11 | 如何实现高性能的异步网络传输？

2019-08-15 李玥

你好，我是李玥。上一节课我们学习了异步的线程模型，异步与同步模型最大的区别是，同步模型会阻塞线程等待资源，而异步模型不会阻塞线程，它是等资源准备好后，再通知业务代码来完成后续的资源处理逻辑。这种异步设计的方法，可以很好地解决IO等待的问题。

我们开发的绝大多数业务系统，它都是IO密集型系统。跟IO密集型系统相对的另一种系统叫计算密集型系统。通过这两种系统的名字，估计你也能大概猜出来IO密集型系统是什么意思。

IO密集型系统大部分时间都在执行IO操作，这个IO操作主要包括网络IO和磁盘IO，以及与计算机连接的一些外围设备的访问。与之相对的计算密集型系统，大部分时间都是在使用CPU执行计算操作。我们开发的业务系统，很少有非常耗时的计算，更多的是网络收发数据，读写磁盘和数据库这些IO操作。这样的系统基本上都是IO密集型系统，特别适合使用异步的设计来提升系统性能。

应用程序最常使用的IO资源，主要包括磁盘IO和网络IO。由于现在的SSD的速度越来越快，对于本地磁盘的读写，异步的意义越来越小。所以，使用异步设计的方法来提升IO性能，我们更加需要关注的问题是，如何来实现高性能的异步网络传输。

今天，咱们就来聊一聊这个话题。

理想的异步网络框架应该是什么样的？

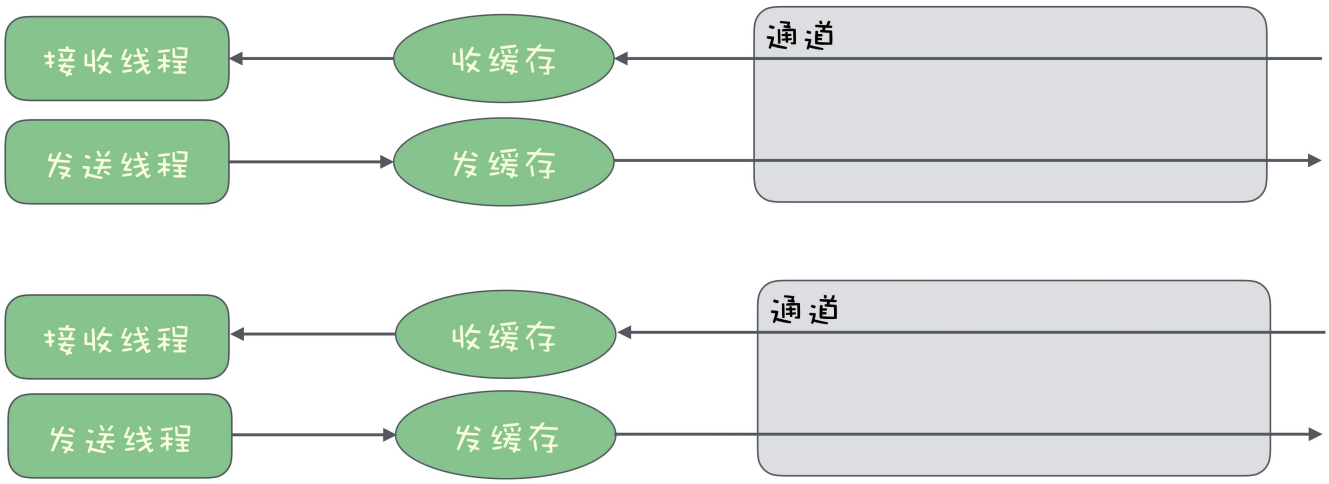
在我们开发的程序中，如果要实现通过网络来传输数据，需要用到开发语言提供的网络通信类

库。大部分语言提供的网络通信基础类库都是同步的。一个TCP连接建立后，用户代码会获得一个用于收发数据的通道。每个通道会在内存中开辟两片区域用于收发数据的缓存。

发送数据的过程比较简单，我们直接往这个通道里面来写入数据就可以了。用户代码在发送时写入的数据会暂存在缓存中，然后操作系统会通过网卡，把发送缓存中的数据传输到对端的服务器上。

