# 第3章 网络库设计与实现

## 3.1 线程操作的相关封装

### 3.1.1 互斥锁类MutexLock

具体实现见WebServer/Base/下的MutexLock.cc和MutexLock.h。MutexLock类内含实际的互斥锁pthread\_mutex\_t。MutexLock的构造函数会初始化互斥锁，MutexLock会实现lock和unlock方法，并在析构时销毁该互斥锁，析构时要先调用lock方法，确保没有存在lock阻塞的地方。

MutexLock.cc中提供了MutexLockGuard函数，构造时lock，析构时自动unlock。

### 3.1.2 条件变量类Condition

具体实现见WebServer/Base/下的Condition.cc和Condition.h。作为MutexLock的友元类，Condition类内含MutexLock和pthread\_cond\_t，构造时对cond\_初始化，析构时调用destroy注销cond\_，并提供wait和signal方法。

### 3.1.3 线程类Thread

具体实现见WebServer/Base/下的CurrentThread.h。namespace CurrentThread保存了当前线程的信息，通过syscall(SYS\_gettid)获取当前线程id。Thread类通过function保存了要执行的操作，真正的创建线程并执行在start里面。Thread类实现了join方法，用于join start里创建的线程。Thread的析构函数会将线程设置为detach状态。

## 3.2 Reactor实现

### 3.2.1 事件及对应操作的封装类Channel

具体实现见WebServer/Net/下的Channel.cc和Channel.h。在2.1.1中我们提到，Serveice Handle内含一个poller和事件-处理机制的对应关系）。poller容易理解，本质即是一个I/O复用，而单个事件的事件-处理机制对应关系需要用一个数据结构来存储，项目中命名为Channel。Channel主要包含以下方法

int fd\_;

int events\_;

int revents\_;

EventCallBack readCallBack\_;

EventCallBack writeCallBack\_;

EventCallBack errorCallBack\_;

EventCallBack closeCallBack\_;

void Channel::handleEvent()；

fd对应要处理的文件描述符，event\_表示要poller关注的事件集合，revent\_表示已经触发了的事件集合。在handleEvent中，程序会根据revents\_选择相应的事件处理程序（后缀为CallBack\_的方法）。

### 3.2.2 I/O复用类EPoller

具体实现见WebServer/Net/下的EPoll.cc和EPoll.h。这里的EPoller即是2.1.1中Serveice Handle包含的poller。本次网络库选用epoll进行I/O复用，epoll封装在EPoller中，并提供以下方法：

void epollAdd(Channel\* channel);

void epollMod(Channel\* channel);

void epollDel(Channel\* channel);

void poll(std::vector<Channel\*>& ret, int timeoutMs = kEpollWaitTime);

以epollAdd为例：

void EPoller::epollAdd(Channel\* channel) {

struct epoll\_event ev;

ev.events = channel->events();

ev.data.ptr = channel;

epoll\_ctl(epollfd\_, EPOLL\_CTL\_ADD, channel->fd(), &ev);

}

poll方法会通过一个vector返回相应的事件：

void EPoller::poll(std::vector<Channel\*>& ret, int timeoutMs) {

Int nfds= epoll\_wait(epollfd\_,&\*events\_.begin(), static\_cast<int>(events\_.size()), timeoutMs);

for (int idx = 0; idx < nfds; ++idx) {

epoll\_event ev = events\_[idx];

Channel\* channel = (Channel\*)ev.data.ptr;

channel->setRevents(ev.events);

ret.push\_back(channel);

}

}

### 3.2.3 事件管理类EventLoop

这里的EventLoop即是2.1.1中Serveice Handle，也是one loop per thread思想中的loop。EventLoop类包含一个EPoller用于探测响应事件，同时，EventLoop应该包含一个Channel集合ChannelList，EPoller返回事件时，EventLoop根据返回的事件类型查找ChannelList，找到对应的channel后，调用channel的handleEvent处理该事件。EventLoop时序图如下：

