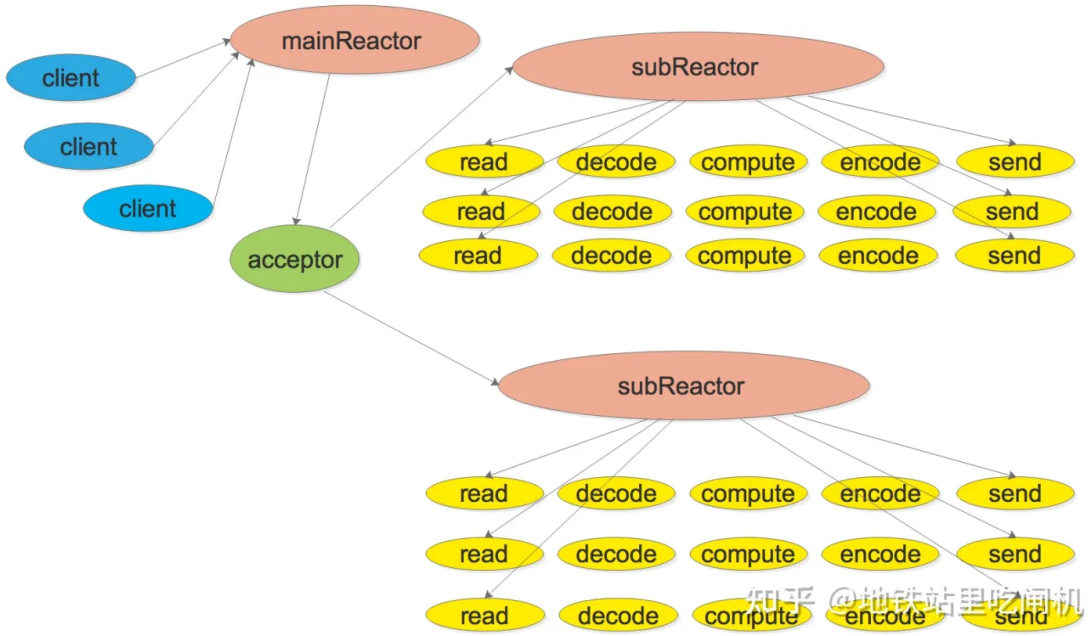
网络库

标签	进行中
更新时间	2023/01/15 12:28
新增属性	

一、概述

1、Multi-Reactor 概述

- muduo 网路库架构采用的是 Reactor 模式，而且将该模式分为多个类，并支持线程池实现多线程并发处理。



- Reactor 是一种服务器设计模式，事件处理模型，事件驱动的反应堆模式。
- 反应堆就是：事件来了就执行，但可能有很多类型的事件，所以需要提前注册好不同类型的事件处理函数以便事件到来时方便执行。
- 事件到来则由 `epoll_wait` 获取同时到来的多个事件，然后根据不同的类型将事件分发给不同的事件处理器，就是提前注册的事件处理函数

- Reactor 作为事件驱动的模式，是通过到来的事件来自动调用相应函数的，函数是由 Reactor 去调用的，而不是在主函数中调用的，所以大量运用了回调函数。

- 该模式的核心思想为将所有要处理的 IO 事件注册到一个 IO 多路复用器上（通常为 epoll 系统调用，我们也只使用 epoll），同时主线程阻塞在多路复用器上，如 `epoll_wait`，一旦有 IO 事件就绪或到来，则返回并将注册好的相应的 IO 事件分到对应的处理器上。
- 简略流程：

  1.注册事件和对应事件处理器

  2.多路复用器等待事件到来

  3.事件到来，事件分发器分发事件到处理器

  4.事件处理器处理事件，然后注册新的事件

2、Multi-Reactor 架构的三大核心模块介绍

2.1 概述

- muduo 库有三个核心组件支撑一个 Reactor 实现持续的监听一组 fd，并根据每个 fd 上发生的事件调用相应的处理函数。
- 三个组件分别是 Channel 类，Poller，EpollPoller 类和 EventLoop 类。

2.2 Channel 类

2.2.1 Channel 类概述

- Channel 类其实相当于一个文件描述符 fd 的保姆

- 作为一个 IO 事件分发器，封装了发生 IO 事件的文件描述符，关心的事件类型，以及事件相应的回调函数。

- Channel 对 fd 和事件进行了一层封装。平常我们写网络编程的相关函数，基本就是创建套接字，绑定地址，转变为可监听状态，然后接受连接。
- 但是得到了一个初始化好的 socket 还不够，我们需要监听这个 socket 上的事件并且处理事件的。
- 比如 Reactor 模型中使用了 epoll 监听该 socket 上的事件，我们还需要将需要被监视的套接字和监视的事件注册到 epoll 对象中。

- 而 Channel 类将文件描述符 fd 和感兴趣的事件 event（需要监听的事件）封装到了一起，而事件监听相关的代码放到了 Poller 和 EpollPoller 类中。

- Channel 扮演了一个 IO 事件分发器的作用，Acceptor 中的 Channel 主要处理连接事件，而每个 TcpConnection 中会有一个 Channel，来检测 fd 的可读，关闭，错误消息等。触发相应的回调函数。

- 一般网络库会把监听的 socket 单独放在一个线程，然后连接到来的多个 socket 放在其他几个线程，所以就有了两个创建 Channel 的地方。
- 一个是在 Acceptor 构造中创建用于监听 socket 的通道，一个是新连接到来，TcpConnection 中创建的 Channel。构造函数中传入了当前所属的事件循环和文件 fd。

- 既然在两个地方创建了 Channel，所以回调的设置也不同，Acceptor 中 Channel 回调指向了 Acceptor 的 handle（处理）函数，进而指向了 TcpServer 的 newConnetion，而 TcpConnection 中的 Channel 则是回调到了 TcpConnection 中。

```C++
acceptChannel.setReadCallback(std::bind(&Acceptor::handleRead, ...
...));

channel->setReadCallback(std::bind(&TcpConnection::handleRead, ...
...));
channel->setWriteCallback(std::bind(&TcpConnection::handleWrite, .
.....));
channel->setCloseCallback(std::bind(&TcpConnection::handleClose, .
.....));
channel->setErrorCallback(std::bind(&TcpConnection::handleError, .
.....));
```

2.2.2 Channel 重要成员变量

- `const int fd`：这个 Channel 对象管理的文件描述符，Poller 监听的对象。
- `int events`：fd 所感兴趣的事件类型的集合
- `int revents`：事件监听器实际监听到的该 fd 发生的事件类型的集合，当事件监听器监听到了一个 fd 发生了什么事件，通过 set_revents 来设置 revent 的值。
- `static const int NoneEvent`：事件状态，相当于封装了 epoll_ctl。
- `EventLoop *loop`：这个 fd 属于哪个 EventLoop 对象。
- `ReadEventCallback readcallback`：各种事件发生时的回调函数。
- `std::weak_ptr<void> tie`：
  - `bool tied`：解决 TcpConnection 和 Channel 生命周期问题。

2.2.3 Channel 重要方法

- `void setReadCallback()`：向 Channel 对象注册各类事件的回调函数
  - 一个 fd 在发生可读可写这类操作时，就需要调用相应处理函数来处理。
  - 外部通过调用这类函数就可以将事件处理函数放进 Channel 类中，当需要调用的时候就可以直接拿出来调用了。

- `void enableReading()`：设置 fd 相应的事件状态（ReadEvent），相当于 epoll_ctl。
  - 外部通过这几个函数来告知 Channel 你所监管的 fd 都对哪些事件类型感兴趣，并把这个 fd 及其感兴趣的事件注册到事件监听器（IO 多路复用模块）上。
  - 这些函数中都有 update 方法，本质上就是调用了 epoll_ctl();
- `void setRevents()`：当事件监听器监听到某个 fd 发生了什么事件，通过这个函数可以将这个 fd 实际（real）发生的事件封装进这个 Channel 中。
- `void handleEvent()`：当调用了 epoll_wait 后，可以得知事件监听器上哪些 Channel（fd）发生了哪些事件，继而要调用 Channel 对应的处理函数。
  - 此函数让每个发生了事件的 Channel 调用自己保管的事件处理函数。
  - 每个 Channel 会根据自己 fd 实际发生的事件（revents）和感兴趣的事件（events）来选择调用 readcallback 这类函数。
- `void handleEventWithGuard()`：真正的事件处理函数。
- `void tie()`：防止当 channel 被手动 remove 掉，channel 还在执行回调操作，成员变量 `weak_ptr<void> tie` 作为引用参数传入。
- `void update()`：通过 channel 所属的 EventLoop，调用 poller 的相应方法，注册 fd 的 events 事件。

2.3 Poller/EpollPoller 类

2.3.1 Poller/EpollPoller 概述

- 负责监听文件描述符事件是否触发以及返回发生事件的文件描述符以及具体事件的模块就是 Poller。
- 所以一个 Poller 对象对应一个事件监听器，在 Multi-Reactor 模型中，有多少 Reactor 就有多少 Poller。

- muduo 提供了 epoll 和 poll 两种 IO 多路复用的方法来实现事件监听，不过我们只使用 epoll。
- 调用一次 poll 方法就能给你返回事件监听器的监听结果（发生事件的 fd 及其发生的事件）