只要这个缓存不满，或者说，我们发送数据的速度没有超过网卡传输速度的上限，那这个发送数据的操作耗时，只不过是一次内存写入的时间，这个时间是非常快的。所以，**发送数据的时候同步发送就可以了，没有必要异步。**

比较麻烦的是接收数据。对于数据的接收方来说，它并不知道什么时候会收到数据。那我们能直接想到的方法就是，用一个线程阻塞在那儿等着数据，当有数据到来的时候，操作系统会先把数据写入接收缓存，然后给接收数据的线程发一个通知，线程收到通知后结束等待，开始读取数据。处理完这一批数据后，继续阻塞等待下一批数据到来，这样周而复始地处理收到的数据。



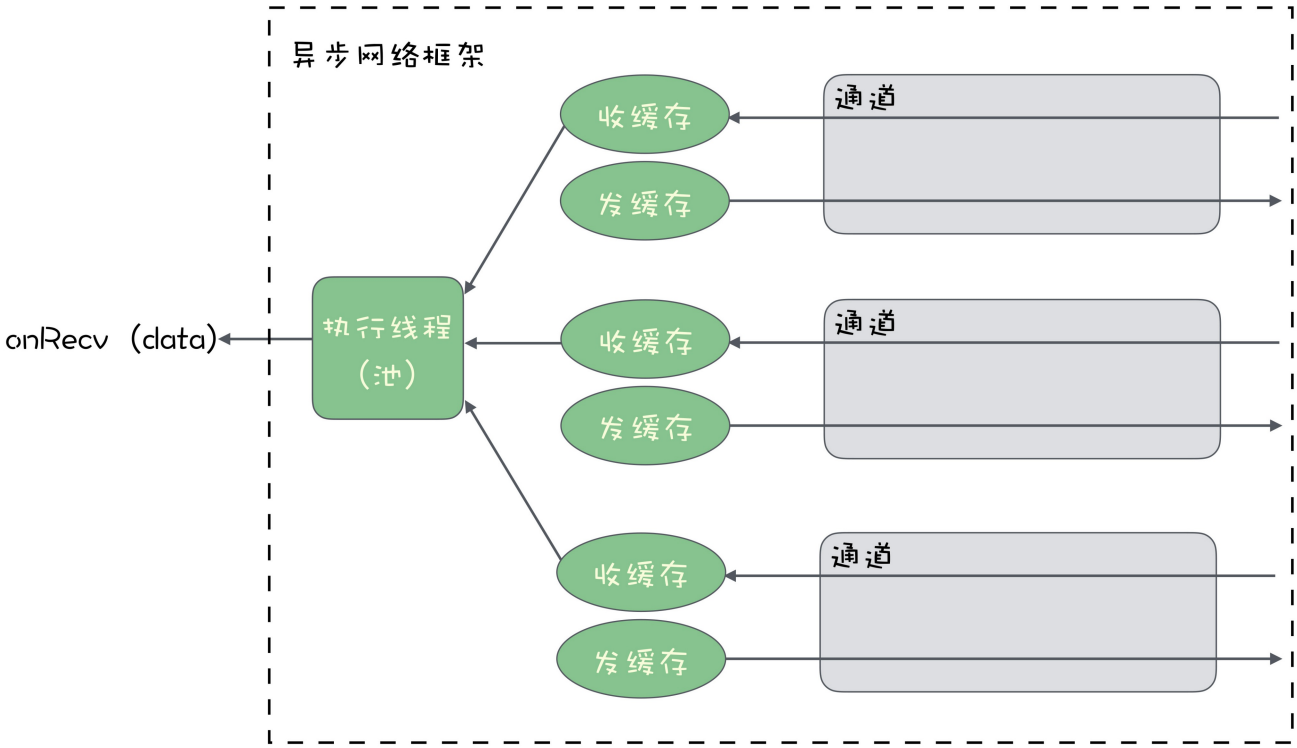
这就是同步网络IO的模型。同步网络IO模型在处理少量连接的时候，是没有问题的。但是如果要同时处理非常多的连接，同步的网络IO模型就有点儿力不从心了。