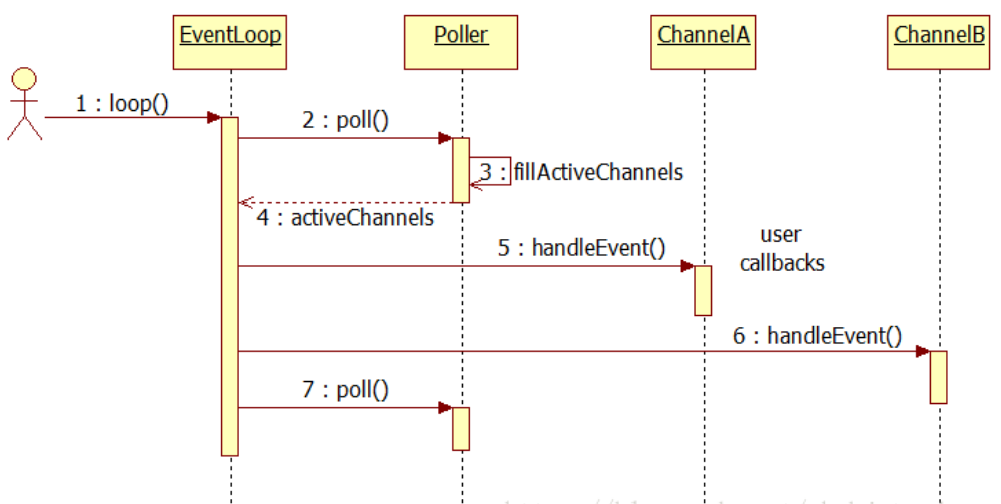


图 3-1 EventLoop时序图

EventLopp的核心在于poll函数，实际上就是调用了EPoller中的poll方法来返回相应的事件列表activeChannels，然后再进行事件分发。EventLoop的poll实现如下：

void EventLoop::loop() {

looping\_ = true;

quit\_ = false;

while (!quit\_) {

activeChannels\_.clear();

poller\_->poll(activeChannels\_);

for (ChannelList::iterator it = activeChannels\_.begin();

it != activeChannels\_.end(); ++it) {

(\*it)->handleEvent();

}

}

looping\_ = false;

}

### 3.2.4 统一计时器事件处理

不同于libevent中通过设置poller超时时间来实现定时，我们使用timerfd及其相关的方法来实现定时的功能，时间到了，timerfd就会触发。我们通过一个有序的数据结构来管理计时器事件，并设置timerfd的响应时间为最近计时器事件中的时间。这样，timerfd一触发，我们就去处理超时的计时器事件。

由此可见，我们可以把timerfd和处理超时事件这个过程封装为一个Channel。上文提到了要用一个有序的数据结构来管理计时器事件，一个计时器事件包括一个时间戳和一个动作function。下文为选用数据结构的思考过程。首先，我们考虑用线性表来实现，插入事件时排序，这样插入和删除都是线性时间的。简单优化，用堆来进行组织，事件复杂度降为logN，但是标准库中的priority\_queue并不支持高效的堆中间数据的增删。考虑用红黑树实现，但直接用时间戳作为key的话，不能处理时间戳相同的计时器事件。最终，我们选择用时间戳-计时器事件对（STL中pair）来作为key，用set管理计时器事件。

为了各个类的功能更加独立，我们把上文提到的timerfd的channel和计时器时间管理set再封装为一个TimerQueue类，channel中响应事件主要做的动作就是通过调用getExpired得到所有的超时事件，然后执行各个计时器事件对应的动作。

