模拟单线程情况下muduo库的工作情况

muduo的源代码对于一个初学者来说还是有一些复杂的，其中有很多的回调函数以及交叉的组件，下面我将追踪一次TCP连接过程中发生的事情，不会出现用户态的源码，都是库内部的运行机制。下文笔者将描述一次连接发生的过程，将Channel到加入到loop循环为止。

监听套接字加入loop循环的完整过程

- 首先创建一个TcpServer对象，在的创建过程中，首先new出来自己的核心组件（Acceptor,loop,connectionMap,threadPool）之后TcpServer会向Acceptor注册一个新连接到来时的Connection回调函数。loop是由用户提供的，并且在最后向Acceptor注册一个回调对象，用于处理：一个新的Client连接到来时该怎么处理。
  TcpServer向Acceptor注册的回调代码主要作用是：当一个新的连接到来时，根据Acceptor创建的可连接描述符和客户的地址，创建一个Connection对象，并且将这个对象加入到TcpServer的ConnectionMap中，由TcpServer来管理上述新建con对象。但是现在监听套接字的事件分发对象Channel还没有加入loop，就先不多提这个新的连接到到来时的处理过程。

```
TcpServer::TcpServer(EventLoop* loop,const InetAddress& listenAddr,const string& nameArg,Option option)
    : loop_(CHECK_NOTNULL(loop)),
      ipPort_(listenAddr.toIpPort()),name_(nameArg),acceptor_(new Acceptor(loop, listenAddr, option == kReusePort)),
    threadPool_(new EventLoopThreadPool(loop, name_)),
    connectionCallback_(defaultConnectionCallback),
    messageCallback_(defaultMessageCallback),
    nextConnId_(1)
{//上面的loop是用户提供的loop
  acceptor_->setNewConnectionCallback(
      boost::bind(&TcpServer::newConnection, this, _1, _2));//注册给acceptor的回调
}//将在Acceptor接受新连接的时候

void TcpServer::newConnection(int sockfd, const InetAddress& peerAddr)
{//将本函数注册个acceptor
  loop_->assertInLoopThread();//断言是否在IO线程
  EventLoop* ioLoop = threadPool_->getNextLoop();//获得线程池中的一个loop
  char buf[64];//获得线程池map中的string索引
  snprintf(buf, sizeof buf, "-%s#%d", ipPort_.c_str(), nextConnId_);
  ++nextConnId_;
  string connName = name_ + buf;

  LOG_INFO << "TcpServer::newConnection [" << name_
           << "] - new connection [" << connName
           << "] from " << peerAddr.toIpPort();
  InetAddress localAddr(sockets::getLocalAddr(sockfd));//获得本地的地址，用于构建Connection
  // FIXME poll with zero timeout to double confirm the new connection
  // FIXME use make_shared if necessary
  TcpConnectionPtr conn(new TcpConnection(ioLoop,
                                          connName,
                                          sockfd,
                                          localAddr,
                                          peerAddr));//构建了一个connection
  connections_[connName] = conn;//将新构建的con加入server的map中
  conn->setConnectionCallback(connectionCallback_);//muduo默认的
  conn->setMessageCallback(messageCallback_);//moduo默认的
  conn->setWriteCompleteCallback(writeCompleteCallback_);//? ?
  conn->setCloseCallback(
      boost::bind(&TcpServer::removeConnection, this, _1)); // FIXME: unsafe
  ioLoop->runInLoop(boost::bind(&TcpConnection::connectEstablished, conn));//在某个线程池的loop中加入这个con
}
```

