多线程构造函数只需要保证构造函数中不要泄露this指针就能保证多线程安全。

对象之间的关系有三种：

composition：对象a是对象b的直接数据成员，a的生命期由b控制；

association：对象a调用了对象b的成员函数，虽然a拥有b的指针/引用，但b的生命期不受a的控制；

aggregation：从形式上来看aggregation和association相同，区别在于聚合关系中a和b有整体和部分的逻辑关系。

association和aggregation关系在多线程中可能会出现安全问题。

多线程析构函数的设计是一个难点，需要保证同一时刻对象只能被一个线程析构，常用shared\_ptr来解决。

#### 多线程安全的观察者模式：

一种观察者模式的写法为:

|  |
| --- |
| class Observer{ public:  Observer(Subject\* subject) {  subject->register(this);  \_subject = s;  }  virtual ~Observer(){  \_subject->unregister();  }  protected:  Subject\* \_subject; }; |
| class Subject{ public:  virtual void register(Observer\* obser);  virtual void unregister(Observer\* obser);  void Notify(){  for(set<Observer\*>::iterator it=observerList.begin(); it!=observerList.end();it++){  (\*\*it).update();  }  } protected:  set<Observer\*> observerList; }; |

在多线程的情况下，这种写法存在问题：1.Oberver::~Oberver()中解注册时如何得知—\_subject还活着。2.Subject::Notify中如何得知update的Observer对象活着。

使用shared\_ptr和weak\_ptr来解决这种多线程中的不安全状况。

* shared\_ptr：强引用，能够控制对象的生命期。只要有指向x对象的shared\_ptr存在，x对象就不会被析构。
* weak\_ptr：弱引用，通过它来判断对象是否或者，如果对象还活着，可以通过weak\_ptr提升为有效的shared\_ptr，否则返回空的shared\_ptr。

用weak\_ptr来保证线程安全。

|  |
| --- |
| class Subject{ public:  virtual void register(weak\_ptr<Observer> obser);  virtual void unregister(weak\_ptr<Observer> obser);  void Notify(){  MutexLockGuard lock(mutex\_);  for(auto it=observerList.begin(); it!=observerList.end();it++){  shared\_ptr<Observer> obj(it->lock());  if(obj){  obj->update();  }  else{  observerList.erase(it);  }  }  } protected:  mutable MutexLock mutex\_;  set<Observer\*> observerList; }; |

用weak\_ptr保存Oberver指针，再调用Oberver对象，如果对象还活着，则可以将weak\_ptr提升为shared\_ptr，否则，提升失败。

用互斥锁+条件变量可以实现信号量的功能，慎用信号量。8

推荐的C++多线程服务器编程模式为：

#### one loop per thread + thread pool

##### one loop per thread

这种模型中，每个IO线程都有一个event loop，用于处理读写和定时事件。event loop 代表了线程的主循环，需要让哪个线程干活，就把timer或IO channel（如TCP连接）注册到哪个线程的loop里即可。

对实时性要求高的connection可以单独用一个线程；数据量大的connection可以单独用一个线程；其他辅助性connections可以共享一个线程。

##### thread pool

线程池用于处理没有IO只有计算任务的线程，线程池中的队列可以是任务队列或生产者消费者队列。

适用多线程的场景：让IO和计算相互重叠，降低延迟，提高平均响应性能。

每个文件描述符只由一个线程操作。

对同一个epoll fd的操作（添加、删除、修改、等待）都放到同一个线程执行。

用socket对象封装文件描述符，对文件描述符的所有读写操作都通过这个对象进行，并且在析构函数中关闭文件描述符，这样一来，只要socket对象还活着，就不会有其他socket对象跟他有一样的文件描述符，也就不可能串话。

在非阻塞网络编程中，假设TCP连接A收到了一个requestA，程序开始处理这个requestA，为了避免阻塞其他request，程序记住发来这个requestA的TCP连接，在线程池中进行处理，处理完成之后，会把response发回给TCP连接A。假如在处理requestA的过程中，客户端关闭了ruquestA，就会出错。