- 这个 Poller 是个抽象虚类，由 EpollPoller 和 PollPoller 继承实现，与监听文件描述符和返回监听结果的具体方法也基本上是在这两个派生类中实现。
- EpollPoller 就是封装了用 epoll 方法实现的与事件监听有关的各种方法。
- PollPoller 则是封装了 poll 方法实现的与事件监听有关的各种方法。我们不使用。

2.3.2 Poller 重要成员变量

- `ChannelMap channels：`类型为 `unordered_map`，负责记录文件描述符到 Channel 的映射，管理了所有注册在 Poller 上的 Channel。
- `EventLoop *ownerLoop`：所属的 EventLoop 对象。

2.3.3 Poller 重要方法

- 所有 IO 复用保留统一接口
  - `virtual Timestamp poll();`
  - `virtual void updateChannel();`
  - `virtual void removeChannel();`

2.3.4 EpollPoller 重要成员变量

- `int epoll_fd`：就是用 epoll_create 方法返回的 epoll 句柄。
- `EventList events`：用于存放 epoll_wait 返回的所有发生事件的文件描述符。

2.3.5 EpollPoller 重要重要方法

- 重写基类抽象方法
  - `Timestamp poll()`：
    - 这个函数作为 Poller 的核心，当外部调用 poll 方法的时候，该方法底层是通过 epoll_wait 来获取这个事件监听起上发生事件的 fd 及其对应发生的事件。
    - 我们自动每个 fd 都是由一个 Channel 封装的，通过 map channels 可以根据 fd 找到封装这个 fd 的 Channel。
    - 将事件监听器监听到该 fd 发生的事件写进这个 Channel 中的 revents 成员变量中，然后把这个 Channel 装进 activeChannels 中。
    - 这样，当外界调用完 poll 之后就能拿到事件监听器的监听结果，也就是 activeChannels 中的成员（活跃事件或者说监听到的事件的 fd），以及每个 fd 都发生了什么事件。
  - `void updateChannel()`：通过 channel 的 update 函数传递到 EventLoop 中，再传递到 Poller 中。其中调用了 `update()`。
  - `void removeChannel()`：同上。
- `void fillActiveChannels()`：填写活跃的连接，即监听到的连接。
- `void update()`：更新 channel 通道，就是调用 epoll_ctl，add/mod/del。

2.4 EventLoop 类

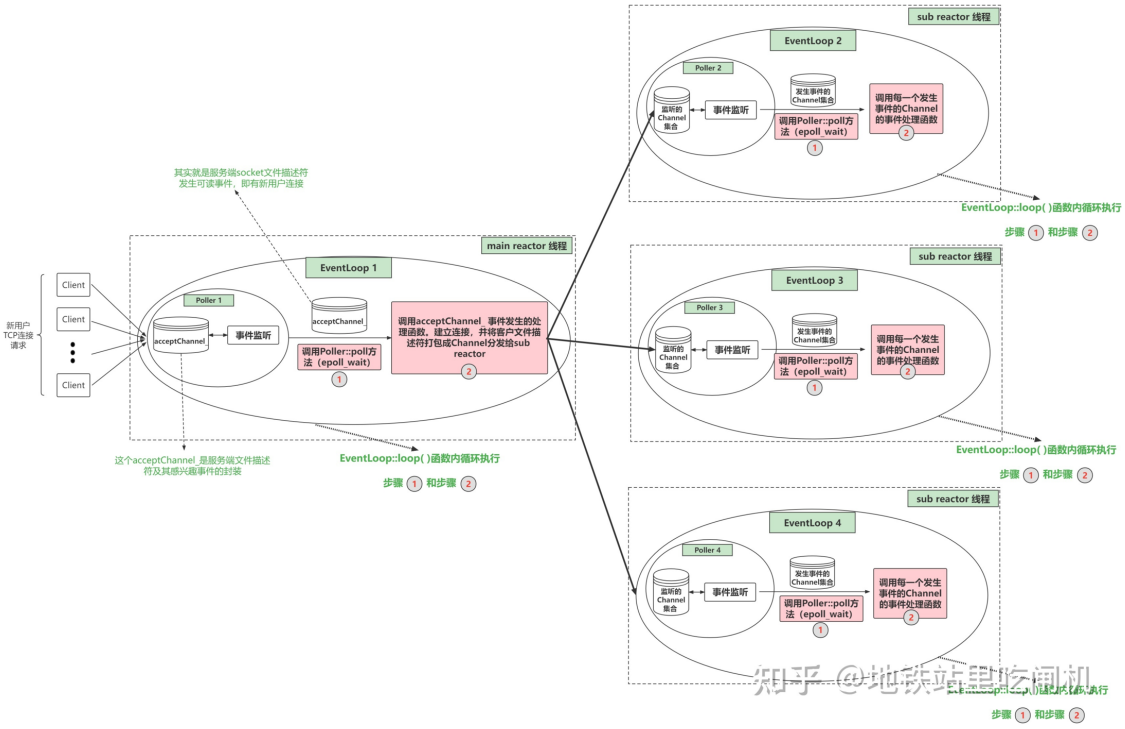
2.4.1 EventLoop 概述

- 作为一个网络服务器，需要有持续监听，持续获取监听结果，持续处理监听结果对应的事件的能力；
- 也就是我们需要循环的去调用 poll 方法来获取实际发生事件的 Channel 集合，然后调用这些 Channel 里保管的不同类型事件的处理函数（调用 HandlerEvent 方法）。

- EventLoop 就是负责实现循环，负责驱动循环的模块，Channel 和 Poller 相当于 EventLoop 的手下，EventLoop 整合封装了两者并向上提供了更方便的接口来使用。

2.4.2 概览 Poller，Channel，EventLoop 在 Reactor 架构中的角色

- EventLoop 起到一个驱动循环的功能，Poller 负责从事件监听器上获取监听结果。
- 而 Channel 类则在其中起到了将 fd 及其相关属性封装的作用，将 fd 及其感兴趣的事件和发生的事件以及不同事件的回调函数封装在一起，这样在各个模块中传递更加方便。



2.4.3 One Loop Per Thread 概述

- 每一个 EventLoop 都绑定了一个线程，每一个线程负责循环监听一组 fd 的集合。
- 每个 EventLoop 对象都是一个 reactor，所以每个 EventLoop 必然存在于子线程，那么我们的主线程如何在接受到事件连接后把 accept 到的 fd 传递给子线程呢?
- 把线程所属的 EventLoop 对象的指针传过去，然后注册回调。

- 综上 EventLoop 实际上就是一个管理类，它管理子线程内的连接，即 channel 对象和 IO 多路复用对象。那它一定也有一个进行事件循环的函数，来进行正常的事件循环。

2.4.4 EventLoop 重要成员变量

- `std::atomic_bool looping`：标识 loop 是否运行。
- `std::atomic_bool quit`：标识 loop 是否退出。
- `const pid_t threadId`：标识当前 EventLoop 的所属 id。
- `Timestamp pollRetureTime`：Poller 返回发生事件的 Channel 的时间点
- `int wakeupFd`：当 mainLoop 获取一个新用户的 Channel 需通过该成员变量唤醒 subLoop 处理 Channel。
- `ChannelList activeChannels`：返回 Poller 检测到当前有事发生的所有 Channel 列表。
- `std::atomic_bool callingPendingFunctors`：标识当前 loop 是否有需要执行的回调操作
- `std::vector<Functor> pendingFunctors`：存储 loop 需要执行的所有回调操作。

2.4.5 EventLoop 重要方法

- `void loop()`：
  - 每个 EventLoop 对象都唯一绑定了一个线程，这个线程其实一直执行这个函数里面的 while 循环，这个 while 循环的大致逻辑就是调用 poll 方法获取事件监听器上的监听结果。
  - 接下来在 loop 里就会调用监听结果中每一个 Channel 的处理函数 HandlerEvent。每一个 Channel 的处理函数会根据 Channel 类中封装的实际发生的事件，执行 Channel 类中封装的各事件处理函数。
  - EventLoop 的主要功能就是持续循环的获取监听结果并且根据结果调用处理函数。

- `void quit()`：退出事件循环。
- `void runInLoop()`：在当前 loop 中执行。
- `void queueInLoop()`：把上层注册的回调函数 cb 放入队列中，唤醒 loop 所在的线程执行 cb。
- `void wakeup()`：通过 eventfd 唤醒 loop 所在的线程。
- EventLoop 的方法通知 Poller 的方法
  - `void updateChannel()`：
  - `void removeChannel()`：
- `void handleRead()`：给 eventfd 返回的文件描述符 wakeupFd 绑定的事件回调，当 wakeup()时，即有事件发生，调用 handleRead()读 wakeupFd，同时唤醒阻塞的 epoll_wait。
- `void doPendingFunctors()`：执行上层回调。

