就是经典的RAII。

```
class SqlConnRAII {
public:
    SqlConnRAII(MYSQL** sql, SqlConnPool* connpool) {
        assert(sql);
        sql_=connpool->GetConn();
        *sql = sql_;
        connpool_=connpool;
    }
    ~SqlConnRAII() {
        if(sql_){
            connpool_->FreeConn(sql_);
        }
    }
private:
    MYSQL* sql_;
    SqlConnPool* connpool_;
};
```

当需要创建或得到sql*时：

```
MYSQL *sql;
SqlConnRAII(&sql,SqlConnPool::Instance());
```

ThreadPool

工作队列和线程池分别为生产者消费者，生产者队列用std::queue<std::function<void()>>,线程在初始化线程池时创建匿名线程，回调函数为死循环取任务，注意：被唤醒后判断任务是否已经被取走\线程池是否已经关闭。执行回调函数前需要释放锁，执行完毕后再拿锁。

向工作队列添加任务函数采用模板，使用万能引用+完美转发，开销最小：无论是左值还是右值都是不会拷贝：左值（引用），右值（移动） 若使用值传递，左值会触发拷贝构造函数。若使用万能引用，左值传递引用，无需拷贝，右值触发移动语义。

```
template<typename T>
void wrapper(T&& arg) {
    target(std::forward<T>(arg));  // 根据原始类型调用正确重载
}
```

根据引用折叠，若传入左值（如 int a），T 推导为 int&，形参类型折叠为 int&（左值引用）。若传入右值（如 42），T 推导为 int，形参类型为 int&&（右值引用）。 即使形参是右值引用，在函数内部具名变量 arg 仍是左值（因其有名称和持久地址）。采用forward能恢复参数的原始值类别（左值/右值）根据 T 的推导结果，std::forward(arg) 返回： 若 T 是左值引用（如 int&），返回左值引用。 若 T 是普通类型（如 int），返回右值引用。

信号量和条件变量 信号量更适用

- 控制固定数量的资源（如数据库连接池限流）
- 简单同步场景（如仅需限制并发线程数）

条件变量更适用

- 条件复杂的等待（如缓冲区有数据且未被锁定）
- 需广播通知所有等待线程（如线程池任务分配）

条件变量通常更轻量，但需配合锁；信号量涉及系统调用，开销较大

```
//消费
sem_wait(&semId_);
std::lock_guard<std::mutex> locker(mtx_);
connQue_.pop();
//生产
std::lock_guard<std::mutex> locker(mtx_);
connQue_.push(conn);
sem_post(&semId_);
```

timer

网络编程中除了处理IO事件之外，定时事件也同样不可或缺，如定期检测一个客户连接的活动状态、游戏中的技能冷却倒计时以及其他需要使用超时机制的功能。我们的服务器程序中往往需要处理众多的定时事件，因此有效的组织定时事件，使之能在预期时间内被触发且不影响服务器主要逻辑，对我们的服务器性能影响特别大。

我们的web服务器也需要这样一个时间堆，定时剔除掉长时间不动的空闲用户，避免他们占着茅坑不拉屎，耗费服务器资源。

一般的做法是将每个定时事件封装成定时器，并使用某种容器类数据结构将所有的定时器保存好，实现对定时事件的统一管理。常用方法有排序链表、红黑树、时间堆和时间轮。这里使用的是时间堆。

时间堆的底层实现是由小根堆实现的。小根堆可以保证堆顶元素为最小的。

堆包括两个主要成员：ref和heap，ref存储fd与其在heap中的位置，实现O(1)查找。

堆有两个基本操作，up和down

- up 自尾节点和父节点比较，移动到父节点
- down 和两个孩子比较，与最小的孩子交换，移动到这个孩子

定时器堆对外提供adjust、add和GetNextTick

- adjust(int id, int newExpires) adjust将指定id的timer设置为新的超时时间，对新的timer执行down函数。
- add(int id, int timeOut, const TimeoutCallBack& cb) 如果有这个timer存在，就设置为新的超时时间；如果不存在，新建一个timer，放到heap最后，执行up函数。
- GetNextTick() tick一下，pop出所有超时的timer，将这些超时timer的回调函数记录下来，计算下次超时时间为顶部timer的超时时间，释放锁。执行所有回调函数关闭连接。

  主Reactor会添加timer，从Reactor会调整timer，因此需要加锁。

  设置超时时间为60s，60s后这里的超时时间非常短，超时触发太频繁，可以容忍一些连接未关闭。

http

http主要完成对http解析和应用层部分，每个http对象封装一个用户相关数据。对外提供read、write和process分别完成读、解析处理、写操作。包括httprequest、httpresponse、readBuff、writeBuff

httpconn类完成对fd的管理，完成读写。调用buffer类的读写完成LT\ET模式读写。读入请求报文后解析，解析完毕后进行应用层操作，然后将响应写入响应报文，写入buffer，完成响应。