因为，每个连接都需要阻塞一个线程来等待数据，大量的连接数就会需要相同数量的数据接收线程。当这些TCP连接都在进行数据收发的时候，会导致什么情况呢？对，会有大量的线程来抢占CPU时间，造成频繁的CPU上下文切换，导致CPU的负载升高，整个系统的性能就会比较慢。

所以，我们需要使用异步的模型来解决网络IO问题。怎么解决呢？

你可以先抛开你知道的各种语言的异步类库和各种异步的网络IO框架，想一想，对于业务开发者来说，一个好的异步网络框架，它的API应该是什么样的呢？

我们希望达到的效果，无非就是，只用少量的线程就能处理大量的连接，有数据到来的时候能第一时间处理就可以了。



对于开发者来说，最简单的方式就是，事先定义好收到数据后的处理逻辑，把这个处理逻辑作为一个回调方法，在连接建立前就通过框架提供的API设置好。当收到数据的时候，由框架自动来执行这个回调方法就好了。

实际上，有没有这么简单的框架呢？

使用Netty来实现异步网络通信

在Java中，大名鼎鼎的Netty框架的API设计就是这样的。接下来我们看一下如何使用Netty实现异步接收数据。

```
// 创建一组线性
EventLoopGroup group = new NioEventLoopGroup();

try{
    // 初始化Server
    ServerBootstrap serverBootstrap = new ServerBootstrap();
    serverBootstrap.group(group);
    serverBootstrap.channel(NioServerSocketChannel.class);
    serverBootstrap.localAddress(new InetSocketAddress("localhost", 9999));

    // 设置收到数据后的处理的Handler
    serverBootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
        protected void initChannel(SocketChannel socketChannel) throws Exception {
            socketChannel.pipeline().addLast(new MyHandler());
        }
    });
    // 绑定端口，开始提供服务
    ChannelFuture channelFuture = serverBootstrap.bind().sync();
    channelFuture.channel().closeFuture().sync();
} catch(Exception e){
    e.printStackTrace();
} finally {
    group.shutdownGracefully().sync();
}
```

这段代码它的功能非常简单，就是在本地9999端口，启动了一个Socket Server来接收数据。我带你一起来看一下这段代码：

1. 首先我们创建了一个EventLoopGroup对象，命名为group，这个group对象你可以简单把它理解为一组线程。这组线程的作用就是来执行收发数据的业务逻辑。
2. 然后，使用Netty提供的ServerBootstrap来初始化一个Socket Server，绑定到本地9999端口上。
3. 在真正启动服务之前，我们给serverBootstrap传入了一个MyHandler对象，这个MyHandler是我们自己来实现的一个类，它需要继承Netty提供的一个抽象类：ChannelInboundHandlerAdapter，在这个MyHandler里面，我们可以定义收到数据后的处理逻辑。这个设置Handler的过程，就是我刚刚讲的，预先来定义回调方法的过程。
4. 最后就可以真正绑定本地端口，启动Socket服务了。

服务启动后，如果有客户端来请求连接，Netty会自动接受并创建一个Socket连接。你可以看到，我们的代码中，并没有像一些同步网络框架中那样，需要用户调用Accept()方法来接受创建连接的情况，在Netty中，这个过程是自动的。

当收到来自客户端的数据后，Netty就会在我们第一行提供的EventLoopGroup对象中，获取一个IO线程，在这个IO线程中调用接收数据的回调方法，来执行接收数据的业务逻辑，在这个例子中，就是我们传入的MyHandler中的方法。

Netty本身它是一个全异步的设计，我们上节课刚刚讲过，异步设计会带来额外的复杂度，所以这个例子的代码看起来会比较多，比较复杂。但是你看，其实它提供了一组非常友好API。

真正需要业务代码来实现的就两个部分：一个是把服务初始化并启动起来，还有就是，实现收发消息的业务逻辑MyHandler。而像线程控制、缓存管理、连接管理这些异步网络IO中通用的、比较复杂的问题，Netty已经自动帮你处理好了，有没有感觉很贴心？所以，非常多的开源项目使用Netty作为其底层的网络IO框架，并不是没有原因的。

