# Netty是什么？为什么要用Netty？

## 为什么要用Netty

1、虽然JAVA NIO框架提供了 多路复用IO的支持，但是并没有提供上层“信息格式”的良好封装。例如前两者并没有提供针对 Protocol Buffer、JSON这些信息格式的封装，但是Netty框架提供了这些数据格式封装（基于责任链模式的编码和解码功能）；

2、直接使用NIO需要需要额外的技能，例如Java多线程，网络编程；

3、要编写一个可靠的、易维护的、高性能的NIO服务器应用。除了框架本身要兼容实现各类操作系统的实现外。更重要的是它应该还要处理很多上层特有服务，例如：客户端的权限、还有上面提到的信息格式封装、简单的数据读取，断连重连，半包读写，心跳等等，这些Netty框架都提供了响应的支持。

4、JAVA NIO框架存在一个poll/epoll bug：Selector doesn’t block on Selector.select(timeout)，不能block意味着CPU的使用率会变成100%（这是底层JNI的问题，上层要处理这个异常实际上也好办）。当然这个bug只有在Linux内核上才能重现。

这个问题在JDK 1.7版本中还没有被完全解决，但是Netty已经将这个bug进行了处理。

这个Bug与操作系统机制有关系的，JDK虽然仅仅是一个兼容各个操作系统平台的软件，但在JDK5和JDK6最初的版本中（严格意义上来将，JDK部分版本都是），这个问题并没有解决，而将这个帽子抛给了操作系统方，这也就是这个bug最终一直到2013年才最终修复的原因(JDK7和JDK8之间)。

## 为什么不用Netty5

1. netty5 中使用了 ForkJoinPool，增加了代码的复杂度，但是对性能的改善却不明显

2. 多个分支的代码同步工作量很大

3. 作者觉得当下还不到发布一个新版本的时候

4. 在发布版本之前，还有更多问题需要调查一下，比如是否应该废弃 exceptionCaught， 是否暴露EventExecutorChooser等等。

## 为什么Netty使用NIO而不是AIO？

Netty不看重Windows上的使用，在Linux系统上，AIO的底层实现仍使用EPOLL，没有很好实现AIO，因此在性能上没有明显的优势，而且被JDK封装了一层不容易深度优化。

AIO还有个缺点是接收数据需要预先分配缓存, 而不是NIO那种需要接收时才需要分配缓存, 所以对连接数量非常大但流量小的情况, 内存浪费很多。

据说Linux上AIO不够成熟，处理回调结果速度跟不上处理需求，有点像外卖员太少，顾客太多，供不应求，造成处理速度有瓶颈。

作者原话：

Not faster than NIO (epoll) on unix systems (which is true)

There is no daragram suppport

Unnecessary threading model (too much abstraction without usage)

# 第一个Netty程序

## Channel

Channel 是Java NIO 的一个基本构造。

它代表一个到实体（如一个硬件设备、一个文件、一个网络套接字或者一个能够执行一个或者多个不同的I/O操作的程序组件）的开放连接，如读操作和写操作

目前，可以把Channel 看作是传入（入站）或者传出（出站）数据的载体。因此，它可以被打开或者被关闭，连接或者断开连接。

## 回调和Future

一个回调其实就是一个方法，一个指向已经被提供给另外一个方法的方法的引用。这使得后者可以在适当的时候调用前者。回调在广泛的编程场景中都有应用，而且也是在操作完成后通知相关方最常见的方式之一。

Netty 在内部使用了回调来处理事件；当一个回调被触发时，相关的事件可以被一个interface-ChannelHandler 的实现处理。

Future 提供了另一种在操作完成时通知应用程序的方式。这个对象可以看作是一个异步操

作的结果的占位符；它将在未来的某个时刻完成，并提供对其结果的访问。

JDK 预置了interface java.util.concurrent.Future，但是其所提供的实现，只允许手动检查对应的操作是否已经完成，或者一直阻塞直到它完成。这是非常繁琐的，所以Netty提供了它自己的实现——ChannelFuture，用于在执行异步操作的时候使用。

ChannelFuture提供了几种额外的方法，这些方法使得我们能够注册一个或者多个ChannelFutureListener实例。监听器的回调方法operationComplete()，将会在对应的操作完成时被调用。然后监听器可以判断该操作是成功地完成了还是出错了。如果是后者，我们可以检索产生的Throwable。简而言之，由ChannelFutureListener提供的通知机制消除了手动检查对应的操作是否完成的必要。

每个Netty 的出站I/O 操作都将返回一个ChannelFuture。

## 事件和ChannelHandler

Netty 使用不同的事件来通知我们状态的改变或者是操作的状态。这使得我们能够基于已经发生的事件来触发适当的动作。

Netty事件是按照它们与入站或出站数据流的相关性进行分类的。

可能由入站数据或者相关的状态更改而触发的事件包括：

连接已被激活或者连接失活；数据读取；用户事件；错误事件。

出站事件是未来将会触发的某个动作的操作结果，这些动作包括：

打开或者关闭到远程节点的连接；将数据写到或者冲刷到套接字。

每个事件都可以被分发给ChannelHandler 类中的某个用户实现的方法。

可以认为每个Channel-Handler 的实例都类似于一种为了响应特定事件而被执行的回调。

Netty 提供了大量预定义的可以开箱即用的ChannelHandler 实现，包括用于各种协议（如HTTP 和SSL/TLS）的ChannelHandler。

# Netty组件再了解

## Channel、EventLoop 和ChannelFuture

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。

#### 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，被用于所有的I/O 操作。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。