为了避免这种情况，把TCP连接A的文件描述符用TcpConnection对象进行封装，并用shared\_ptr进行管理。这样，客户端就不可能在服务端处理TCP连接A描述符时关闭描述符。

**上面的这种做法也就体现了一种思想：用对象来包装资源，把资源管理和对象生命期统一起来（RAII）。**

多线程编程与fork和signal不太兼容。

**多线程日志：**

保证所有的线程只写一个日志文件，实现时用一个背景线程收集日志信息，并写入日志文件，其他业务线程只管往这个日志线程发送日志消息，称为“异步日志”。

muduo采用“双缓冲”技术实现异步日志，维护两个Buffer:A和B

前端负责往bufferA写日志消息，后端负责将bufferB的日志消息写入文件。

bufferA写满后，交换A和B，前端向bufferB写日志消息，后端将bufferA的消息写入文件。

#### TCP编程的本质

非阻塞网络编程是高性能并发网络编程的主流模式，通过注册回调函数来实现。

处理三个半事件：  
1.连接的建立，包括客户端的connect和服务端的accept，TCP连接一旦建立，客户端和服务端是平等的。

2.连接的断开，包括主动断开（close，shutdown）和被动断开（read返回0）

3.消息到达，文件描述符可读。对其的处理方式决定了网络编程的风格（阻塞还是非阻塞，如何处理分包，应用层的缓冲如何设计等）。

4.消息发送完毕，只代表写入了OS的缓冲区，并不代表对方收到。

**注意的点：**

1.如果主动关闭连接，如何保证对方已经收到全部数据，如果应用层有缓冲，如何保证先发送缓冲区内容，在断开连接？

在非阻塞IO中，应用层的buffer是必须的。因为非阻塞IO的核心思想就是避免阻塞在read()或write()等其他IO系统调用上。

a) TcpConnection需要一个output buffer：对于应用程序而言，只关心生成数据，而不应该关心数据是一次性发送还是几次发送。如程序想发送100kB的数据，但由于Tcp窗口的限制，只发送了80kB，还有20kB的数据没有发送。程序只需要send()，send()之后的数据由网络库负责，所以剩下的20kB由网络库接管，保存在output buffer中，然后注册POLLOUT事件，一旦socket可写就发送数据。

如果程序又发送了50kB，则将这50kB的数据append到这20kB后面，等socket可写时一并发送。

如果断开连接后还有待发的数据，由网络库去做，程序也不用关心。

b) TcpConnection需要一个input buffer：一条完整的消息可能需要接收端接收几次才能完成，需要应用层buffer构成一条完成消息后再交给程序。

2.如果主动发起连接，但对方拒绝，如何定期重试。

3.采用ET还是LT

4.在非阻塞网络编程中，为什么使用应用层发送缓冲？

5.在非阻塞网络编程中，为什么使用应用层接收缓冲？

6.在非阻塞网络编程中，如何设计并使用缓冲？

7.如果使用缓冲区，万一接收方处理缓慢，数据会不会一直堆在发送方，造成内存暴涨，如果做应用层流量控制

8.如何设计定时器，并使之与网络IO共用一个线程，以避免锁

#### 常见的并发服务器编程设计方案

0.accept+read/write

迭代服务器，只有一个线程，循环处理每个连接请求，一次只能服务一个客户。

1.accept+fork()

也称为child-per-client/fork()-per-client，每个新的连接用一个子进程去处理。适用于并发连接数不大的情况。适合长连接，不适合短链接。

业务逻辑初步从网络框架中分离出来，但仍与IO紧密结合。

2.accept+thread()

和1的方式一样，只不过对每个新的连接新建一个线程处理。同样不适合短连接。

3.prefork

方案1的优化，服务器在启动阶段调用fork()创建一个进程池。每个客户请求由当前可用子进程池中的某个子进程处理。

4.prethreaded

方案2的优化，服务器在启动阶段创建一个线程池。

以上都是阻塞式网络编程，程序流程往往阻塞在read上()，等待数据到达。但TCPshi 全双工，如果需要阻塞在read时，需要write，此时应该如何处理？

a.用两个线程，一个用于读，一个用于写

b.IO多路复用（IO multiplexing），让一个thread of control能处理多个连接。使用IO多路复用肯定要配合非阻塞IO，而使用非阻塞IO就要用到应用层buffer（本质是event-driven事件驱动）。**event-driven事件驱动经典模式reactor模式。**

