Netty In Action中文版 - 第三章：Netty核心概念

目录(?) [-]

在这一章我们将讨论Netty的10个核心类，清楚了解他们的结构对使用Netty很有用。可能有一些不会再工作中用到，但是也有一些很常用也很核心，你会遇到。

- Bootstrap or ServerBootstrap
- EventLoop
- EventLoopGroup
- ChannelPipeline
- Channel
- Future or ChannelFuture
- ChannelInitializer
- ChannelHandler

本节的目的就是介绍以上这些概念，帮助你了解它们的用法。

3.1 Netty Crash Course

在我们开始之前，如果你了解Netty程序的一般结构和大致用法(客户端和服务器都有一个类似的结构)会更好。

一个Netty程序开始于Bootstrap类，Bootstrap类是Netty提供的一个可以通过简单配置来设置或"引导"程序的一个很重要的类。Netty中设计了Handlers来处理特定的"event"和设置Netty中的事件，从而来处理多个协议和数据。事件可以描述成一个非常通用的方法，因为你可以自定义一个handler,用来将Object转成byte[]或将byte[]转成Object；也可以定义个handler处理抛出的异常。

你会经常编写一个实现ChannelInboundHandler的类，ChannelInboundHandler是用来接收消息，当有消息过来时，你可以决定如何处理。当程序需要返回消息时可以在ChannelInboundHandler里write/flush数据。可以认为应用程序的业务逻辑都是在ChannelInboundHandler中来处理的，业务罗的生命周期在ChannelInboundHandler中。

Netty连接客户端端或绑定服务器需要知道如何发送或接收消息，这是通过不同类型的handlers来做的，多个Handlers是怎么配置的？Netty提供了ChannelInitializer类用来配置Handlers。ChannelInitializer是通过ChannelPipeline来添加ChannelHandler的，如发送和接收消息，这些Handlers将确定发的是什么消息。ChannelInitializer自身也是一个ChannelHandler，在添加完其他的handlers之后会自动从ChannelPipeline中删除自己。

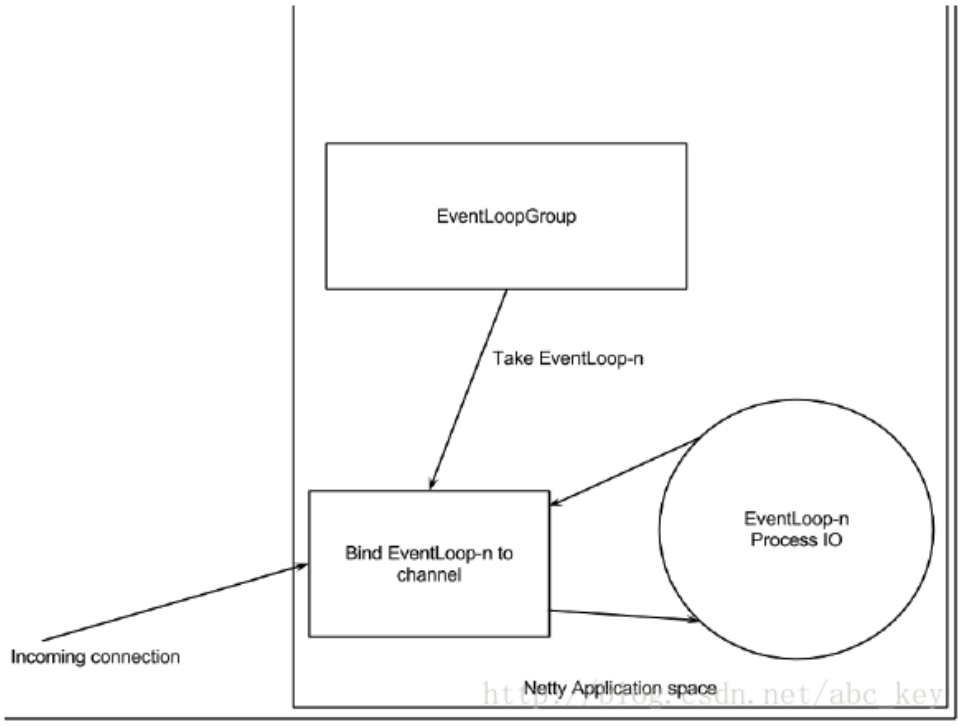
所有的Netty程序都是基于ChannelPipeline。ChannelPipeline和EventLoop和EventLoopGroup密切相关，因为它们三个都和事件处理相关，所以这就是为什么它们处理IO的工作由EventLoop管理的原因。