- 下面接着讲述在TcpServer的构造过程中发生的事情：创建Acceptor对象。TcpServer用unique_ptr持有唯一的指向Acceptor的指针。Acceptor的构造函数完成了一些常见的选项。最后的一个向Acceptor->Channel注册一个回调函数，用于处理：listening可读时（新的连接到来），该怎么办？答案是：当新的连接到来时，创建一个已连接描述符，然后调用TcpServe注册给Acceptor的回调函数，用于处理新的连接。

```cpp
Acceptor::Acceptor(EventLoop* loop, const InetAddress& listenAddr, bool reuseport)
  : loop_(loop),
    acceptSocket_(sockets::createNonblockingOrDie(listenAddr.family())),
    acceptChannel_(loop, acceptSocket_.fd()),
    listenning_(false),
    idleFd_(::open("/dev/null", O_RDONLY | O_CLOEXEC))
{
  assert(idleFd_ >= 0);
  acceptSocket_.setReuseAddr(true);
  acceptSocket_.setReusePort(reuseport);
  acceptSocket_.bindAddress(listenAddr);
  acceptChannel_.setReadCallback(
      boost::bind(&Acceptor::handleRead, this));//Channel设置回调，当sockfd可读时掉用设置的回调
}

void Acceptor::handleRead()
{
  loop_->assertInLoopThread();//判断是否在IO线程
  InetAddress peerAddr;//客户的地址
  //FIXME loop until no more
  int connfd = acceptSocket_.accept(&peerAddr);//获得连接的描述符
  if (connfd >= 0)
  {
    // string hostport = peerAddr.toIpPort();
    // LOG_TRACE << "Accepts of " << hostport;
    if (newConnectionCallback_)
    {
      newConnectionCallback_(connfd, peerAddr);//TcpServer注册的，创建新的con,并且加入TcpServer的ConnectionMap中。
    }
    else
    {
      sockets::close(connfd);
    }
  }
  else
  {
    LOG_SYSERR << "in Acceptor::handleRead";
    // Read the section named "The special problem of
    // accept()ing when you can't" in libev's doc.
    // By Marc Lehmann, author of libev.
    if (errno == EMFILE)
    {
      ::close(idleFd_);
      idleFd_ = ::accept(acceptSocket_.fd(), NULL, NULL);
      ::close(idleFd_);
      idleFd_ = ::open("/dev/null", O_RDONLY | O_CLOEXEC);
    }
  }
}
```

- 在上述Acceptor对象的创建过程中，Acceptor会创建一个用于处理监听套接字事件的Channel对象，以下Acceptor的Channel对象的创造过程，很常规的处理过程。

```cpp
Channel::Channel(EventLoop* loop, int fd__)
  : loop_(loop),
    fd_(fd__),
    events_(0),
    revents_(0),
    index_(-1),
    logHup_(true),
    tied_(false),
    eventHandling_(false),
    addedToLoop_(false)
{
}
```

- 到此，在muduo库内部的初始化过程已经基本处理完毕，然后由用户调用TcpServer的setThreadNum()和start()函数。在start()函数中会将打开Acceptor对象linten套接字。

```
void TcpServer::setThreadNum(int numThreads)
{//设置线程池的开始数目
  assert(0 <= numThreads);
  threadPool_->setThreadNum(numThreads);
}

void TcpServer::start()
{//TcpServer开始工作
  if (started_.getAndSet(1) == 0)//获得原子计数
  {
    threadPool_->start(threadInitCallback_);//线程池开始工作

    assert(!acceptor_->listenning());//打开accepor的监听状态
    loop_->runInLoop(
        boost::bind(&Acceptor::listen, get_pointer(acceptor_)));//打开acceptor的listening
  }
}
```

- 打开Acceptor对象的listenfd的详细过程。

```
void Acceptor::listen()
{
  loop_->assertInLoopThread();//判断是否在IO线程
  listenning_ = true;//进入监听模式
  acceptSocket_.listen();
  acceptChannel_.enableReading();//让监听字的channel关注可读事件
}
```

- 接着使用了Channel对象中的的enableReading()函数，让这个Channel对象关注可读事件。关键在于更新过程，应该是这个流程中最重要的操作。

```
void enableReading() { events_ |= kReadEvent; update(); }//将关注的事件变为可读，然后更新
```

- 使用了Channel的更新函数：update()

```
void Channel::update()
{
  addedToLoop_ = true;//更新channel的状态
  loop_->updateChannel(this);//调用POLLER的更新功能
}
```