**reactor模式的思想：**

网络编程中有很多事务性的工作，可以提取为公用的框架或库，而用户只需要填上关键的业务逻辑代码，并将其回调注册到框架中，就可以实现完整的网络服务。

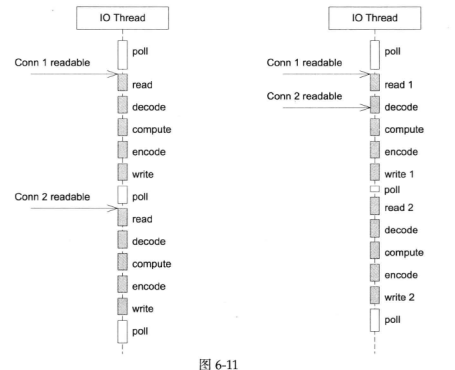
reactor的意义在于将消息（IO事件）分发到用户提供的处理函数，并保持网络部分的通用代码不变，独立于用户的业务逻辑。

5.poll(reactor)

单线程reactor，执行顺序如下左图：

没有事件的时候，线程等待在select/poll/epoll\_wait上，事件到达后由网络库处理IO，再把消息回调客户端代码。reactor事件循环所在的线程叫IO线程。通常由网络库负责读写socket，用户代码负责解码，计算，编码。

由于是单线程，因此事件是顺序处理的，非抢占。如果想要延迟计算，也不能用sleep()之类的阻塞调用，应该注册超时回调，以避免阻塞当前IO线程。



优点在于将网络框架和业务逻辑分离，缺点在于适合IO密集的应用，不适合CPU密集的应用，难以发挥多核的威力。

6.reactor + thread-per-task

对于每个请求而不是每个连接，创建一个新线程去计算。也就是可能会创建多个线程去计算同一连接上收到的多个请求，则计算结果的次序不确定。

7.reactor + worker thread

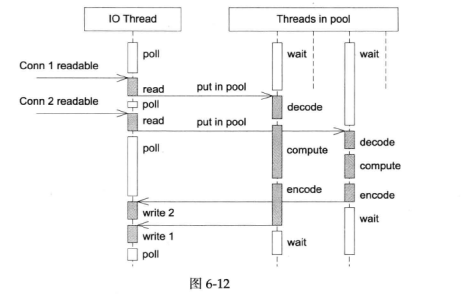
为每个连接创建一个计算线程，每个连接上的请求固定发给同一线程计算。

8.reactor + thread pool

全部的IO工作在reactor线程完成，计算任务交给thread pool。

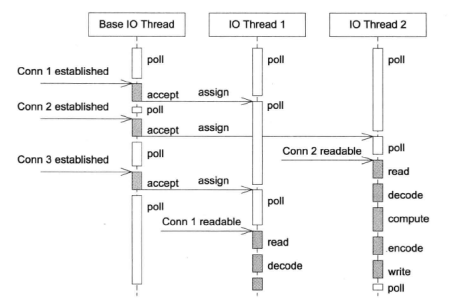
和方案5相比变化不大，只是把decode,compute,encode独立成一个函数，然后交给线程池去计算。

线程池的另一个作用就是执行阻塞操作，如在客户端连接之后，把数据库查询操作放到线程池中，就可以避免阻塞IO线程，不会影响其他客户连接。



9.reactors in threads

muduo采用的方式，一个主reactor负责accept连接，然后把连接挂在某个sub reactor中，这样该连接的所有操作都在那个sub reactor所处的线程中完成。



muduo采用固定大小的reactor pool，也就是说线程数是固定的。与8相比，减少了进出thread pool的两次上下文切换，在把多个连接分散到多个reactor线程之后，小规模的计算可以在当前IO线程完成并返回结果，从而降低延迟。

