万字长文梳理Muduo库核心代码及优秀编程细节思想剖析

地铁站里吃闸机
修养

关注他

原文地址：

理Muduo库核心代码及优秀编程细节剖析_我在地铁站里吃闸机的博客-CSDN博客

一、前言：

Muduo库是陈硕个人开发的Tcp网络编程库，支持Reactor模型。本人前段时间出于个人学习目的用c++11重构了Muduo库中核心的Multi-Reactor架构。这篇博文对Muduo库中的Multi-reactor架构代码进行逻辑梳理，同时认真剖析了作者每一处精妙的代码设计思想。

目前我只重构并剖析了Muduo库中的核心部分，即Multi-Reactor架构部分。但是**这部分已经足够支撑起一个基本的高并发TCP服务器的运行了。**另外，Muduo库中剩余的rpc、HTTP等通信协议还暂时没有完成，日后一定补上。

在第二章概述篇中会概述一下每一个类的作用，其意图在于对每一个类建立一个直觉，第二章不会讲的很详细。重点留在第三章主线篇。主线篇会围绕连接建立、消息读取、消息发送、连接断开这几条主线来进行梳理，**同时会对代码中优秀编程思想和编程细节进行讨论。**最后在第四章线程篇会讲到Muduo库中涉及到的线程控制、线程通信的机制。

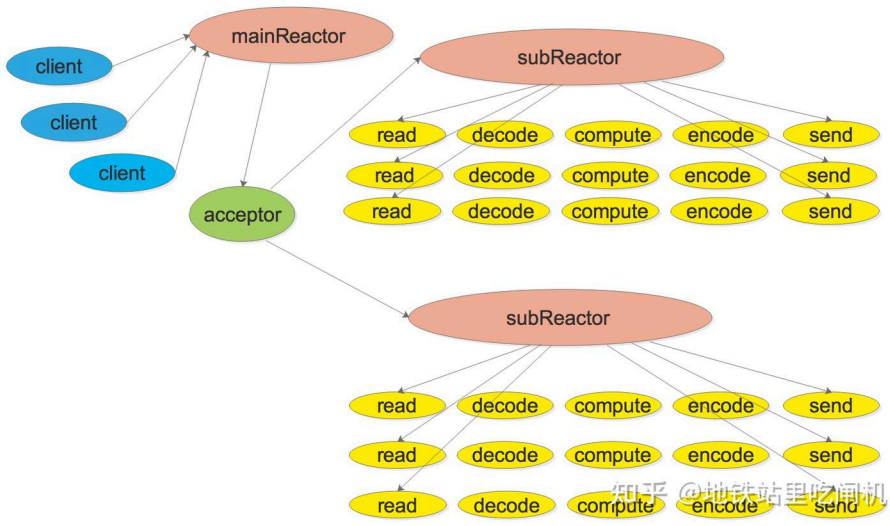
另外提一下，目前这是小弟第一次写博文，而且文章内容很长，并且这还是第一版初稿，可能有一些写的不知所云的东西，请多担待。建议先点个关注收藏，先把我养起来，日后一定不断更新矫正（**同时你们还能体会到免费包养我的快感，何乐而不为啊！！！[doge]**）。

二、 概述篇

在这一章中会介绍Multi-Reactor代码中的几个类，注意这一章只会简要介绍每个类的功能和作用。为下一章铺垫。

1. Multi-Reactor概述

Muduo库是基于Reactor模式实现的TCP网络编程库。该文章后续篇幅都是围绕Multi-reactor模型进行展开。Multi-Reactor模型如下所示（网上找的图，不是我画的）：



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。这三个组件分别是 `Channel` 类、 `Poller/EpollPoller` 类以及 `EventLoop` 类。

2.2 三大核心模块之一：Channel类 （后面到处都离不开Channel）

2.2.1 Channel类概述：

Channel类其实相当于一个文件描述符的保姆！

在TCP网络编程中，想要IO多路复用监听某个文件描述符，就要把这个fd和该fd感兴趣的事件通过epoll_ctl**注册**到IO多路复用模块（我管它叫**事件监听器**）上。当事件监听器监听到该fd发生了某个事件。事件监听器返回 [发生事件的fd集合]以及[每个fd都发生了什么事件]

Channel类则封装了一个 [fd] 和这个 [fd感兴趣事件] 以及事件监听器监听到 [该fd实际发生的事件]。同时Channel类还提供了**设置**该fd的感兴趣事件，以及将该fd及其感兴趣事件**注册**到事件监听器或从事件监听器上**移除**，以及**保存**了该fd的每种事件对应的处理函数。

2.2.2 Channel类重要的成员变量：

- `int fd_` 这个Channel对象照看的文件描述符
- `int events_` 代表fd感兴趣的事件类型集合
- `int revents_` 代表事件监听器实际监听到该fd发生的事件类型集合，当事件监听器监听到一个fd发生了什么事件，通过 `Channel::set_revents()` 函数来设置revents值。
- `EventLoop* loop` 这个fd属于哪个EventLoop对象，这个暂时不解释。
- `read_callback_` 、 `write_callback_` 、 `close_callback_` 、 `error_callback_` ：这些是std::function类型，代表着这个Channel为这个文件描述符保存的各事件类型发生时的处理函数。比如这个fd发生了可读事件，需要执行可读事件处理函数，这时候Channel类都替你保管好了这些可调用函数，真是贴心啊，要用执行的时候直接管保姆要就可以了。

2.2.3 Channel类重要的成员方法：

- **向Channel对象注册各类事件的处理函数**

```
void setReadCallback(ReadEventCallback cb) {read_callback_ = std::move(cb);}
void setWriteCallback(Eventcallback cb) {write_callback_ = std::move(cb);}
void setCloseCallback(EventCallback cb) {close_callback_ = std::move(cb);}
void setErrorCallback(EventCallback cb) {error_callback_ = std::move(cb);}
```

一个文件描述符会发生可读、可写、关闭、错误事件。当发生这些事件后，就需要调用相应的处理函数来处理。外部通过调用上面这四个函数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。

- **将Channel中的文件描述符及其感兴趣事件注册事件监听器上或从事件监听器上移除**

```
void enableReading() {events_ |= kReadEvent; upadte();}
void disableReading() {events_ &= ~kReadEvent; update();}
void enableWriting() {events_ |= kWriteEvent; update();}
void disableWriting() {events_ &= ~kWriteEvent; update();}
void disableAll() {events_ |= kNonEvent; update();}
```

外部通过这几个函数来告知Channel你所监管的文件描述符都对哪些事件类型感兴趣，并把这个文件描述符及其感兴趣事件注册到事件监听器（IO多路复用模块）上。这些函数里面都有一个 `update()` 私有成员方法，这个update其实本质上就是调用了 `epoll_ctl()` 。

- `void HandlerEvent(TimeStamp receive_time)`

当调用 `epoll_wait()` 后，可以得知事件监听器上哪些Channel（文件描述符）发生了哪些事件，事件发生后自然就要调用这些Channel对应的处理函数。 `Channel::HandleEvent` ，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会根据自己文件描述符实际发生的事件（通过Channel中的 `revents_` 变量得知）和感兴趣的事件（通过Channel中的 `events_` 变量得知）来选择调用 `read_callback_` 和/或 `write_callback_` 和/或 `close_callback_` 和/或 `error_callback_` 。

2.3 三大核心模块之二：Poller / EpollPoller

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**（这里我不确定要不要把Poller就当作事件监听器）。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多路复用方法来实现事件监听。不过默认是使用epoll来实现，也可以通过选项选择poll。但是我自己重构的muduo库只支持epoll。