3、其他类

3.1 Acceptor 类

- 接受新用户的连接并分发连接给 SubReactor（SubEventLoop）。

3.1.1 Acceptor 概述

- Acceptor 封装了服务器监听套接字 fd 以及相关处理方法，主要是对其他类的方法调用进行了封装。

3.1.2 Acceptor 重要成员变量

- `Socket acceptSocket：`服务器监听套接字的文件描述符。

- `Channel acceptChannel`：把 acceptSocket 及其感兴趣的事件和事件对应的处理函数都封装进去。
- `EventLoop *loop`：监听套接字的 fd 由这个 EventLoop 负责循环监听以及处理相应事件。一般是主 EventLoop。
- `NewConnectionCallback newConnectionCallback`：TcpServer 构造函数中将 newConnection 函数注册给了这个成员变量。
    - 这个 newConnection 函数的功能是公平的选择一个 subEventLoop，并把已经接受的连接分发给这个 subEventLoop

3.1.3 Acceptor 重要成员方法

- `void listen()`：该函数底层调用了 linux 的函数 listen，开启对 acceptSocket 的监听，同时将 acceptChannel 及其感兴趣的事件（可读事件）注册到主 EventLoop 的事件监听器上。
    - 就是让主 EventLoop 事件监听器去监听 acceptSocket。
- `void handleRead()`：这是一个私有方法，这个方法是要注册到 acceptChannel 上的，同时 handleRead 方法内部还调用了成员变量 newConnectionCallback 保存的函数。
    - 当主 EventLoop 监听到 acceptChannel 上发生了可读事件时（新用户连接事件），就是调用这个 handleRead 方法。
    - 当监听的 fd 有事件发生了，该函数就接受新连接，并且以负载均衡的方式选择一个 subEventLoop，并把这个新连接分发到这个 subEventLoop 上。
- `void setNewConnectionCallback()`：设置新连接的回调函数。

3.2 Socket 类

Socket 类是对 socket 文件描述符的封装，也包括一些对 socket 操作的工具函数。

3.2.1 Socket 成员变量

- `int socketFd`：服务器监听套接字的文件描述符。

3.2.2 Socket 方法

- `int fd()`：获取文件描述符。
- `void bindAddress()`：调用 bind 绑定服务器 IP 端口。
- `void listen()`：监听套接字。
- `int accept()`：调用 accept 接受新用户连接请求。
- `void setTcpNoDelay()`：调用 setsockopt 来设置一些 socket 选项。
- `void shutdownWrite()`：使用 shutdown 设置 SHUT_WR，关闭写端。

3.3 TcpConnection 类

3.3.1 TcpConnection 概述

- 这个类主要封装了一个已建立的 TCP 连接，以及控制该 TCP 连接的方法（连接建立/关闭/销毁），以及该连接发生的各种事件（读/写/错误/连接）对应的处理函数，和这个 TCP 连接的服务端和客户端的套接字地址信息等。

- Acceptor 用于 mainEventLoop 中，对服务器监听套接字 fd 及其相关方法进行封装（监听/接受连接/分发连接给 subEventLoop）
- TcpConnection 用于 subEventLoop 中，对服务器监听套接字 fd 及其相关方法进行封装（读消息事件/发送消息事件/连接关闭事件/错误事件）。

3.3.2 TcpConnection 重要变量

- `std::unique_ptr<Socket> socket`：用于保存已连接套接字的文件描述符。

- `std::unique_ptr<Channel> channel`：封装了 socket 及其各类事件的处理函数（读/写/错误/关闭等事件处理函数 ）。这个 Channel 保存的各类事件的处理函数是在 TcpConnection 对象的构造函数中注册的。
- `EventLoop *loop`：该 Tcp 连接的 Channel 注册到了哪一个 subEventLoop 上，这个 loop 就是哪一个 subEventLoop。
- `Buffer inputBuffer`：Buffer 类，是该 TCP 连接对应的用户接收缓冲区。
- `Buffer outputBuffer`：Buffer 类，用于暂存那些暂时发送不出去的待发送数据。因为 TCP 发送缓冲区是有大小限制的，若到达高水位线，则无法把发送的数据通过 send 直接拷贝到 TCP 发送缓冲区，而是暂存在这个 outputBuffer 中，等 TCP 发送缓冲区有空间了，触发可写事件了，再把 outputBuffer 中的数据拷贝到 TCP 发送缓冲区中。
- `enum state`：标识了当前 TCP 连接的状态
  - connected：已连接
  - connecting：正在连接
  - disconnecting：正在断开连接
  - disconnected：已经断开连接
- `ConnectionCallback connectionCallback`：用户会自定义连接建立关闭后的处理函数，收到消息后的处理函数，消息发送完毕后的处理函数，连接关闭后的处理函数，然后这些函数会注册给相应成员变量保存。

3.3.3 TcpConnection 重要成员方法

- 在一个已经建立好的 TCP 连接上主要会发生四类事件：可读事件，可写事件，连接关闭事件，错误事件。
- 当事件监听器监听到一个连接发生了以上的事件，那么就会在 EventLoop 中调用这些事件对应的处理函数。

    - `void handleRead()`：负责处理 TCP 连接的可读事件，它会将客户端发送来的数据拷贝到用户缓冲区中（intputBuffer），然后再调用 connectionCallback 保存的连接建立后的处理函数。
    - `void handleWrite()`：负责处理 TCP 连接的可写事件。
    - `void handleClose()`：负责处理 TCP 连接的关闭事件，就是将这个 TcpConnection 对象中的 channel 从事件监听器中移除。然后调用 connectionCallback 和 closeCallback 保存的回调函数。
    - `void handleError()`：负责处理 TCP 连接的错误事件。
- `void send()`：发送数据。
- `void shutdown()`：关闭连接。
- `void connectEstablished()`：连接建立。
- `void connectDestroyed()`：连接销毁。
- `void setState()`：设置连接状态。
- `void sendInLoop()`：当前 Loop 线程中发送数据。
- `void shutdownInLoop()`：关闭当前 Loop 线程写端

3.4 Buffer 类

3.4.1 Buffer 类概述

Buffer 类其实就是封装了一个用户缓冲区，以及向这个缓冲区读写数据等一系列控制方法。

3.4.2 为什么要有缓冲区

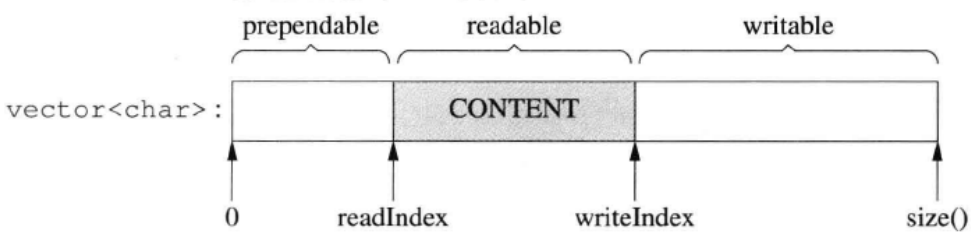
- 非阻塞网络编程中应用层 buffer 是必须的，非阻塞 IO 的核心思想是避免阻塞在 read 或 write 或其他 IO 系统调用上，这样可以最大限度复用 thread-of-control（控制线程），让一个线程能服务于多个 socket 连接。

    - IO 线程只能阻塞在 IO 复用函数上，如 `select/poll/epoll_wait`。这样一来，应用层的缓冲就是必须的，每个 Tcp socket 都要有 inputBuffer 和 outputBuffer

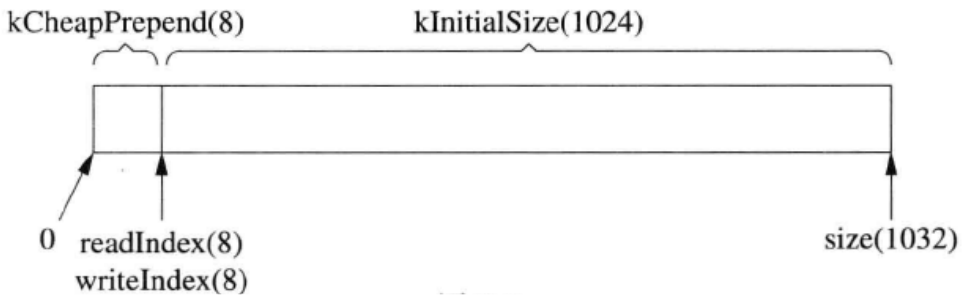
- TcpConnection 必须要有 outputBuffer：使程序在 write 操作上不会产生阻塞，当 write 操作后，操作系统一次性没有接受完时，网络库把剩余数据放入 outputBuffer 中，然后注册 POLLOUT 事件，一旦 socket 变得可写，则立刻调用 write 进行写入数据。即应用层 buffer 到操作系统 buffer。
- TcpConnection 也必须要有 intputBuffer：当发送方 send 数据后，接受方收到的数据不一定是完整的数据，网络库在处理 socket 可读事件的时候，必须一次性把 socket 里的数据读完，否则会反复触发 POLLIN 事件，造成 busy-loop
    - 当连接到达文件描述符上限时，此时没有可供你保存新连接套接字的文件描述符了，那么新来的连接就会一直放在 accept 队列中，于是其可读事件就会一直触发读事件，因为一直不读，也没办法读，这就是 busy-loop
- 所以网络库为了应对数据不完整的情况，收到的数据先放到 inputBuffer 里。即操作系统 buffer 到应用层 buffer。

