# 陈硕大佬认为一个合格的网络库:

# 

# 这里不支持跨平台个人认为是跟编译器在处理epoll\_wait()的返回值有关系. linux

# 下gcc编译器在处理函数以值的形式返回,不会调用拷贝构造,而是直接移动构造. 但是windows下msvc会拷贝一次

**一 01 02  
以一个最简单的epoll服务端程序着手, 即不做封装,把全部操作都暴露在外面,之后再一步步地封装.  
为什么最开始一点都不封装? 为了更好地分析数据的组织,各种操作之间的联系.  
并且,这种模式与REACTOR的层层回调十分相似.  
基本的准备:  
一. 监听采用水平触发, 通信采用边缘触发.  
二. 准备好socket需要设置的属性:  
 1) 必须的**

**SO\_REUSEADDR选项允许在服务器套接字关闭后立即重新绑定到相同的端口。这对于快速重启服务器并在同一端**

**在多个套接字绑定到相同的端口时，此选项可以确保套接字被成功绑定而不会报告“地址已在使用中”的错误。**

**2)必须的**

**TCP\_NODELAY选项用于禁用Nagle算法，该算法用于延迟发送小数据包，以便将多个小数据包合并成一个大的数**

**在某些情况下（例如实时通信），禁用Nagle算法可以减少延迟，但可能会增加网络流量。  
网络编程中，使用TCP\_NODELAY选项可以禁用Nagle算法，使得小数据包也可以立即发送，这对于实时性要求较高的应用会有帮助**

**3)有用,Reactor模型中意义不大,用于监听的socket只有一个**

**SO\_REUSEPORT选项允许多个套接字绑定到相同的IP地址和端口，这对于实现负载平衡或多线程服务器很有用**

**通过使用SO\_REUSEPORT，多个套接字可以同时接收传入的连接，而不会被内核限制在一个套接字上。**

**4)可能有用,但是不如自己做心跳,根据业务设置,可以降低资源占用,提高性能**

**SO\_KEEPALIVE选项允许在连接空闲时发送保持活动（keep-alive）消息以检测对等端的连接状态。**

**如果对等端意外关闭连接或网络中断，服务器可以及时发现并采取相应的措施。**

**这对于长时间空闲的连接或与不可靠网络连接的情况下很有用，但要注意，它会在网络上增加轻微的负载。**

1. **监听的socket 与通信的socket  
   监听的socket专门负责接受客户端的连接. (读事件)  
   通信的socket有四种情况: 对端已关闭 有数据可以读(读事件) 有数据可以写 发生错误**

**其中,将监听的socket的读事件与通信的socket的读事件放到同一模块中**

**原因: Reactor模型是基于对象的(object-base) 而不是面向对象的(object-oriented). 这么处理是为了后面基于同一类事件注册不同的回调函数打下基础.**

**二 03 04 05**

**将上述代码中的冗余部分抽取出来: socket的api epoll的api   
,封装为 地址协议类(InetAddress), 套接字类(Socket), I/O多路复用epoll类(Epoll, 内部存储了epoll\_event[] 数组)   
这里有第一个细节: 每个事件的处理采用分支进行过滤, 这意味着在这些分支内创建的对象是局部的,那么为了延长生命周期,  
 可以从堆区new出对象, 外部存储这些指针,然后程序退出时再一一销毁.  
第二个细节: 虽然目前是单线程的程序,但是设计的时候不能只考虑到单线程的执行效率,这里我利用编译器的优化,选择拷贝一份epoll\_event[]数组  
的内容,而不是直接使用这个数组.  
 为什么? 因为在多线程中,为了追求效率,我会考虑使用线程池来分担事件处理,让事件循环执行得更快.这样子就存在异步执行,那么就会有多个线程同时  
使用epoll\_event[]数组, 这就引出了一个问题: 加锁还是拷贝?   
肯定是选择拷贝,因为拷贝可以保证异步执行,再说了一个epoll\_event[]数组能有多大?(epoll\_event内部还使用了联合体进一步缩小了内存占用)**

**, 加锁保证不了异步执行,甚至可能拖累速度.**

**三 06 07 08**

Channel的目标是提供一个简单的接口让应用程序能知道它的文件描述符何时可以进行非阻塞的读写操作  
在Muduo网络库的语境中，它表示的是一种负责文件描述符事件分发的工具，你可以将它理解为事件发生与其处理之间的一座"桥梁"或"通道"。