Netty中所有的IO操作都是异步执行的，例如你连接一个主机默认是异步完成的；写入/发送消息也是同样是异步。也就是说操作不会直接执行，而是会等一会执行，因为你不知道返回的操作结果是成功还是失败，但是需要有检查是否成功的方法或者是注册监听来通知；Netty使用Futures和ChannelFutures来达到这种目的。Future注册一个监听，当操作成功或失败时会通知。ChannelFuture封装的是一个操作的相关信息，操作被执行时会立刻返回ChannelFuture。

3.2 Channels,Events and Input/Output(IO)

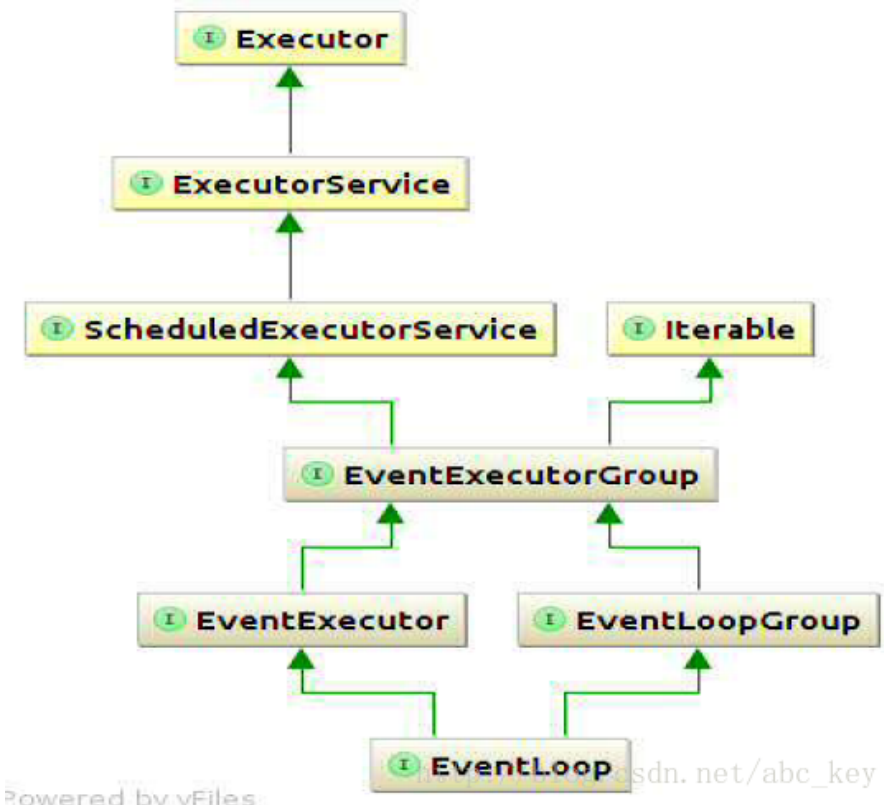
Netty是一个非阻塞、事件驱动的网络框架。Netty实际上是使用多线程处理IO事件，对于熟悉多线程编程的读者可能会需要同步代码。这样的方式不好，因为同步会影响程序的性能，Netty的设计保证程序处理事件不会有同步。

下图显示一个EventLoopGroup和一个Channel关联一个单一的EventLoop，Netty中的EventLoopGroup包含一个或多个EventLoop，而EventLoop就是一个Channel执行实际工作的线程。EventLoop总是绑定一个单一的线程，在其生命周期内不会改变。



当注册一个Channel后，Netty将这个Channel绑定到一个EventLoop，在Channel的生命周期内总是被绑定到一个EventLoop。在Netty IO操作中，你的程序不需要同步，因为一个指定通道的所有IO始终由同一个线程来执行。