每个Channel 都将会被分配一个ChannelPipeline 和ChannelConfig。ChannelConfig 包含了该Channel 的所有配置设置，并且支持热更新。由于Channel 是独一无二的，所以为了保证顺序将Channel 声明为java.lang.Comparable 的一个子接口。因此，如果两个不同的Channel 实例都返回了相同的散列码，那么AbstractChannel 中的compareTo()方法的实现将会抛出一个Error。

**Channel 的生命周期状态**

ChannelUnregistered Channel 已经被创建，但还未注册到EventLoop

ChannelRegistered Channel 已经被注册到了EventLoop

ChannelActive Channel 处于活动状态（已经连接到它的远程节点）。它现在可以接收和发送数据了

ChannelInactive Channel 没有连接到远程节点

**最重要Channel 的方法**

eventLoop 返回分配给Channel 的EventLoop

pipeline 返回分配给Channel 的ChannelPipeline

isActive 如果Channel 是活动的，则返回true。活动的意义可能依赖于底层的传输。例如，一个Socket 传输一旦连接到了远程节点便是活动的，而一个Datagram 传输一旦被打开便是活动的。

localAddress 返回本地的SokcetAddress

remoteAddress 返回远程的SocketAddress

write 将数据写到远程节点。这个数据将被传递给ChannelPipeline，并且排队直到它被冲刷

flush 将之前已写的数据冲刷到底层传输，如一个Socket

writeAndFlush 一个简便的方法，等同于调用write()并接着调用flush()

#### EventLoop和EventLoopGroup

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。io.netty.util.concurrent 包构建在JDK 的java.util.concurrent 包上。，一个EventLoop 将由一个永远都不会改变的Thread 驱动，同时任务（Runnable 或者Callable）可以直接提交给EventLoop 实现，以立即执行或者调度执行。

根据配置和可用核心的不同，可能会创建多个EventLoop 实例用以优化资源的使用，并且单个EventLoop 可能会被指派用于服务多个Channel。

Netty的EventLoop在继承了ScheduledExecutorService的同时，只定义了一个方法，parent()。在Netty 4 中，所有的I/O操作和事件都由已经被分配给了EventLoop的那个Thread来处理。

**任务调度**

偶尔，你将需要调度一个任务以便稍后（延迟）执行或者周期性地执行。例如，你可能想要注册一个在客户端已经连接了5 分钟之后触发的任务。一个常见的用例是，发送心跳消息到远程节点，以检查连接是否仍然还活着。如果没有响应，你便知道可以关闭该Channel 了。

**在内部，当提交任务到如果（**当前）调用线程正是支撑EventLoop 的线程，那么所提交的代码块将会被（直接）执行。否则，EventLoop 将调度该任务以便稍后执行，并将它放入到内部队列中。当EventLoop下次处理它的事件时，它会执行队列中的那些任务/事件。

**异步传输**

异步传输实现只使用了少量的EventLoop（以及和它们相关联的Thread），而且在当前的线程模型中，它们可能会被多个Channel 所共享。这使得可以通过尽可能少量的Thread 来支撑大量的Channel，而不是每个Channel 分配一个Thread。EventLoopGroup 负责为每个新创建的Channel 分配一个EventLoop。在当前实现中，使用顺序循环（round-robin）的方式进行分配以获取一个均衡的分布，并且相同的EventLoop可能会被分配给多个Channel。

一旦一个Channel 被分配给一个EventLoop，它将在它的整个生命周期中都使用这个EventLoop（以及相关联的Thread）。请牢记这一点，因为它可以使你从担忧你的Channel-

Handler 实现中的线程安全和同步问题中解脱出来。

另外，需要注意的是，EventLoop 的分配方式对ThreadLocal 的使用的影响。因为一个EventLoop 通常会被用于支撑多个Channel，所以对于所有相关联的Channel 来说，ThreadLocal 都将是一样的。这使得它对于实现状态追踪等功能来说是个糟糕的选择。然而，

在一些无状态的上下文中，它仍然可以被用于在多个Channel 之间共享一些重度的或者代价昂贵的对象，甚至是事件。

#### ChannelFuture 接口

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。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。

## ChannelHandler、ChannelPipeline和ChannelHandlerContext

#### ChannelHandler 接口

从应用程序开发人员的角度来看，Netty 的主要组件是ChannelHandler，它充当了所有处理入站和出站数据的应用程序逻辑的容器。ChannelHandler 的方法是由网络事件触发的。事实上，ChannelHandler 可专门用于几乎任何类型的动作，例如将数据从一种格式转换为另外一种格式，例如各种编解码，或者处理转换过程中所抛出的异常。

举例来说，ChannelInboundHandler 是一个你将会经常实现的子接口。这种类型的ChannelHandler 接收入站事件和数据，这些数据随后将会被你的应用程序的业务逻辑所处理。当你要给连接的客户端发送响应时，也可以从ChannelInboundHandler 冲刷数据。你的应用程序的业务逻辑通常驻留在一个或者多个ChannelInboundHandler 中。

这种类型的ChannelHandler 接收入站事件和数据，这些数据随后将会被你的应用程序的业务逻辑所处理。

ChannelHandler 的生命周期

下面列出了interface ChannelHandler 定义的生命周期操作，在ChannelHandler被添加到ChannelPipeline 中或者被从ChannelPipeline 中移除时会调用这些操作。这些方法中的每一个都接受一个ChannelHandlerContext 参数。

**handlerAdded** 当把ChannelHandler 添加到ChannelPipeline 中时被调用

**handlerRemoved** 当从ChannelPipeline 中移除ChannelHandler 时被调用

**exceptionCaught** 当处理过程中在ChannelPipeline 中有错误产生时被调用

Netty 定义了下面两个重要的ChannelHandler 子接口：

ChannelInboundHandler——处理入站数据以及各种状态变化；

ChannelOutboundHandler——处理出站数据并且允许拦截所有的操作。

##### ChannelInboundHandler 接口

下面列出了interface ChannelInboundHandler 的生命周期方法。这些方法将会在数据被接收时或者与其对应的Channel 状态发生改变时被调用。正如我们前面所提到的，这些方法和Channel 的生命周期密切相关。