注意，时间戳设置我们用的是绝对时间。Timerfd设置超时时间用的是相对时间。

统一计时器事件后的EventLoop时序图如下：

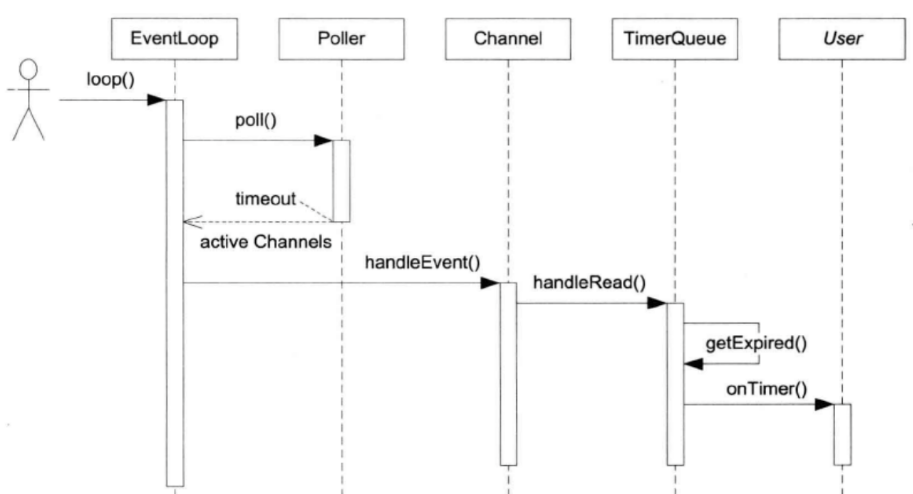


图 3-2 计时器事件时序图

timerfd触发后主要就干了两件事，首先，read读timerfd事件，避免持续触发通知。实现如下：

void TimerQueue::readTimerFd() {

uint64\_t buf;

ssize\_t ret = ::read(timerFd\_, &buf, sizeof buf);

}

获取超时事件并执行部分代码如下：

void TimerQueue::handleRead() {

readTimerFd();

auto expiredTimers = getExpired();

for (auto it = expiredTimers.begin(); it != expiredTimers.end(); ++it) {

it->second->run();

delete it->second;

}

}

获取的超时计时器事件放在vector中，这里每个vector元素都是时间戳-计时器事件对。getExpired函数实现如下：

std::vector<std::pair<timespec, Timer\*> > TimerQueue::getExpired() {

//获取当前的绝对时间

timespec now;

clock\_gettime(CLOCK\_REALTIME, &now);

//把set中时间戳小于当前事件的事件都找出来

std::vector<std::pair<timespec, Timer\*> > ret;

auto it = timers\_.begin();

while (it != timers\_.end() && it->first < now) {

ret.push\_back(\*it);

++it;

}

//删除掉超时事件

auto start = timers\_.begin();

auto end = timers\_.end();

timers\_.erase(start, end);

//根据set中具有最小时间戳的事件时间设置timerfd新的超时时间

if (it != timers\_.end()) {

timespec tim;

tim.tv\_sec = it->first.tv\_sec - now.tv\_sec;

tim.tv\_nsec = it->first.tv\_nsec - now.tv\_nsec;

resetTime(tim);

}

return ret;

}

## 3.3 网络部分的相关设计

这部分的代码都在Webserver/Net目录下。

### 3.3.1 基本封装Socket类和InetAddress类

详细实现见Socket.cc和InetAddress.cc。Socket类构造时创建一个socket，析构时关闭，并实现对bind，listen，accept等方法的简单封装。InetAddress则是对IPv4地址及端口号的简单封装。

### 3.3.2 接受新连接Acceptor类

详细实现见Acceptor.cc和Acceptor.h。考虑网络编程中accpet做的事，阻塞监听一个套接字，有响应时则做出响应动作，也可以封装为一个Channel并放到EventLoop中去监听。

Acceptor类包含一个server Socket（listen Socket）和对应该Socket的channel，channel观察该Socket的readable事件，并在handleEvent中回调Acceptor的handleRead，该handleRead会调用Socket的accept来创建新的TCP连接，并将新创建的socket传回给Acceptor的创建者。这部分代码如下：

void Acceptor::handleRead() {

InetAddress peerAddr(0);

、 //peerAddr保存了对端地址信息，也用于回传

int connfd = acceptSocket\_.accept(&peerAddr);

if (connfd > 0) {

newConnectionCallBack\_(connfd, peerAddr);

} else {

::close(connfd);

}

}

Acceptor的Socket会提前调用Socket的listen，并把对应channel加到EventLoop中监听，实现如下：

void Acceptor::listen() {

listenning\_ = true;

acceptSocket\_.listen();

acceptChannel\_.enableReading();

loop\_->addChannel(&acceptChannel\_);

