Nginx/Netty/ZeroMQ网络模型

为啥会有线程或者进程模型这种东西，因为计算机CPU主频已经很难再提高了，现在更倾向于设计多核系统，那么要发挥现在计算机的处理能力，就需要将系统设计成支持多处理器的，最简单的那肯定就是多线程（进程）程序了。。。尤其是在网络编程中，特别是对于每个事物都是独立的情况下，例如没有会话的HTTP请求啥的，这种方式可以极大的提高程序的吞吐量和伸缩性。。。。

当然，多线程，多进程的方式必然会增加程序设计的复杂性，尤其是在一些同步，加锁的地方会使编码难度变大，不过其实遵循良好的模式来设计程序的话，可以避免陷入太多的坑。。。。

好了，下面先来看看Nginx的设计吧：

嗯，Nginx属于比较单纯的HTTP服务器，对于它来说，每一个http请求都是独立的，所以不用考虑session啥的这种共享的东西，所以在需要进行同步的地方很少。。。。

Nginx采用的是Master/Worker模型，一个master进程，多个Worker进程。。。

所有的Worker进程之间只有很少的共享变量，首先是一段共享内存，这里面我现在还能想起来的大概有：

（1）其中的一段用于锁的实现，Nginx所有worker进程之间的同步都是通过这把锁来完成的

（2）好像还有一个计数器，用于保存所有的http请求数量吧。。。好像是。。记不太清了。。。

另外所有的由于监听Socket是在Master进程中创建的，所以它的文件描述符被所有的Worker进程共享。、。。。

好啦，用一张图来描述一下整个上面这些东西的大致结构吧：

其实画了这张图之后，觉得好像读过的所有源码的实现其实都差不太多。。主体基本上都是这样子的。。。。

由于监听器Socket如果同时被注册到多个epoll里面之后，会引起惊群现象，所以Nginx中所有的worker进程需要获取共享内存中的一个锁之后才能将这个监听Socket放入到自己的Epoll里面，这样就将外部的TCP连接分布到了每一个Worker进程，每个Woker进程处理自己的连接的请求就ok了。。。够简单吧。。。其实这也体现了在并发编程中特别重要的一个原则：**封闭原则**，将可变的状态信息封闭在线程或者进程的内部，这样就没有了并发环境下的不一致的情况了。。。

其实这里还有另外一种实现，不再将监听器Socket交给Worker进程来共享，而是将其放到另外一个地方，然后将接收到的连接的socket的文件描述符传递给Worker进程。。然后Worker进程中再具体的处理这些socket的IO请求就好了。。这样就省去了对监听Socket进行同步的过程。。。

好了，接下来来看看Netty吧：

首先Netty中也严格遵循了封闭性，不过这里就是线程封闭性了，每一个执行线程都有自己的selector，然后所有外部的TCP连接都分属于这些Selector，也就分属于这些执行线程，对于这些TCP连接的所有IO操作，都将会封闭在所属的线程内部。。。

其实对于Netty最重要的就是要知道它是如何实现线程封闭的，因为在java多线程程序中会存在许多的线程，有用户线程，Netty自己的IO线程。其余的线程池啥的，而这些对象会在线程之间传递。。如何确保属于某个Netty的IO线程的连接上面的IO操作只由这个线程来执行就需要一些额外的机制了来实现了。。。。

其实要搞清楚整个Netty的运行情况,将它的EventLoopGroup以及EventLoop搞清楚就差不多了:

（1）EventLoopGroup继承自EventExecutorGroup，而EventExecutorGroup可以将它看成是一个线程池，而EventLoopGroup只不过是扩展了了线程池的功能，加入了一些事件的处理。例如channel的注册啥的。。

（2）EventLoop继承自EventExecutor，而一个EventExecutor可以将其看成是一个线程池中的执行线程，而EventLoop扩展了EventExecutor的主循环，一般情况下线程池中的线程要做的就是不断的从任务队列里面后去Task，然后执行他们，而EventLoop中加入了select以及IO的处理过程。。。