- EventLoop持有唯一的Poller，也就是说，这个Poller将负责最后的更新过程。如果是新的Channel对象，则在Poller的pollfd数组中增加席位；如果不是新的Channel对象，则更新它目前所发生的事件（将目前发生的事件设置为0）。

```
void EventLoop::updateChannel(Channel* channel)
{
  assert(channel->ownerLoop() == this);//判断channel的LOOP是否是当前的LOOP
  assertInLoopThread();//判断是否在IO线程
  poller_->updateChannel(channel);//使用POLLER来更新channel
}
```

- 紧接着使用了Poller的updateChannel函数

```cpp
void PollPoller::updateChannel(Channel* channel)
{//将channel关注的事件与pollfd同步
  Poller::assertInLoopThread();//如果不再loop线程直接退出
  LOG_TRACE << "fd = " << channel->fd() << " events = " << channel->events();
  if (channel->index() < 0)//获得channel在map中的位置
  {
    // a new one, add to pollfds_
    assert(channels_.find(channel->fd()) == channels_.end());
    struct pollfd pfd;//新建一个pfd与channel相关联
    pfd.fd = channel->fd();
    pfd.events = static_cast<short>(channel->events());//关注的事件设置为channel关注的事件
    pfd.revents = 0;//正在发生的事件为0
    pollfds_.push_back(pfd);//将设置好的pollfd加入关注事件列表
    int idx = static_cast<int>(pollfds_.size())-1;//并且获得加入的位置
    channel->set_index(idx);//channel保存自己在pollfds中的位置
    channels_[pfd.fd] = channel;//channel将自己加入到channelmap中
  }
  else
  {
    // update existing one
    assert(channels_.find(channel->fd()) != channels_.end());
    assert(channels_[channel->fd()] == channel);//判断位置是否正确
    int idx = channel->index();//获得channel在pollfd中的索引
    assert(0 <= idx && idx < static_cast<int>(pollfds_.size()));
    struct pollfd& pfd = pollfds_[idx];//获得索引
    assert(pfd.fd == channel->fd() || pfd.fd == -channel->fd()-1);
    pfd.events = static_cast<short>(channel->events());//修改关注的事件
    pfd.revents = 0;//将当前发生的事件设置为0
    if (channel->isNoneEvent())//如果channel没有任何事件，一个暂时熄火的channel
    {
      // ignore this pollfd
      pfd.fd = -channel->fd()-1;//将索引设置为原来索引的负数
    }
  }
}
```

- 至此，调用EventLoop的loop函数，进行loop循环，开始处理事件。

```cpp
void EventLoop::loop()
{
  assert(!looping_);//判断是否在LOOPING
  assertInLoopThread();//判断这个函数在LOOP线程调用
  looping_ = true;//进入LOOPING状态
  quit_ = false;  // FIXME: what if someone calls quit() before loop() ?
  LOG_TRACE << "EventLoop " << this << " start looping";

  while (!quit_)
  {
    activeChannels_.clear();//将活动线程队列置空
    pollReturnTime_ = poller_->poll(kPollTimeMs, &activeChannels_);//获得活动文件描述符的数量，并且获得活动的channel队列
    ++iteration_;//增加Poll次数
    if (Logger::logLevel() <= Logger::TRACE)
    {
      printActiveChannels();
    }
    // TODO sort channel by priority
    eventHandling_ = true;//事件处理状态
    for (ChannelList::iterator it = activeChannels_.begin();
        it != activeChannels_.end(); ++it)
    {
      currentActiveChannel_ = *it;//获得当前活动的事件
      currentActiveChannel_->handleEvent(pollReturnTime_);//处理事件，传递一个poll的阻塞时间
    }
    currentActiveChannel_ = NULL;//将当前活动事件置为空
    eventHandling_ = false;//退出事件处理状态
    doPendingFunctors();//处理用户在其他线程注册给IO线程的事件
  }

  LOG_TRACE << "EventLoop " << this << " stop looping";
  looping_ = false;//推出LOOPING状态
}
```

一个监听套接字已经进入循环，如果此时一个新的连接到来又会发生什么事情呢?