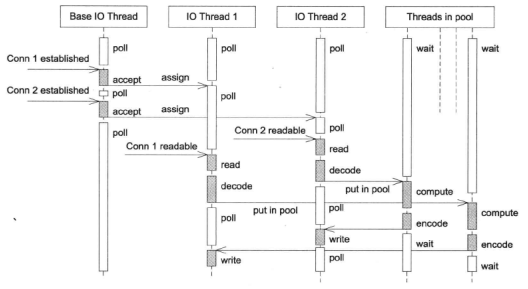
也就是说，方案9主reactor在accept之后，由子reactor处理该连接的所有操作，包括IO和计算。而方案8只有一个reactor，也就是所有连接的IO都在reactor中进行，把计算任务分配给thread pool。

10.reactors in processes

Nginx采用

11.reactors + thread pool

方案8和方案9的结合，既采用多个reactor来处理IO，又采用线程池来处理计算。这种方案适合既有突发IO（采用多个reactor处理多个连接上的IO），又有突发计算的应用（利用线程池把一个连接上的计算任务分配给多个线程去做）。



#### **muduo的实现**

**公开接口：**

**Buffer**：数据的读写通过Buffer进行

**InetAddress**：封装IPV4地址

**EventLoop**事件循环：每个线程只能有一个EventLoop，负责IO和定时器事件的分派，用eventfd来异步唤醒，用TimerQueue作为计时器管理，Epoll作为IO multiplexing

**EventloopThread**启动一个线程，在其中运行EventLoop::loop()

**TcpConnection**封装一次TCP连接，shared\_ptr管理其生命期

**TcpClient**客户端

**TcpServer**服务器

内部接口

**Channel** 负责注册和响应IO事件

**Socket**封装文件描述符，RAII

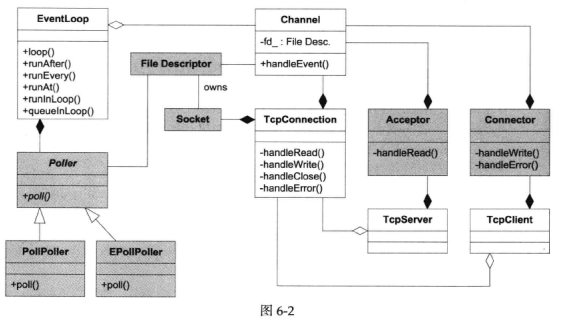
**Poller**是PollPoller和EpollPoller的基类，**PollPoller**和**EpollPoller**封装poll和epoll

**Connector**发起Tcp连接，是TcpClient的成员

**Acceptor**接收Tcp连接，是TcpServer的成员

**TimerQueue**用timerfd实现定时，设置epoll\_wait的等待时长，是EventLoop的成员

**EventLoopThreadPool**用于创建IO线程池，把TcpConnection分配到EventLoop线程，是TcpSercer的成员。

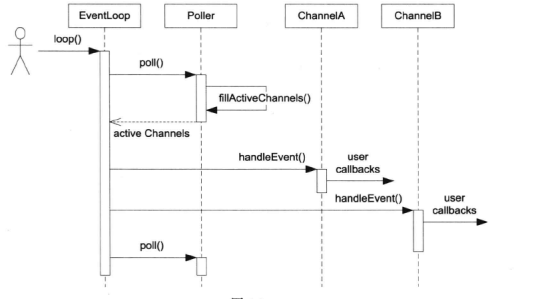


**reactor的核心是事件分发机制，即将IO多路复用拿到的IO事件分发给各个文件描述符的事件处理函数**

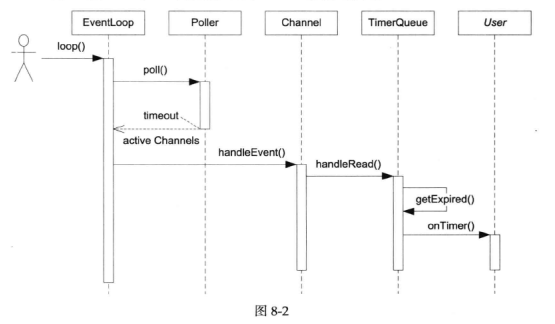
channel负责注册和响应IO，每个eventloop对象只属于一个eventloop，因此每个eventloop只属于一个IO线程。**每个channel对象自始至终只负责一个文件描述符的IO事件分发**，但不拥有这个fd，也不会析构。