为了帮助理解，下图显示了EventLoop和EventLoopGroup的关系：



EventLoop和EventLoopGroup的关联不是直观的，因为我们说过EventLoopGroup包含一个或多个EventLoop，但是上面的图显示EventLoop是一个EventLoopGroup，这意味着你可以只使用一个特定的EventLoop。

3.3 什么是Bootstrap?为什么使用它？

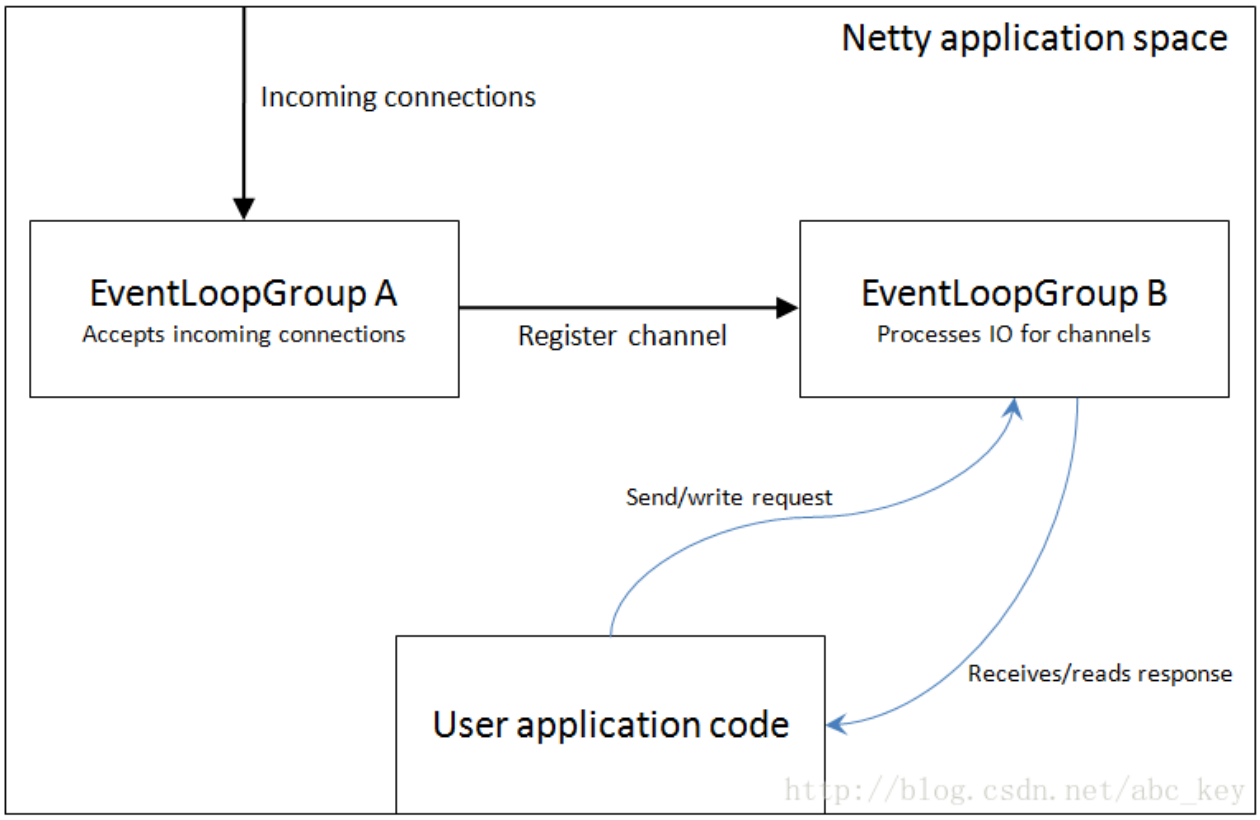
“引导”是Netty中配置程序的过程，当你需要连接客户端或服务器绑定指定端口时需要使用bootstrap。如前面所述，“引导”有两种类型，一种是用于客户端的Bootstrap(也适用于DatagramChannel)，一种是用于服务端的ServerBootstrap。不管程序使用哪种协议，无论是创建一个客户端还是服务器都需要使用“引导”。

