模拟单线程情况下muduo库的工作情况

在上篇中，笔者追踪了Connetfd（连接套接字）和Listenfd（监听套接字）的Channel对象加入到loop循环的过程。其中包括了网络连接过程中，muduo会创建的对象。本文将会追踪Connetfd（连接套接字）和Listenfd（监听套接字）从loop循环退出并且销毁，一直到main函数终止的过程。

连接套接字正常情况下完整的销毁情况（read == 0）

由TcpConnection对象向自己所拥有的Channel对象注册的可读事件结束时，会出现`read == 0`的情况，此时会直接调用TcpConnection对象的handleClose函数。因为在向Channel对象注册可读事件时，使用了如下的语句：

```
channel_->setReadCallback(&TcpConnection::handleRead,this);
```

this使得Channel对象可以直接在TcpConnection向它注册的handleClose函数内部使用TcpConnetion的函数。

```
void TcpConnection::handleRead(Timestamp receiveTime)
{//都是向channel注册的函数
  loop_->assertInLoopThread();//断言在loop线程
  int savedErrno = 0;//在读取数据之后调用用户提供的回调函数
  ssize_t n = inputBuffer_.readFd(channel_->fd(), &savedErrno);
  if (n > 0)
  {//这个应该时用户提供的处理信息的回调函数
    messageCallback_(shared_from_this(), &inputBuffer_, receiveTime);
  }
  else if (n == 0)
  {//读到了0,直接关闭
    handleClose();
  }
  else
  {//如果有错误
    errno = savedErrno;
    LOG_SYSERR << "TcpConnection::handleRead";
    handleError();//处理关闭
  }
}
void TcpConnection::handleClose()
{//处理关闭事件
  loop_->assertInLoopThread();//断言是否在loop线程
  LOG_TRACE << "fd = " << channel_->fd() << " state = " << stateToString();
  assert(state_ == kConnected || state_ == kDisconnecting);
  // we don't close fd, leave it to dtor, so we can find leaks easily.
  setState(kDisconnected);//设置关闭状态
  channel_->disableAll();//不再关注任何事情

  TcpConnectionPtr guardThis(shared_from_this());//获得shared_ptr交由tcpsever处理
  connectionCallback_(guardThis);//这他妈就是记录一点日志
  // must be the last line
  closeCallback_(guardThis);
}
```

在以上的handleClose代码中，首先会设置TcpConnection对象的关闭状态，其次让自己Channel对象不再关注任何事情。
因为TcpConnection在创建时使用了如下语句：

```
class TcpConnection : boost::noncopyable, public boost::enable_shared_from_this<TcpConnection>
```

便可以使用shared_fron_this()获得指向本TcpConnection对象的shared_ptr指针，然后在后续的过程中，对指向本对象的shared_ptr进行操作，则可以安全的将本对象从其他依赖类中安全的移除。
继续跟踪上述的最后一句，且closeCallback是由TcpServer在创建TcpConnection对象时向它注册的：

```
conn->setCloseCallback(boost::bind(&TcpServer::removeConnection, this, _1));
```

目的在于在TcpServer的TcpconnectionMap中移除指向指向这个TcpConnection对象的指针。

```
void TcpServer::removeConnection(const TcpConnectionPtr& conn)
{
  // FIXME: unsafe
  loop_->runInLoop(boost::bind(&TcpServer::removeConnectionInLoop, this, conn));//注册到loop线程中移除这个con
}

void TcpServer::removeConnectionInLoop(const TcpConnectionPtr& conn)
{
  loop_->assertInLoopThread();//断言是否在IO线程
  LOG_INFO << "TcpServer::removeConnectionInLoop [" << name_
           << "] - connection " << conn->name();
  size_t n = connections_.erase(conn->name());//删除该con
  (void)n;
  assert(n == 1);
  EventLoop* ioLoop = conn->getLoop();//获得线程Loop
  ioLoop->queueInLoop(
      boost::bind(&TcpConnection::connectDestroyed, conn));//将线程销毁动作添加到loop中去
}
```

目前的步骤还在于处理TcpConnection对象。

```
void TcpConnection::connectDestroyed()
{//销毁连接
  loop_->assertInLoopThread();//断言是否在loop线程
  if (state_ == kConnected)//如果此时处于连接状态
  {
    setState(kDisconnected);//将状态设置为不可连接状态
    channel_->disableAll();//channel不再关注任何事件

    connectionCallback_(shared_from_this());//记录作用，好坑的一个作用
  }
  channel_->remove();//在poller中移除channel
}
```