3.4.3 Buffer 类设计

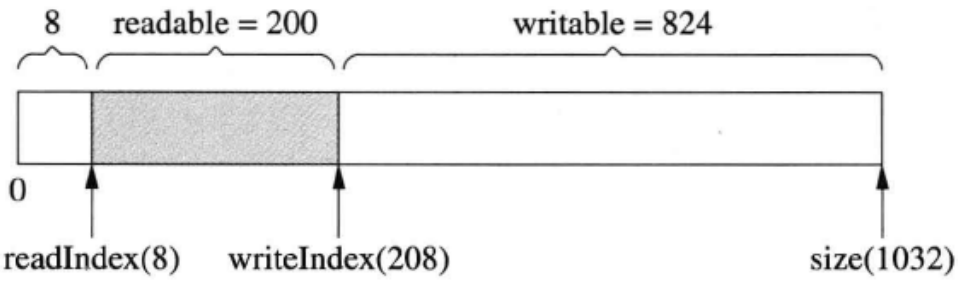
- 这个类用两个游标标记了可读缓冲区和可写缓冲区（空闲）的起始位置。



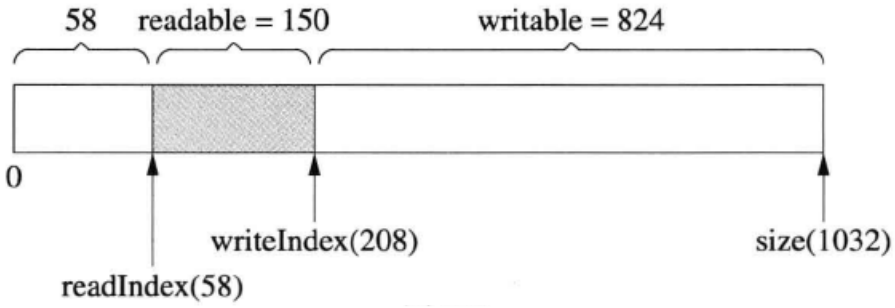
- 初始状态 `readIndex` 和 `writeIndex` 在同一位置（8），`kCheapPrepend` 定义 `prependable` 初始大小，`kInitalSize` 定义 `writable` 初始大小



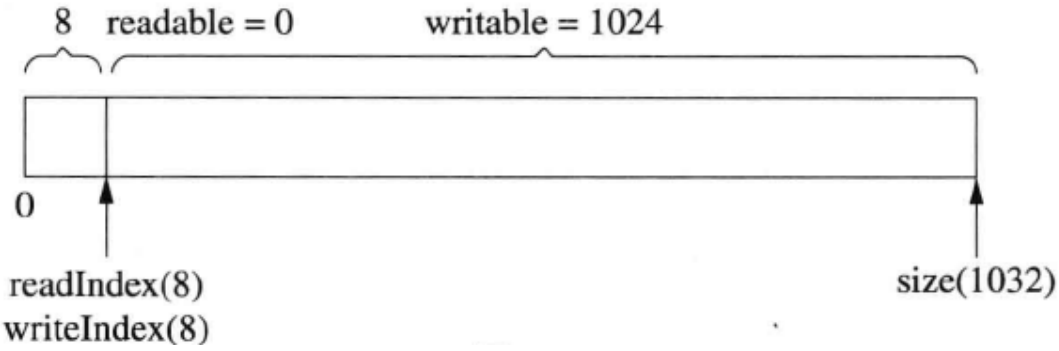
- 若向 Buffer 写入 200 字节则 `writeIndex` 游标则会向后移动。



- 若从 BUffer 中读入了 50 字节， `readIndex` 则会向后移动 50 字节，`writeIndex` 保持不变。



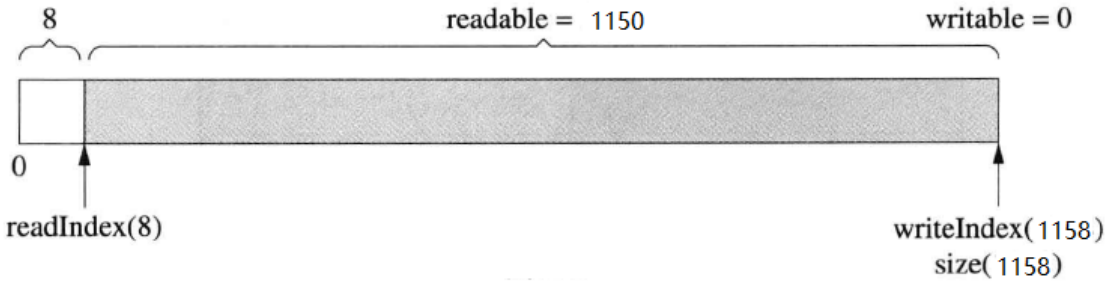
- 若一次性读完全部数据（`writeIndex` – `readIndex`），则两游标都返回原位以备新一轮使用。



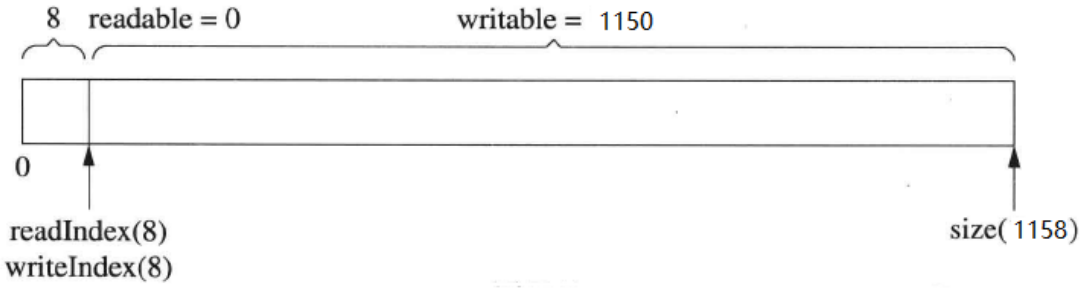
- Buffer 可以自动增长，初始值为 1024，在写入字节超过最大长度时，会进行扩容。

比如一次性写入 1000 字节，若 writeable（可写缓冲区）不够，则需加上 `readIndex` – `kCheapPrepare`（已读缓冲区）进行判断。

若依然不够，则进行扩容。此时 `readIndex` 游标将回到初始位置（8），而 `writeIndex` 的位置则是 `readable`（新可读缓冲区，即前可读缓冲区 150+ 新写入长度 1000）+ 8



- 在将这 1150 字节可读缓冲区读完后，`readIndex` 和 `writeIndex` 依然返回 `kCheapPrepend`（8），size 则是保持在了 1158，`writeable` 可写缓冲区为 1150，这样就实现了一次扩容。



3.4.4 为什么要预留 prepareable

- **什么是粘包**
  - 首先我们要了解一下粘包这个概念：
  - 客户端发送的多个数据包被当作一个数据包接收，就称作粘包，也称数据的无边界性，read/recv 函数不知道数据包的开始或结束标志，只把它们当作连续的数据流来处理。
  - 而 UDP 则是把内容一个个的发送过去，不管客户端能否完全接收到内容，它都会发送，而内容大于设定大小时，多余部分会被丢掉。
  - 所以只有 TCP 才会出现粘包，UDP 则不会出现粘包。

- **为什么产生粘包**
  - 主要原因：TCP 称为流失协议，数据流会杂糅在一起，接收端不清楚每个消息的界限，不知道每次应该读取多少字节的数据。
  - 次要原因：TCP 为了提高传输效率会有一个 nagle 优化算法，当多次 send 的数据字节都非常少，nagle 算法就会将这些数据合并成一个 TCP 段再发送，这就无形中产生了粘包。
- **如何解决粘包**
  - 这里就提到 prepareable 字段的作用了。
  - 它为 Buffer 提供了一个 8 字节的预留空间，可以用作存放数据包的字节长度，以便接收端判断数据包的边界。

3.4.5 Buffer 成员变量

- `static const size_t kCheapPrepend`：预留区大小。
- `static const size_t kInitialSize`：初始化 buffer 大小。
- `std::vector<char> buffer`：buffer 缓冲区。
- `size_t readerIndex`：可读区起始位置。
- `size_t writerIndex`：可写区起始位置。

3.4.6 Buffer 重要成员方法

- `void append()`：将 data 数据添加到缓冲区
- `size_t retrieveAsString()`：获取缓冲区中长度为 len 的数据，并以 string 返回。
- `size_t retrieveAllString()`：获取缓冲区所有数据，并以 string 返回。
- `void ensureWritableBytes()`：在向缓冲区写入长度为 len 的数据之前，先调用此函数，会检查你的缓冲区可写空间是否能装下长度为 len 的数据，如果不能，则动态扩容。

- `size_t readFd()`：客户端发来数据，readFd 从该 Tcp 接收缓冲区中将数据读出来并放到 Buffer 中。
- `size_t writeFd()`：服务端要向这条 Tcp 连接发送数据，通过该方法将 Buffer 中的数据拷贝到 Tcp 发送缓冲区中。
- `void makeSpace()`：扩容操作。
- `size_t readableBytes()`：获取可读区域字节数。
- `size_t writableBytes()`：获取可写区域字节数。
- `size_t prependableBytes()`：获取预留区域字节数。