读写通过do...while(isET)完成ET\LT模式，读在readbuff内部实现分块读。写在httpconn实现分块写，响应行和响应头放在writebuff中，进而放在iov[0]，响应体中请求的文件（若有）放在iov[1]中。一块传输buff里面的内容，另一块传输内存映射的文件指针。

写的逻辑：

```
do{
    len=writev(fd_,iov_,iovCnt_);
    if(len<=0){
        *saveErrno = errno;
        break;
    }
    if(iov_[0].iov_len + iov_[1].iov_len == 0){
        break;
    }else if(static_cast<size_t>(len) > iov_[0].iov_len){
        //写入长度超过第一个缓冲区
        iov_[1].iov_base=(uint8_t*)iov_[1].iov_base + (len-
iov_[0].iov_len);
        iov_[1].iov_len -= (len - iov_[0].iov_len);
        if(iov_[0].iov_len){
```

```
            //响应头已经发送完毕，释放writeBuff_
            writeBuff_.RetrieveAll();
            iov_[0].iov_len=0;
        }
    }else{
        //写入长度未超第一个缓冲区
        iov_[0].iov_base=(uint8_t*)iov_[0].iov_base + len;
        iov_[0].iov_len-=len;
        writeBuff_.Retrieve(len);
    }
}while(isET || ToWriteBytes() > 10240);
```

请求报文的解析采用有限状态机，分别解析请求行、请求头和请求体，将其中的内容保存到类内相关成员，如方法、url、版本、相关状态等。

解析请求体中应用层相关内容，如解析application/x-www-form-urlencoded获取登录、注册信息，进而进入应用层。

这里可以考虑应用层单定义一个类，作为接口，后续开发网盘业务使用。

完成解析请求行后，状态转为HEADERS，完成解析请求头后，状态转为BODY，完成解析请求体后，状态转为FINISH。parse中当状态不为FINISH以及readbuffer中还有可读数据时进行循环处理请求的每一行，根据状态使用特定的解析函数。

响应报文的构造，主要使用MakeResponse，根据请求文件是否可以获取得到响应码，然后添加响应行、响应头和响应体。响应体中包括两部分，响应行和响应头放在writebuff中，进而放在iov[0]，响应体中请求的文件（若有）放在iov[1]中。

```
int* mmRet=(int*)mmap(0,mmFileStat_.st_size,PROT_READ,MAP_PRIVATE,srcFd,0);
mmFile_=(char*)mmRet;
//mmFile_即对应的文件地址，直接可以读取的
```

这里可以考虑尝试sendFile()

这里采用内存映射的零拷贝方式，避免把文件拷贝o到内核，再拷贝到用户空间，再拷贝到内核，再拷贝到网卡四次拷贝。 内存映射mmap允许程序直接在用户态中访问内核空间中的数据，这样能避免一次无意义的Copy，建立共享映射后，就不需要从内核缓冲区拷贝到用户缓冲区了，这就避免了一次拷贝了。

sendfile（两次上下文切换，最少两次数据拷贝）sendfile() 系统调用在两个文件描述符之间直接传递数据（完全在内核中操作），从而避免了数据在内核缓冲区和用户缓冲区之间的拷贝，操作效率很高

Linux 2.1 版本提供了 sendFile() 函数：数据根本不经过用户态，直接从内核缓冲区进入到 Socket Buffer 中；同时由于完全和用户态无关，就减少了一次上下文切换

2.3、mmap 和 sendfile 的区别

- 都是 Linux 内核提供，实现零拷贝的 API
- mmap 适合小数据量读写，sendFile() 适合大文件传输

- mmap 需要 4 次上下文切换，3 次数据拷贝
- sendFile() 需要 3 次上下文切换，最少 2 次数据拷贝
- sendFile() 可以利用DMA 方式，减少CPU 拷贝；mmap 则不能（必须从内存缓冲区拷贝到Socket 缓冲区） 基于此基础，RocketMQ 使用了 mmap；Kafka 使用了 sendFile()

server

server包括epoll方法封装的epoller类，主Reactor的webserver类和从Reactor的subreactor类。

Epoller提供AddFd、ModFd、DelFd、Wait，完成对epoll_ctl和epoll_wait的封装。包括int epollFd和std::vector events。通过accept将要监听的fd和时间存储在events中。