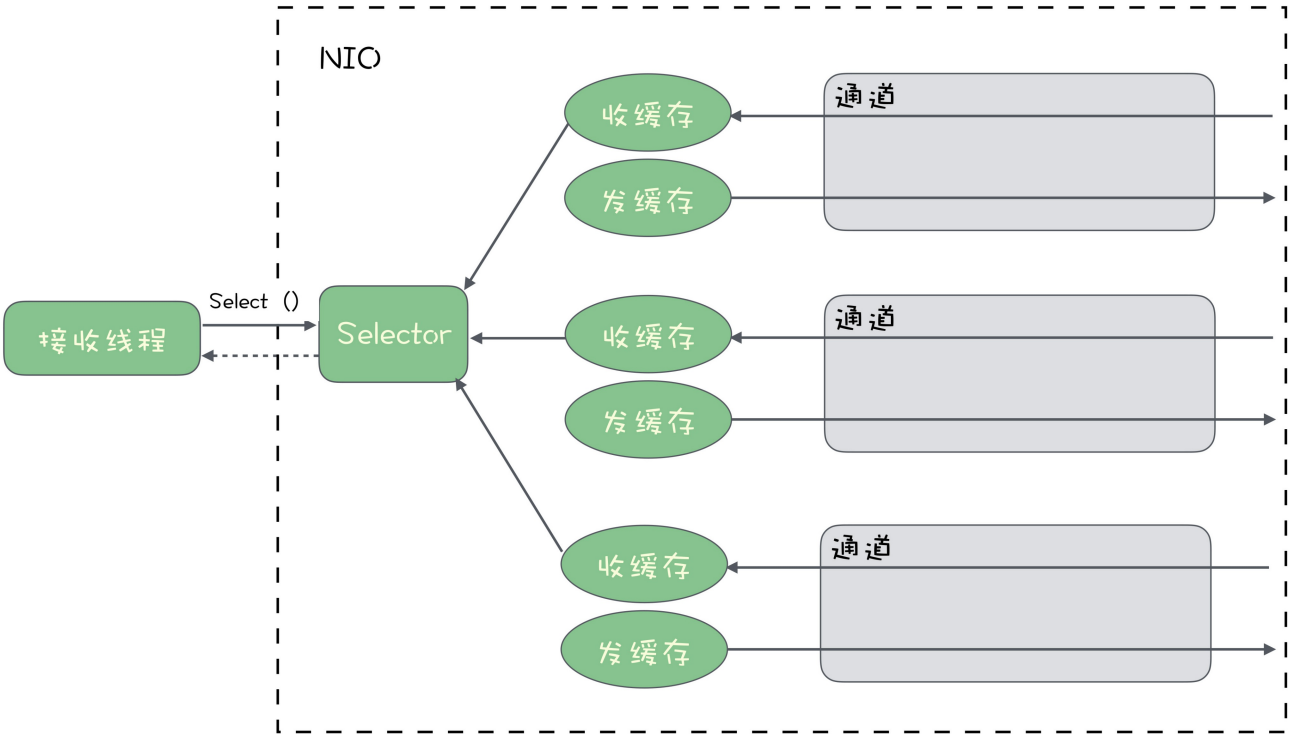
在这种设计中，Netty自己维护一组线程来执行数据收发的业务逻辑。如果说，你的业务需要更灵活的实现，自己来维护收发数据的线程，可以选择更加底层的Java NIO。其实，Netty也是基于NIO来实现的。

使用NIO来实现异步网络通信

在Java的NIO中，它提供了一个Selector对象，来解决一个线程在多个网络连接上的多路复用问题。什么意思呢？在NIO中，每个已经建立好的连接用一个Channel对象来表示。我们希望能实现，在一个线程里，接收来自多个Channel的数据。也就是说，这些Channel中，任何一个Channel收到数据后，第一时间能在同一个线程里面来处理。

我们可以想一下，一个线程对应多个Channel，有可能会出现这两种情况：

1. 线程在忙着处理收到的数据，这时候Channel中又收到了新数据；
2. 线程闲着没事儿干，所有的Channel中都没收到数据，也不能确定哪个Channel会在什么时候收到数据。



Selecor通过一种类似于事件的机制来解决这个问题。首先你需要把你的连接，也就是Channel绑定到Selector上，然后你可以在接收数据的线程来调用Selector.select()方法来等待数据到来。这个select方法是一个阻塞方法，这个线程会一直卡在这儿，直到这些Channel中的任意一个有数据到来，就会结束等待返回数据。它的返回值是一个迭代器，你可以从这个迭代器里面获取所有Channel收到的数据，然后来执行你的数据接收的业务逻辑。