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就是封装了poll方法实现的与事件监听有关的各种方法。**以后谈到Poller希望大家都知道我说的其实是EpollPoller。**

2.3.2 Poller/EpollPoller的重要成员变量：

- `epollfd_` 就是用 `epoll_create` 方法返回的epoll句柄，这个是常识。
- `channels_` ：这个变量是 `std::unordered_map<int, Channel*>` 类型，负责记录 文件描述符 ---> Channel的映射，也帮忙保管所有注册在你这个Poller上的Channel。
- `ownerLoop_` ：所属的EventLoop对象，看到后面你懂了。

2.3.3 EpollPoller给外部提供的最重要的方法：

```
TimeStamp poll(int timeoutMs, ChannelList *activeChannels)
```

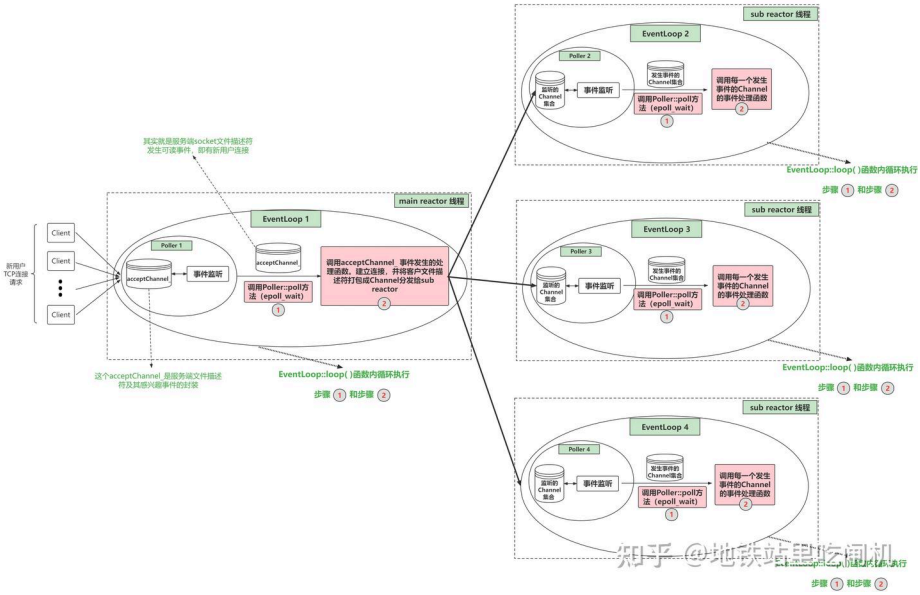
这个函数可以说是**Poller的核心**了，当外部调用 `poll` 方法的时候，该方法底层其实是通过 `epoll_wait` 获取这个事件监听器上发生事件的fd及其对应发生的事件，我们知道每个fd都是由一个Channel封装的，通过哈希表 `channels_` 可以根据fd找到封装这个fd的Channel。**将事件监听器监听到该fd发生的事件写进这个Channel中的revents成员变量中。**然后把这个Channel装进 `activeChannels` 中（它是一个vector<Channel*>）。这样，当外界调用完 `poll` 之后就能拿到事件监听器的**监听结果（ `activeChannels_` ）**，【这里标红是因为后面会经常提到这个“监听结果”这四个字，希望你明白这代表什么含义】，这个activeChannels就是事件监听器监听到的发生事件的fd，以及每个fd都发生了什么事件。

2.4 三大核心模块之三：EventLoop

2.4.1 EventLoop概述：

刚才的Poller是封装了和事件监听有关的方法和成员，调用一次 `Poller::poll` 方法它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）。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 `Poller:poll` 方法获取实际发生事件的Channel集合，然后调用这些Channel里面保管的不同类型事件的处理函数（调用 `Channel::HandlerEvent` 方法）。】

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及其感兴趣事件和发生的事件以及不同事件对应的回调函数封装在一起，这样在各个模块中传递更加方便。接着EventLoop调用



另外上面这张图我没有画出Acceptor，因为Acceptor和EventLoop和Poller之间有点错杂，可能画出来效果不好。

2.4.3 One Loop Per Thread 含义介绍

有没有注意到上面图中，每一个EventLoop都绑定了一个线程（一对一绑定），这种运行模式是Muduo库的特色！！充份利用了多核cpu的能力，每一个核的线程负责循环监听一组文件描述符的集合。至于这个One Loop Per Thread是怎么实现的，后面还会交代。

2.4.4 EventLoop重要方法 `EventLoop:loop()`：

```
void EventLoop::loop()
{ //EventLoop 所属线程执行
    省略代码 省略代码 省略代码
    while(!quit_)
    {
        activeChannels_.clear();
        pollReturnTime_ = poller_->poll(kPollTimeMs, &activeChannels_);//此时activeChan
        for(Channel *channel : activeChannels_)
            channel->HandlerEvent(pollReturnTime_);
        省略代码 省略代码 省略代码
    }
    LOG_INFO("EventLoop %p stop looping. \n", t_loopInThisThread);
}
```

每个EventLoop对象都唯一绑定了一个线程，这个线程其实就在一直执行这个函数里面的while循环，这个while循环的大致逻辑比较简单。就是调用 `Poller::poll` 方法获取事件监听器上的监听结果。接下来在loop里面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。（比如一个Channel发生了可读事件，可写事件，则这个Channel的 `HandlerEvent( )` 就会调用提前注册在这个Channel的可读事件和可写事件处理函数，又比如另一个Channel只发生了可读事件，那么 `HandlerEvent( )` 就只会调用提前注册在这个Channel中的可读事件处理函数)

看完上面的代码，感受到EventLoop的主要功能了吗？就是持续循环的获取监听结果并且根据结

3.1 Acceptor: 接受新用户连接并分发连接给SubReactor（SubEventLoop）

3.3.1 Acceptor概述

Accetpor封装了服务器监听套接字fd以及相关处理方法。Acceptor类内部其实没有贡献什么核心的处理函数，主要是对其他类的方法调用进行封装。这里也不好概述，具体看下面的内容吧。

3.3.2 Acceptor封装的重要成员变量

- `acceptSocket_`：这个是服务器监听套接字的文件描述符
- `acceptChannel_`：这是个Channel类，把 `acceptSocket_` 及其感兴趣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由哪个EventLoop负责循环监听以及处理相应事件，其实这个EventLoop就是main EventLoop。
- `newConnectionCallback_`：TcpServer构造函数中将 `TcpServer::newConnection( )` 函数注册给了这个成员变量。这个 `TcpServer::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。

3.3.3 Acceptor封装的重要成员方法

- `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` 及其感兴趣事件（可读事件）注册到main EventLoop的事件监听器上。换言之就是让main EventLoop事件监听器去监听 `acceptSocket_`
- `handleRead( )`：这是一个私有成员方法，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main 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。
  简单说一下这个 `handleRead( )` 最终实现的功能是什么，接受新连接，并且以负载均衡的选择方式选择一个sub EventLoop，并把这个新连接分发到这个subEventLoop上。

3.4 Socket类

```
public:
    int fd() const {return sockfd_;}
    void bindAddress(const InetAddress &localaddr); //调用bind绑定服务器IP端口
    void listen(); //调用listen监听套接字
    int accept(InetAddress *peeradd); //调用accept接受新客户连接请求
    void shutdownWrite();  //调用shutdown关闭服务端写通道

    /**  下面四个函数都是调用setsockopt来设置一些socket选项  **/
    void setTcpNoDelay(bool on); //不启用naggle算法，增大对小数据包的支持
    void setReuseAddr(bool on);
    void setReusePort(bool on);
    void setKeepAlive(bool on);

private:
    const int sockfd_; //服务器监听套接字文件描述符
```