**channelRegistered** 当Channel 已经注册到它的EventLoop 并且能够处理I/O 时被调用

**channelUnregistered** 当Channel 从它的EventLoop 注销并且无法处理任何I/O 时被调用

**channelActive** 当Channel 处于活动状态时被调用；Channel 已经连接/绑定并且已经就绪

**channelInactive** 当Channel 离开活动状态并且不再连接它的远程节点时被调用

**channelReadComplete** 当Channel上的一个读操作完成时被调用①

**channelRead** 当从Channel 读取数据时被调用

**ChannelWritability-Changed**

当Channel 的可写状态发生改变时被调用。用户可以确保写操作不会完成得太快（以避免发生OutOfMemoryError）或者可以在Channel 变为再次可写时恢复写入。可以通过调用Channel 的isWritable()方法来检测Channel 的可写性。与可写性相关的阈值可以通过Channel.config().setWriteHighWaterMark()和Channel.config().setWriteLowWater-Mark()方法来设置

**userEventTriggered** 当ChannelnboundHandler.fireUserEventTriggered()方法被调用时被调用。

当某个ChannelInboundHandler 的实现重写channelRead()方法时，它要负责显式地释放与池化的ByteBuf 实例相关的内存。Netty 为此提供了一个实用方法ReferenceCount-Util.release()

Netty 将使用WARN 级别的日志消息记录未释放的资源，使得可以非常简单地在代码中发现违规的实例。但是以这种方式管理资源可能很繁琐。一个更加简单的方式是使用Simple-

ChannelInboundHandler，SimpleChannelInboundHandler 会自动释放资源。

##### ChannelOutboundHandler 接口

出站操作和数据将由ChannelOutboundHandler 处理。它的方法将被Channel、Channel-

Pipeline 以及ChannelHandlerContext 调用。

所有由ChannelOutboundHandler 本身所定义的方法：

**bind(ChannelHandlerContext,SocketAddress,ChannelPromise)**

当请求将Channel 绑定到本地地址时被调用

**connect(ChannelHandlerContext,SocketAddress,SocketAddress,ChannelPromise)**

当请求将Channel 连接到远程节点时被调用

**disconnect(ChannelHandlerContext,ChannelPromise)**

当请求将Channel 从远程节点断开时被调用

**close(ChannelHandlerContext,ChannelPromise)** 当请求关闭Channel 时被调用

**deregister(ChannelHandlerContext,ChannelPromise)**

当请求将Channel 从它的EventLoop 注销时被调用

**read(ChannelHandlerContext)** 当请求从Channel 读取更多的数据时被调用

**flush(ChannelHandlerContext)** 当请求通过Channel 将入队数据冲刷到远程节点时被调用

**write(ChannelHandlerContext,Object,ChannelPromise)** 当请求通过Channel 将数据写到远程节点时被调用

##### ChannelHandler的适配器

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。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。

你可以使用ChannelInboundHandlerAdapter 和ChannelOutboundHandlerAdapter类作为自己的ChannelHandler 的起始点。这两个适配器分别提供了ChannelInboundHandler和ChannelOutboundHandler 的基本实现。通过扩展抽象类ChannelHandlerAdapter，它们获得了它们共同的超接口ChannelHandler 的方法。

ChannelHandlerAdapter 还提供了实用方法isSharable()。如果其对应的实现被标注为Sharable，那么这个方法将返回true，表示它可以被添加到多个ChannelPipeline。

#### ChannelPipeline 接口

当Channel 被创建时，它会被自动地分配到它专属的ChannelPipeline。每一个新创建的Channel 都将会被分配一个新的ChannelPipeline。这项关联是永久性的；Channel 既不能附加另外一个ChannelPipeline，也不能分离其当前的。在Netty 组件的生命周期中，这是一项固定的操作，不需要开发人员的任何干预。

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。

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，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。

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。

##### ChannelPipeline上的方法

**addFirst、addBefore、addAfter、addLast**

将一个ChannelHandler 添加到ChannelPipeline 中

**remove** 将一个ChannelHandler 从ChannelPipeline 中移除

**replace** 将ChannelPipeline 中的一个ChannelHandler 替换为另一个ChannelHandler

**get** 通过类型或者名称返回ChannelHandler

**context** 返回和ChannelHandler 绑定的ChannelHandlerContext

**names** 返回ChannelPipeline 中所有ChannelHandler 的名称

ChannelPipeline 的API 公开了用于调用入站和出站操作的附加方法。

##### ChannelHandlerContext

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。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。

ChannelHandlerContext 代表了ChannelHandler 和ChannelPipeline 之间的关联，每当有ChannelHandler 添加到ChannelPipeline 中时，都会创建ChannelHandler-Context。ChannelHandlerContext 的主要功能是管理它所关联的ChannelHandler 和在同一个ChannelPipeline 中的其他ChannelHandler 之间的交互。

ChannelHandlerContext 有很多的方法，其中一些方法也存在于Channel 和Channel-Pipeline 本身上，**但是有一点重要的不同。**如果调用Channel 或者ChannelPipeline 上的这些方法，它们将沿着整个ChannelPipeline 进行传播。而调用位于ChannelHandlerContext上的相同方法，则将从当前所关联的ChannelHandler 开始，并且只会传播给位于该ChannelPipeline 中的下一个（入站下一个，出站上一个）能够处理该事件的ChannelHandler。