下面是一些检测事件：

```
POLLIN ：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。
1. 客户端直接调用close，会触犯EPOLLRDHUP事件
2. 通过EPOLLRDHUP属性，来判断是否对端已经关闭，这样可以减少一次系统调用。
```

webserver作为主Reactor，对外开放为服务器的入口。

webserver的初始化完成自身参数设置、socket初始化、EventMode初始化、log初始化、连接池初始化、线程池初始化、subReactor初始化（创建启动每个subReactor）。

socket初始化：

```
struct sockaddr_in addr;
addr.sin_family=AF_INET;
addr.sin_addr.s_addr=htonl(INADDR_ANY);
addr.sin_port=htons(port_);
listenFd_=socket(AF_INET,SOCK_STREAM,0);
ret = setsockopt(listenFd_, SOL_SOCKET, SO_LINGER, &optLinger,
sizeof(optLinger));
ret=setsockopt(listenFd_,SOL_SOCKET,SO_REUSEADDR,(const
void*)&optval,sizeof(int));
ret=bind(listenFd_,(struct sockaddr*)&addr,sizeof(addr));
ret=listen(listenFd_,1024);
ret=epoller_->AddFd(listenFd_,listenEvent_ | EPOLLIN);
SetFdNonblock(listenFd_);
```

主Reactor只负责监听新的连接，只需while(1)->epoll_wait->DealListen_();DealListen_()中同样利用do...while实现监听的ET\LT模式，accept到新的连接后AddClient_到某个subReactor中，这里采用随机分配和哈希分配方法。

SubReactor负责对每个连接进行读写关闭，包含了实际的用户数据和连接的映射std::unordered_map<int, HttpConn> users_及其timer std::unique_ptr timer_。

SubReactor启动后进入loop循环，完成对其负责的用户的读写。这里考虑主Reactor添加用户，导致subReactor资源的竞争问题。

```
void SubReactor::Start() {
    isRunning_ = true;
    thread_ = std::thread([this] { Loop(); });
}
```

loop循环：

大量短连接时，非常多的timer依次到达超时时间，每次GetNextTick获取最近的超时时间非常短，等待时间过短，cpu空转严重，因此设置timeMS至少为100ms，或改进计时器结构

```
void SubReactor::Loop() {
    int timeMS = -1;
    while (isRunning_) {
        // 处理计数器，清除超时timer，返回最近的超时时间
        // 容易发生雪崩，频繁处理计时器，因此设置timeMS至少为100ms，或改进计时器结构
        if(timeoutMS_ > 0) {
            timeMS = timer_->GetNextTick();
        }
        //若无请求到来就等待timeMS后唤醒，
        //-1代表一直等待，
        //0代表立刻返回（易造成cpu空转）
        int eventCnt = epoller_->Wait(timeMS);
        for (int i = 0; i < eventCnt; ++i) {
            //依次处理有请求的fd
            int fd = epoller_->GetEventFd(i);
            uint32_t events = epoller_->GetEvents(i);

            auto it = users_.find(fd);
            if (it == users_.end()) continue;
            // 没有这个用户了，连接已关闭
            if (events & EPOLLIN) {
                // 接收可选线程池处理业务，当业务耗时较长时采用线程池处理业务
                if (openThreadPool_) {
                    ThreadPoolDealRead_(&it->second);  // 线程池处理业务
                }else{
                    OnRead_(&it->second);
                }
            }
            else if(events & EPOLLOUT){
                OnWrite_(&it->second);
```

```
            }
            else if (events & (EPOLLRDHUP | EPOLLHUP | EPOLLERR)) {
                CloseConn_(&it->second);  // 连接关闭事件直接处理
            }
        }
    }
}
```

读函数首先更新计时器为新的超时时间，然后通过调用httpconn的read函数，然后调用处理函数。

处理函数处理这个连接，完成解析和业务，需要写就设置EPOLLOUT，否则设置EPOLLIN。

写函数也是更新计时器，然后调用httpconn的write函数，如果写完修改epoll为EPOLLIN，如果没写完返回了EAGAIN，继续设置EPOLLOUT（因为这里设置了EPOLLONESHOT所以要重新设置，将fd添加到epoll）

添加用户包括初始化httpconn、timer、epoller，注意顺序，先初始化资源再注册epoll监听。

遇到的问题

添加用户由主Reactor线程负责，读写由从Reactor负责，会发生资源竞争问题，需要对从Reactor的资源加锁或通过无锁结构实现。

这个问题最初是把程序改成多reactor后发生的，表现为timer->adjust中，还没有初始化timer就发生了查找timer,原因是添加timer和添加epoll顺序反了，先添加了epoll,此时从reactor就收到请求，要调整timer,而主reactor还没添加timer,发生了报错。