直接看它上面的代码注释就懂了。

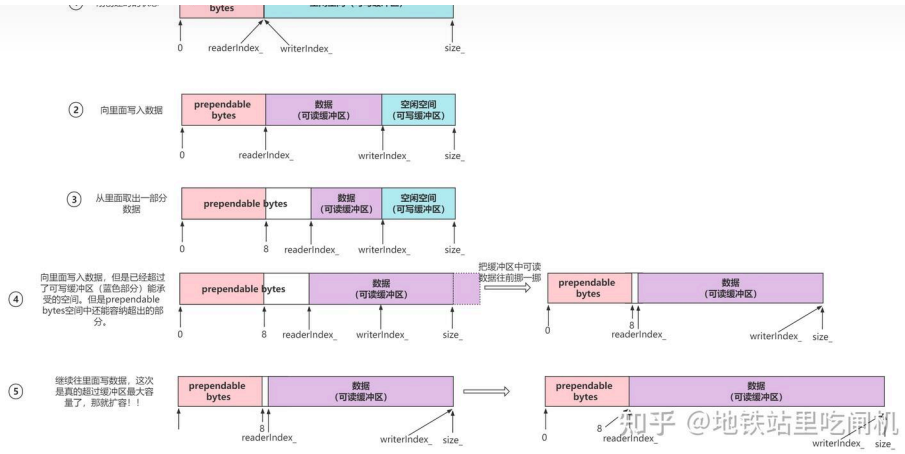
3.5 Buffer类

3.5.1 Buffer概述:

Buffer类其实是封装了一个用户缓冲区，以及向这个缓冲区写数据读数据等一系列控制方法。

3.5.2 Buffer类主要设计思想（读写配合，缓冲区内部调整以及动态扩容）

我个人觉得这个缓冲区类的实现值得参考和借鉴，以前自己写的只支持一次性全部读出和写入，而这个Buffer类可以读一点，写一点，内部逻辑稳定。这个Buffer类是 `vector<char>`（方便动态扩



Buffer类设计思想，向Buffer读写数据会遇到的问题以及Buffer类内部如何解决这些问题

其中需要关注的一个思想就是，随着写入数据和读入数据，蓝色的空闲空间会越来越少，prependable空间会越来越大，当什么时候空用空间耗尽了，就会向步骤4一样，把所有数据拷贝前移，重新调整。

另外当整个缓冲区的prependable空间和蓝色的空闲空间都无法装下新来的数据时，那就会调用vector的resize，实现扩容机制。

3.5.3 重要的成员方法：

- append(const char* data, size_t len)：将data数据添加到缓冲区中。
- retrieveAsString(size_t len); :获取缓冲区中长度为len的数据，并以strnig返回。
- retrieveAllString();获取缓冲区所有数据，并以string返回。
- ensureWritableByts(size_t len);当你打算向缓冲区写入长度为len的数据之前，先调用这个函数，这个函数会检查你的缓冲区可写空间能不能装下长度为len的数据，如果不能，就动态扩容。

下面两个方法主要是封装了调用了上面几个方法：

- ssize_t Buffer::readFd(int fd, int* saveErrno);：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。
- ssize_t Buffer::writeFd(int fd, int* saveErrno);：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。

其实readFd和writeFd函数的设计还有一些值得讨论的地方，这个放在以后再讲把。

3.6 TcpConnection 类

3.6.1 概述

在上面讲Acceptor的时候提到了这个TcpConnection类。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（连接建立和关闭和销毁），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，以及这个TCP连接的服务端和客户端的套接字地址信息等。

我个人觉得TcpConnection类和Acceptor类是兄弟关系，Acceptor用于main EventLoop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等），TcpConnection用于SubEventLoop中，对连接套接字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、错误事件等）。

3.6.2 TcpConnection的重要变量

- socket_：用于保存已连接套接字文件描述符。
- channel_：封装了上面的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种保存的各类事件的处理函数是在TcpConnection对象构造函数中注**

- inputBuffer_：这是一个Buffer类，是该TCP连接对应的用户接收缓冲区。
- outputBuffer_：也是一个Buffer类，不过是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，假如达到了高水位线，就没办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。
- state_：这个成员变量标识了当前TCP连接的状态（Connected、Connecting、Disconnecting、Disconnected）
- connetionCallback_、messageCallback_、writeCompleteCallback_、closeCallback_：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**后**的处理函数]以及Muduo库中定义的[连接关闭**后**的处理函数]。这四个函数都会分别注册给这四个成员变量保存。

3.6.3 TcpConnection的重要成员方法：

`handleRead()` 、 `handleWrite()` 、 `handleClose()` 、 `handleError()` ：
这四个函数都是私有成员方法，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。

- `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（ `inputBuffer_` ），然后再调用 `connectionCallback_` 保存的 [连接建立后的处理函数]。
- `handleWrite( )` 负责处理Tcp连接的可写事件。这个函数的情况有些复杂，留到下一篇讲解。
- `handleClose( )` 负责处理Tcp连接关闭的事件。大概的处理逻辑就是将这个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。这 `closeCallback_` 中保存的函数是由Muduo库提供的，`connectionCallback_` 保存的回调函数则由用户提供的（可有可无其实）