你可以选择直接在这个线程里面来执行接收数据的业务逻辑，也可以将任务分发给其他的线程来执行，如何选择完全可以由你的代码来控制。

小结

传统的同步网络IO，一般采用的都是一个线程对应一个Channel接收数据，很难支持高并发和高吞吐量。这个时候，我们需要使用异步的网络IO框架来解决问题。

然后我们讲了Netty和NIO这两种异步网络框架的API和他们的使用方法。这里面，你需要体会一下这两种框架在API设计方面的差异。Netty自动地解决了线程控制、缓存管理、连接管理这些问题，用户只需要实现对应的Handler来处理收到的数据即可。而NIO是更加底层的API，它提供了Selector机制，用单个线程同时管理多个连接，解决了多路复用这个异步网络通信的核心问题。

思考题

刚刚我们提到过，Netty本身就是基于NIO的API来实现的。课后，你可以想一下，针对接收数据这个流程，Netty它是如何用NIO来实现的呢？欢迎在留言区与我分享讨论。

感谢阅读，如果你觉得这篇文章对你有一些启发，也欢迎把它分享给你的朋友。

精选留言

游弋云端

24

关于JAVA的网络，之前有个比喻形式的总结，分享给大家：

例子：有一个养鸡的农场，里面养着来自各个农户（Thread）的鸡（Socket），每家农户都在农场中建立了自己的鸡舍（SocketChannel）

1、BIO：Block IO，每个农户盯着自己的鸡舍，一旦有鸡下蛋，就去做捡蛋处理；

2、NIO：No-Block IO-单Selector，农户们花钱请了一个饲养员（Selector），并告诉饲养员（register）如果哪家的鸡有任何情况（下蛋）均要向这家农户报告（select keys）；

3、NIO：No-Block IO-多Selector，当农场中的鸡舍逐渐增多时，一个饲养员巡视（轮询）一次所需时间就会不断地加长，这样农户知道自己家的鸡有下蛋的情况就会发生较大的延迟。怎么解决呢？没错，多请几个饲养员（多Selector），每个饲养员分配管理鸡舍，这样就可以减轻一个饲养员的工作量，同时农户们可以更快的知晓自己家的鸡是否下蛋了；

4、Epoll模式：如果采用Epoll方式，农场问题应该如何改进呢？其实就是饲养员不需要再巡视鸡舍，而是听到哪间鸡舍的鸡打鸣了（活跃连接），就知道哪家农户的鸡下蛋了；

5、AIO：Asynchronous I/O, 鸡下蛋后，以前的NIO方式要求饲养员通知农户去取蛋，AIO模式出现以后，事情变得更加简单了，取蛋工作由饲养员自己负责，然后取完后，直接通知农户来拿即可，而不需要农户自己到鸡舍去取蛋。

2019-08-15

作者回复

这个比喻非常赞

2019-08-16

业余阜

14

多回到队列上来吧。Netty几乎很多Java课都会讲到。。。

2019-08-15

作者回复

我们还是需要一个例子能让大家理解异步网络传输的。

2019-08-16

芥末小龙

6

玥哥秋安：我今天看了这个课程我觉得就有同学要说，老师你是不是跑题了，首先我要说一下为什么要说今天这讲课，并且用netty来举栗子。

第一：了解一下异步网络传输的原理

第二：用Netty来举栗子是因为Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。

第三：rocketMQ的底层就是用Netty实现的。

第四：可以看一下第9节课如果学习开源代码，然后在看一下rocketMQ的源码。然后根据MQ的Producer，Consumer，Broker，NameSrv等的底层实现是不是理解了玥哥的良苦用心。