两种bootsstraps之间有一些相似之处，其实他们有很多相似之处，也有一些不同。Bootstrap和ServerBootstrap之间的差异：

- Bootstrap用来连接远程主机，有1个EventLoopGroup
- ServerBootstrap用来绑定本地端口，有2个EventLoopGroup

事件组(Groups)，传输(transports)和处理程序(handlers)分别在本章后面讲述，我们在这里只讨论两种"引导"的差异(Bootstrap和ServerBootstrap)。第一个差异很明显，“ServerBootstrap”监听在服务器监听一个端口轮询客户端的“Bootstrap”或DatagramChannel是否连接服务器。通常需要调用“Bootstrap”类的connect()方法，但是也可以先调用bind()再调用connect()进行连接，之后使用的Channel包含在bind()返回的ChannelFuture中。

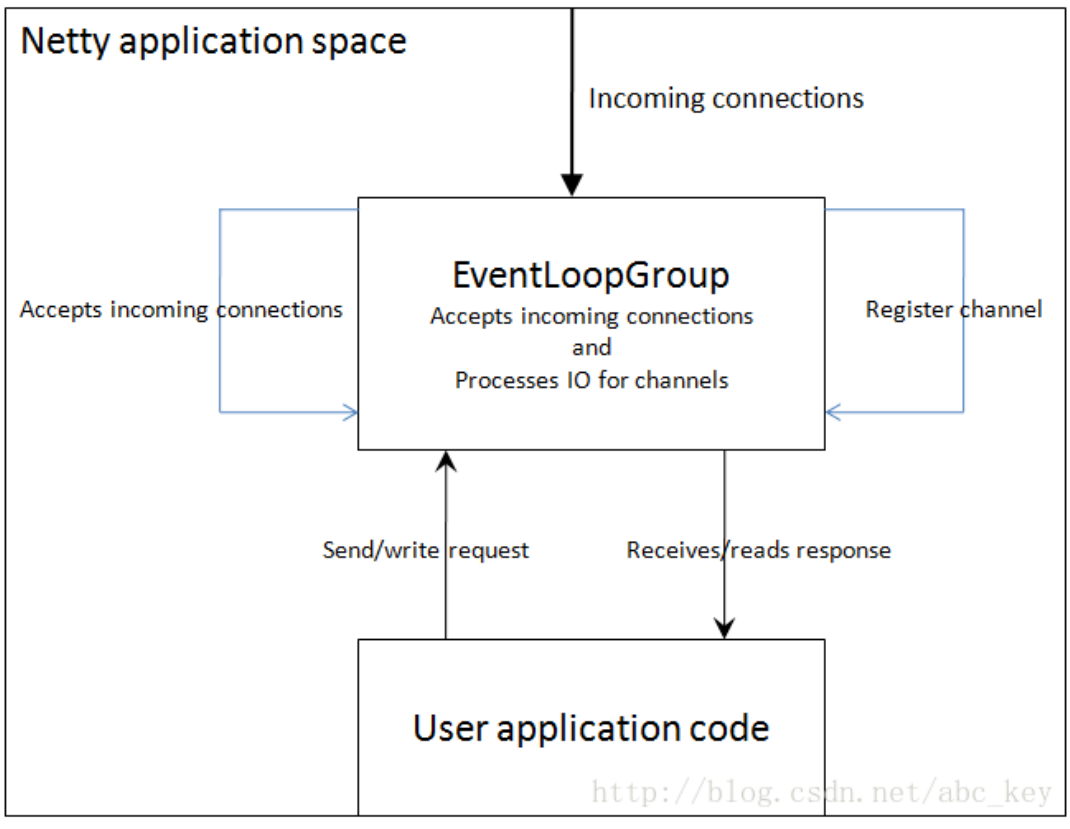
第二个差别也许是最重要的。客户端bootstraps/applications使用一个单例EventLoopGroup，而ServerBootstrap使用2个EventLoopGroup(实际上使用的是相同的实例)，它可能不是显而易见的，但是它是个好的方案。一个ServerBootstrap可以认为有2个channels组，第一组包含一个单例ServerChannel，代表持有一个绑定了本地端口的socket；第二组包含所有的Channel，代表服务器已接受了的连接。下图形象的描述了这种情况：