调换顺序后，有时还是报错，发现仍热是主从reactor之间的资源竞争问题。最初考虑加锁方案，但这样就要对所有资源加锁，考虑到从reactor负责读写，需要高并发，因此采用无锁方式。

因此考虑采用无锁队列实现主reactor向从reactor添加用户，避免了主从reactor之间的资源竞争。

无锁队列

无锁队列（Lock-Free Queue）是一种不使用锁机制（如互斥锁或读写锁）来实现线程安全的数据结构，是lock-free中最基本的数据结构。它通过复杂的原子操作（如CAS操作，Compare-And-Swap，通过比较内存中的值与预期值是否相等来实现原子操作）来确保在多线程环境下的正确性和一致性。无锁队列的设计目标是在高并发场景下提供高性能的入队和出队操作，避免了锁机制带来的性能开销和潜在的死锁问题。

CAS 操作包含三个操作数：

- 内存位置（V）：需要更新的变量的内存地址。
- 期望值（A）：当前线程认为变量应该具有的值。
- 新值（B）：如果变量的当前值等于期望值，则将其更新为新值。

CAS 的操作逻辑如下：

- 如果内存位置 V 的当前值等于期望值 A，则将 V 的值更新为新值 B。
- 如果 V 的当前值不等于 A，则说明其他线程已经修改了 V，操作失败。
- CAS 是一个原子操作，由硬件（通常是 CPU）直接支持，确保在多线程环境下不会出现竞争条件。

ABA 问题：CAS 可能会遇到 ABA 问题，即变量的值从 A 变为 B 又变回 A，CAS 无法检测到这种变化。可以通过版本号或标记来解决。

实现了单生产者-单消费者（SPSC）的无锁环形队列，无锁队列采用循环链表实现，头尾使用atomic类型。

```
head_.load(std::memory_order_acquire)
禁止重排序：当前线程中所有后续的读写操作不会被重排到此操作之前。
可见性保证：若其他线程通过 release 写入同一原子变量，则当前线程能看到该线程在 release 之前的所有修改（包括非原子变量）

tail_.store(next_tail, std::memory_order_release)
禁止重排序：当前线程中所有先前的读写操作不会被重排到此操作之后。
可见性发布：此操作前的所有修改（包括非原子变量）对其他线程通过 acquire 读取同一原子变量时可见
```

现代CPU缓存以 64字节缓存行（Cache Line） 为单位加载数据。若两个线程频繁修改的变量（如 head_ 和 tail_）位于同一缓存行，即使它们逻辑独立，也会因缓存一致性协议（如MESI）导致：

- 一个线程修改变量 → 整个缓存行失效 → 其他线程需从内存重新加载 → 性能骤降（10倍以上）

alignas(64) 强制变量对齐到64字节边界，确保每个变量独占一个缓存行，避免伪共享：

```
alignas(64) std::atomic<size_t> head_{0};   // 缓存行对齐，避免伪共享
alignas(64) std::atomic<size_t> tail_{0};
```

主Reactor通过PushClient向无锁队列中添加，添加完后通过epoll通知从reactor，每个从reactor拥有一个notify_fd_，如果监听到notify_fd_就执行把无锁队列中的内容拷贝出来，循环结束执行所有用户添加操作。

closeConn_占比过大问题

perf工具支持生成堆栈跟踪数据，利用perf进行性能测试：

- 使用perf record命令记录程序运行时的堆栈数据。例如，下面的命令会以每秒99次的频率对整个系统进行采样，并生成包含调用堆栈数据的perf.data文件：

  ```
  perf record -F 99 -g ./mywebserver
  ```

- 将数据转换为火焰图格式 使用perf script将perf.data文件中的数据导出为文本格式，以便后续处理：

  ```
  perf script > out.perf
  ```

这条命令会将perf.data中的采样数据转换为堆栈跟踪格式并保存为out.perf文件。

- 生成火焰图： 要生成火焰图，需要使用Flamegraph工具。假设你已经从GitHub下载并解压了Flamegraph工具包，可以通过以下命令将out.perf文件转换为火焰图：

```
./stackcollapse-perf.pl out.perf > out.folded
./flamegraph.pl out.folded > flamegraph.svg
```

发现CloseConn_占比过大（23%），因此将这个函数放到系统线程池运行，占用降低到了11%。

超时计时器频繁触发，引发雪崩

原本计时器的逻辑是每次循环计算最近超时的时间，然后设置epoll等待时间。最初超时时间设置60s,这样的话，等到60s后，超时计时器会频繁触发，通过设置最小超时时间和最少超时个数，比如把超时时间设置为第64个超时的计时器。

log花时间太长