个人见解勿喷！

2019-08-16

linqw

4

学习完高性能网络传输，写下自己的理解和疑惑，老师有空帮忙看下哦

Netty服务端会存在两个线程池NioEventLoopGroup，一个线程池主要用来处理客户端的连接，一般设置单线程NioEventLoop，在Linux中可能是EpollEventLoop，要是服务端监控多个端口可以设置多个线程，服务端接收到客户端的连接会创建Channel通道，Channel通道中会有收发缓存，服务端会定时监控Channel通道是否已经断开，在一定时间没有收到客户端的心跳包，把客户端的Channel从服务端移除，还可以设置服务端接收连接的队列，还有一个处理线程池NioEventLoopGroup，里面会有多个线程NioEventLoop，然后每个NioEventLoop都会有一个Selector，然后可以多个channel绑定到NioEventLoop的Selector中，即一个Channel只能被一个NioEventLoop处理，一个NioEventLoop可以处理多个Channel，即收到Channel数据，NioEventLoop执行Handler，包括解码、拆包等Handler，服务端返回响应消息对Channel进行编码等Handler。

尝试回答下课后习题接收数据这个流程Netty是一个NioEventLoop会有一个Selector，原先的Nio是只有一个Selector进行处理所有的连接收发事件，这样的话比如NioEventLoopGroup中有10个NioEventLoop，这样的话就有10个Selector，比如有10000读写请求，每个Selector就可以维持1000

2019-08-15

leslie

3

Java基础太差：几乎不懂；故而其实本课程学习让我觉得越多Java相关的非常吃力，希望老师后面的课程里面纯Java的东西能浅一点或者告知Java的理解大概要什么水平。

刘超老师的趣谈linux在跟着学、张磊的深入剖析Kubernetes目前学了一遍。我想从用这种方式去理解或解释不知道原理是否类似正确吧：希望老师提点或者下堂课时解答。

其实Netty基于NIO就像Kubernetes其实是基于Cgroup和Namespace一样：其实Netty是使用了NIO的Selecor去处理线程的异步机制，Netty在它的基础上去优化了其线程控制和连接管理并追

加了缓存管理，请老师指正；谢谢。
努力跟着学习，努力跟着做题；希望完课的时候能从另外一个高度/层次去理解和使用消息队列。

2019-08-15

作者回复

放心，我们这门课使用的任何语言都不会特别深入，更多的是讲实现原理，语言只是讲解和举例的载体。

2019-08-16

xhrg 2

1 本文按理说应该讲的是消息队列中网络的使用，但是实际作者讲的是java的网络基础编程和netty的入门介绍。

2 实际的网络传输，比如rocketmq在发送端也是异步的，而不是同步。

3 对于java的BIO,NIO，包括netty的入门demo，网上文章很多。如何在消息队列的中间件开发中，能高效使用netty，或者说NIO，也不是一件容易的事。

2019-08-15

川杰 1

老师，以下是我的理解：异步网络框架中，通过线程池处理接收消息的情况，和同步相比，好处在于，同步框架下，一个连接必须有两个线程（等数据的线程、处理数据的线程），当连接过多时会有大量频繁的上下文切换；而异步框架利用线程池接管了（等数据的线程）的作用，减少了上下文切换、线程的创建销毁的开销；
问题是：虽然线程池完成了数据接收的功能，但加入消息发送方发来了大量的消息，因为线程池的线程数量毕竟是有限的，此时是否就会出现消息不能及时转发给数据处理线程的情况呢？

2019-08-15

作者回复

你说的这个问题是有可能出现的。

2019-08-16

一步 1

希望老师多讲讲MQ 相关的,不要深入某一个语言某一个类库或者框架进行讲解的，有的同学语言不熟悉，代码也看不懂的，某一个类库或者框架实现的细节也不知道的

2019-08-15

作者回复

我们这节课和接下来的几节课讲的都是实现消息队列必须的一些技术，同学应该重点来理解我们讲解的原理，但是要真正掌握这些原理，还是需要依托与某一个语言或者框架来实际操作一下的。