三、主线篇

本来一开始写这个博客的时候，很想把每一行代码全给分析一遍，后来越整理越乱，最后直接放弃了，所以这里决定对Muduo库的几条大的主线进行脉络梳理。

TCP网络编程的本质其实是处理下面这几个事件：

- 连接的建立。
- 连接的断开：包括主动断开和被动断开。
- 消息到达，客户端连接文件描述符可读。
- 消息发送，向客户端连接文件描述符写数据。

所以我们这一篇内容也是围绕上面四个主线展开！

1. 用Muduo库搭一个最简单的Echo服务器

要理顺Muduo库的代码逻辑，当然还是先学会一下怎么使用吧，这一小节就看代码注释就好了。一定要会最简单的使用。

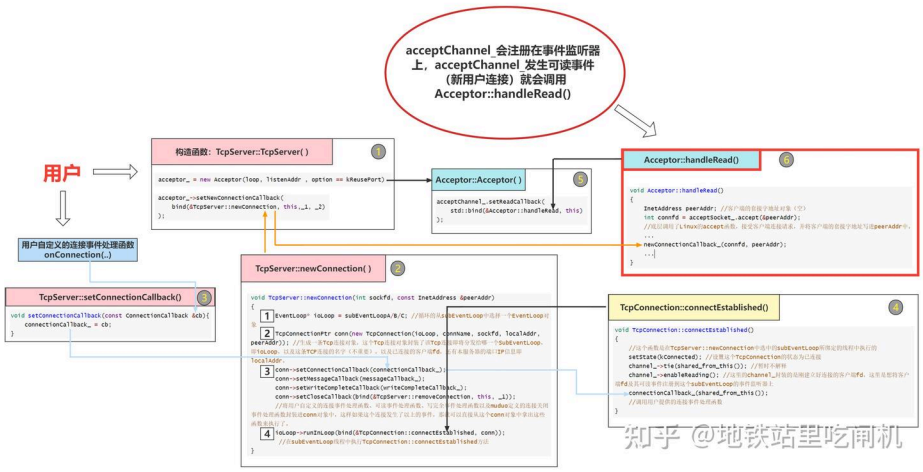
```
int main()
{
    EventLoop loop;
    //这个EventLoop就是main EventLoop，即负责循环事件监听处理新用户连接事件的事件循环器。第一

    InetAddress addr(4567);
```

```
    server.start();
    //启动TcpServer服务器

    loop.loop(); //执行EventLoop::Loop()函数，这个函数在概述篇的EventLoop小节有提及，自己去
    return 0;
}

class EchoServer
{
public:
    EchoServer(EventLoop *loop,
            const InetAddress &addr,
            const std::string &name)
        : server_(loop, addr, name)
        , loop_(loop)
    {
        server_.setConnectionCallback(
            std::bind(&EchoServer::onConnection, this, std::placeholders::_1)
        );
        // 将用户定义的连接事件处理函数注册进TcpServer中，TcpServer发生连接事件时会执行onConr

        server_.setMessageCallback(
            std::bind(&EchoServer::onMessage, this,
                std::placeholders::_1, std::placeholders::_2, std::placeholders::_3)
        );
        //将用户定义的可读事件处理函数注册进TcpServer中，TcpServer发生可读事件时会执行onMessc


        server_.setThreadNum(3);
        //设置sub reactor数量，你这里设置为3，就和概述篇图2中的EventLoop2 EventLoop3 Eventl
    }
    void start(){
        server_.start();
    }
private:
    void onConnection(const TcpConnectionPtr &conn)
    {
        //用户定义的连接事件处理函数：当服务端接收到新连接建立请求，则打印Connection UP，如果是
        if (conn->connected())
            LOG_INFO("Connection UP : %s", conn->peerAddress().toIpPort().c_str());
        else
            LOG_INFO("Connection DOWN : %s", conn->peerAddress().toIpPort().c_str());
    }

    void onMessage(const TcpConnectionPtr &conn,
                   Buffer *buf,
                   Timestamp time)
    {
        //用户定义的可读事件处理函数：当一个Tcp连接发生了可读事件就把它这个接收到的消息原封不动
        std::string msg = buf->retrieveAllAsString();
        conn->send(msg);
        conn->shutdown();
    }
    EventLoop *loop_;
    TcpServer server_;
};
```

1. 建立事件循环器EventLoop: `EventLoop loop;`
2. 建立服务器对象，即TcpServer类对象： `TcpServer server_;`
3. 向TcpServer注册各类事件的用户自定义的处理函数: `setMessageCallback()` 、

2. 建立连接

2.1 连接建立的代码逻辑



当acceptChannel_发生可读事件时，即新用户连接时，就会调用`Acceptor::handleRead()函数`

注意下面的标号分别对应上图中的代码方框标号！！！

1: `TcpServer::TcpServer( )`
当我们创建一个TcpServer对象，即执行代码 `TcpServer server(&loop, listenAddr);` 调用了TcpServer的构造函数，TcpServer构造函数最主要的就是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()` 。

5: `Acceptor::Acceptor( )`
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_` ，该Channel对象封装了服务器监听套接字文件描述符（尚未注册到main EventLoop的事件监听器上）。
接着Acceptor构造函数将 `Acceptor::handleRead( )` 方法注册进 `acceptChannel_` 中，这也意味着，日后如果事件监听器监听到 `acceptChannel_` 发生可读事件，将会调用 `AcceptorC::handleRead( )` 函数。

至此，TcpServer对象创建完毕，用户调用 `TcpServer::start( )` 方法，开启TcpServer。我们来直接看一下 `TcpServer::start( )` 方法都干了什么。

```
/**** TcpServer.cc *******/
void TcpServer::start() //开启服务器监听
{
    省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略
    loop_->runInLoop(std::bind(&Acceptor::listen, acceptor_.get()));
    //让这个EventLoop，也就是mainLoop来执行Acceptor的listen函数，开启服务端监听
    省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略 省略
}
/***** Acceptor.cc ********/
void Acceptor::listen()
{
    listenning_ = true;
    acceptSocket_.listen();
    acceptChannel_.enableReading();
}
```

其实就是将其实主要就是调用 `Acceptor::listen( )` 函数（底层是调用了linux的函数 `listen(`

监听器的监听结果，并且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。

6: `Acceptor::handleRead( )`
当程序如果执行到了这个函数里面，说明 `acceptChannel_` 发生可读事件，程序处理新客户连接请求。
该函数首先调用了Linux的函数 `accept( )` 接受新客户连接。
接着调用了 `TcpServer::newConnection( )` 函数，这个函数是在步骤1中注册给Acceptor并由成员变量 `newConnectionCallback_` 保存。

7: `TcpServer::newConnection( )`
该函数的主要功能就是将建立好的连接进行封装（封装成TcpConnection对象），并使用选择算法公平的选择一个sub EventLoop，并调用 `TcpConnection::connectEstablished( )` 将 `TcpConnection::channel_` 注册到刚刚选择的sub EventLoop上。

2.2 编程细节启发！！！什么时候用智能指针管理对象最合适！！！

我平时对智能指针的使用缺乏权衡感。在一些情况下使用智能指针会带来额外的性能开销，所以不能无脑梭哈。但是智能指针又能保护的内存安全。但是这里的编程细节给了我一些启发。

```
/******** Callbacks.h  ********/
using TcpConnectionPtr = std::shared_ptr<TcpConnection>;
/******** TcpServer.cc ********/
void TcpServer::newConnection(int sockfd, const InetAddress &peerAddr)
{
    代码省略
        TcpConnectionPtr conn(new TcpConnection(ioLoop, connName, sockfd, localAddr, p
    connections_[connName] = conn;
    代码省略
    ioLoop->runInLoop(bind(&TcpConnection::connectEstablished, conn));
}
```

在 `TcpServer::newConnection()` 函数中，当接受了一个新用户连接，就要把这个Tcp连接封装成一个TcpConnection对象，也就是上面代码中的 `new TcpConnection(...)` 。然后用一个共享型智能指针来管理这个对象。**所以为什么这里要把TcpConnection用智能指针来管理啊?**

这里使用智能指针管理TcpConnetion的最重要原因在于防止指针悬空，而指针悬空可能会来自以下这三个方面：
1.**TcpConnection会和用户直接交互，用户可能会手欠删除。**
在我们编写Echo服务器的时候，我们用户可以自定义连接事件发生后的处理函数（如下所示），并将这个函数注册到TcpServer中。

```
/**** 用户自定义的连接事件发生后的处理函数 *****/
void onConnection(const TcpConnectionPtr &conn)
{
    .......
}
```

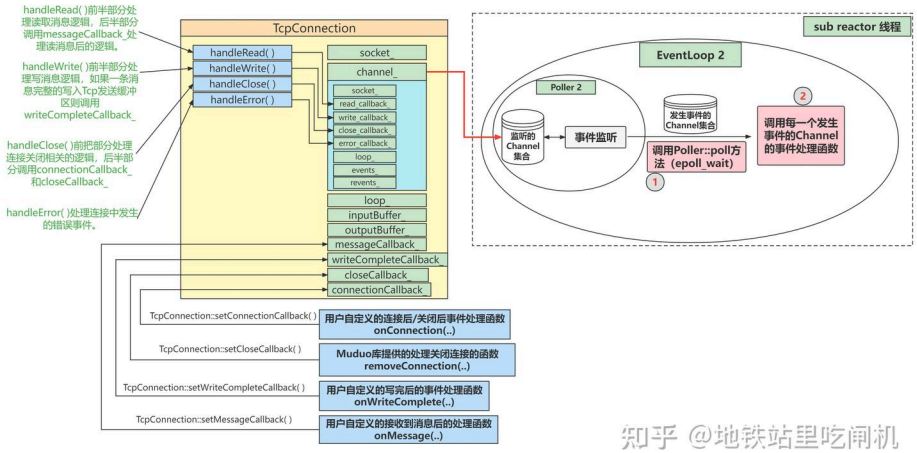
假如这里的onConnection函数传入的是TcpConnection而不是TcpConnectionPtr，用户在onConnection函数中把TcpConnection对象给Delete了怎么办？删除了之后，程序内部还要好几处地方都在使用TcpConnection对象。结果这个对象的内存突然消失了，服务器访问非法内存崩溃。虽然这一系列连锁反应会让人觉得用户很笨。但是作为设计者的我们必须要保证，**编程设计不可以依赖用户行为，一定要尽可能地封死用户的误操作。**所以这里用了共享智能指针。