上图中，EventLoopGroup A唯一的目的就是接受连接然后交给EventLoopGroup B。Netty可以使用两个不同的Group，因为服务器程序需要接受很多客户端连接的情况下，一个EventLoopGroup将是程序性能的瓶颈，因为事件循环忙于处理连接请求，没有多余的资源和空闲来处理业务逻辑，最后的结果会是很多连接请求超时。若有两EventLoops， 即使在高负载下，所有的连接也都会被接受，因为EventLoops接受连接不会和哪些已经连接了的处理共享资源。

EventLoopGroup和EventLoop是什么关系？EventLoopGroup可以包含很多个EventLoop，每个Channel绑定一个EventLoop不会被改变，因为EventLoopGroup包含少量的EventLoop的Channels，很多Channel会共享同一个EventLoop。这意味着在一个Channel保持EventLoop繁忙会禁止其他Channel绑定到相同的EventLoop。我们可以理解为EventLoop是一个事件循环线程，而EventLoopGroup是一个事件循环集合。

如果你决定两次使用相同的EventLoopGroup实例配置Netty服务器，下图显示了它是如何改变的：



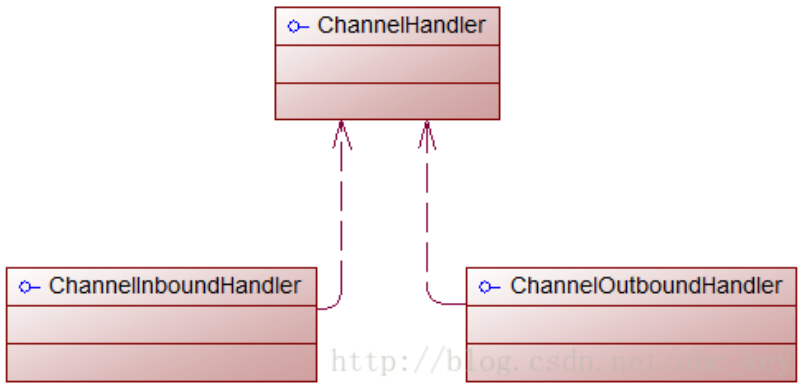
Netty允许处理IO和接受连接使用同一个EventLoopGroup，这在实际中适用于多种应用。上图显示了一个EventLoopGroup处理连接请求和IO操作。

下一节我们将介绍Netty是如何执行IO操作以及在什么时候执行。

3.4 Channel Handlers and Data Flow(通道处理和数据流)

本节我们一起来看看当你发送或接收数据时发生了什么？回想本章开始提到的handler概念。要明白Netty程序wirte或read时发生了什么，首先要对Handler是什么有一定的了解。Handlers自身依赖于ChannelPipeline来决定它们执行的顺序，因此不可能通过ChannelPipeline定义处理程序的某些方面,反过来不可能定义也不可能通过ChannelHandler定义ChannelPipeline的某些方面。没必要说我们必须定义一个自己和其他的规定。本节将介绍ChannelHandler和ChannelPipeline在某种程度上细微的依赖。

在很多地方，Netty的ChannelHandler是你的应用程序中处理最多的。即使你没有意思到这一点，若果你使用Netty应用将至少有一个ChannelHandler参与，换句话说，ChannelHandler对很多事情是关键的。那么ChannelHandler究竟是什么？给ChannelHandler一个定义不容易，我们可以理解为ChannelHandler是一段执行业务逻辑处理数据的代码，它们来来往往的通过ChannelPipeline。实际上，ChannelHandler是定义一个handler的父接口，ChannelInboundHandler和ChannelOutboundHandler都实现ChannelHandler接口，如下图：