channel把不同的IO事件分发为不同的回调，用bind实现。channel的生命期尤其owner class负责管理，它一般是其他类的直接或间接成员。

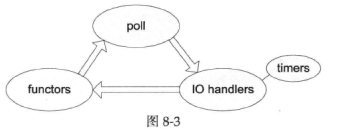
eventloop就是一个reactor，处理IO，每个channel对应一个文件描述符。以下为muduo的reactor模式。



通过timerfd可以用和处理IO事件相同的方式来处理定时。定时器的功能由三个class实现：Timerfd/Timer/TimerQueue



Eventloop只是一个类，需要在线程中执行，一个线程对应一个eventloop，定义EventloopThread类创建新线程，并在新线程中定义Eventloop对象，执行loop循环。

网络库的执行顺序，以上的类都是工具：  


Acceptor：  
用accept接收新连接，并通过回调函数通知使用者

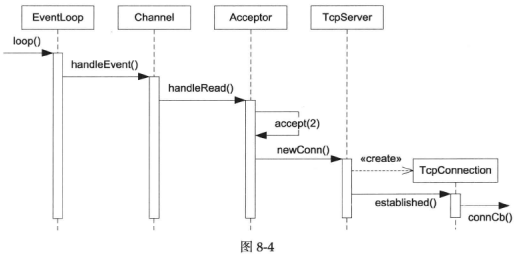
TcpServer：

首先通过Acceptor创建监听描述符并接收新连接。对每个新连接创建TcpConnection对象进行管理。

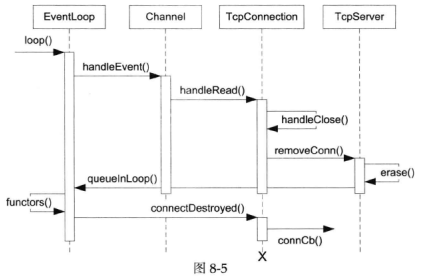
TcpConnection：

通过shared\_ptr来管理新连接，设置回调函数，handleRead。

建立新连接



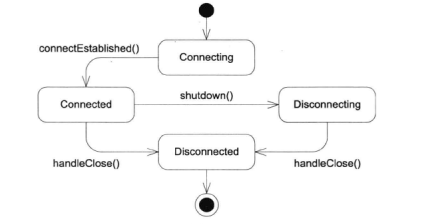
断开连接



目前只有一种关闭连接的方式，被动关闭。对端关闭连接后，read返回0，则调用TcpConnection::handleClose()关闭连接，handleClose()主要调用回调函数closeCallback\_，而closeCallback\_在TcpServer定义时绑定为TcpServer::removeConnection。

以上是连接以及handleRead()

TcpConnection状态图



TcpConnection::send()用于发送数据，如果一次性不能发送完毕，则把剩余的数据写入到发送缓存，并把socket可写事件加入监听。然后在socket可写时，调用TcpConnection：：handleWrite()继续发送剩余的数据。

多线程TcpServer

新建TcpConnetion时从event loop pool中挑选一个poll给TcpConnection用。EventLoopThreadPool类，也就是一个线程池，每个线程运行eventloop::loop()。

TcpServer和TcpConnection只处理单线程的情况，借助EventLoop::runInLoop并引入EventLoopThreadPool实现多线程TcpServer

Connetor类

主动发起连接，需要注意几个问题：

1.socket是一次性的，一旦出错如对方拒绝连接，则无法恢复，只能关闭重来，connector类每次使用新的socket和channel去建立连接

2.错误代码与accept不同，EAGAIN表示错误，表明本地端口用完，需要关闭socket再延期重试。即使socket可写，不一定意味这连接成功建立，用getsockopt确认。