}

### 3.3.3 Acceptor创建者：TcpServer类

具体实现见TcpServer.cc和TcpServer.h。TcpServer类是网络库使用者直接接触到的一个类，它根据用户提供的服务器地址信息创建了Acceptor类，并根据Acceptor回传的socket和peerAddress来控制一个Tcp连接实际处理的事。其实这里TcpServer会根据socket和peerAdderess创建一个TcpConnection类来处理此次连接，TcpServer通过set(key为TcpServer为该连接创建的名字)对多个TcpConnection进行管理。TcpConnection类会在后续介绍。至此，网络部分的设计已具雏形。

TcpServer包括的部分属性如下：

class TcpServer {

private:

void newConnection(int sockfd, const InetAddress& peerAddr);

typedef std::map<std::string, TcpConnectionPtr> ConnectionMap;

EventLoop\* loop\_;

std::shared\_ptr<Acceptor> acceptor\_;

bool started\_;

int nextConnId\_;

ConnectionMap connections\_;

｝

回到上一部分，accept成功后回调函数newConnectionCallBack\_(connfd, peerAddr);该函数由TcpServer设置，具体定义如下：

void TcpServer::newConnection(int sockfd, const InetAddress& peerAddr) {

loop\_->assertInLoopThread();

char buf[32];

snprintf(buf, sizeof (buf), "#%d", nextConnId\_);

++nextConnId\_;

std::string connName = name\_ + buf;

InetAddress localAddr(Socket::getSockName(sockfd));

//创建TcpConnection类

TcpConnectionPtr conn(

new TcpConnection(ioLoop, connName, sockfd, localAddr, peerAddr));

//通过map对TcpConnection管理

connections\_[connName] = conn;

//设置TcpConnection相关信息，下节会有叙述

conn->setConnectionCallBack(connectionCallBack\_);

conn->setMessageCallBack(messageCallBack\_);

conn->setCloseCallBack(std::bind(&TcpServer::removeConnection,this, std::placeholders::\_1));

//新建连接时回调函数，有TcpServer的创建者（用户）指定

conn->connectEstablish();

}

TcpServe的处理逻辑可用下图表述：

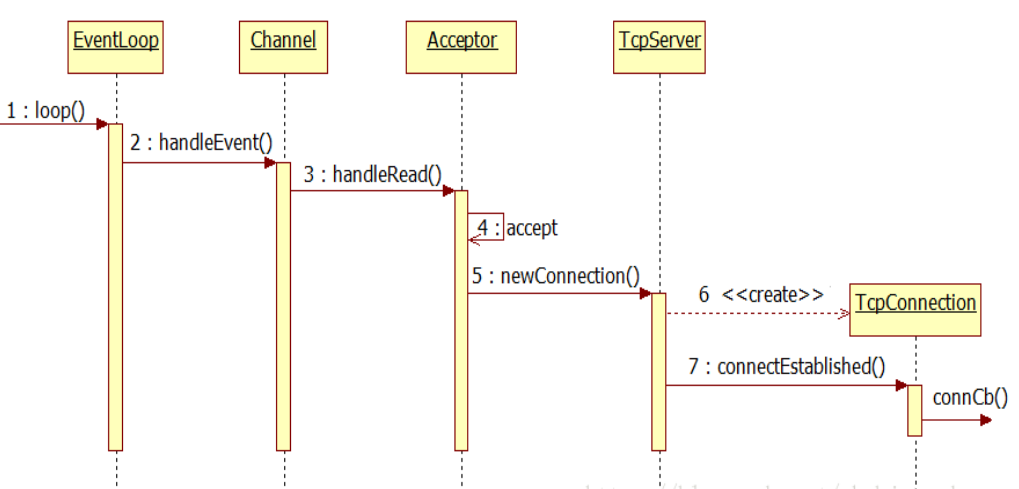


图 3-3 TcpServer函数调用顺序

### 3.3.4 TcpConnection类

TcpConnection类表示实际的一次TCP连接。TcpConnection是个内部类，也是网络库中最复杂的一个类。首先我们分析上文3.3.3中根据connectionName，socketfd，本地地址InetAddress，对端地址InetAddress创建的TcpConnection类，这是是我们已知的信息要存在TcpConnection中的信息。本次连接的消息处理方式要怎么处理？应该由用户指定，即TcpServer的创建者指定，所以TcpServer在创建TcpConnection时也设置了其消息处理函数，即回调用户指定的消息处理函数。连接建立以及释放的回调函数处理过程亦是如此。