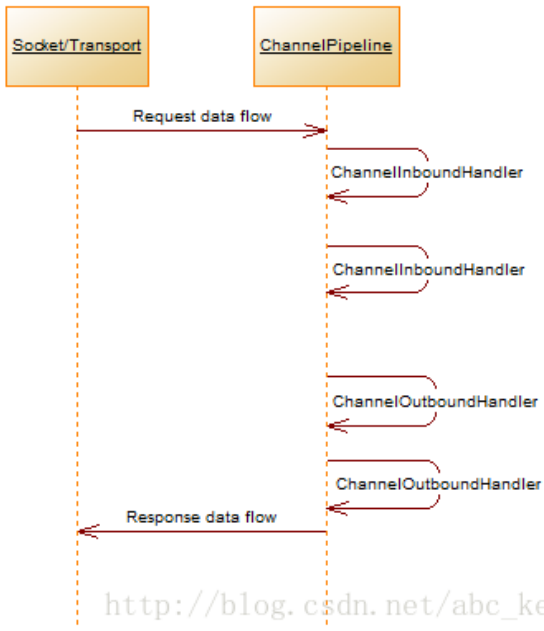


上图显示的比较容易，更重要的是ChannelHandler在数据流方面的应用，在这里讨论的例子只是一个简单的例子。ChannelHandler被应用在许多方面，在本书中会慢慢学习。

Netty中有两个方向的数据流，上图显示的入站(ChannelInboundHandler)和出站(ChannelOutboundHandler)之间有一个明显的区别：若数据是从用户应用程序到远程主机则是“出站(outbound)”，相反若数据时从远程主机到用户应用程序则是“入站(inbound)”。

为了使数据从一端到达另一端，一个或多个ChannelHandler将以某种方式操作数据。这些ChannelHandler会在程序的“引导”阶段被添加ChannelPipeline中，并且被添加的顺序将决定处理数据的顺序。ChannelPipeline的作用我们可以理解为用来管理ChannelHandler的一个容器，每个ChannelHandler处理各自的数据(例如入站数据只能由ChannelInboundHandler处理)，处理完成后将转换的数据放到ChannelPipeline中交给下一个ChannelHandler继续处理，直到最后一个ChannelHandler处理完成。

下图显示了ChannelPipeline的处理过程：



上图显示ChannelInboundHandler和ChannelOutboundHandler都要经过相同的ChannelPipeline。

在ChannelPipeline中，如果消息被读取或有任何其他的入站事件，消息将从ChannelPipeline的头部开始传递给第一个ChannelInboundHandler，这个ChannelInboundHandler可以处理该消息或将消息传递到下一个ChannelInboundHandler中，一旦在ChannelPipeline中没有剩余的ChannelInboundHandler后，ChannelPipeline就知道消息已被所有的饿Handler处理完成了。

反过来也是如此，任何出站事件或写入将从ChannelPipeline的尾部开始，并传递到最后一个ChannelOutboundHandler。ChannelOutboundHandler的作用和ChannelInboundHandler相同，它可以传递事件消息到下一个Handler或者自己处理消息。不同的是ChannelOutboundHandler是从ChannelPipeline的尾部开始，而ChannelInboundHandler是从ChannelPipeline的头部开始，当处理完第一个ChannelOutboundHandler处理完成后会出发一些操作，比如一个写操作。

一个事件能传递到下一个ChannelInboundHandler或上一个ChannelOutboundHandler，在ChannelPipeline中通过使用ChannelHandlerContext调用每一个方法。Netty提供了抽象的事件基类称为ChannelInboundHandlerAdapter和ChannelOutboundHandlerAdapter。每个都提供了在ChannelPipeline中通过调用相应的方法将事件传递给下一个Handler的方法的实现。我们能覆盖的方法就是我们需要做的处理。

可能有读者会奇怪，出站和入站的操作不同，能放在同一个ChannelPipeline工作？Netty的设计是很巧妙的，入站和出站Handler有不同的实现，Netty能跳过一个不能处理的操作，所以在出站事件的情况下，ChannelInboundHandler将被跳过，Netty知道每个handler都必须实现ChannelInboundHandler或ChannelOutboundHandler。