**ChannelHandlerContext 的API**

**alloc** 返回和这个实例相关联的Channel 所配置的ByteBufAllocator

**bind** 绑定到给定的SocketAddress，并返回ChannelFuture

**channel** 返回绑定到这个实例的Channel

**close** 关闭Channel，并返回ChannelFuture

**connect** 连接给定的SocketAddress，并返回ChannelFuture

**deregister** 从之前分配的EventExecutor 注销，并返回ChannelFuture

**disconnect** 从远程节点断开，并返回ChannelFuture

**executor** 返回调度事件的EventExecutor

**fireChannelActive** 触发对下一个ChannelInboundHandler 上的channelActive()方法（已连接）的调用

**fireChannelInactive** 触发对下一个ChannelInboundHandler 上的channelInactive()方法（已关闭）的调用

**fireChannelRead** 触发对下一个ChannelInboundHandler 上的channelRead()方法（已接收的消息）的调用

**fireChannelReadComplete** 触发对下一个ChannelInboundHandler 上的channelReadComplete()方法的调用

**fireChannelRegistered** 触发对下一个ChannelInboundHandler 上的fireChannelRegistered()方法的调用

**fireChannelUnregistered** 触发对下一个ChannelInboundHandler 上的fireChannelUnregistered()方法的调用

**fireChannelWritabilityChanged** 触发对下一个ChannelInboundHandler 上的fireChannelWritabilityChanged()方法的调用

**fireExceptionCaught** 触发对下一个ChannelInboundHandler 上的fireExceptionCaught(Throwable)方法的调用

**fireUserEventTriggered** 触发对下一个ChannelInboundHandler 上的fireUserEventTriggered(Object evt)方法的调用

**handler** 返回绑定到这个实例的ChannelHandler

**isRemoved** 如果所关联的ChannelHandler 已经被从ChannelPipeline中移除则返回true

**name** 返回这个实例的唯一名称

**pipeline** 返回这个实例所关联的ChannelPipeline

**read** 将数据从Channel读取到第一个入站缓冲区；如果读取成功则触发一个channelRead事件，并（在最后一个消息被读取完成后）通知ChannelInboundHandler 的channelReadComplete

(ChannelHandlerContext)方法

当使用ChannelHandlerContext 的API 的时候，有以下两点：

* ChannelHandlerContext 和ChannelHandler 之间的关联（绑定）是永远不会改变的，所以缓存对它的引用是安全的；
* 如同我们在本节开头所解释的一样，相对于其他类的同名方法，ChannelHandler Context的方法将产生更短的事件流，应该尽可能地利用这个特性来获得最大的性能。

## 选择合适的内置通信传输模式

**NIO** io.netty.channel.socket.nio 使用java.nio.channels 包作为基础——基于选择器的方式

**Epoll** io.netty.channel.epoll 由 JNI 驱动的 epoll()和非阻塞 IO。这个传输支持只有在Linux 上可用的多种特性，如SO\_REUSEPORT，比NIO 传输更快，而且是完全非阻塞的。将NioEventLoopGroup替换为EpollEventLoopGroup ， 并且将NioServerSocketChannel.class 替换为EpollServerSocketChannel.class 即可。

**OIO** io.netty.channel.socket.oio 使用java.net 包作为基础——使用阻塞流

**Local** io.netty.channel.local 可以在VM 内部通过管道进行通信的本地传输

**Embedded** io.netty.channel.embedded Embedded 传输，允许使用ChannelHandler 而又不需要一个真正的基于网络的传输。在测试ChannelHandler 实现时非常有用

## 引导Bootstrap

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。

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。

比较**Bootstrap** 类

|  |  |  |
| --- | --- | --- |
|  | **Bootstrap** | **ServerBootstrap** |
| 网络编程中的作用 | 连接到远程主机和端口 | 绑定到一个本地端口 |
| EventLoopGroup 的数目 | 1 | 2 |

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）。

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。

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。

**在引导过程中添加多个ChannelHandler**

Netty 提供了一个特殊的ChannelInboundHandlerAdapter 子类：

public abstract class ChannelInitializer<C extends Channel> ext ends ChannelInboundHandlerAdapter

它定义了下面的方法：

protect ed abstract void initChannel(C ch) throws Exception;

这个方法提供了一种将多个ChannelHandler 添加到一个ChannelPipeline 中的简便方法。你只需要简单地向Bootstrap 或ServerBootstrap 的实例提供你的ChannelInitializer 实现即可，并且一旦Channel 被注册到了它的EventLoop 之后，就会调用你的initChannel()版本。在该方法返回之后，ChannelInitializer 的实例将会从ChannelPipeline 中移除它自己。

## ChannelOption

ChannelOption的各种属性在套接字选项中都有对应。

1、ChannelOption.SO\_BACKLOG

ChannelOption.SO\_BACKLOG对应的是tcp/ip协议listen函数中的backlog参数，函数listen(int socketfd,int backlog)用来初始化服务端可连接队列，

服务端处理客户端连接请求是顺序处理的，所以同一时间只能处理一个客户端连接，多个客户端来的时候，服务端将不能处理的客户端连接请求放在队列中等待处理，backlog参数指定了队列的大小

2、ChannelOption.SO\_REUSEADDR

ChanneOption.SO\_REUSEADDR对应于套接字选项中的SO\_REUSEADDR，这个参数表示允许重复使用本地地址和端口，

比如，某个服务器进程占用了TCP的80端口进行监听，此时再次监听该端口就会返回错误，使用该参数就可以解决问题，该参数允许共用该端口，这个在服务器程序中比较常使用，比如某个进程非正常退出，该程序占用的端口可能要被占用一段时间才能允许其他进程使用，而且程序死掉以后，内核一需要一定的时间才能够释放此端口，不设置SO\_REUSEADDR　　　　就无法正常使用该端口。

