# 【第03章-Netty的组件和设计】

从高层次的角度来看，Netty 解决了两个相应的关注领域，我们可将其大致标记为技术的和体系结构的。首先，它的基于Java NIO 的异步的和事件驱动的实现，保证了高负载下应用程序性能的最大化和可伸缩性。其次，Netty 也包含了一组设计模式，将应用程序逻辑从网络层解耦，简化了开发过程，同时也最大限度地提高了可测试性、模块化以及代码的可重用性。

## 3.1 Channel、EventLoop 和ChannelFuture

Channel、EventLoop 和ChannelFuture这些类合在一起，可以被认为是Netty 网络抽象的代表：

* Channel—Socket；
* EventLoop—控制流、多线程处理、并发；
* ChannelFuture—异步通知。

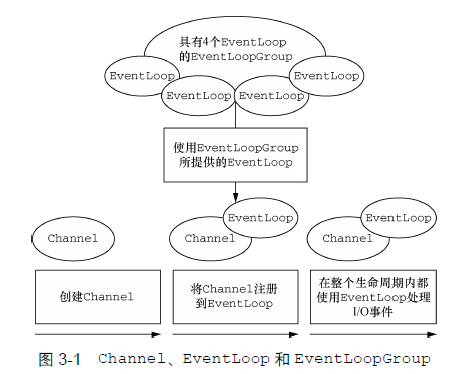
### 3.1.1 Channel 接口

基本的I/O 操作（bind()、connect()、read()和write()）依赖于底层网络传输所提供的原语。在基于Java 的网络编程中，其基本的构造是class Socket。Netty 的Channel 接口所提供的API，大大地降低了直接使用Socket 类的复杂性。此外，Channel 也是拥有许多预定义的、专门化实现的广泛类层次结构的根，下面是一个简短的部分清单：

* EmbeddedChannel；
* LocalServerChannel；
* NioDatagramChannel；
* NioSctpChannel；
* NioSocketChannel。

### 3.1.2 EventLoop 接口

EventLoop 定义了Netty 的核心抽象，用于处理连接的生命周期中所发生的事件。图3-1在高层次上说明了Channel、EventLoop、Thread 以及EventLoopGroup 之间的关系。



这些关系是：

* 一个EventLoopGroup 包含一个或者多个EventLoop；
* 一个EventLoop 在它的生命周期内只和一个Thread 绑定；
* 所有由EventLoop 处理的I/O 事件都将在它专有的Thread 上被处理；
* 一个Channel 在它的生命周期内只注册于一个EventLoop；
* 一个EventLoop 可能会被分配给一个或多个Channel。

注意，在这种设计中，一个给定Channel 的I/O 操作都是由相同的Thread 执行的，实际上消除了对于同步的需要。

### 3.1.3 ChannelFuture 接口

正如我们已经解释过的那样，Netty 中所有的I/O 操作都是异步的。因为一个操作可能不会立即返回，所以我们需要一种用于在之后的某个时间点确定其结果的方法。为此，Netty 提供了ChannelFuture 接口，其addListener()方法注册了一个ChannelFutureListener，以便在某个操作完成时（无论是否成功）得到通知。

关于ChannelFuture 的更多讨论

可以将ChannelFuture 看作是将来要执行的操作的结果的占位符。它究竟什么时候被执行则可能取决于若干的因素，因此不可能准确地预测，但是可以肯定的是它将会被执行。此外，所有属于同一个Channel 的操作都被保证其将以它们被调用的顺序被执行。

## 3.2 ChannelHandler 和ChannelPipeline

现在，我们将更加细致地看一看那些管理数据流以及执行应用程序处理逻辑的组件。

### 3.2.2 ChannelPipeline 接口

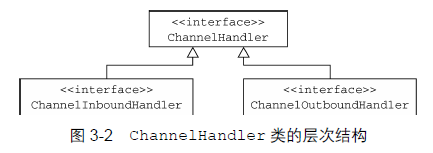
ChannelPipeline 提供了ChannelHandler 链的容器，并定义了用于在该链上传播入站和出站事件流的API。当Channel 被创建时，它会被自动地分配到它专属的ChannelPipeline。

ChannelHandler 安装到ChannelPipeline 中的过程如下所示：

* 一个ChannelInitializer的实现被注册到了ServerBootstrap中；
* 当ChannelInitializer.initChannel()方法被调用时，ChannelInitializer将在ChannelPipeline 中安装一组自定义的ChannelHandler；
* ChannelInitializer 将它自己从ChannelPipeline 中移除。

为了审查发送或者接收数据时将会发生什么，让我们来更加深入地研究ChannelPipeline和ChannelHandler 之间的共生关系吧。

ChannelHandler 是专为支持广泛的用途而设计的，可以将它看作是处理往来Channel-Pipeline 事件（包括数据）的任何代码的通用容器。图3-2 说明了这一点，其展示了从Channel-Handler 派生的ChannelInboundHandler 和ChannelOutboundHandler 接口。