当一个ChannelHandler添加到ChannelPipeline中时获得一个ChannelHandlerContext。通常是安全的获得这个对象的引用，但是当一个数据报协议如UDP时这是不正确的，这个对象可以在之后用来获取底层通道，因为要用它来read/write消息，因此通道会保留。也就是说Netty中发送消息有两种方法：直接写入通道或写入ChannelHandlerContext对象。这两种方法的主要区别如下：

- 直接写入通道导致处理消息从ChannelPipeline的尾部开始
- 写入ChannelHandlerContext对象导致处理消息从ChannelPipeline的下一个handler开始

3.5 编码器、解码器和业务逻辑：细看Handlers

如前面所说，有很多不同类型的handlers，每个handler的依赖于它们的基类。Netty提供了一系列的“Adapter”类，这让事情变的很简单。每个handler负责转发时间到ChannelPipeline的下一个handler。在*Adapter类(和子类)中是自动完成的，因此我们只需要在感兴趣的*Adapter中重写方法。这些功能可以帮助我们非常简单的编码/解码消息。有几个适配器(adapter)允许自定义ChannelHandler，一般自定义ChannelHandler需要继承编码/解码适配器类中的一个。Netty有一下适配器：

- ChannelHandlerAdapter

- ChannelInboundHandlerAdapter
- ChannelOutboundHandlerAdapter

三个ChannelHandler涨，我们重点看看ecoders,decoders和SimpleChannelInboundHandler<I>，SimpleChannelInboundHandler<I>继承ChannelInboundHandlerAdapter。

3.5.1 Encoders(编码器), decoders(解码器)

发送或接收消息后，Netty必须将消息数据从一种形式转化为另一种。接收消息后，需要将消息从字节码转成Java对象(由某种解码器解码)；发送消息前，需要将Java对象转成字节(由某些类型的编码器进行编码)。这种转换一般发生在网络程序中，因为网络上只能传输字节数据。

有多种基础类型的编码器和解码器，要使用哪种取决于想实现的功能。要弄清楚某种类型的编解码器，从类名就可以看出，如“ByteToMessageDecoder”、“MessageToByteEncoder”，还有Google的协议“ProtobufEncoder”和“ProtobufDecoder”。

严格的说其他handlers可以做编码器和适配器，使用不同的Adapter classes取决你想要做什么。如果是解码器则有一个ChannelInboundHandlerAdapter或ChannelInboundHandler，所有的解码器都继承或实现它们。“channelRead”方法/事件被覆盖，这个方法从入站(inbound)通道读取每个消息。重写的channelRead方法将调用每个解码器的“decode”方法并通过ChannelHandlerContext.fireChannelRead(Object msg)传递给ChannelPipeline中的下一个ChannelInboundHandler。

类似入站消息，当你发送一个消息出去(出站)时，除编码器将消息转成字节码外还会转发到下一个ChannelOutboundHandler。

3.5.2 业务逻辑(Domain logic)

也许最常见的是应用程序处理接收到消息后进行解码，然后供相关业务逻辑模块使用。所以应用程序只需要扩展SimpleChannelInboundHandler<I>，也就是我们自定义一个继承SimpleChannelInboundHandler<I>的handler类，其中<I>是handler可以处理的消息类型。通过重写父类的方法可以获得一个ChannelHandlerContext的引用，它们接受一个ChannelHandlerContext的参数，你可以在class中当一个属性存储。

处理程序关注的主要方法是“channelRead0(ChannelHandlerContext ctx, I msg)”，每当Netty调用这个方法，对象“I”是消息，这里使用了Java的泛型设计，程序就能处理I。如何处理消息完全取决于程序的需要。在处理消息时有一点需要注意的，在Netty中事件处理IO一般有很多线程，程序中尽量不要阻塞IO线程，因为阻塞会降低程序的性能。

必须不阻塞IO线程意味着在ChannelHandler中使用阻塞操作会有问题。幸运的是Netty提供了解决方案，我们可以在添加ChannelHandler到ChannelPipeline中时指定一个EventExecutorGroup，EventExecutorGroup会获得一个EventExecutor，EventExecutor将执行ChannelHandler的所有方法。EventExecutor将使用不同的线程来执行和释放EventLoop。