3.5 TcpServer 类

- TcpServer 主要的功能是管理新连接到来时创建的 TcpConnection，是直接提供给用户使用的类。

3.5.1 TcpServer 重要成员变量

- `EventLoop *loop`：mainEventLoop 中的 baseloop，用户自定义的 loop。
- `std::unique_ptr<Acceptor> acceptor`：Acceptor 对象，用于监听新连接事件。
- `std::shared_ptr<EventLoopThreadPool>`：线程池对象。
- `ConnectionCallback connectionCallback`：各种类型事件的回调函数。
- `ConnectionMap connections`：保存所有连接的 Map。

3.5.2 TcpServer 重要成员函数

- `void setThreadInitCallback()`：设置各种类型事件的回调函数。
- `void setThreadNum()`：设置底层 subLoop 的个数。
- `void start()`：开启服务器监听。

- `void newConnection()`：当有新用户连接时，acceptor 会执行找个回调操作，负责将 mainLoop 接收到的请求连接通过回调轮询分发给 subLoop 去处理。
- `void removeConnection()`：移除连接。

4、工具类

4.1 noncopyable 不可拷贝基类

- 将拷贝构造函数和拷贝赋值函数用 delete 修饰

4.2 Thread 线程类

- 对线程的封装，使用的是 pthread

4.2.1 Thread 成员变量

- `bool started`：线程开启标识。
- `bool joined`：线程退出标识。
- `std::shared_ptr<std::thread> thread`：线程类型智能指针。
- `pid_t tid`：线程号。
- `ThreadFunc func`：线程回调函数。
- `std::string name`：线程名。
- `static std::atomic_int numCreated`：线程序号。

4.2.2 Thread 成员函数

- `void start()`：开启线程。
- `void join()`：终止线程。
- `bool started()`：返回线程是否开启。
- `pid_t tid()`：返回线程号。
- `const std::string &name()`：返回线程名。
- `static int numCreated()`：返回线程序号。
- `void setDefaultName()`：设置默认线程名。

4.3 EventLoopThread 事件循环线程类

- 封装了当前 EventLoop 所对应的线程。

4.3.1 EventLoopThread 重要成员变量

- `EventLoop *loop`：当前线程对应的事件循环。
- `Thread thread`：当前线程。
- `ThreadInitCallback callback`：保存线程初始化回调函数。

4.3.2 EventLoopThread 成员方法

- `EventLoop *startLoop()`：启用底层线程 Thread 类对象 thread 中的 start() 创建线程，然后绑定事件循环，返回新线程中的 EventLoop 指针。
- `void threadFunc()`：该方法在单独的新线程里运行，先创建一个独立的 EventLoop 对象，然后执行 EventLoop 的 loop()，这样就开启了底层 Poller 的 poll()。

4.4 EventLoopThreadPool 线程池类

- 线程池负责管理 subEventLoop。
- 负责对线程的创建，结束，可以用轮询的方式获取线程。

4.4.1 EventLoopThreadPool 重要成员变量

- `EventLoop *baseloop`：mainLoop 对应的事件循环。
- `std::vector<std::unique_ptr<EventLoopThread>> threads`：线程集合。
- `std::vector<EventLoop *> loops`：事件循环集合。

4.4.2 EventLoopThreadPool 重要成员方法

- `EventLoop *getNextLoop()`：以轮询方式分配 Channel 给 subLoop，返回获取到的 loop。
- `void start()`：开启线程池，初始创建 mainEventLoop 对应的线程，以及加入后续创建的 subEventLoop 对应的线程。

4.5 Timestamp 事件戳类

- 封装当前时间以及格式化输出。

4.6 CurrentThread 当前线程类

- 该类获取线程标识，即 tid

4.7 InetAddress 地址类

- 封装 socket 地址类型。

二、线程

1、One Loop Pre Thread

- One Loop Pre Thread 的含义就是，一个 EventLoop 和一个线程唯一绑定，

被这个 EventLoop 管辖的一切操作都必须在这个 EventLoop 绑定的线程中执行。

- 比如负责新连接建立的操作都要在 mainEventLoop 线程中运行，已建立的连接分发到某个 subEventLoop 上，这个已建立连接的任何操作，都必须在这个 subEventLoop 线程上运行，不能到别的 subEventLoop 去执行。

1.1 eventfd()

- eventfd 是一种系统调用，可以用来实现事件通知。
- eventfd 包含一个 64 位无符号整型计数器，创建 eventfd 时会返回一个文件描述符，进程可以通过这个文件描述符进行 read/write 来读取或改变计数器的值，从而实现进程间通信。

```C++
#include<sys/eventfd.h>
int eventfd(unsigned int initval, int flags);
```

- `initval`：创建 eventfd 时，64 位计数器的初始值。
- `flags`：eventfd 文件描述符的标志。
  - `EFD_CLOEXEC`：返回的 eventfd 文件描述符在 fork 后 exec 其他程序时会自动关闭这个文件描述符。
  - `EFD_NONBLOCK`：设置返回的 eventfd 为非阻塞。
  - `EFD_SEMAPHORE`：将 eventfd 作为一个信号量来使用

1.1.1 read

- 读取计数器的值
- 如果计数器中的值大于 0
  - 设置了 `EFD_SEMAPHORE` 标志位，则返回 1，且计数器中的值减去 1。
  - 没有设置 `EFD_SEMAPHORE` 标志位，则返回计数器中的值，且计数器设置为 0。
- 如果计数器中的值为 0
  - 设置了 `EFD_NONBLOCK` 标志位就直接返回−1。
  - 没有设置 `EFD_NONBLOCK` 标志位就会一直阻塞直到计数器中的值大于 0。

1.1.2 write

- 向计数器中写入值
- 如果写入的值和小于 `0xFFFFFFFFFFFFFFFE`,则写入成功。
- 如果写入的值和大于 `0xFFFFFFFFFFFFFFFE`
  - 设置了 `EFD_NONBLOCK` 标志位就直接返回−1
  - 如果没有设置 `EFD_NONBLOCK` 标志位，则会一直阻塞直到 read 操作执行。

1.2 保证 EventLoop 和线程唯一绑定

1.2.1 __thread 线程局部存储

- __thread 变量在每一个线程都会有一份独立实体，各个线程的值互不干扰。

```C++
__thread EventLoop *t_loopInThisThread = nullptr;
```

- 因为一般全局变量都是被同一个进程中的多个线程共享，但是我们不希望共享。
- 在 EventLoop 对象的构造函数中，如果当前线程没有绑定 EventLoop 对象，那么 `t_loopInThisThread` 为 nullptr，然后就让该指针变量指向 EventLoop 对象的地址。
- 如果 `t_loopInThisThread` 不为 nulptr，说明当前线程已经绑定了一个 EventLoop 对象了，这时候 EventLoop 对象构造失败。

```C++
EventLoop::EventLoop()
    : wakeupFd(createEventfd())
    , wakeupChannel(new Channel(this, wakeupFd))
    , ......
{
    if (t_loopInThisThread)
    {
        LOG_FATAL("Another EventLoop %p exists in this thread %d \n", t_loopInThisThread, threadId);
    }
    else
    {
        t_loopInThisThread = this;
    }

    ......
}
```

1.3 EventLoop 线程只执行自己的操作

- 比如负责新连接建立的操作都要在 mainEventLoop 线程中运行，已建立的连接分发到某个 subEventLoop 上，这个已建立连接的任何操作，都必须在这个 subEventLoop 线程上运行，不能到别的 subEventLoop 去执行。

1.3.1 EventLoop 构造

- `createEventfd()` 返回一个 eventfd 文件描述符，并且该文件描述符设置为非阻塞和子进程不拷贝模式。该 eventfd 文件描述符赋给了 EventLoop 对象的成员变量 wakeupFd。
- 然后将 wakeupFd 用 Channel 封装起来，得到 wakeupChannel。

```C++
int createEventfd()
{
    int evtfd = eventfd(0, EFD_NONBLOCK | EFD_CLOEXEC);
    if (evtfd < 0)
    {
        LOG_FATAL("eventfd error:%d\n", errno);
    }
    return evtfd;
}
```

- 在 EventLoop 构造函数的内部

```C++
// 给Channel注册一个读事件处理函数
wakeupChannel->setReadCallback(
    std::bind(&EventLoop::handleRead, this));

// 然后将wakeupChannel注册到事件监听器上监听其可读事件，当监听器监听到wakeupChannel的可读事件时就会调用EventLoop::handleRead()函数
wakeupChannel->enableReading();
```

1.3.2 Loop 执行

- 当 mainEventLoop 接受一个新连接请求，并把新连接封装成一个 TcpConnection 对象，并且希望在 subEventLoop 线程中执行 `TcpConnection::connectEstablished()` 函数。
- 那我们怎么在 mainEventLoop 线程中通知 subEventLoop 线程来执行该函数（即执行分派任务）。
- `EventLoop::runInLooo()` 函数接受一个可调用的函数对象 `Functor cb`，如果当前 cpu 正在运行的线程就是该 EventLoop 对象绑定的线程，那么就同步调用 cb 回调函数，否则就说明这是跨线程调用，需要异步将 cb 传给 `queueInLoop()` 函数。

```C++
void EventLoop::runInLoop(Functor cb)
{
    // 在当前EventLoop中执行回调
    if (isInLoopThread())
    {
        cb();
    }

    // 在非当前EventLoop线程中执行cb，就需要唤醒EventLoop所在线程执行cb
    else
    {
        queueInLoop(cb);
    }
}
```