3.重试的间隔应该逐渐延长。

4.处理自连接，客户端没有bind，有可能出现(sourceIP,sourcePort) = (destIP,destPort)的情况，此时需要断开连接重试。

#### muduo分析

在一个完整的网络框架中，需要考虑网络socket事件，信号，定时。现只分析muduo中对于socket的实现。

对于单线程Reactor模式

从用户使用的角度来看，以echo为例：



1.初始化一个EventLoop作为reactor驱动器

2.初始化一个TcpServer，并设置回调函数。定义了一个Acceptor对象，完成了服务端套接字的socket()和bind()。

3.调用TcpServer的start函数，进行Acceptor::listen()，并将监听描述符的read事件加入channel监听。

4.调用EventLoop的loop函数，进行事件循环，在单线程中，只有一个EventLoop。EventLoop中，首先调用Poller获取当前就绪的channel，然后调用channel::handleEvent()进行处理。

此时，Poller关注的描述符也就是channel只有一个，那就是监听描述符，当监听描述符可写之后，表明收到

muduo中用到的回调函数：

|  |  |  |
| --- | --- | --- |
| **TcpServer类** | | |
| **回调函数** | **赋值** | **作用** |
| connectionCallback\_ | 用户定义 | 建立新连接之后调用 |
| messageCallback\_ | 用户定义 | 收到消息后执行业务逻辑 |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **Acceptor类** | | |
| **回调函数** | **赋值** | **作用** |
| newConnectionCallback\_\_ | TcpServer::newConnection | accept返回之后调用，创建TcpConnection对象 |

|  |  |  |
| --- | --- | --- |
| **TcpConnection类** | | |
| **回调函数** | **赋值** | **作用** |
| connectionCallback\_ | TcpServer:: connectionCallback\_ |  |
| messageCallback\_ | TcpServer:: messageCallback\_ |  |
| writeCompleteCallback\_ | TcpServer:: writeCompleteCallback\_ |  |
| highWaterMarkCallback\_ |  |  |
| closeCallback\_ | TcpServer::  removeConnection |  |

|  |  |  |
| --- | --- | --- |
| **Channel类** | | |
| **TcpConnection的channel** | | |
| **回调函数** | **赋值** | **作用** |
| readCallback\_ | TcpConnection::handleRead | 根据读写情况调用messageCallback\_  writeCompleteCallback  closeCallback |
| writeCallback\_ | TcpConnection::handleWrite |
| closeCallback\_ | TcpConnection::handleClose |
| errorCallback\_ | TcpConnection::handleError |
| **Acceptor的channel** | | |
| readCallback\_ | Acceptor::handleRead | Accept新连接，然后调用  newConnectionCallback |

Acceptor类监听连接，TcpConnection负责管理已连接描述符。

Acceptor在回调函数newConnectionCallback中定义TcpConnection对象，并且调用TcpConnection::connectEstablished执行connectionCallback，并且设置新连接TcpConnection的channel的监听事件，加入读。然后将TcpConnection加入Poller的监听。

也就是说TcpServer管理一个Acceptor和一个TcpConnection的指针集合(ConnectionMap)。

Acceptor包含一个channel对象，一个监听socket对象，以及所属EventLoop的指针。

TcpConnection包含所属EventLoop的指针，已连接socket的指针（已连接socket在Acceptor中创建），channel指针（TcpConnection构造时定义）。

channel起桥接作用，本身并不拥有socket套接字，需要知道socket描述符。socket描述符和channel是绑定的关系，套接字的拥有者是Acceptor和TcpConnection。

Poller维护一个ChannelMap，指示要监听的描述符以及对应描述符的监听事件，这个函数用来更新ChannelMap。

EventLoop是一个reactor，拥有Poller用于IO复用，其中Poller所要监听的套接字及对应的监听事件通过Channel::update()来实现，调用关系为：

Channel:: enableReading() / enableWriting() / disableReading()🡪 Channel::update()🡪EventLoop::updateChannel🡪 Poller::updateChannel