3、ChannelOption.SO\_KEEPALIVE

Channeloption.SO\_KEEPALIVE参数对应于套接字选项中的SO\_KEEPALIVE，该参数用于设置TCP连接，当设置该选项以后，连接会测试链接的状态，这个选项用于可能长时间没有数据交流的连接。当设置该选项以后，如果在两小时内没有数据的通信时，TCP会自动发送一个活动探测数据报文。

4、ChannelOption.SO\_SNDBUF和ChannelOption.SO\_RCVBUF

ChannelOption.SO\_SNDBUF参数对应于套接字选项中的SO\_SNDBUF，ChannelOption.SO\_RCVBUF参数对应于套接字选项中的SO\_RCVBUF这两个参数用于操作接收缓冲区和发送缓冲区

的大小，接收缓冲区用于保存网络协议站内收到的数据，直到应用程序读取成功，发送缓冲区用于保存发送数据，直到发送成功。

5、ChannelOption.SO\_LINGER

ChannelOption.SO\_LINGER参数对应于套接字选项中的SO\_LINGER,Linux内核默认的处理方式是当用户调用close（）方法的时候，函数返回，在可能的情况下，尽量发送数据，不一定保证会发生剩余的数据，造成了数据的不确定性，使用SO\_LINGER可以阻塞close()的调用时间，直到数据完全发送

6、ChannelOption.TCP\_NODELAY

ChannelOption.TCP\_NODELAY参数对应于套接字选项中的TCP\_NODELAY,该参数的使用与Nagle算法有关，Nagle算法是将小的数据包组装为更大的帧然后进行发送，而不是输入一次发送一次,因此在数据包不足的时候会等待其他数据的到了，组装成大的数据包进行发送，虽然该方式有效提高网络的有效负载，但是却造成了延时，而该参数的作用就是禁止使用Nagle算法，使用于小数据即时传输，于TCP\_NODELAY相对应的是TCP\_CORK，该选项是需要等到发送的数据量最大的时候，一次性发送数据，适用于文件传输。

## ByteBuf

ByteBuf API 的优点：

它可以被用户自定义的缓冲区类型扩展；

通过内置的复合缓冲区类型实现了透明的零拷贝；

容量可以按需增长（类似于JDK 的StringBuilder）；

在读和写这两种模式之间切换不需要调用ByteBuffer 的flip()方法；

读和写使用了不同的索引；

支持方法的链式调用；

支持引用计数；

支持池化。

ByteBuf 维护了两个不同的索引，名称以read 或者write 开头的ByteBuf 方法，将会推进其对应的索引，而名称以set 或者get 开头的操作则不会

如果打算读取字节直到readerIndex 达到和writerIndex 同样的值时会发生什么。在那时，你将会到达“可以读取的”数据的末尾。就如同试图读取超出数组末尾的数据一样，试图读取超出该点的数据将会触发一个IndexOutOf-BoundsException。

可以指定ByteBuf 的最大容量。试图移动写索引（即writerIndex）超过这个值将会触发一个异常。（默认的限制是Integer.MAX\_VALUE。）

#### 分配

**堆缓冲区**

最常用的ByteBuf 模式是将数据存储在JVM 的堆空间中。这种模式被称为支撑数组（backing array），它能在没有使用池化的情况下提供快速的分配和释放。可以由hasArray()来判断检查ByteBuf 是否由数组支撑。如果不是，则这是一个直接缓冲区

**直接缓冲区**

直接缓冲区是另外一种ByteBuf 模式。

直接缓冲区的主要缺点是，相对于基于堆的缓冲区，它们的分配和释放都较为昂贵。如果你正在处理遗留代码，你也可能会遇到另外一个缺点：因为数据不是在堆上，所以你不得不进行一次复制。

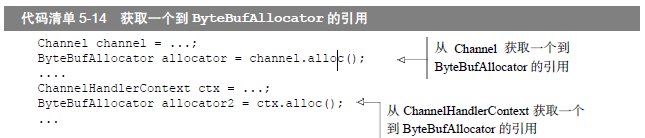
显然，与使用支撑数组相比，这涉及的工作更多。因此，如果事先知道容器中的数据将会被作为数组来访问，你可能更愿意使用堆内存。

**ByteBufAllocator**

Netty 通过interface ByteBufAllocator分配我们所描述过的任意类型的ByteBuf 实例。

|  |  |
| --- | --- |
| 名称 | 描述 |
| buffer() | 返回一个基于堆或者直接内存存储的ByteBuf |
| heapBuffer() | 返回一个基于堆内存存储的ByteBuf |
| directBuffer() | 返回一个基于直接内存存储的ByteBuf |
| compositeBuffer() | 返回一个可以通过添加最大到指定数目的基于堆的或者直接内存存储的缓冲区来扩展的CompositeByteBuf |
| ioBuffer() | 返回一个用于套接字的I/O 操作的ByteBuf，当所运行的环境具有sun.misc.Unsafe 支持时，返回基于直接内存存储的ByteBuf，否则返回基于堆内存存储的ByteBuf；当指定使用PreferHeapByteBufAllocator 时，则只会返回基于堆内存存储的ByteBuf。 |

可以通过Channel（每个都可以有一个不同的ByteBufAllocator 实例）或者绑定到ChannelHandler 的ChannelHandlerContext 获取一个到ByteBufAllocator 的引用。



Netty提供了两种ByteBufAllocator的实现：PooledByteBufAllocator和Unpooled-ByteBufAllocator。前者池化了ByteBuf的实例以提高性能并最大限度地减少内存碎片。后者的实现不池化ByteBuf实例，并且在每次它被调用时都会返回一个新的实例。