- `queueInLoop()` 函数其实就是主线程用来给子线程传递回调的。
- 我们希望这个 cb 能在某个 EventLoop 对象所绑定的线程（subEventLoop）上运行，所以把 cb 这个可调用对象保存在 EventLoop 对象的 `pendingFunctors` 这个数组中。

```C++
void EventLoop::queueInLoop(Functor cb)
{
    {
        std::unique_lock<std::mutex> lock(mutex);
        pendingFunctors.emplace_back(cb);
    }

    // callingPendingFunctors为true的意思是，如果EventLoop正在处理当前的PendingFunctors函数时有新的回调函数加入，我们也要继续唤醒loop所在线程，如果不唤醒，那么新加入的函数就不会得到处理，会因为下一轮的epoll_wait而继续阻塞住。

    if (!isInLoopThread() || callingPendingFunctors)
    {
        // 唤醒Eventloop所在线程
        wakeup();
    }
}
```

- 在 wakeup 函数中，向 wakeupFd 写数据，这样会触发 EventLoop 读事件，当前 loop 线程就会被唤醒，epoll_wait 就会返回，EventLoop::loop 中阻塞的情况被打断。

```C++
void EventLoop::wakeup()
{
    uint64_t one = 1;
    ssize_t n = write(wakeupFd, &one, sizeof(one));

    ......
}
```

- `EventLoop::loop()` 肯定是运行在其所绑定的 EventLoop 线程中，在该函数内会调用 `doPendingFunctors()` 函数，该函数就是把其所绑定的 EventLoop 对象中的 pendingFunctors 数组中保存的可调用对象拿出来执行。

```C++
void EventLoop::loop()
{
    looping = true;
    quit = false;

    while (!quit)
    {
        ......
        doPendingFunctors();
    }
    looping = false;
}
```

三、主线

- TCP 网络编程的本质其实是处理三个半事件：
  - 连接的建立：服务器被动接收连接（accept）和客户端主动发起连接（connect）。
  - 连接的断开：包括主动断开（close，shutdown）和被动断开（read 返回 0）。
  - 消息到达：文件描述符可读。
  - 消息发送完毕：算半个，发送完毕是指数据写入操作系统缓冲区（内核缓冲区），将由 TCP 协议栈负责数据的发送与重传。
    - **不代表对方已经收到数据。**

1、Echo 服务器

- 要实现网络库的代码，首先要知道该库对外是怎么提供服务的。

```C++
class EchoServer
{
public:
    EchoServer(......)
    {
        // 将用户自定义连接事件处理函数注册进TcpServer里，TcpServer发生连接事件时会执行该函数
        server.setConnectionCallback(......);

        server.setMessageCallback(......);

        // 设置合适的subLoop（subReactor）线程数量
        server.setThreadNum(......);
    }

    void start()
    {
        server.start();
    }

private:
    // 用户自定义连接事件处理函数
    // 连接建立或断开的回调函数
    void onConnection(......)
    {
        if (//新连接建立请求)
        {
            printf("Connection UP");
        }
        //关闭连接请求
        else
        {
            printf("Connection DOWN");
        }
    }

    // 可读写事件回调
    void onMessage(......)
    {
```

```
        ......
    }

    EventLoop *loop;
    TcpServer server;
};

int main() {
    // mainEventLoop主事件循环，负责循环监听处理新用户连接事件的事件循环
器
    EventLoop loop;

    // InetAddress是对sockaddr_in进行封装
    InetAddress addr(......);

    // 构造回显服务器对象
    EchoServer server(&loop, addr, "EchoServer");

    // 启动回显服务器
    server.start();

    // 开启事件循环
    loop.loop();
    return 0;
}
```

1. 建立事件循环器 EventLoop：`EventLoop loop;`
2. 建立服务器对象：`TcpServer server;`
3. 向 TcpServer 注册各类事件的用户自定义处理函数：`setConnectionCallback;`
4. 启动 server：`server.start();`
5. 开启事件循环：`loop.loop();`

2、连接的建立

2.0 用户开始使用

- 用户首先在回显服务器上自定义连接事件的处理函数 `onConnection`

```C++
void onConnection(const TcpConnectionPtr &conn)
{
    ......
}
```

2.0.1 TcpServer::setConnectionCallback()

- `ConnectionCallback` 是一个通过 `TcpConnectionPtr` 封装的回调类型。

```C++
using ConnectionCallback = std::function<void(const TcpConnectionPtr &)>;
```

- 所以可以在 TcpServer 中为用户自定义连接事件设置回调函数。

```C++
void setConnectionCallback(const ConnectionCallback &cb)
{
    connectionCallback = cb;
}
```

2.1 TcpServer::TcpServer()构造

- 当我们创建一个 TcpServer 对象时，TcpServer 构造函数做的最主要的事就是在类的内部实例化了一个 Acceptor 对象，并往这个 Acceptor 对象注册了一个回调函数 `TcpServer::newConnection()` 。

```C++
TcpServer::TcpServer(......)
    : acceptor(new Acceptor(......))
    , ......
{
    acceptor->setNewConnectionCallback(
        std::bind(&TcpServer::newConnection, ......));
}
```

2.1.1 TcpServer::newConnection()

- 该函数的功能就是将建立好的连接进行封装，封装成一个 TcpConnection 对象。
- 有一个新用户连接，Acceptor 会执行这个回调，负责将 mainLoop 接收到的请求连接通过回调分发给 subLoop 去处理

```C++
void TcpServer::newConnection(......)
{
    // 轮询算法 选择一个subLoop 来管理connfd对应的channel
    EventLoop *ioLoop = threadPool->getNextLoop();

    // 生成一条TCP连接对象，这个TCP连接对象封装了该TCP连接即将分发给哪一个subEventLoop，即ioLoop。
    TcpConnectionPtr conn(new TcpConnection(......));

    // 下面的回调都是用户设置给TcpServer中的TcpConnection的
    conn->setConnectionCallback(connectionCallback);
    conn->setMessageCallback(messageCallback);
    conn->setWriteCompleteCallback(writeCompleteCallback);
    conn->setCloseCallback(
        std::bind(&TcpServer::removeConnection, ......));

    // 调用TcpConnection::connectEstablished()将TcpConnection::channel注册到刚刚选择的subEventLoop上
    ioLoop->runInLoop(
        std::bind(&TcpConnection::connectEstablished, .....))
;
}
```

2.1.2 TcpConnection::connectEstablisher()

- 此函数是在 subEventLoop 线程中执行的。

```C++
void TcpConnection::connectEstablished(......)
{
    // 设置这个TcpConnection的状态为已连接
    setState(......);

    channel->tie(shared_from_this());

    //channel封装的是建立好连接的客户端fd，将客户端fd及其可读事件注册到subEventLoop的事件监听器上
    channel->enableReading();

    // 新连接建立,调用用户提供的连接事件处理函数
    connectionCallback(shared_from_this());
}
```

2.2 Acceptor::Acceptor()构造

- 当我们在 TcpServer 构造函数实例化 Acceptor 对象时，Acceptor 的构造函数中实例化了一个 Channel 对象，即 acceptChannel。

```C++
Acceptor::Acceptor(......)
    : acceptChannel(......)
    , ......
{

    ......

    acceptChannel.setReadCallback(
        std::bind(&Acceptor::handleRead, ......));
}
```

- 该 Channel 对象封装了服务器监听套接字文件描述符，但是尚未注册到 mainEventLoop 的事件监听器上。
- 接着 Acceptor 构造函数将 `Acceptor::handleRead()` 方法注册进 acceptChannel 中，这就意味着，日后如果事件监听器监听到 acceptChannel 发生可读事件，将会调用 `Acceptor::handleRead()` 函数。

2.2.1 Acceptor::handleRead()

- 当 acceptChannel 发生可读事件发生时，即新用户连接时，Acceptor 会执行这个 `handleRead()` 函数。
- 在 TcpServer 构造函数中就已经通过 `acceptor->setNewConnectionCallback` 设置过回调函数。

```C++
class Acceptor : noncopyable
{
public:

    ......

    void setNewConnectionCallback(const NewConnectionCallback &cb) { NewConnectionCallback_ = cb; }

    ......
};
```

```C++
void Acceptor::handleRead()
{
    InetAddress peerAddr;

    //调用linux函数accept()接受新客户连接
    int connfd = acceptSocket.accept(&peerAddr);

    ......

    //NewConnectionCallback实际指向TcpServer::newConnection
    NewConnectionCallback(connfd, peerAddr);

    ......
}
```

2.3 整体流程

2.3.1 两条代码主线

用户自定义处理函数

- 2.0 用户开始使用：自定义连接事件处理函数 `onConnectionn` 。
- 2.0.1 `TcpServer::setConnectionCallback()`：TcpServer 中设置用户自定义连接事件 onConnection 的回调函数。
- 2.1.1 `TcpServer::newConnection()`：为新连接分发好 subEventLoop 后，给该连接设置回调函数。
- 2.1.2 `TcpConnection::connectEstablisher()`：在连接建立好后，执行回调。