TcpConnection内设置的消息处理函数何时调用呢？Socket触发时。这样，我们又可以把该Socket和调用的消息处理函数封装成一个Channel放在TcpConnection中。

综上，TcpConnection的部分属性如下：

class TcpConnection : public std::enable\_shared\_from\_this<TcpConnection>{

private:

enum StateE {kConnecting, kConnected, kDisConnecting, kDisConnected};

StateE state\_;

void setState(StateE s) {

state\_ = s;

}

void handleRead();

EventLoop\* loop\_;

std::string name\_;

std::shared\_ptr<Socket> socket\_;

std::shared\_ptr<Channel> channel\_;

InetAddress localAddr\_;

InetAddress peerAddr\_;

typedef std::function<void (const TcpConnectionPtr& conn)> ConnectionCallBack;

typedef std::function<void (const TcpConnectionPtr& conn,

const char\* data, ssize\_t len)> MessageCallBack;

ConnectionCallBack connectionCallBack\_;

MessageCallBack messageCallBack\_;

｝

上述代码中的handleRead即是channel中的事件回调函数，定义如下：

void TcpConnection::handleRead() {

char buf[65536];

ssize\_t n = ::read(channel\_->fd(), buf, sizeof(buf));

messageCallBack\_(shared\_from\_this(), buf, n);

}

读取消息并传递给用户定义的消息处理函数进行处理。

TcpConnection断开连接如何处理？此时sockedfd会变得可读，但只能读取0个字节。这里要处理两件事，一是把该TcpConnetion对应的socketfd关掉，并把socketfd对应的channel从EventLoop中移除，二是删除TcpServer中存储的该TcpConnection对象。我们做出如下考虑，Tcp断开连接时，TcpConnetion中的channel->handleEvent()调用了TcpServer中removeConnection函数，如果此时TcpConnection对象直接被彻底删除了，而上述channel正属于该对象，程序就会core dump。为此，我们要设法让TcpConnection在channel->handleEvent()返回后才被销毁。程序运行逻辑如下：

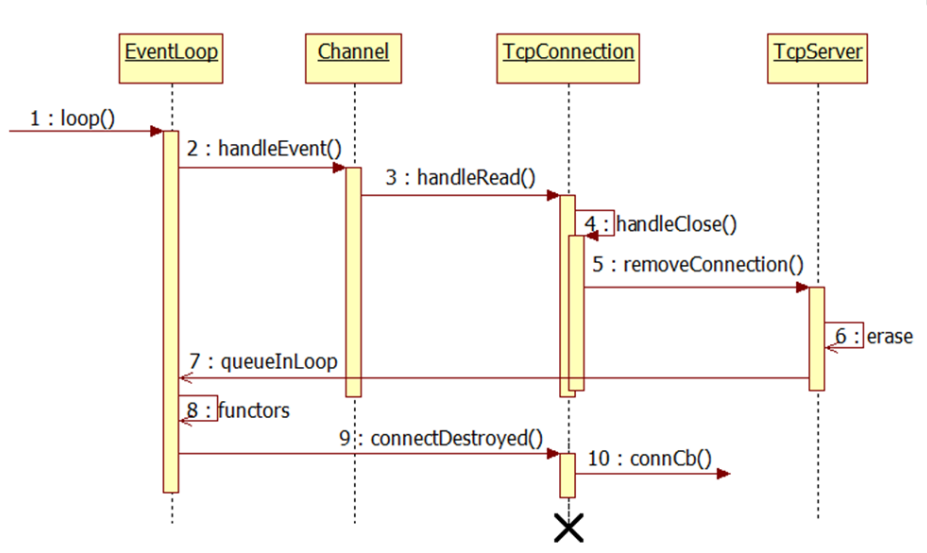


图 3-4 TcpConnection断开连接

这里的queueInLoop(function cb)暂且可以理解为调用者立即返回，等一段时间再去执行cb函数，后续多线程部分会细述其作用及实现。回到上图所示逻辑，removeConnection把TcpConnection的某个函数绑定到cb上，然后调用runInLoop立即返回，这样，channel->handleEvent()也得以执行完毕。此时，由于TcpConnection的某个函数绑定到了cb上，要等过段时间cb执行完TcpConnection才会被析构。