2. **TcpConnection对象的多线程安全问题：**
假如服务器要关闭了这个时候MainEventLoop线程中的 `TcpServer::~TcpServer()` 函数开始把所有TcpConnection对象都删掉。那么其他线程还在使用这个TcpConnection对象，如果你把它的内存空间都释放了，其他线程访问了非法内存，会直接崩溃。

这种情况我们留到将连接关闭的时候再来讨论，这一部分也是有很好的编程启发的！

3. 消息读取

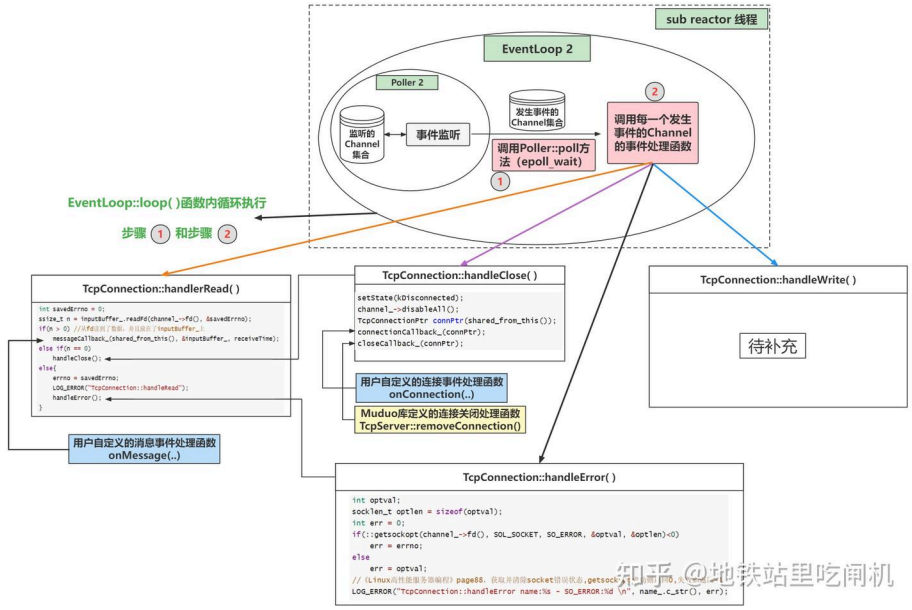
3.1 前情回顾



TcpConnection分发到某一个SubEventLoop后

在MainEventLoop中接受新连接请求之后，将这条Tcp连接封装成TcpConnection对象。TcpConnection对象的内容如上图所示，主要就是封装了连接套接字的fd(上图中的 `socket_`)、连接套接字的 `channel_` 等。在TcpConnection的构造函数中会将TcpConnection::handleRead( )等四个上图中的蓝色方法注册进这个 `channel_` 内。
当TcpConnection对象建立完毕之后，MainEventLoop的Acceptor会将这个TcpConnection对象中的 `channel_` 注册到某一个SubEventLoop中。

3.2 消息读取逻辑



如上图所示，SubEventLoop中的 `EventLoop::loop()` 函数内部会**循环的执行**上图中的步骤1和步骤2。步骤1就是调用 `Poller::poll()` 方法获取事件监听结果，这个事件监听结果是一个Channel集合，每一个Channel封装着 [一个fd] 及 [fd感兴趣的事件] 和 [事件监听器监听到该fd实际发生的事件]。步骤2就是调用每一个Channel的 `Channel::HandlerEvent` 方法。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（ `readCallback_` 、 `writeCallback_` 、 `closeCallback_` 、 `errorCallback_`)。

```
void TcpConnection::handleRead(TimeStamp receiveTime)
{
    int savedErrno = 0;
    ssize_t n = inputBuffer_.readFd(channel_->fd(), &savedErrno);
    if(n > 0) //从fd读到了数据，并且放在了inputBuffer_上
    {
        messageCallback_(shared_from_this(), &inputBuffer_, receiveTime);
    }
    else if(n == 0)
        handleClose();
    else
    {
        errno = savedErrno;
        LOG_ERROR("TcpConnection::handleRead");
        handleError();
    }
}
```

TcpConnection::handleRead( ) 函数首先调用 Buffer_.readFd(channel_->fd(), &saveErrno) ，该函数底层调用Linux的函数 readv( ) ，将Tcp接收缓冲区数据拷贝到用户定义的缓冲区中（ inputBuffer_ ）。如果在读取拷贝的过程中发生了什么错误，这个错误信息就会保存在 savedErrno 中。
当 readFd( ) 返回值大于0，说明从接收缓冲区中读取到了数据，那么会接着调用 messageCallback_ 中保存的用户自定义的读取消息后的处理函数。
readFd( ) 返回值等于0，说明客户端连接关闭，这时候应该调用 TcpConnection::handleClose( ) 来处理连接关闭事件
readFd( ) 返回值等于-1，说明发生了错误，调用 TcpConnection::handleError( ) 来处理 savedErrno 的错误事件。Moduo库只支持LT模式，所以读事件不会出现EAGAIN的错误，所以一旦出现错误，说明肯定是比较不好的非正常错误了。而EAGAIN错误只不过是非阻塞IO调用时的一种常见错误而已。

3.3 Buffer::readFd()函数剖析：

剖析这个函数是因为这个函数的设计有可取之处。这个readFd巧妙的设计，可以让用户一次性把所有TCP接收缓冲区的所有数据全部都读出来并放到用户自定义的缓冲区Buffer中。
用户自定义缓冲区Buffer是有大小限制的，我们一开始不知道TCP接收缓冲区中的数据量有多少，如果一次性读出来会不会导致Buffer装不下而溢出。所以在 readFd( ) 函数中会在栈上创建一个临时空间 extrabuf ，然后使用 readv 的分散读特性，将TCP缓冲区中的数据先拷贝到Buffer中，如果Buffer容量不够，就把剩余的数据都拷贝到 extrabuf 中，然后再调整Buffer的容量(动态扩容)，再把 extrabuf 的数据拷贝到Buffer中。当这个函数结束后， extrabuf 也会被释放。另外 extrabuf 是在栈上开辟的空间，速度比在堆上开辟还要快。

```
ssize_t Buffer::readFd(int fd, int* saveErrno)
{
    char extrabuf[65536] = {0}; //栈上的内存空间
    struct iovec vec[2];
    const size_t writableSpace = writableBytes(); //可写缓冲区的大小
    vec[0].iov_base = begin() + writerIndex_; //第一块缓冲区
    vec[0].iov_len = writableSpace; //当我们用readv从socket缓冲区读数据，首先会先填满这个ve
    vec[1].iov_base = extrabuf; //第二块缓冲区，如果Buffer缓冲区都填满了，那就填到我们临时创
    vec[1].iov_len = sizeof(extrabuf); //栈空间上。
    const int iovcnt = (writableSpace < sizeof(extrabuf) ? 2 : 1);
    const ssize_t n = ::readv(fd, vec, iovcnt);
    if(n < 0){
        *saveErrno = errno; //出错了！！
    }
    else if(n <= writableSpace){ //说明Buffer空间足够存了
        writerIndex_ += n;  //
```

```
    }
    return n;
}
```