TcpServer 构造

- 2.1 `TcpServer::TcpServer()`：Acceptor 对象通过设置新连接回调函数将 `TcpServer::newConnection` 绑定。
  - 2.2 `Acceptor::Acceptor()`：TcpServer 构造后，Acceptor 构造函数将 `Acceptor::handleRead()` 读回调注册进 acceptorChannel。
  - 2.1.1 `TcpServer::newConnection()`：在分配好 subEventLoop 后，执行该事件循环对应的 TcpConnection 连接的 `connectEstablished` 函数。
- 2.2.1 `Acceptor::handleRead()`：当事件监听器监听到 acceptorChannel 发生读事件，则会调用 handleRead()读回调。

2.3.2 主要流程

- 在 TcpServer 对象创建完毕后，用户调用 `TcpServer::start()` 方法，开启 TcpServer。

```C++
void TcpServer::start()
{
    // 启动底层的loop线程池
    threadPool->start(threadInitCallback);
    loop->runInLoop(std::bind(&Acceptor::listen, ......));

}
```

- 主要调用了 `Acceptor::listen()` 函数（底层 Linux 函数 listen）来监听服务器套接字。

```C++
void Acceptor::listen()
{
    listenning = true;
    acceptSocket.listen();
    acceptChannel.enableReading();
}
```

- 以及将 acceptor 注册到 mainEventLoop 的事件监听器上监听它的可读事件（新用户连接事件）。
- 接着调用 loop，循环获取事件监听器的监听结果，并根据监听结果调用注册在事件监听器上的 Channel 对象的事件处理函数。

3、消息读取

3.1 新连接初始化

- 在连接建立完成，并且 mainEventLoop 接受新连接请求之后，这条 Tcp 连接就被封装成了 TcpConnection 对象，主要封装了连接套接字的 fd（socket），连接套接字的 channel 等。
- 在 TcpConnection 构造时，下面四个方法会被注册进这个 channel 里。

```C++
//handleRead前半部分处理读取消息逻辑，后半部分通过调用messageCallback处理读消息后的逻辑
void handleRead(Timestamp receiveTime);

//handleWrite前半部分处理写消息逻辑，当一条消息完整写入Tcp发送缓冲区中时，则调用writeCompleteCallback
void handleWrite();

//handleClose前半部分处理连接关闭逻辑，后半部分调用connectionCallback和closeCallback
void handleClose();

//handleError处理连接错误事件
void handleError();
```

- 在 TcpConnection 中，通过 `set......Callback` 方法来设置相应事件的回调函数。

```C++
//上层分别对应了用户自定义的连接后/关闭后事件处理函数onConnection
void setConnectionCallback(const ConnectionCallback &cb)
{ connectionCallback = cb; }

//接收到消息后的函数onMessage
void setMessageCallback(const MessageCallback &cb)
{ messageCallback = cb; }

//写完后的事件处理函数onWriteComplete
void setWriteCompleteCallback(const WriteCompleteCallback &cb)
{ writeCompleteCallback = cb; }

//处理关闭连接的函数（网络库提供）removeConnection
void setCloseCallback(const CloseCallback &cb)
{ closeCallback = cb; }

// 这些回调TcpServer也有 用户通过写入TcpServer注册，TcpServer再将注册的回调传递给TcpConnection，TcpConnection再将回调注册到Channel中
ConnectionCallback connectionCallback;
MessageCallback messageCallback;
WriteCompleteCallback writeCompleteCallback;
CloseCallback closeCallback;
```

- 当 TcpConnection 对象建立完毕后，mainEventLoop 的 Acceptor 会将这个 TcpConnection 对象中的 channel 注册到某一个 subEventLoop 中。

3.2 EventLoop 循环监听

- 在 TcpConnection 对象中，已经封装好了套接字的 fd 以及 channel。而这个 channel 会被当前 subEventLoop 事件循环中的 Poller 对象加入 `ChannelList` 监听集合进行事件监听。

```C++
//vector中的每一个Channel封装着
//一个fd
//fd感兴趣的事件
//事件监听器监听到该fd实际发生的事件
using ChannelList = std::vector<Channel *>;
```

- 当 Channel 有事件发生时，Poller 会调用 `poll` 方法，也就是底层 `epoll_wait` 获取事件监听结果，来调用每一个发生事件的 Channel 的 handleEvent 事件处理函数。
- 该方法会根据每一个 Channel 的感兴趣事件以及实际发生的事件调用提前注册在 Channel 内的对应的事件处理函数。

C++

```cpp
void Channel::handleEvent(......)
{
    ......

    // 读
    if (revents & (EPOLLIN | EPOLLPRI))
    {
        if (readCallback)
        {
            readCallback(receiveTime);
        }
    }

    // 写
    if (revents & EPOLLOUT)
    {
        if (writeCallback)
        {
            writeCallback();
        }
    }

    //关闭
    if ((revents & EPOLLHUP) && !(revents_ & EPOLLIN))
    {
        if (closeCallback)
        {
            closeCallback();
        }
    }

    // 错误
    if (revents & EPOLLERR)
    {
        if (errorCallback)
        {
            errorCallback();
        }
    }
}
```

3.3 消息读取处理

- handleEvent 中的 `readCallback` 事件处理函数其实保存的是 `TcpConnection::handleRead`。
- 该方法首先调用 `Buffer.readFd`，底层为 linux 函数 readv 实现。将 Tcp 接受缓冲区数据拷贝到用户定义的 inputBuffer 缓冲区中。

```C++
void TcpConnection::handleRead(......)
{
    ......

    ssize_t n = inputBuffer.readFd(channel->fd(), ......);

    // 有数据到达
    if (n > 0)
    {
        // 表明已建立连接的用户有可读事件发生了,调用用户传入的回调操作onMessage
        messageCallback(shared_from_this(), &inputBuffer, receiveTime);
    }
    else if (n == 0)
    {
        handleClose();
    }
    else
    {
        ......

        handleError();
    }
}
```

3.3.1 Buffer::readFd()

- Buffer 缓冲区是有大小的，但是从 fd 上读取数据的时候，却不知道数据的最终大小。如果一次性读出来可能导致 Buffer 装不下而溢出。
- 通过 readFd 的设计能够让用户一次性把所有 TCP 缓冲区的所有数据全部读出来并放到用户自定义的缓冲区 Buffer 中。

C++

```cpp
ssize_t Buffer::readFd(int fd, ......)
{
    // 栈额外空间，用于从套接字往出读时，当buffer暂时不够用时暂存数据，待buffer重新分配足够空间后，在把数据交换给buffer。
    char extrabuf[65536] = {0};

    /*
    struct iovec {
        ptr_t iov_base; // iov_base指向的缓冲区存放的是readv所接收的数据或是writev将要发送的数据
        size_t iov_len; // iov_len在各种情况下分别确定了接收的最大长度以及实际写入的长度
    };
    */

    // 使用iovec分配两个连续的缓冲区
    struct iovec vec[2];

    // 这是Buffer底层缓冲区剩余的可写空间大小，不一定能完全存储从fd读出的数据
    const size_t writable = writableBytes();

    // 第一块缓冲区，指向可写空间
    vec[0].iov_base = begin() + writerIndex;
    vec[0].iov_len = writable;
    // 第二块缓冲区，指向栈空间
    vec[1].iov_base = extrabuf;
    vec[1].iov_len = sizeof(extrabuf);

    //如果第一个缓冲区writable大于等于64K，那就只采用一个缓冲区，而不使用栈空间extrabuf的内容
    const int iovcnt = (writable < sizeof(extrabuf)) ? 2 : 1;
    const ssize_t n = readv(fd, vec, iovcnt);

    ......

    // Buffer的可写缓冲区已经够存储读出来的数据了
```

```
    else if (n <= writable)
    {
        writerIndex += n;
    }

    // extrabuf里面也写入了n-writable长度的数据
    else
    {
        writerIndex = buffer.size();

        // 对buffer扩容 并将extrabuf存储的另一部分数据追加至buffer
        append(extrabuf, n - writable);
    }
    return n;
}
```

- extrabuf 是在栈上开辟的空间，函数结束后会自动释放，而且速度也比在堆上开辟要快。

4、消息发送

- 当用户调用了 `TcpConnection::send(buf)` 函数时，相当于要求网络库把 buf 数据发送给该 Tcp 连接的客户端。`TcpConnection::send(buf)` 内部实际是调用了 linux 函数 write。
  - 如果 Tcp 发送缓冲区能一次性容纳 buf，那这个 write 函数将 buf 数据全部拷贝到发送缓冲区中。
  - 如果 Tcp 发送缓冲区不能一次性容纳 buf，这时候 write 函数将 buf 数据尽可能的拷贝到 Tcp 发送缓冲区中，并将 errno 设置为 EWOULDBLOCK。
- 剩余没拷贝到 Tcp 发送缓冲区中的 buf 数据会被放在 `TcpConnection::outputBuffer` 中，并且向事件监听器上注册该 `TcpConnection::channel` 的可写事件。