下面考虑代码实现，TcpConnection中对于连接关闭的处理如下，修改handleRead代码如下：

void TcpConnection::handleRead() {

char buf[65536];

ssize\_t n = ::read(channel\_->fd(), buf, sizeof(buf));

if (n > 0) {

messageCallBack\_(shared\_from\_this(), buf, n);

} else if (n == 0) {

//socketfd可读但读不到数据表示连接关闭

handleClose();

} else {

handleError();

}

}

handleClose部分代码如下：

void TcpConnection::handleClose() {

loop\_->assertInLoopThread();

assert(state\_ == kConnected || state\_ == kDisConnecting);

setState(kDisConnecting);

channel\_->disableAll();

loop\_->modChannel(channel\_.get());

closeCallBack\_(shared\_from\_this());

}

CloseBack回调函数被TcpServer设置为removeConnection:

void TcpServer::removeConnection(const TcpConnectionPtr &conn) {

ssize\_t n = connections\_.erase(conn->name());

assert(n == 1);

loop\_->queueInLoop(std::bind(&TcpConnection::connectDestroyed, conn));

}

这里绑定的connectionDestroyed函数用来实现上文所说的关闭socketfd、注销channel等，实现如下：

void TcpConnection::connectDestroyed() {

loop\_->assertInLoopThread();

assert(state\_ == kConnected || state\_ == kDisConnecting);

setState(kDisConnected);

channel\_->disableAll();

loop\_->modChannel(channel\_.get());

connectionCallBack\_(shared\_from\_this());

loop\_->delChannel(channel\_.get());

::close(channel\_->fd());

}

### 3.3.5 数据缓冲Buffer类

Buffer类的基本结构如下：

+--------------------------+--------------------------+-------------------------+

| prependable bytes | readable bytes | writable bytes |

| | (CONTENT) | |

+-----------------------+--------------------------+--------------------------+

| | | |

0 <= readerIndex <= writerIndex <= size

开始时 writerIndex和readerIndex处于同一个位置，往缓冲区写数据时writeindex会向后移动。Readindex和writeindex之间表示还未读取的数据。

该Buffer的设计有两个特色。一是写缓冲不够大时会检测能否通过将CONTENT部分向前移动来扩大writeable bytes。实现如下：

void makeSpace(size\_t len) {

if (writableBytes() + prependableBytes() < len + kCheapPrepend) {

buffer\_.resize(writeIndex\_+len);

}

else {

// move readable data to the front, make space inside buffer

size\_t readable = readableBytes();

std::copy(begin()+readIndex\_,

begin()+writeIndex\_,

begin()+kCheapPrepend);

readIndex\_ = kCheapPrepend;

writeIndex\_ = readIndex\_ + readable;

}

}

二是缓冲区前面预留了较小的prependable bytes，防止出现需要在缓冲区前面插入数据的情况，避免此时需要移动全部数据。

### 3.3.6 TcpConnection发送读取数据

读取数据上文已经介绍，只需要把之前的char\*和len改成buffer类就可以。发送数据流程如下，发送时将数据暂存到缓存，打开socketfd对写的关注，socketfd可写触发，将缓存中的数据发送出去，关闭socketfd对写的关注。

这里该socketfd会一直可写，如果不及时关闭EPoller对socketfd可写的关注，就会造成busy loop。

TcpConnection处理写事件的实现如下：

void TcpConnection::handleWrite() {

loop\_->assertInLoopThread();

if (channel\_->isWriting()) {

ssize\_t n = ::write(channel\_->fd(),outputBuffer\_.peek(),outputBuffer\_

.readableBytes());

if (n > 0) {

outputBuffer\_.retrieve(n);

channel\_->disableWriting();

if (outputBuffer\_.readableBytes() == 0) {

channel\_->disableWriting();

loop\_->modChannel(channel\_.get());