4. 消息发送

4.1 消息发送的代码逻辑

当用户调用了 `TcpConnetion::send(buf)` 函数时，相当于要求muduo库把数据 `buf` 发送给该Tcp连接的客户端。此时该TcpConnection注册在事件监听器上的感兴趣事件中是没有可写事件的。`TcpConnection::send(buf)` 函数内部其实是调用了Linux的函数 `write( )` （别说你不知道 `write( )` 是啥）
如果TCP发送缓冲区能一次性容纳buf，那这个 `write( )` 函数将buf全部拷贝到发送缓冲区中。

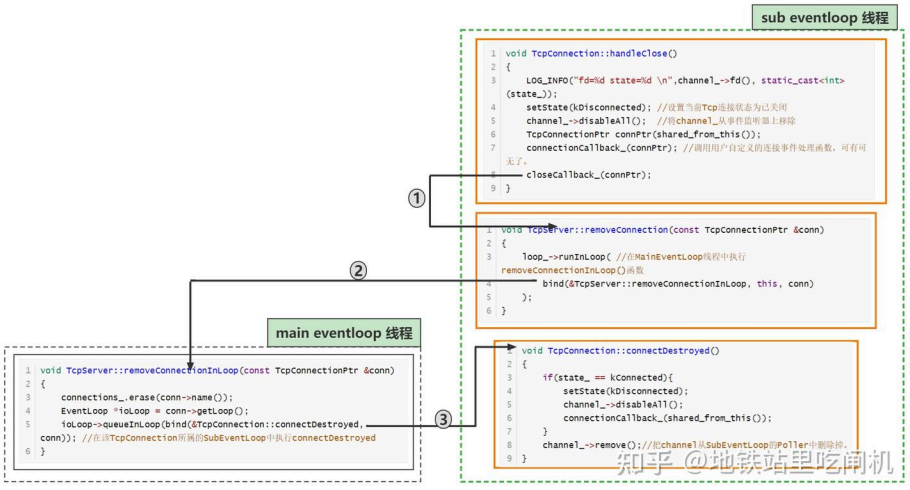
如果TCP发送缓冲区内不能一次性容纳buf：

- 这时候 `write( )` 函数 `buf` 数据尽可能地拷贝到TCP发送缓冲区中，并且将errno设置为 `EWOULDBLOCK` 。
- 剩余未拷贝到TCP发送缓冲区中的 `buf` 数据会被存放在 `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连接可写事件，就会调用 `TcpConnection::handleWrite( )` 函数把 `TcpConnection::outputBuffer_` 中剩余的数据发送出去。
  - 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，如果Tcp发送缓冲区能容纳全部剩余的未发送数据，那最好不过了。如果Tcp发送缓冲区依旧没法容纳剩余的未发送数据，那就尽可能地将数据拷贝到Tcp发送缓冲区中，继续保持可写事件的监听。
- 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。**（移除可写事件是为了提高效率，不会让 `epoll_wait()` 毫无意义的频繁触发可写事件。因为大多数时候是没有数据需要发送的，频繁触发可写事件但又没有数据可写。）

5.连接断开

5.1 连接被动断开

服务端 `TcpConnection::handleRead()` 中感知到客户端把连接断开了。`TcpConnection::handleRead( )` 函数内部调用了Linux的函数 `readv( )` ，当 `readv( )` 返回0的时候，服务端就知道客户端断开连接了。然后就接着调用 `TcpConnection::handleClose( )` 。



上图中的标号1、2、3是函数调用顺序，我们可以看到：

`TcpServer::removeConnection( )` 函数

2. `TcpServer::removeConnection( )` 函数调用了 `remvoveConnectionInLoop( )` 函数，该函数的运行是在MainEventLoop线程中执行的，这里涉及到线程切换技术，后面再讲。
3. `removeConnectionInLoop( )` 函数：TcpServer对象中有一个 `connections_` 成员变量，这是一个unordered_map，负责保存【string --> TcpConnection】的映射，其实就是保存着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_` 中删掉。然后再调用 `TcpConnection::connectDestroyed` 函数。
   另外为什么removeConnectionInLoop()要在MainEventLoop中运行，因为该函数主要是从TcpServer对象中删除某条数据。而TcpServer对象是属于MainEventLoop的。这也是贯彻了One Loop Per Thread的理念。
4. `TcpConnection::connectDestroyed( )` 函数的执行是又跳回到了subEventLoop线程中。该函数就是将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内的数据结构中删除。

再提醒一下，线程切换后面会讲，这里不知道为什么函数的执行为什么可以在线程之间来回跳来跳去的话，先将就将就。

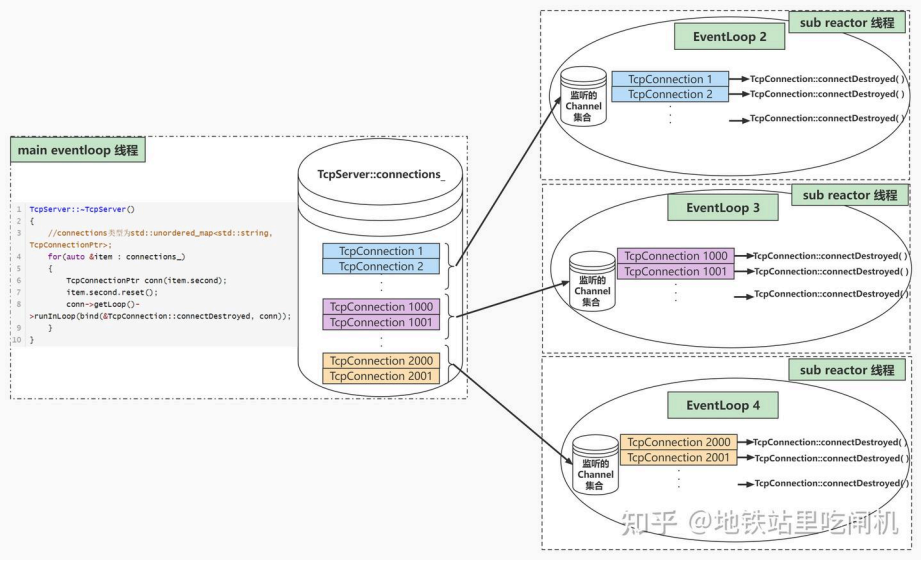
5.2 服务器主动关闭导致连接断开

当服务器主动关闭时，调用 `TcpServer::~TcpServer()` 析构函数。

我们之间讨论了为什么TcpConnection要用智能指针管理，留下了一个TcpConnection对象的多线程安全问题没讨论。这里涉及到两个很好的编程设计思想。好好学起来！

5.2.1 TcpConnection对象的析构问题：

这里在提示一下 `EventLoop::runInLoop()` 函数的意义，假如你有一个EventLoop对象 `loop_` ，当你调用了 `loop_->runInLoop(function)` 函数时，这个 `function` 函数的执行会在这个 `loop_` 绑定的线程上运行！
所以我们画了下面这幅图，在创建TcpConnection对象时，Acceptor都要将这个对象分发给一个SubEventLoop来管理。这个TcpConnection对象的一切函数执行都要在其管理的SubEventLoop线程中运行。再一次贯彻One Loop Per Thread的设计模式。比如**要想彻底删除一个TcpConnection对象，就必须要调用这个对象的 `connecDestroyed()` 方法，这个方法执行完后才能释放这个对象的堆内存。**每个TcpConnection对象的 `connectDestroyed()` 方法都必须在这个TcpConnection对象所属的SubEventLoop绑定的线程中执行。



所有上面的 `TcpServer::~TcpServer()` 函数就是干这事儿的，不断循环的让这个TcpConnection对象所属的SubEventLoop线程执行 `TcpConnection::connectDestroyed()` 函数，同时在