**封装对象是回调函数的基础,之前的封装可以说都是与业务无关的封装,但是封装channel至关重要,channel中的函数就是事件处理函数,直接与业务挂钩.  
这里不封装不是更好? 与业务相关的直接显示出来,一目了然.   
不好,这里仅仅只是网络库的开端,把核心业务固定之后,应该考虑的是扩展功能,那就应该封装. 采用层层回调的方式不断提高程序的可用性.**

**四.09 10  
经过上面的封装,已经基本实现了逻辑上一个epollfd(一颗红黑树)对应多个channel(桥梁),由红黑树将发生的事件送入桥梁**

**与实现上一致的效果了. I/O多路复用中,对事件是批处理的,把这抽象成一棵树不断地抽取能量然后结果的过程. 那么这一过程应该是  
这棵树的一个方法,于是抽取为一个包含Epoll类的新类EventLoop类(一棵真正的树).**

1. **11  
   tcpServer: 一个tcpServer有多个eventloop(多棵树, 一棵签到树, 其余是结果树), 暂时只写一个(单线程)  
     
   六.12 13 14 15 16 17**

**考虑到监听的socketfd和客户端连接的socketfd功能是不同的,生命周期也不同.  
一个用于接收连接, 其余是用于I/O操作的.  
所以抽取为两部分: Acceptor 与 Connection, 归属于TcpServer  
  
七.18 19 20 21   
解决TCP分包粘包的问题,加入收发缓冲区Buffer  
网络库中，接收缓冲区和发送缓冲区各自有其特殊的职能。**

**接收缓冲区（input buffer）：负责贮存从socket中读取的数据，之后应用程序可以在任何时候从中取出数据进行处理。由于网络I/O的不确定性，接收到的数据可能是不完整的，需要通过缓冲区来暂存数据，以便实现粘包处理和半包读取。**

**发送缓冲区（output buffer）：负责存储待发送的数据，当socket可写时，将数据写入socket。这可以实现应用层的缓冲，避免了频繁的系统调用。当你一次性要发送大量数据时，可能会大于系统socket缓冲区的容量，此时就需要发送缓冲区来暂存这些数据，等待发送。  
定长报文 加报文长度的报文 类似http协议\r\n\r\n结尾分隔符  
  
八.22  
上面已经基本解决了 建立连接 处理消息 发送消息 关闭连接 错误处理.**

**这里主要是增加一些通知的方法. 例如消息发送完毕之后的通知, epoll超时的通知**

**九.23 24  
到这里,已经实现了 底层类:**

**Socket：无论是客户端还是服务器端，Socket都是建立网络通信的基础。**

**InetAddress：整个网络通信中，需要确定的就是网络地址，所以这个类的重要性很高。**

**Channel：所有的I/O操作都由Channel提供，所以它的重要性也是毋容置疑的。**

**Buffer：Buffer中存放了实际的数据，Channel会将数据读到Buffer内，或者将Buffer里的数据写到通道内。没有这个类，我们就无法操作数据。**

**Connection：建立和处理Connection是网络通信的重要步骤。**

**Epoll：操作系统级别的I/O复用技术，用于提升服务器性能，管理多个Socket连接。**

**EventLoop：负责将不同的事件分发到对应的处理程序。**

**Acceptor：在服务端接受客户端连接的组件，比起其他核心组件其重要性相对要小一些。**

**TcpServer：具体的应用往往是围绕某个具体的Server进行的，如HttpServer，FtpServer，所以TcpServer重要.**

**这里将底层类运用起来, 先制作一个回显业务类,一个回显服务类对应一个TCPserver, 多线程线程池以后再tcpServer中实现即可.  
  
十.25  
开始实现线程池 C11线程, 生产者消费者模型(锁,条件变量), lambda函数(原地构造) this指针  
 包装器 绑定器   
  
因为之前已经将不同功能的socketfd封装为了Connection和 Acceptor对象,所以可以使用线程池开通事件循环,  
在事件循环中监视不同的对象.  
  