一个新连接到达时的处理过程。

- 此时在loop循环中的监听套接字变得可读，然后便调用一个可读事件的处理对象。首先调用Acceptor注册的handleRead对象，完成连接套接字的创建，其次在handleRead对象的内部调用TcpServer注册给Acceptor的函数对象，用于将新建con对象加入TcpServer的ConnectionMap中去。

```
void Channel::handleEvent(Timestamp receiveTime)
{
  boost::shared_ptr<void> guard;
  if (tied_)
  {
    guard = tie_.lock();//提升成功说明con存在
    if (guard)//这样做比较保险
    {
      handleEventWithGuard(receiveTime);
    }
  }
  else
  {
    handleEventWithGuard(receiveTime);
  }
}

void Channel::handleEventWithGuard(Timestamp receiveTime)
{//真正的处理各种事件
  eventHandling_ = true;//处理事件状态
  LOG_TRACE << reventsToString();
  if ((revents_ & POLLHUP) && !(revents_ & POLLIN))
  {
    if (logHup_)
    {
      LOG_WARN << "fd = " << fd_ << " Channel::handle_event() POLLHUP";
    }
    if (closeCallback_) closeCallback_();
  }

  if (revents_ & POLLNVAL)
  {
    LOG_WARN << "fd = " << fd_ << " Channel::handle_event() POLLNVAL";
  }

  if (revents_ & (POLLERR | POLLNVAL))
  {
    if (errorCallback_) errorCallback_();
  }
  if (revents_ & (POLLIN | POLLPRI | POLLRDHUP))
  {
    if (readCallback_) readCallback_(receiveTime);
  }
  if (revents_ & POLLOUT)
  {
    if (writeCallback_) writeCallback_();
  }
  eventHandling_ = false;
}
```

- 此时，监听套接字处理的时可读事件，调用之前由Acceptor注册的handleRead回调函数

```
void Acceptor::handleRead()
{
  loop_->assertInLoopThread();//判断是否在IO线程
  InetAddress peerAddr;//客户的地址
  //FIXME loop until no more
  int connfd = acceptSocket_.accept(&peerAddr);//获得连接的描述符
  if (connfd >= 0)
  {
    // string hostport = peerAddr.toIpPort();
    // LOG_TRACE << "Accepts of " << hostport;
    if (newConnectionCallback_)
    {
      newConnectionCallback_(connfd, peerAddr);//这是个关键步骤，重点在于这个回调是谁注册的
    }
    else
    {
      sockets::close(connfd);
    }
  }
  else
  {
    LOG_SYSERR << "in Acceptor::handleRead";
    // Read the section named "The special problem of
    // accept()ing when you can't" in libev's doc.
    // By Marc Lehmann, author of libev.
    if (errno == EMFILE)
    {
      ::close(idleFd_);
      idleFd_ = ::accept(acceptSocket_.fd(), NULL, NULL);
      ::close(idleFd_);
      idleFd_ = ::open("/dev/null", O_RDONLY | O_CLOEXEC);
    }
  }
}
```

- 在上述函数中又调用，由TcpServer注册给Acceptor的回调函数

```
void TcpServer::newConnection(int sockfd, const InetAddress& peerAddr)
{//将本函数注册个acceptor
  loop_->assertInLoopThread();//断言是否在IO线程
  EventLoop* ioLoop = threadPool_->getNextLoop();//获得线程池中的一个loop
  char buf[64];//获得线程池map中的string索引
  snprintf(buf, sizeof buf, "-%s#%d", ipPort_.c_str(), nextConnId_);
  ++nextConnId_;
  string connName = name_ + buf;

  LOG_INFO << "TcpServer::newConnection [" << name_
           << "] - new connection [" << connName
           << "] from " << peerAddr.toIpPort();
  InetAddress localAddr(sockets::getLocalAddr(sockfd));//获得本地的地址，用于构建Connection
  // FIXME poll with zero timeout to double confirm the new connection
  // FIXME use make_shared if necessary
  TcpConnectionPtr conn(new TcpConnection(ioLoop,
                                          connName,
                                          sockfd,
                                          localAddr,
                                          peerAddr));//构建了一个connection
connections_[connName] = conn;//将新构建的con加入server的map中
  conn->setConnectionCallback(connectionCallback_);//muduo默认的
  conn->setMessageCallback(messageCallback_);//moduo默认的
  conn->setWriteCompleteCallback(writeCompleteCallback_);//? ?
  conn->setCloseCallback(
      boost::bind(&TcpServer::removeConnection, this, _1)); // FIXME: unsafe
  ioLoop->runInLoop(boost::bind(&TcpConnection::connectEstablished, conn));//在某个线程池的loop中加入这个
con
}
```