Netty4.1默认使用了PooledByteBufAllocator。

**Unpooled 缓冲区**

可能某些情况下，你未能获取一个到ByteBufAllocator 的引用。对于这种情况，Netty 提

供了一个简单的称为Unpooled 的工具类，它提供了静态的辅助方法来创建未池化的ByteBuf

实例。

buffer() 返回一个未池化的基于堆内存存储的ByteBuf

directBuffer()返回一个未池化的基于直接内存存储的ByteBuf

wrappedBuffer() 返回一个包装了给定数据的ByteBuf

copiedBuffer() 返回一个复制了给定数据的ByteBuf

Unpooled 类还使得ByteBuf 同样可用于那些并不需要Netty 的其他组件的非网络项目，使得其能得益于高性能的可扩展的缓冲区API。

#### 随机访问索引/顺序访问索引/读写操作

如同在普通的Java 字节数组中一样，ByteBuf 的索引是从零开始的：第一个字节的索引是0，最后一个字节的索引总是capacity() - 1。使用那些需要一个索引值参数(**随机访问**,也即是数组下标)的方法（的其中）之一来访问数据既不会改变readerIndex 也不会改变writerIndex。如果有需要，也可以通过调用readerIndex(index)或者writerIndex(index)来手动移动这两者。**顺序访问**通过索引访问

有两种类别的读/写操作：

get()和set()操作，从给定的索引开始，并且保持索引不变；get+数据字长（bool.byte,int,short,long,bytes）

read()和write()操作，从给定的索引开始，并且会根据已经访问过的字节数对索引进行调整。

**更多的操作**

isReadable() 如果至少有一个字节可供读取，则返回true

isWritable() 如果至少有一个字节可被写入，则返回true

readableBytes() 返回可被读取的字节数

writableBytes() 返回可被写入的字节数

capacity() 返回ByteBuf 可容纳的字节数。在此之后，它会尝试再次扩展直到达到maxCapacity()

maxCapacity() 返回ByteBuf 可以容纳的最大字节数

hasArray() 如果ByteBuf 由一个字节数组支撑，则返回true

array() 如果 ByteBuf 由一个字节数组支撑则返回该数组；否则，它将抛出一个UnsupportedOperationException 异常

**可丢弃字节**

为可丢弃字节的分段包含了已经被读过的字节。通过调用discardRead-Bytes()方法，可以丢弃它们并回收空间。这个分段的初始大小为0，存储在readerIndex 中，会随着read 操作的执行而增加（get\*操作不会移动readerIndex）。

缓冲区上调用discardReadBytes()方法后，可丢弃字节分段中的空间已经变为可写的了。频繁地调用discardReadBytes()方法以确保可写分段的最大化，但是请注意，这将极有可能会导致内存复制，因为可读字节必须被移动到缓冲区的开始位置。建议只在有真正需要的时候才这样做，例如，当内存非常宝贵的时候。

**可读字节**

ByteBuf 的可读字节分段存储了实际数据。新分配的、包装的或者复制的缓冲区的默认的readerIndex 值为0。

**可写字节**

可写字节分段是指一个拥有未定义内容的、写入就绪的内存区域。新分配的缓冲区的writerIndex 的默认值为0。任何名称以write 开头的操作都将从当前的writerIndex 处开始写数据，并将它增加已经写入的字节数。

**索引管理**

调用markReaderIndex()、markWriterIndex()、resetWriterIndex()和resetReaderIndex()来标记和重置ByteBuf 的readerIndex 和writerIndex。

也可以通过调用readerIndex(int)或者writerIndex(int)来将索引移动到指定位置。试图将任何一个索引设置到一个无效的位置都将导致一个IndexOutOfBoundsException。

可以通过调用clear()方法来将readerIndex 和writerIndex 都设置为0。注意，这并不会清除内存中的内容。

**查找操作**

在ByteBuf中有多种可以用来确定指定值的索引的方法。最简单的是使用indexOf()方法。

较复杂的查找可以通过调用forEach Byte()。

代码展示了一个查找回车符（\r）的例子。



**派生缓冲区**

派生缓冲区为ByteBuf 提供了以专门的方式来呈现其内容的视图。这类视图是通过以下方

法被创建的：

duplicate()；

slice()；

slice(int, int)；

Unpooled.unmodifiableBuffer(…)；

order(ByteOrder)；

readSlice(int)。

每个这些方法都将返回一个新的ByteBuf 实例，它具有自己的读索引、写索引和标记索引。其内部存储和JDK 的ByteBuffer 一样也是共享的。

**ByteBuf** 复制 如果需要一个现有缓冲区的真实副本，请使用copy()或者copy(int, int)方法。不同于派生缓冲区，由这个调用所返回的ByteBuf 拥有独立的数据副本。

**引用计数**

引用计数是一种通过在某个对象所持有的资源不再被其他对象引用时释放该对象所持有的资源来优化内存使用和性能的技术。Netty 在第4 版中为ByteBuf引入了引用计数技术， interface ReferenceCounted。

**工具类**

**ByteBufUtil** 提供了用于操作ByteBuf 的静态的辅助方法。因为这个API 是通用的，并且和池化无关，所以这些方法已然在分配类的外部实现。

这些静态方法中最有价值的可能就是hexdump()方法，它以十六进制的表示形式打印ByteBuf 的内容。这在各种情况下都很有用，例如，出于调试的目的记录ByteBuf 的内容。十六进制的表示通常会提供一个比字节值的直接表示形式更加有用的日志条目，此外，十六进制的版本还可以很容易地转换回实际的字节表示。