（3）每一个channel都注册到了其中一个EventLoop的selector上面，在这个channel上面产生的所有IO事件的处理都将会是在这个EventLoop中进行处理。。。

下面用一张图来说明一下上面提到的之间的关系吧：

上面图形算是刻画了几个东西之间的对应关系吧，其实这里更重要的还是分析Netty中如何保证Channel上面的IO操作都保证在其所属的EventLoop中执行。。。来看一段代码吧：

```
    private void write(Object msg, boolean flush, ChannelPromise promise) {
        DefaultChannelHandlerContext next = findContextOutbound();
        EventExecutor executor = next.executor();
        if (executor.inEventLoop()) {
            next.invokeWrite(msg, promise);
            if (flush) {
                next.invokeFlush();
            }
        } else {
            int size = channel.estimatorHandle().size(msg);
            if (size > 0) {
                ChannelOutboundBuffer buffer = channel.unsafe().outboundBuffer();

                if (buffer != null) {
                    buffer.incrementPendingOutboundBytes(size, false);
                }
            }
            executor.execute(WriteTask.newInstance(next, msg, size, flush,
promise));
        }
    }
```

其实这里的实现挺简单的，一般情况下针对Channel的IO操作都会在Channel对象的pipeline上面调用相应的方法，这里举出了write方法，其余的IO操作的方法其实都一样，可以看到这里其实会判断当前的执行线程是否是当前channel所属的EventLoop的执行线程，如果是的话，那么可以直接在当前线程中具体的执行IO操作了，如果不是的话，那么就需要创建一个Task，将IO操作封装在这个task里面，然后提交给所属的EventLoop，这样，也就保证了Channel的所有IO操作都封闭在其所属的EventLoop中了。。

够简单的吧。。。

这里需要注意的是，每一个EventLoop都有自己的task队列，而不是所有的EventLoop共享同一个task队列，这与一般的线程池的实现有不太一样的地方。。。。

下面先来看看ZeroMQ（java）的设计实现吧：

个人感觉ZeroMQ(java)中的实现算是比较奇葩的一种了。。。。

在ZeroMQ中有专门封装出了IO线程，也就是IOThread类型，可以就将其理解为工作线程，而且一般情况下所有的对象都要依附于相应的IOThread对象才能运行。。。

这里要做的就是如何将这些对象的方法的执行都封闭在各自所属的IOThread对象的工作线程中运行。。。。嗯，这个个人觉的实现的比较的婉转。。。

来看一张图形吧：

嗯，看起来挺复杂的，来具体的说明一下这张图上的东西是怎么运行的吧：

（1）CTX是整个ZeroMQ的执行上下文，它会维护当前所有的socket对象以及IOThread对象，每一个IOThread对象都有自己的Poller对象，用于维护channel的注册，以及事件的回调，当然这些都是在当前的工作线程中执行的

（2）每一个具体的连接将会注册到其中一个IO线程的poller上面去，这里还有一个比较奇特的东西，那就是MailBox，每一个IO线程都有一个自己的mailbox，这里与其说对象方法的执行依赖于IO线程，还不如说依赖于某一个mailbox，

（3）ZObject可以理解为用于响应命令以及发送命令的对象，它可以给其他的ZObject对象发送命令，让其执行某个方法，也要接受别的ZObject发送过来的命令，执行相应的操作，那么其实所有的命令都是其实都是直接放到ZObject对象所属的MailBox的命令队列里面去的。。。然后命令里将会含有目的对象的引用。。那么接下来的事情就是如何让目的对象的相应的方法执行。。。

（4）这个就比较囧了，每一个mailbox都有一个channel注册到poller上面，当别的地方给这个mailbox发送命令的时候，会同时给这个channel写一个标志数据，这样就会被poller上面的selector感应到，这个时候的IO回调是执行mailbox里面的所有命令，这样也就保证了对象执行命令的时候是在其所依赖的IO线程中执行的。。。

够婉转的了。。。