- 上述对象的最后一行，是调用新建的TcpConnection对象的函数，用设置新建的con对象中的channel的关注事件。

```
void TcpConnection::connectEstablished()
{//建立连接
  loop_->assertInLoopThread();//断言是否在IO线程
  assert(state_ == kConnecting);//正处于连接建立过程
  setState(kConnected);
  channel_->tie(shared_from_this());//使channel的tie的指向不为空
  channel_->enableReading();//将connection设置为可读的

  connectionCallback_(shared_from_this());//用户提供的回调函数，muduo有提供默认的
}
```

- 至此以后的过程与将listen->channel添加到loop中的过程一样。

```
void enableReading() { events_ |= kReadEvent; update(); }//将关注的事件变为可读，然后更新
```

- 使用了Channel的更新函数：update()

```
void Channel::update()
{
  addedToLoop_ = true;//更新channel的状态
  loop_->updateChannel(this);//调用POLLER的更新功能
}
```

- 使用了EventLoop的updateChannel()功能

```
void EventLoop::updateChannel(Channel* channel)
{
  assert(channel->ownerLoop() == this);//判断channel的LOOP是否是当前的LOOP
  assertInLoopThread();//判断是否在IO线程
  poller_->updateChannel(channel);//使用POLLER来更新channel
}
```

- 在poller中更新channel

```
void PollPoller::updateChannel(Channel* channel)
{//将channel关注的事件与pollfd同步
  Poller::assertInLoopThread();//如果不再loop线程直接退出
  LOG_TRACE << "fd = " << channel->fd() << " events = " << channel->events();
  if (channel->index() < 0)//获得channel在map中的位置
  {
    // a new one, add to pollfds_
    assert(channels_.find(channel->fd()) == channels_.end());
    struct pollfd pfd;//新建一个pfd与channel相关联
    pfd.fd = channel->fd();
    pfd.events = static_cast<short>(channel->events());//关注的事件设置为channel关注的事件
    pfd.revents = 0;//正在发生的事件为0
    pollfds_.push_back(pfd);//将设置好的pollfd加入关注事件列表
    int idx = static_cast<int>(pollfds_.size())-1;//并且获得加入的位置
    channel->set_index(idx);//channel保存自己在pollfds中的位置
    channels_[pfd.fd] = channel;//channel将自己加入到channelmap中
  }
  else
  {
    // update existing one
    assert(channels_.find(channel->fd()) != channels_.end());
    assert(channels_[channel->fd()] == channel);//判断位置是否正确
    int idx = channel->index();//获得channel在pollfd中的索引
    assert(0 <= idx && idx < static_cast<int>(pollfds_.size()));
    struct pollfd& pfd = pollfds_[idx];//获得索引
    assert(pfd.fd == channel->fd() || pfd.fd == -channel->fd()-1);
    pfd.events = static_cast<short>(channel->events());//修改关注的事件
    pfd.revents = 0;//将当前发生的事件设置为0
    if (channel->isNoneEvent())//如果channel没有任何事件，一个暂时熄火的channel
    {
      // ignore this pollfd
      pfd.fd = -channel->fd()-1;//将索引设置为原来索引的负数
    }
  }
}
```

最后一个连接的channel加入loop循环，新的循环已经开始了。