- 事件监听器监听到该 Tcp 连接的可写事件，就会调用 `TcpConnection::handleWrite` 函数把 `TcpConnection::outputBuffer` 中的剩余数据发送出去。
- 在 `TcpConnection::handleWrite` 函数中，通过调用 `Buffer::writeFd` 函数将 outputBuffer 的数据写入到 Tcp 发送缓冲区。

```C++
void TcpConnection::handleWrite()
{
    //监听到可写事件
    if (channel->isWriting())
    {

        //将outputBuffer中readable部分写入Tcp发送缓冲区
        ssize_t n = outputBuffer.writeFd(channel->fd(), ......);
        ......

        //若完全写完
        if (outputBuffer.readableBytes() == 0)
        {
            channel->disableWriting();

            ......
        }
     }
     ......
}
```

- 如果 Tcp 发送缓冲区不能完全容纳全部剩余的未发送数据，那就尽可能的将数据拷贝到 Tcp 发送缓冲区中，结束该函数，继续保持可写事件的监听。一次次调用 handleWrite，直到数据写完。
- 当数据全部拷贝到 Tcp 发送缓冲区之后，就会调用用户自定义的写完后的事件处理函数 ，并且移除该 TcpConnection 在事件监听器上的可写事件。

```C++
//写完成
if (writeCompleteCallback)
{
    // TcpConnection对象在其所在的subloop中，加入用户自定义回调函数
    loop->queueInLoop(
      std::bind(writeCompleteCallback, shared_from_this()));
}

if (state == Disconnecting)
{
    // 在当前所属的loop中把TcpConnection删除掉
    shutdownInLoop();
}
```

5、连接断开

5.1 连接被动断开

- 当 `TcpConnection::handleRead` 函数内部的 readv 返回 0 时，就说明没有数据可读或读到了文件尾部，服务端就知道客户端断开连接了。
- 接着就调用 `TcpConnection::handleClose`。

```C++
void TcpConnection::handleClose()
{
    ......

    setState(Disconnected);

    //将channel从事件监听器上移除
    channel->disableAll();

    TcpConnectionPtr connPtr(shared_from_this());

    // 执行用户自定义的连接事件处理函数
    connectionCallback(connPtr);

    // 执行关闭连接的回调,执行的是TcpServer::removeConnection回调方法
    closeCallback(connPtr);
}
```

5.1.1 TcpServer::removeConnection

- 和 handleClose 一样运行在 subEventLoop 中

```C++
void TcpServer::removeConnection(const TcpConnectionPtr &conn)
{
    loop->runInLoop(
        std::bind(&TcpServer::removeConnectionInLoop, ......));
}
```

5.1.2 TcpServer::removeConnectionInLoop

- 该函数运行在 mainEventLoop 中，主要从 TcpServer 对象中删除某条数据.
- TcpServer 对象中有一个 connections 成员变量，这是一个 `unordered_map`，负责保存 Tcp 连接的名字到 TcpConnection 对象的映射。

- 因为这个 Tcp 连接要关闭了，所以也要把这个 TcpConnection 对象从 connections 中删除，然后再调用 `TcpConnection::connectDestroyed` 函数。

```C++
void TcpServer::removeConnectionInLoop(const TcpConnectionPtr &co
nn)
{
    ......

    connections.erase(conn->name());
    EventLoop *ioLoop = conn->getLoop();

    //在该TcpConnection所属的subEvenntLoop中执行connectDestroyed
    ioLoop->queueInLoop(
        std::bind(&TcpConnection::connectDestroyed, conn));
}
```

5.1.3 TcpConnection::connectDestroyed

- 该函数的执行又跳回到 subEventLoop 线程中，将 Tcp 连接的监听描述符从事件监听器中移除。
- 另外 subEventLoop 中的 Poller 类对象还保存着这条 Tcp 连接的 channel，所以调用 `channel.remove` 将这个 Tcp 连接的 channel 对象从 Poller 内删除。

```C++
void TcpConnection::connectDestroyed()
{
    if (state == Connected)
    {
        setState(Disconnected);

        //// 把channel的所有感兴趣的事件从poller中删除掉
        channel->disableAll();
        connectionCallback(shared_from_this());
    }

    //// 把channel从poller中删除掉
    channel->remove();
}
```

5.2 服务器主动关闭导致连接断开

5.2.1 TcpServer::~TcpServer

- 当服务器主动关闭时，调用 `TcpServer::~TcpServer` 析构函数。
- 不断循环的让这个 TcpConnetion 对象所属的 subEventLoop 线程执行 `TcpConnection::connectDestroyed` 函数。
- 同时在 mainEventLoop 的 TcpServer::~TcpServer 函数中调用 `item.second.reset` 释放保管 TcpConnection 对象的共享智能指针，以达到释放 TcpConnection 对象的堆内存空间的目的。

```C++
TcpServer::~TcpServer()
{
    for(auto &item : connections)
    {
        TcpConnectionPtr conn(item.second);

        //当conn出了其作用域,即可释放智能指针指向的TcpConnection对象，即引用计数减为0，这个TcpConnection对象的堆内存就会被释放。
        item.second.reset();

        // 销毁连接
        conn->getLoop()->runInLoop(
            std::bind(&TcpConnection::connectDestroyed, conn));
    }
}
```

- TcpServer::connections 是一个 `unordered_map<string, TcpConnectionPtr>`，其中 TcpConnectionPtr 是一个指向 TcpConnection 的 shared_ptr。

```C++
using ConnectionMap = std::unordered_map<std::string, TcpConnectionPtr>;

......

using TcpConnectionPtr = std::shared_ptr<TcpConnection>;
```

- 一开始，每一个 TcpConnection 对象都被一个共享智能指针 TcpConnectionPtr 持有，当执行了 `TcpConnectionPtr conn(item.second)` 时，这个 TcpConnection 对象就被 conn 和这个 item.second 共同持有，但是这个 conn 的生存周期很短，只要离开了当前 for 循环，conn 就会被释放。
- 接着调用 item.second.reset 释放掉 TcpServer 中保存的该 TcpConnection 对象的智能指针，但是还剩 conn 依然持有这个 TcpConnection 对象，因此当前 TcpConnection 对象还不会被析构。
- 接下来调用 `conn->getLoop()->runInLoop(function)`，这个 function 函数会在 loop 绑定的线程，也就是 subEventLoop 上执行 connectDestroyed。

5.2.2 TcpConnection::connectDestroyed

- 在创建 TcpConnection 对象时，Acceptor 都要将这个对象分发给 subEventLoop 来管理，这个 TcpConnection 对象的一切函数都要在其管理的 subEventLoop 线程中运行。
- 所以 connectDestroyed 方法必须在这个 TcpConnection 对象所属的 subEventLoop 绑定的线程中执行。
- mainEventLoop 线程将当前 for 循环跑完后，共享智能指针 conn 离开代码块，因此被析构，但是 TcpConnection 对象还不会被释放，因为还有一个共享智能指针指向这个 TcpConnection 对象（item.second），我们看不到这个智能指针，这个智能指针在 connectDestroyed 中，是一个隐式的 this。
- 当这个函数执行完后，智能指针就彻底被释放了。至此，就没有任何智能指针指向这个 TcpConnection 对象了，TcpConnection 对象就被彻底析构了。

5.2.3 TcpConnection 数据未发完

- 我们需要在触发 TcpConnection 关闭机制后，让 TcpConnection 先把数据发送完再关闭。
- 先了解一下 `shared_from_this` 是什么。

- `enable_shared_from_this` 是一个模板类

```C++
template<class T> class enable_shared_from_this;
```

- 我们继承 `enable_shared_from_this` 的目的，就是使用 `shared_from_this` 函数

```C++
shared_ptr<T>  shared_from_this();
```

- 把当前类对象作为参数传给其他函数时，如果直接传递 this 指针，我们将不会得知调用者会如何使用。
- 若是在类内用 `shared_ptr<TcpConnection> p1(this);` 的方式显示初始化一个指向自身的智能指针，外部的其他智能指针并不知情，他们不共享引用计数，
- 在类外这样构造的智能指针，不知道类内部有相同的指针，会造成两个非共享的 shared_ptr 指向一个对象，所以会造成两次析构。

```C++
shared_ptr<TcpConnection> x(new TcpConnection);
```

- 我们在类内使用 shared_from_this 就不会出现这样的情况

```C++
shared_ptr<TcpConnection> p2 = shared_from_this();
```

- 在 `TcpConnection::connectEstablished` 中，我们有这样一段代码。

```C++
connectionCallback(shared_from_this());
```

- 该函数返回一个 shared_ptr。

```C++
using ConnectionCallback = std::function<void(const TcpConnectionPtr &)>;

......

using TcpConnectionPtr = std::shared_ptr<TcpConnection>;
```

- 接下来这个 shared_ptr 就作为 Channel 的 `Channel::tie()` 函数的参数。

```C++
std::weak_ptr<void> tie;

......

//tie可以解决TcpConnection和Channel的生命周期时长问题，从而保证了Channel对象能够在TcpConnection销毁前销毁
void Channel::tie(const std::shared_ptr<void> &obj)
{
```