2019-08-16

godtrue 0

课后思考及建议
没有对比就没有伤害，尤其对于学习，对比一下就知道那个课程优秀，那个更优秀。

首先，我觉得如何实现高性能的网络通信，是必须要讲的，这个原理是脱离具体语言的，和什么实现框架也没什么关系。
不过篇幅有限老师只能讲解一下她的精髓，如果想一点点弄明白，建议看看李林峰大哥的《netty权威指南(第二版)》她用了三章来讲解网络通信模型的演进。
另外，我同时在学kafka的专栏，我发现一个现象，两位老师都没有先将一个消息的全生命历程先细致的讲一下，学习过丁奇老师的MySQL，他上来就讲解了一下一个SQL语句是怎么执行的。我觉得很有整体感，知道整个过程之后其实下面再细致的讲解都是性能优化的事情啦！
我猜想其他各种系统，尤其是和数据打交道的都类似，只要一个完整的流程知道了，下面好多知识都是在为这个系统的性能、健壮性、高可用性、自身的其他特性在加强。
所以，我提过这样的问题，也建议先讲一下一条消息从发送到接收都经历了那些关键环节或组件，对一条消息的全生命历程有个整体的认识。然后再讲每个关键环节为什么这么实现，其性能最佳吞吐量最高。然后再讲各个组件是怎么紧密配合的，如果我知道一个软件是什么？又清楚他由什么组成？每一部分为什么如此设计？那些设计是通用的那些设计比较独特？我觉得我就理解了这个软件
老师的课程非常优秀，哈哈，我觉得听了我的建议还可以再优秀一点点

2019-08-23

作者回复

感谢你的建议！

2019-08-23

凌霄 0

jsf中netty使用一个线程组boss线程（一个端口一个boss线程）专门处理accept，使用一个线程组work专门处理io，work线程再调用业务线程。

2019-08-22

奎因 0

我是 Python 开发者，基于 Python 内置的 low level api 中的异步 io 对象编写了一个异步的 web socket

2019-08-19

作者回复

方便的话可以给大家分享一下代码哟

2019-08-19

Colin 0

我觉得这一章完全可以带过

2019-08-18

来碗绿豆汤 0

今天坐了8个小时火车，正好把课程看一遍。老师课程写的非常好，一口气读完毫无枯燥敢。回答一下今天的问题:同一个数据在不同操作系统中的表示形式不同，可能导致程序无法被读出;内

存里面的数据还会包含很多对象引用，类应用，这些存的都是内存地址，当你再次加载到内存时，这些引用都是失效的。

2019-08-17

jack 0

老师，有点晕，前边讲的completablefuture和nio之间的区别和联系是什么呢？好像两者并没有直接关系？

2019-08-17

作者回复

简单地说，异步的方法和类似CompletableFuture这样的异步框架，可以解决我们自己编写代码的异步化问题。

而NIO等异步网络通信框架，解决的是异步收发网络数据的问题。

2019-08-18

明日 0

没有使用过netty，但感觉这里的实现方式就像是用消息队列的方式进行异步处理。。。
使用固定数量的线程持有selector监听端口的数据，接收的数据被封装成channel并与MyHandler业务处理逻辑绑定，被监听线程发送到缓存队列中（看到addLast就感觉是这样的），然后由前面注册的线程池去异步的处理这些数据

2019-08-16

oscarwin 0

不小心发出去了，继续上一个回答。Netty使用了NIO的IO多路复用能力，采用线程池来增加对多核CPU的利用。在Linux编程里epoll加非阻塞IO，组成了传说中的reactor模式，那么Netty为每一个线程实现一个reactor，使得这个吞吐量非常强大，对应到C++的框架，就类似muduo网络库了。

2019-08-15

作者回复

其实这些语言中，它们的很多基础类库实现的原理都是一样的。

2019-08-16

oscarwin 0

不是很懂Java，强答一发，如有错误还望指正。我理解的异步框架应该是提供提供对客户端端连接，然后转发到业务服务器上，并通过回调的方式来响应这个异步的请求。实现整个完整的过程就是一个异步框架了。Java的NIO我不是很懂，但是我猜想NIO只是为Java实现了Linux下的IO多路复用能力，更准确的只是一种IO模型框架。

2019-08-15

作者回复

[illegible]

2019-08-16

许童童 0

Netty 它是如何用 NIO 来实现的呢？
不是写Java的，尝试来答一下，应该是类似Nginx，一个主线程，多个worker线程，用epoll管理多路复用套接字，主线程将连接通过负载均衡算法分配到worker线程。

2019-08-15

ly

0

netty用得很少，其原理不太清楚，但是用java的nio写过简单的非生产程序。我的大致猜测：他的serverbootstrap的bind就是调用serversocketchannel的bind并且accept

2019-08-15