TcpConnectino对象的堆内存释放操作是在 `TcpConnection::connectDestroyed()` 调用完后。
这个析构函数巧妙利用了共享智能指针的特点，当没有共享智能指针指向这个TcpConnection对象时（引用计数为0），这个TcpConnection对象就会被析构删除（堆内存释放）。

我们解读一下TcpServer::~TcpServer()中的代码逻辑

```
TcpServer::~TcpServer()
{
    //connections类型为std::unordered_map<std::string, TcpConnectionPtr>;
    for(auto &item : connections_)
    {
        TcpConnectionPtr conn(item.second);
        item.second.reset();
        conn->getLoop()->runInLoop(bind(&TcpConnection::connectDestroyed, conn));
    }
}
```

- 首先 `TcpServer::connections_` 是一个 `unordered_map<string, TcpConnectionPtr>` ，其中 `TcpConnectionPtr` 的含义是指向TcpConnection的shared_ptr。
- 在一开始，每一个TcpConnection对象都被一个共享智能指针TcpConnetionPtr持有，当执行了 `TcpConnectionPtr conn(item.second)` 时，这个TcpConnetion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，只要离开了当前的这一次for循环， `conn` 就会被释放。
- 紧接着调用 `item.second.reset()` 释放掉TcpServer中保存的该TcpConnectino对象的智能指针。此时在当前情况下，只剩下 `conn` 还持有这个TcpConnection对象，因此当前TcpConnection对象还不会被析构。
- 接着调用了 `conn->getLoop()->runInLoop(bind(&TcpConnection::connectDestroyed, conn));`
  这句话的含义是让SubEventLoop线程去执行 `TcpConnection::connectDestroyed()` 函数。**当你把这个conn的成员函数传进去的时候，conn所指向的资源的引用计数会加1。因为传给runInLoop的不只有函数，还有这个函数所属的对象conn。**
- SubEventLoop线程开始运行 `TcpConnection::connectDestroyed()`
- MainEventLoop线程当前这一轮for循环跑完，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，而且这个智能指针在 `TcpConnection::connectDestroyed()` 中，只不过这个智能指针你看不到，它在这个函数中是一个隐式的this的存在。当这个函数执行完后，智能指针就真的被释放了。到此，就没有任何智能指针指向这个TcpConnection对象了。TcpConnection对象就彻底被析构删除了。

5.2.2 如果TcpConnection中有正在发送的数据，怎么保证在触发TcpConnection关闭机制后，能先让TcpConnection先把数据发送完再释放TcpConnection对象的资源?

这个问题就要好好参考这部分代码的设计了，这部分代码也是很值得吸收的精华。

```
/***** TcpConnection.cc *****/
void TcpConnection::connectEstablished()
{
    setState(kConnected);
    channel_->tie(shared_from_this());
    channel_->enableReading(); //向poller注册channel的epollin事件
    //新连接建立，执行回调
    connectionCallback_(shared_from_this());
}
```

我们先了解一下 `shared_from_this()` 是什么意思，首先TcpConnection类继承了一个类（其实这一部分当然最好是自行谷歌了，我这里提一嘴而已。） 继承了这个类之后才能使用shared_from_this()函数。

TCA。
接着这个shared_ptr就作为channel_的 `Channel::tie()` 函数的函数参数。

```
/*****  Channel.h   ******/
std::weak_ptr<void> tie_;
/*****  Channel.cc   ******/
void Channel::tie(const shared_ptr<void>& obj)
{
    tie_ = obj;
    tied_ = true;
}
void Channel::HandlerEvent(TimeStamp receiveTime)
{
    if(tied_){
        shared_ptr<void> guard = tie_.lock();
        if (guard)
            HandleEventWithGuard(receiveTime);
    }
    else{
        。。。。一般不会执行到这里其实。我实在想不到正常运行的情况下怎么会执行到这里，可能是我比
        HandleEventWithGuard(receiveTime);
    }
}
```

当事件监听器返回监听结果，就要对每一个发生事件的channel对象调用他们的 `HandlerEvent()` 函数。在这个 `HandlerEvent` 函数中，会先把 `tie_` 这个weak_ptr提升为强共享智能指针。这个强共享智能指针会指向当前的TcpConnection对象。就算你外面调用删除析构了其他所有的指向该TcpConnection对象的智能指针。你只要 `HandleEventWithGuard()` 函数没执行完，你这个TcpConnetion对象都不会被析构释放堆内存。而 `HandleEventWithGuard()` 函数里面就有负责处理消息发送事件的逻辑。当 `HandleEventWithGuard()` 函数调用完毕，这个 `guard` 智能指针就会被释放。

四、线程篇 — One Loop Per Thread

One Loop Per Thread的含义就是，一个EventLoop和一个线程唯一绑定，和这个EventLoop有关的，被这个EventLoop管辖的一切操作都必须在这个EventLoop绑定线程中执行，比如在MainEventLoop中，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，比如接收数据发送数据，连接断开等事件处理都必须在这个SubEventLoop线程上运行，还不准跑到别的SubEventLoop线程上运行。那这到底到底怎么实现的呢？这里终于有机会讲了。

1. 预备知识：eventfd()的使用

- 函数原型：

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

调用函数 `eventfd()` 会创建一个eventfd对象，或者也可以理解打开一个eventfd类型的文件，类似普通文件的open操作。eventfd的在内核空间维护一个**无符号64位整型计数器**， 初始化为 `initval` 的值。

- flags是以下三个标志位OR结果

read操作的时候将会阻塞直到计数器中有值，如果设置了这个这个标志位，计数器没有值得时候也会立刻返回-1。
    - `EFD_SEMAPHORE(2.6.30～)`：这个标志位会影响read操作。后面讲。
- 使用方法：
  - write向eventfd中写值
    - 如果写入的值和小于 `0xFFFFFFFFFFFFFFFE` 则写入成功
    - 如果大于 `0xFFFFFFFFFFFFFFFE`
      - 如果设置了EFD_NONBLOCK标志位就直接返回-1
      - 如果没有设置EFD_NONBLOCK标志位就会一直阻塞到read操作执行。
  - read读取eventfd的值：
    - 如果计数器中的值大于0
      - 如果设置了 `EFD_SEMAPHORE` 标志位，则返回1，且计数器的值减去1
      - 如果没有设置 `EFD_SEMAPHORE` 标志位，则返回计数器中的值，并且设置计数器值为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.

2. 如何保证一个EventLoop对象和一个线程唯一绑定（该线程只能绑定一个EventLoop对象，该EventLoop对象也必须绑定一个线程）

下图是EventLoop构造函数（我把不相关的代码全删了。所以看上去可能会有一点点光秃秃）

```
/***** EventLoop.cc *****/
__thread EventLoop *t_loopInThisThread = nullptr;

EventLoop::EventLoop() :
    wakeupFd_(createEventfd()), //生成一个eventfd，每个EventLoop对象，都会有自己的eventfd
        wakeupChannel_(new Channel(this, wakeupFd_))
{
    LOG_DEBUG("EventLoop created %p in thread %d \n", this, threadId_);
    if(t_loopInThisThread) //如果当前线程已经绑定了某个EventLoop对象了，那么该线程就无法创建
        LOG_FATAL("Another EventLoop %p exits in this thread %d \n", t_loopInThisThrea
    else
        t_loopInThisThread = this;
    wakeupChannel_->setReadCallback(std::bind(&EventLoop::handleRead, this));
    wakeupChannel_->enableReading();
}
```