另一个有用的方法是boolean equals(ByteBuf, ByteBuf)，它被用来判断两个ByteBuf实例的相等性。

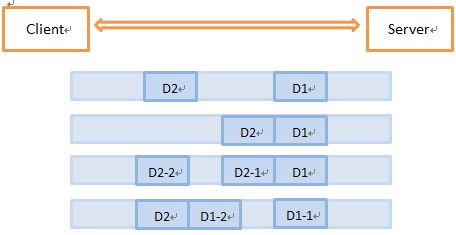
# 解决粘包/半包问题

## 回顾我们的的Hello,Netty

再讲解一遍，加上注释

# 什么是TCP粘包半包？

**改造程序，客户端发送100遍消息**



假设客户端分别发送了两个数据包D1和D2给服务端，由于服务端一次读取到的字节数是不确定的，故可能存在以下4种情况。

（1）服务端分两次读取到了两个独立的数据包，分别是D1和D2，没有粘包和拆包；

（2）服务端一次接收到了两个数据包，D1和D2粘合在一起，被称为TCP粘包；

（3）服务端分两次读取到了两个数据包，第一次读取到了完整的D1包和D2包的部分内容，第二次读取到了D2包的剩余内容，这被称为TCP拆包；

（4）服务端分两次读取到了两个数据包，第一次读取到了D1包的部分内容D1\_1，第二次读取到了D1包的剩余内容D1\_2和D2包的整包。

如果此时服务端TCP接收滑窗非常小，而数据包D1和D2比较大，很有可能会发生第五种可能，即服务端分多次才能将D1和D2包接收完全，期间发生多次拆包。

## TCP粘包/半包发生的原因

由于TCP协议本身的机制（面向连接的可靠地协议-三次握手机制）客户端与服务器会维持一个连接（Channel），数据在连接不断开的情况下，可以持续不断地将多个数据包发往服务器，但是如果发送的网络数据包太小，那么他本身会启用Nagle算法（可配置是否启用）对较小的数据包进行合并（基于此，TCP的网络延迟要UDP的高些）然后再发送（超时或者包大小足够）。那么这样的话，服务器在接收到消息（数据流）的时候就无法区分哪些数据包是客户端自己分开发送的，这样产生了粘包；服务器在接收到数据库后，放到缓冲区中，如果消息没有被及时从缓存区取走，下次在取数据的时候可能就会出现一次取出多个数据包的情况，造成粘包现象

UDP：本身作为无连接的不可靠的传输协议（适合频繁发送较小的数据包），他不会对数据包进行合并发送（也就没有Nagle算法之说了），他直接是一端发送什么数据，直接就发出去了，既然他不会对数据合并，每一个数据包都是完整的（数据+UDP头+IP头等等发一次数据封装一次）也就没有粘包一说了。

分包产生的原因就简单的多：可能是IP分片传输导致的，也可能是传输过程中丢失部分包导致出现的半包，还有可能就是一个包可能被分成了两次传输，在取数据的时候，先取到了一部分（还可能与接收的缓冲区大小有关系），总之就是一个数据包被分成了多次接收。

更具体的原因有三个，分别如下。

1. 应用程序写入数据的字节大小大于套接字发送缓冲区的大小

2. 进行MSS大小的TCP分段。MSS是最大报文段长度的缩写。MSS是TCP报文段中的数据字段的最大长度。数据字段加上TCP首部才等于整个的TCP报文段。所以MSS并不是TCP报文段的最大长度，而是：MSS=TCP报文段长度-TCP首部长度

3. 以太网的payload大于MTU进行IP分片。MTU指：一种通信协议的某一层上面所能通过的最[**大数据**](http://lib.csdn.net/base/hadoop)包大小。如果IP层有一个数据包要传，而且数据的长度比链路层的MTU大，那么IP层就会进行分片，把数据包分成托干片，让每一片都不超过MTU。注意，IP分片可以发生在原始发送端主机上，也可以发生在中间路由器上。

## 解决粘包半包问题

由于底层的TCP无法理解上层的业务数据，所以在底层是无法保证数据包不被拆分和重组的，这个问题只能通过上层的应用协议栈设计来解决，根据业界的主流协议的解决方案，可以归纳如下。

（1）在包尾增加分割符，比如回车换行符进行分割，例如FTP协议；linebase包和delimiter包下，分别使用 LineBasedFrameDecoder和DelimiterBasedFrameDecoder

（2）消息定长，例如每个报文的大小为固定长度200字节，如果不够，空位补空格；fixed包下，使用FixedLengthFrameDecoder

（3）将消息分为消息头和消息体，消息头中包含表示消息总长度（或者消息体长度）的字段，通常设计思路为消息头的第一个字段使用int32来表示消息的总长度，LengthFieldBasedFrameDecoder；。

# 编解码器框架

## 什么是编解码器

每个网络应用程序都必须定义如何解析在两个节点之间来回传输的原始字节，以及如何将其和目标应用程序的数据格式做相互转换。这种转换逻辑由编解码器处理，编解码器由编码器和解码器组成，它们每种都可以将字节流从一种格式转换为另一种格式。那么它们的区别是什么呢？

如果将消息看作是对于特定的应用程序具有具体含义的结构化的字节序列—它的数据。那么编码器是将消息转换为适合于传输的格式（最有可能的就是字节流）；而对应的解码器则是将网络字节流转换回应用程序的消息格式。因此，编码器操作出站数据，而解码器处理入站数据。我们前面所学的解决粘包半包的其实也是编解码器框架的一部分。

## 解码器

将字节解码为消息——ByteToMessageDecoder

将一种消息类型解码为另一种——MessageToMessageDecoder。

因为解码器是负责将入站数据从一种格式转换到另一种格式的，所以Netty 的解码器实现了ChannelInboundHandler。

什么时候会用到解码器呢？很简单：每当需要为ChannelPipeline 中的下一个Channel-InboundHandler 转换入站数据时会用到。此外，得益于ChannelPipeline 的设计，可以将多个解码器链接在一起，以实现任意复杂的转换逻辑。