使得事件流经ChannelPipeline 是ChannelHandler 的工作，它们是在应用程序的初始化或者引导阶段被安装的。这些对象接收事件、执行它们所实现的处理逻辑，并将数据传递给链中的下一个ChannelHandler。它们的执行顺序是由它们被添加的顺序所决定的。实际上，被我们称为ChannelPipeline 的是这些ChannelHandler 的编排顺序。

图3-3 说明了一个Netty 应用程序中入站和出站数据流之间的区别。从一个客户端应用程序的角度来看，如果事件的运动方向是从客户端到服务器端，那么我们称这些事件为出站的，反之则称为入站的。

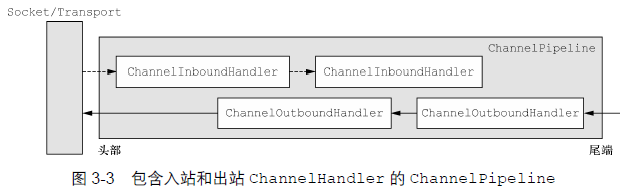


图3-3 也显示了入站和出站ChannelHandler 可以被安装到同一个ChannelPipeline中。如果一个消息或者任何其他的入站事件被读取，那么它会从ChannelPipeline 的头部开始流动，并被传递给第一个ChannelInboundHandler。这个ChannelHandler 不一定会实际地修改数据，具体取决于它的具体功能，在这之后，数据将会被传递给链中的下一个ChannelInboundHandler。最终，数据将会到达ChannelPipeline 的尾端，届时，所有处理就都结束了。

数据的出站运动（即正在被写的数据）在概念上也是一样的。在这种情况下，数据将从ChannelOutboundHandler 链的尾端开始流动，直到它到达链的头部为止。在这之后，出站数据将会到达网络传输层，这里显示为Socket。通常情况下，这将触发一个写操作。

关于入站和出站ChannelHandler 的更多讨论

通过使用作为参数传递到每个方法的ChannelHandlerContext，事件可以被传递给当前ChannelHandler 链中的下一个ChannelHandler。因为你有时会忽略那些不感兴趣的事件，所以Netty提供了抽象基类ChannelInboundHandlerAdapter 和ChannelOutboundHandlerAdapter。通过调用ChannelHandlerContext 上的对应方法，每个都提供了简单地将事件传递给下一个ChannelHandler的方法的实现。随后，你可以通过重写你所感兴趣的那些方法来扩展这些类。

鉴于出站操作和入站操作是不同的，你可能会想知道如果将两个类别的ChannelHandler都混合添加到同一个ChannelPipeline 中会发生什么。虽然ChannelInboundHandle 和ChannelOutboundHandle 都扩展自ChannelHandler，但是Netty 能区分ChannelInboundHandler实现和ChannelOutboundHandler 实现，并确保数据只会在具有相同定向类型的两个ChannelHandler 之间传递。

当ChannelHandler 被添加到ChannelPipeline 时，它将会被分配一个ChannelHandler-Context，其代表了ChannelHandler 和ChannelPipeline 之间的绑定。虽然这个对象可以被用于获取底层的Channel，但是它主要还是被用于写出站数据。

在Netty 中，有两种发送消息的方式。你可以直接写到Channel 中，也可以写到和Channel-Handler相关联的ChannelHandlerContext 对象中。前一种方式将会导致消息从Channel-Pipeline 的尾端开始流动，而后者将导致消息从ChannelPipeline 中的下一个Channel-Handler 开始流动。

### 3.2.3 更加深入地了解ChannelHandler

正如我们之前所说的，有许多不同类型的ChannelHandler，它们各自的功能主要取决于它们的超类。Netty 以适配器类的形式提供了大量默认的ChannelHandler 实现，其旨在简化应用程序处理逻辑的开发过程。你已经看到了，ChannelPipeline中的每个ChannelHandler将负责把事件转发到链中的下一个ChannelHandler。这些适配器类（及它们的子类）将自动执行这个操作，所以你可以只重写那些你想要特殊处理的方法和事件。

为什么需要适配器类

有一些适配器类可以将编写自定义的ChannelHandler 所需要的努力降到最低限度，因为它们提供了定义在对应接口中的所有方法的默认实现。

下面这些是编写自定义ChannelHandler 时经常会用到的适配器类：

* ChannelHandlerAdapter
* ChannelInboundHandlerAdapter
* ChannelOutboundHandlerAdapter
* ChannelDuplexHandler

### 3.2.4 编码器和解码器

当你通过Netty 发送或者接收一个消息的时候，就将会发生一次数据转换。入站消息会被解码；也就是说，从字节转换为另一种格式，通常是一个Java 对象。如果是出站消息，则会发生相反方向的转换：它将从它的当前格式被编码为字节。这两种方向的转换的原因很简单：网络数据总是一系列的字节。

对应于特定的需要，Netty 为编码器和解码器提供了不同类型的抽象类。例如，你的应用程序可能使用了一种中间格式，而不需要立即将消息转换成字节。你将仍然需要一个编码器，但是它将派生自一个不同的超类。为了确定合适的编码器类型，你可以应用一个简单的命名约定。