介绍一下这个 `__thread` ，这个 `__thread` 是一个关键字，被这个关键字修饰的全局变量 `t_loopInThisThread` 会具备一个属性，那就是该变量在每一个线程内都会有一个独立的实体。因为一般的全局变量都是被同一个进程中的多个线程所共享，但是这里我们不希望这样。
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nullptr，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！

3. Muduo库如何实现每个EventLoop线程只运行隶属于该EventLoop的操作?

这个小标题有点拗口，这一小节主要解决这样的一个问题，比如在MainEventLoop中，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上之后，这个已建立连接的任何操作，比如接收数据发送数据，连接断开等事件处理都**必须在这个SubEventLoop线程上运行，不准跑到别的SubEventLoop线程上运行。**

EventLoop构造函数的初始化列表中，如下所示：

```
int createEventfd()
{
```

```
    return evtfd;
}
EventLoop::EventLoop()
    : wakeupFd_(createEventfd()), //生成一个eventfd，每个EventLoop对象，都会有自己的eventf
        ...
{...}
```

在EventLoop的初始化列表中：

- `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_` 。
- 随即将 `wakeupFd_` 用Channel封装起来，得到 `wakeupChannel_` 。接着在EventLoop构造函数中

在EventLoop的构造函数体内：

- 先是给这个Channel注册一个读事件处理函数 `EventLoop::handleRead()`
- 随即将这个 `wakeupChannel_` 注册到事件监听器上监听其可读事件。当事件监听器监听到 `wakeupChannel_` 的可读事件时就会调用 `EventLoop::handleRead()` 函数。

我们来描绘一个情景，我们知道每个EventLoop线程主要就是在执行其EventLoop对象的loop函数（该函数就是一个while循环，循环的获取事件监听器的结果以及调用每一个发生事件的Channel的事件处理函数）。此时SubEventLoop上注册的Tcp连接都没有任何动静，整个SubEventLoop线程就阻塞在 `epoll_wait()` 上。
此时MainEventLoop接受了一个新连接请求，并把这个新连接封装成一个TcpConnection对象，并且希望在SubEventLoop线程中执行 `TcpConnection::connectEstablished()` 函数，因为该函数的目的是将TcpConnection注册到SubEventLoop的事件监听器上，并且调用用户自定义的连接建立后的处理函数。当该TcpConnection对象注册到SubEventLoop之后，这个TcpConnection对象的任何操作（包括调用用户自定义的连接建立后的处理函数。）都必须要在这个SubEventLoop线程中运行，所以 `TcpConnection::connectEstablished()` 函数必须要在SubEventLoop线程中运行。

那么我们怎么在MainEventLoop线程中通知SubEventLoop线程起来执行 `TcpConnection::connectEstablished()` 函数呢？这里就要好好研究一下 `EventLoop::runInLoop()` 函数了。

```
void EventLoop::runInLoop(Functor cb)
{//该函数保证了cb这个函数对象一定是在其EventLoop线程中被调用。
    if(isInLoopThread())//如果当前调用runInLoop的线程正好是EventLoop的运行线程，则直接执行此
        cb();
    else//否则调用 queueInLoop 函数
        queueInLoop(cb);
}
void EventLoop::queueInLoop(Functor cb)
{
    {
        unique_lock<mutex> lock(mutex_);
        pendingFunctors_.emplace_back(cb);
    }
    if(!isInLoopThread() || callingPendingFunctors_)
        wakeup();
}

void EventLoop::wakeup()
{
    uint64_t one = 1;
    ssize_t n = write(wakeupFd_, &one, sizeof(one));
    if(n != sizeof(n))
```

```
    looping_ = true;
    quit_ = false;
    LOG_INFO("EventLoop %p start looping \n", this);
    while(!quit_)
    {
        activeChannels_.clear();
        pollReturnTime_ = poller_->poll(kPollTimeMs, &activeChannels_);//此时activeChan
        for(Channel *channel : activeChannels_)
            channel->HandlerEvent(pollReturnTime_);
        doPendingFunctors(); //执行当前EventLoop事件循环需要处理的回调操作。
    }
}
void EventLoop::doPendingFunctors()
{
   std::vector<Functor> functors;
   callingPendingFunctors_ = true;
   {
       unique_lock<mutex> lock(mutex_);
       functors.swap(pendingFunctors_); //这里的swap其实只是交换的vector对象指向的内存空间
   }
   for(const Functor &functor:functors)
   {
       functor();
   }
   callingPendingFunctors_ = false;
}
```

首先 `EventLoop::runInLoop` 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，那么就直接执行cb函数。否则就把cb传给 `queueInLoop()` 函数。

在 `queueInLoop()` 函数中主要就是把cb这个可调用对象保存在EventLoop对象的 `pendingFunctors_` 这个数组中，我们希望这个cb能在某个EventLoop对象所绑定的线程上运行，但是由于当前cpu执行的线程不是我们期待的这个EventLoop线程，我们只能把这个可调用对象先存在这个EventLoop对象的数组成员 `pendingFunctors_` 中。

我们再把目光转移到上面代码中的 `EventLoop::loop()` 函数中。我们知道 `EventLoop::loop()` 肯定是运行在其所绑定的EventLoop线程内，在该函数内会调用 `doPendingFunctors()` 函数，这个函数就是把自己这个EventLoop对象中的 `pendingFunctors_` 数组中保存的可调用对象拿出来执行。 `pendingFunctors_` 中保存的是其他线程希望你这个EventLoop线程执行的函数。

还有一个问题，假如EventLoop A线程阻塞在 `EventLoop::loop()` 中的 `epoll_wait()` 调用上（EventLoop A上监听的文件描述符没有任何事件发生），这时候EventLoop线程要求EventLoopA赶紧执行某个函数，那其他线程要怎么唤醒这个阻塞住的EventLoopA线程呢？这时候我们就要把目光聚焦在上面的 `wakeup()` 函数了。

`wakeup()` 函数就是向我们想唤醒的线程所绑定的EventLoop对象持有的 `wakeupFd_` 随便写入一个8字节数据，因为 `wakeupFd_` 已经注册到了这个EventLoop中的事件监听器上，这时候事件监听器监听到有文件描述符的事件发生， `epoll_wait()` 阻塞结束而返回。这就相当于起到了唤醒线程的作用！你这个EventLoop对象既然阻塞在事件监听上，那我就通过 `wakeup()` 函数给你这个EventLoop对象一个事件，让你结束监听阻塞。

编辑于 2022-04-08 18:43

muduo　C++　网络编程

理性发言，友善互动

评论内容由作者筛选后展示

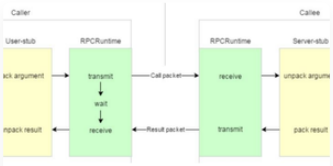
还没有评论，发表第一个评论吧

推荐阅读

muduo源码剖析

原文链接： muduo 源码剖析 muduo是陈硕大神个人开发的C++的TCP网络编程库。muduo基于Reactor模式实现。Reactor模式也是目前大多数Linux端高性能网络编程框架和网络应用所选择的主...

cyhone



基于muduo网络库+protobuf实现c++分布式网络通信框架

linux 发表于linux...

校招｜shein提前批Java南京现场一面

作者：鸡你太美美美美 链接：https://www.nowcoder.com/discu from=zhnkw来源：牛客网 首先这个公司的hr小姐姐以及面试官真的非常好，友善颜值又高，吹爆一面 面经:说说Java的util...

牛客 发表于笔经面经

面经：一篇文章总结大厂广告\推荐\搜索的机器学习面试（...

（因为需要整理，持续更新中） 前一阵准备换工作，四月到五月一个多月的时间面试了各家大厂，岗位都是广告、推荐、搜索相关。 上一篇文章看到好多人点赞+收藏，觉得正好趁机会把这些面试总...

丢丢