十一.26  
多线程的主从Reactor模型. (接收连接线程+I/O线程)  
运行多个事件循环, 主事件循环运行于主线程,从事件循环丢到线程池中.  
由主事件循环接收连接请求, 把连接对象注册给从事件循环.**

1. **27**

**I/O密集型与CPU密集型的处理方式?  
I/O密集型使用上面的模式就可以了;  
但是CPU密集型把大量时间花在计算上,  
I/O不会阻塞事件循环(采用非阻塞的socket), 但是计算可能会阻塞.   
所以还要再次发挥多线程的优势. 增加工作线程处理客户端发来的信息**

**这里工作线程中会用到I/O线程中的资源,所以资源释放就成了问题? 跨线程使用资源要使用智能指针shared\_ptr**

1. **28 29 30**

**使用 智能指针shared\_ptr 智能指针unique\_ptr 原始指针 修改类中的成员. 少数使用栈  
工作线程与I/O线程都会使用到Connection对象中的资源buffer,为了防止客户端关闭连接后**

**I/O线程销毁Connection对象时工作线程还在使用  
 必须使用shared\_ptr延长对象生命周期.  
为了防止同一原始指针初始化多个智能指针的情况,要继承 enable\_shared\_from\_this.  
这里还不够, 因为延长了对象生命周期,那么水平触发下会不断地通知(busy loop),虽然I/O中使用的是边缘触发.  
但是也不应该暴露潜在的风险, 所以应该直接把对应的fd()先关闭,也就是先移除Connection内部的channel, 然后也应该让工作线程停止执行了.  
要注意改为智能指针后在使用绑定器和包装器时指针是指内部的指针.  
  
其他指针如果属于类自己就使用智能指针.  
如果不属于类自己那么还是使用原始指针.  
因为如果不属于类自己,那么就不是在类中生成的了,需要外界传入,如果外界原本传入的是this指针,那么就  
没法使用unique\_ptr了.  
  
十三. 31 32**

**解决了工作线程和I/O线程使用对象的生命周期之后,同步又是一个问题.  
I/O线程是发送buffer中的数据,而工作线程却是处理数据后追加到buffer中. 这个时候要采取合适的同步方法.  
直接给共同操作的资源加锁? 个人认为肯定不行. 如果加锁,还不如直接在I/O线程中处理业务.  
  
要采取线程间通信的方法. socket pipe eventfd ,**

**采用eventfd加入事件循环中(wakeupchannel), 归属为I/O线程即可.**

**(加是累加,读是一次读完,不会出现空读的现象,我设计为I/O线程获得锁之后也是一次性执行完与eventfd中读取的数据是一比一的关系).   
这里也使用到了锁,不过不是操作buffer的锁,而是注册回调函数的锁,回调函数加锁解锁是非常快的.  
之后工作线程把数据追加和发送的操作注册到回调函数中,然后使用eventfd通知I/O线程取出来执行就行了.**

**十四. 33 34**

**除了客户端断开连接, 也存在客户端占线的情况.这个时候就需要定时器来定期清理Connection连接了.  
也可以避免被攻击.(恶意发起多个TCP连接)  
定时器也使用epoll来管理,新技术总是好的嘛. timerchannel归属到eventloop中.  
创建一个时间类归属于Connection类,然后每次有连接时再把Connection的shared\_ptr拷贝一份到Eventloop中.  
使用锁,主事件循环和从事件循环会操作同一资源,使用互斥锁加锁.  
  
十五. 35  
网络库的退出.  
主线程的退出+I/O线程的退出+工作线程的退出  
使用原子变量+定时器:  
将原来的while(true)循环更改为基于原子变量的循环,使用信号并注册信号处理函数, 在信号处理函数中让服务器修改原子变量的值,达到终止服务的目的.**

**同时,为了防止阻塞在epoll\_wait,使用wakeup激活定时器.**

1. **36**

**Buffer的优化,忽略粘包分包 处理报文长度+报文体 处理http协议(get请求)**

1. **37 38**

**性能测试: 根据前面的回显服务器  
I/O密集型, qps 一秒处理 一百万  
CPU密集型, 采用诸如开根号,浮点数多次相乘等方法. qps约为 I/O密集型的一半**

1. **39**

**模拟真正的业务  
,加入心跳报文, 登录, 注册, 转账等.**