通常来说，这些基类的名称将类似于ByteToMessageDecoder 或MessageToByteEncoder。对于特殊的类型，你可能会发现类似于ProtobufEncoder 和ProtobufDecoder这样的名称——预置的用来支持Google 的Protocol Buffers。

严格地说，其他的处理器也可以完成编码器和解码器的功能。但是，正如有用来简化ChannelHandler 的创建的适配器类一样，所有由Netty 提供的编码器/解码器适配器类都实现了ChannelOutboundHandler 或者ChannelInboundHandler 接口。

你将会发现对于入站数据来说，channelRead 方法/事件已经被重写了。对于每个从入站Channel 读取的消息，这个方法都将会被调用。随后，它将调用由预置解码器所提供的decode()方法，并将已解码的字节转发给ChannelPipeline 中的下一个ChannelInboundHandler。

出站消息的模式是相反方向的：编码器将消息转换为字节，并将它们转发给下一个ChannelOutboundHandler

### 3.2.5 抽象类SimpleChannelInboundHandler

最常见的情况是，你的应用程序会利用一个ChannelHandler 来接收解码消息，并对该数据应用业务逻辑。要创建一个这样的ChannelHandler，你只需要扩展基类SimpleChannel-InboundHandler<T>，其中T 是你要处理的消息的Java 类型。在这个ChannelHandler 中，你将需要重写基类的一个或者多个方法，并且获取一个到ChannelHandlerContext 的引用，这个引用将作为输入参数传递给ChannelHandler 的所有方法。

在这种类型的ChannelHandler 中， 最重要的方法是channelRead0(Channel-HandlerContext,T)。除了要求不要阻塞当前的I/O 线程之外，其具体实现完全取决于你。

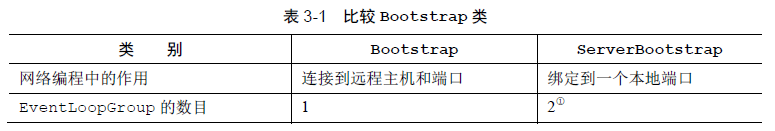
## 3.3 引导

Netty 的引导类为应用程序的网络层配置提供了容器，这涉及将一个进程绑定到某个指定的端口，或者将一个进程连接到另一个运行在某个指定主机的指定端口上的进程。

通常来说，我们把前面的用例称作引导一个服务器，后面的用例称作引导一个客户端。虽然这个术语简单方便，但是它略微掩盖了一个重要的事实，即“服务器”和“客户端”实际上表示了不同的网络行为；换句话说，是监听传入的连接还是建立到一个或者多个进程的连接。

**面向连接的协议 请记住，严格来说，“连接”这个术语仅适用于面向连接的协议，如TCP，其保证了两个连接端点之间消息的有序传递。**

因此，有两种类型的引导：一种用于客户端（简单地称为Bootstrap），而另一种（ServerBootstrap）用于服务器。无论你的应用程序使用哪种协议或者处理哪种类型的数据，唯一决定它使用哪种引导类的是它是作为一个客户端还是作为一个服务器。表3-1 比较了这两种类型的引导类。

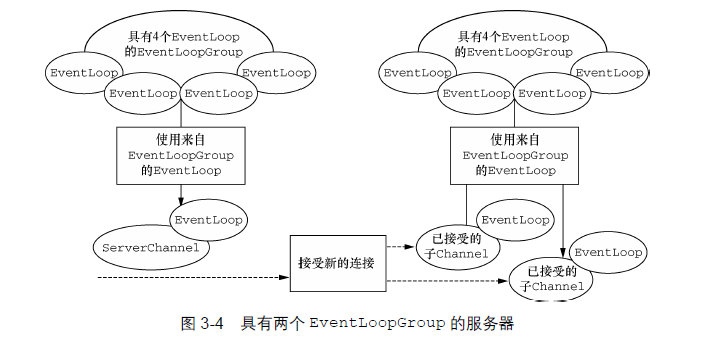


实际上，ServerBootstrap 类也可以只使用一个EventLoopGroup，此时其将在两个场景下共用同一个EventLoopGroup

这两种类型的引导类之间的第一个区别已经讨论过了：ServerBootstrap 将绑定到一个端口，因为服务器必须要监听连接，而Bootstrap 则是由想要连接到远程节点的客户端应用程序所使用的。

第二个区别可能更加明显。引导一个客户端只需要一个EventLoopGroup，但是一个ServerBootstrap 则需要两个（也可以是同一个实例）。为什么呢？

因为服务器需要两组不同的Channel。第一组将只包含一个ServerChannel，代表服务器自身的已绑定到某个本地端口的正在监听的套接字。而第二组将包含所有已创建的用来处理传入客户端连接（对于每个服务器已经接受的连接都有一个）的Channel。图3-4 说明了这个模型，并且展示了为何需要两个不同的EventLoopGroup。



与ServerChannel 相关联的EventLoopGroup 将分配一个负责为传入连接请求创建Channel 的EventLoop。一旦连接被接受，第二个EventLoopGroup 就会给它的Channel分配一个EventLoop。