TcpConnection对象的声明周期随着将Channel对象移除出loop循环而结束。

```
void Channel::remove()
{//将channel从loop中移除
  assert(isNoneEvent());//判断此时的channel是否没有事件发生
  addedToLoop_ = false;//此时没有loop拥有此channel
  loop_->removeChannel(this);//调用POLLER的删除功能
}
```

因为EventLoop对象中的poller对象也持有Channel对象的指针，所以需要将channel对象安全的从poller对象中移除。

```
void EventLoop::removeChannel(Channel* channel)
{//每次间接的调用的作用就是将需要改动的东西与当前调用的类撇清关系
  assert(channel->ownerLoop() == this);
  assertInLoopThread();//如果没有在loop线程调用直接退出
  if (eventHandling_)//判断是否在事件处理状态。判断当前是否在处理这个将要删除的事件以及活动的事件表中是否有这个事件
  {
    assert(currentActiveChannel_ == channel ||
        std::find(activeChannels_.begin(), activeChannels_.end(), channel) == activeChannels_.end());
  }
  poller_->removeChannel(channel);//在POLLER中删除这个事件分发表
}
```

以下时poller对象移除Channel对象的具体操作步骤。

```
void PollPoller::removeChannel(Channel* channel)
{
  Poller::assertInLoopThread();//判断是否在IO线程
  LOG_TRACE << "fd = " << channel->fd();
  assert(channels_.find(channel->fd()) != channels_.end());
  assert(channels_[channel->fd()] == channel);
  assert(channel->isNoneEvent());
  int idx = channel->index();//获得pfd位置的索引
  assert(0 <= idx && idx < static_cast<int>(pollfds_.size()));
  const struct pollfd& pfd = pollfds_[idx]; (void)pfd;//获得pfd
  assert(pfd.fd == -channel->fd()-1 && pfd.events == channel->events());
  size_t n = channels_.erase(channel->fd());//在Map中删除channel
  assert(n == 1); (void)n;//准备删除pollfd中的关注事件
  if (implicit_cast<size_t>(idx) == pollfds_.size()-1)//获得pollfd的索引
  {
    pollfds_.pop_back();
  }
  else//想方设法的删除pollfd中与channel相关的pfd
  {
    int channelAtEnd = pollfds_.back().fd;
    iter_swap(pollfds_.begin()+idx, pollfds_.end()-1);
    if (channelAtEnd < 0)
    {
      channelAtEnd = -channelAtEnd-1;
    }
    channels_[channelAtEnd]->set_index(idx);
    pollfds_.pop_back();
  }
}
```

以上便是一个连接的销毁过程，现在依然让人迷惑的时Channel对象到底被谁持有过？以及TcpConnection对象的生命周期到底在什么时候结束？

Channel与TcpConnection对象的创建与销毁

创建

下面，在让我们进入上一篇文章，具体的看看Channel对象的生命期到底是个什么样子？

- 当新连接过来时，由TcpServer创建一个TcpConnection对象，这个对象中包括一个与此连接相关的Channel对象。
- 然后紧接着TcpServer使用创建的TcpConnection对象向Loop中注册事件。此时的控制权回到TcpConnection对象手中。它操作自己的Channel对象更新EventLoop。
- 最后由EventLoop对象去操作自己的Poller更新Poller的Channel队列。
  在上述过程中，Channel对象的创建操作有这样的顺序：

```
TcpServer->TcpConnection->Channel->EventLoop->Poller
```

TcpConnection对象的创建过程相比于Channel简单的多：

```
TcpServer->TcpConnection
```

在TcpServer中创建Connection对象，然后让TcpConnection对象去操作自己的Channel对象，将Channel加入到EventLoop中去，最后由EventLoop操作自己的Poller收尾。总而言之，Channel对象在整个过程中只由Poller和TcpConnection对象持有，销毁时也应该是如此过程。

销毁

由于Channel是TcpConnection对象的一部分，所以Channel的生命周期一定会比TcpConnection的短。
Channel与TcpConnection对象的销毁基本与上述创建过程相同：

```
TcpConnection->TcpServer->Channel->EventLoop->Poller
```

随着，Channel对象从Poller中移除，TcpConnection的生命周期也随之结束。
TcpConnection对象在整个生命周期中只由TcpServer持有，但是TcpConnetion对象中的Channel又由Poller持有，Poller又是EventLoop的唯一成员，所以造成了如此麻烦的清理与创建过程。那如果能将Channel移出TcpConnection对象，那muduo的创建与清理工作会不会轻松很多？