}

}

}

}

## 3.4 one thread per loop+多线程相关实现

### 3.4.1 one thread per loop

One thread per loop即一个线程运行了一个EventLoop，主线程和其他线程事件处理模型一致，都是通过I/O复用监听事件并同步执行事件处理函数，只不过主线程关注的是一开始listen的socket，处理过程是把新建的socketfd交给工作线程去处理。工作线程则是关注多个实际的用于连接的socketfd，处理过程则是通过该socketfd与客户端（对端，peerAddress）通信。我们将这个模型抽象为一个类，EventLoopThread，用于管理单个线程和对应于该线程的EventLoop。

每个线程的工作则是loop，代码如下：

void EventLoopThread::threadFunc() {

EventLoop loop;

if (callback\_) {

callback\_(&loop);

}

{

MutexLockGuard lock(mutex\_);

loop\_ = &loop;

cond\_.notify();

}

loop.loop();

//assert(exiting\_);

MutexLockGuard lock(mutex\_);

loop\_ = NULL;

}

### 3.4.2 线程池EventLoopThreadPool实现

由于之前已经对线程及其要做的工作有了较好地封装，此处线程池的实现也较为容易。此处通过vector对EventLoopThread进行管理。创建部分代码如下：

void EventLoopThreadPool::start(const ThreadInitCallback& cb) {

assert(!started\_);

baseLoop\_->assertInLoopThread();

started\_ = true;

for (int i = 0; i < numThreads\_; ++i) {

char buf[name\_.size() + 32];

snprintf(buf, sizeof buf, "%s%d", name\_.c\_str(), i);

EventLoopThread\* t = new EventLoopThread(cb, buf);

threads\_.push\_back(std::unique\_ptr<EventLoopThread>(t));

loops\_.push\_back(t->startLoop());

｝

}

获取一个线程：

EventLoop\* EventLoopThreadPool::getNextLoop()

{

baseLoop\_->assertInLoopThread();

assert(started\_);

EventLoop\* loop = baseLoop\_;

if (!loops\_.empty()) {

// round-robin

loop = loops\_[next\_];

++next\_;

if (static\_cast<size\_t>(next\_) >= loops\_.size()) {

next\_ = 0;

}

}

return loop;

}

### 3.4.3 线程间任务调度runInLoop实现

如何在本线程中来指定其他线程执行某个任务，比如，主线程新建了TcpConnection后，如何让线程池中的一个线程去执行TcpConnection相关的任务。我们获取到线程池中的某个个线程EventLoop，将任务添加到该线程的执行队列中，唤醒该线程让他去执行任务队列中的任务。

如何唤醒这个线程，由于该线程平时都在poll出轮询，故可以让该线程额外监听一个wakeup事件，需要唤醒该线程时，写一下wakeup就可以了。此时，监听该wakeup的线程会从poll处返回，然后该wakeup fd的handle被设置为处理任务队列中的任务。

实现代码如下：

void EventLoop::wakeup() {

uint64\_t one = 1;

ssize\_t n = write(wakeupFd\_, &one, sizeof one);

}

void EventLoop::handleRead()

{

uint64\_t one = 1;

ssize\_t n = ::read(wakeupFd\_, &one, sizeof one);

}

runInLoop实现如下：

void EventLoop::runInLoop(Functor cb) {

//没有跨线程

if (isInLoopThread()) {

cb();

} else {

queueInLoop(std::move(cb));

}

}

void EventLoop::queueInLoop(Functor cb) {

{

MutexLockGuard lock(mutex\_);

//添加到任务队列

pendingFunctors\_.push\_back(std::move(cb));

}

if (!isInLoopThread() || callingPendingFunctors\_) {

wakeup();

}

}

## 3.5 双缓冲异步日志系统实现

### 3.5.1 设计方案

以下为日志系统设计的思考过程。

为什么是异步，同步不行吗？假设线程1线程2同时输出日志，因为是同步输出，线程1在输出时线程2则只能等待，速度太慢。

为什么使用缓冲？解决CPU和磁盘速度不匹配的情况，避免线程在磁盘写入时占用CPU资源。线程成功写入缓冲即可成功返回，由后台线程负责将缓冲数据写入磁盘，主线程可继续去处理其他任务。

为什么使用双缓冲？尽量避免主线程写入缓冲和后台线程将缓冲写入磁盘产生的冲突。

日志输出风格采用C++形式的输出风格，即LOG << ....

这部分的具体代码实现在WebServer/Base/log目录下。