#### 将字节解码为消息

**抽象类ByteToMessageDecoder**

将字节解码为消息（或者另一个字节序列）是一项如此常见的任务，以至于Netty 为它提供了一个抽象的基类：ByteToMessageDecoder。由于你不可能知道远程节点是否会一次性地发送一个完整的消息，所以这个类会对入站数据进行缓冲，直到它准备好处理。

它最重要方法

decode(ChannelHandlerContext ctx,ByteBuf in,List<Object> out)

这是你必须实现的唯一抽象方法。decode()方法被调用时将会传入一个包含了传入数据的ByteBuf，以及一个用来添加解码消息的List。对这个方法的调用将会重复进行，直到确定没有新的元素被添加到该List，或者该ByteBuf 中没有更多可读取的字节时为止。然后，如果该List 不为空，那么它的内容将会被传递给ChannelPipeline 中的下一个ChannelInboundHandler。

#### 将一种消息类型解码为另一种

在两个消息格式之间进行转换（例如，从String->Integer）

decode(ChannelHandlerContext ctx,I msg,List<Object> out)

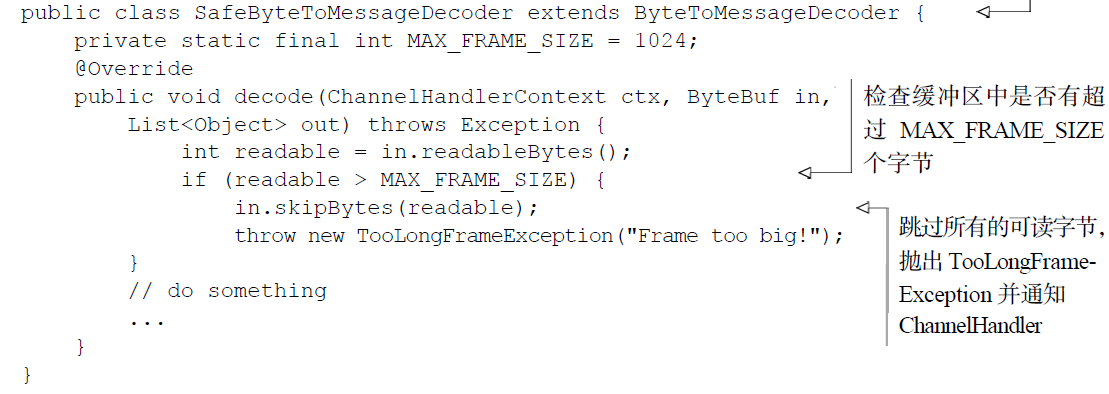
对于每个需要被解码为另一种格式的入站消息来说，该方法都将会被调用。解码消息随后会被传递给ChannelPipeline中的下一个ChannelInboundHandler

MessageToMessageDecoder<T>，T代表源数据的类型

#### TooLongFrameException

由于Netty 是一个异步框架，所以需要在字节可以解码之前在内存中缓冲它们。因此，不能让解码器缓冲大量的数据以至于耗尽可用的内存。为了解除这个常见的顾虑，Netty 提供了TooLongFrameException 类，其将由解码器在帧超出指定的大小限制时抛出。

为了避免这种情况，你可以设置一个最大字节数的阈值，如果超出该阈值，则会导致抛出一个TooLongFrameException（随后会被ChannelHandler.exceptionCaught()方法捕获）。然后，如何处理该异常则完全取决于该解码器的用户。某些协议（如HTTP）可能允许你返回一个特殊的响应。而在其他的情况下，唯一的选择可能就是关闭对应的连接。



## 编码器

解码器的功能正好相反。Netty 提供了一组类，用于帮助你编写具有以下功能的编码器：

将消息编码为字节；MessageToByteEncoder

将消息编码为消息：MessageToMessageEncoder<T>，T代表源数据的类型

#### 将消息编码为字节

encode(ChannelHandlerContext ctx,I msg,ByteBuf out)

encode()方法是你需要实现的唯一抽象方法。它被调用时将会传入要被该类编码为ByteBuf 的（类型为I 的）出站消息。该ByteBuf 随后将会被转发给ChannelPipeline中的下一个ChannelOutboundHandler

#### 将消息编码为消息

encode(ChannelHandlerContext ctx,I msg,List<Object> out)

这是你需要实现的唯一方法。每个通过write()方法写入的消息都将会被传递给encode()方法，以编码为一个或者多个出站消息。随后，这些出站消息将会被转发给ChannelPipeline中的下一个ChannelOutboundHandler

## 编解码器类

我们一直将解码器和编码器作为单独的实体讨论，但是你有时将会发现在同一个类中管理入站和出站数据和消息的转换是很有用的。Netty 的抽象编解码器类正好用于这个目的，因为它们每个都将捆绑一个解码器/编码器对，以处理我们一直在学习的这两种类型的操作。这些类同时实现了ChannelInboundHandler 和ChannelOutboundHandler 接口。

为什么我们并没有一直优先于单独的解码器和编码器使用这些复合类呢？因为通过尽可能地将这两种功能分开，最大化了代码的可重用性和可扩展性，这是Netty 设计的一个基本原则。

相关的类：

抽象类ByteToMessageCodec

抽象类MessageToMessageCodec

## Netty内置的编解码器和ChannelHandler

Netty 为许多通用协议提供了编解码器和处理器，几乎可以开箱即用，这减少了你在那些相当繁琐的事务上本来会花费的时间与精力。

#### 通过SSL/TLS 保护Netty 应用程序

SSL和TLS这样的安全协议，它们层叠在其他协议之上，用以实现数据安全。我们在访问安全网站时遇到过这些协议，但是它们也可用于其他不是基于HTTP的应用程序，如安全SMTP（SMTPS）邮件服务器甚至是关系型数据库系统。

为了支持SSL/TLS，Java 提供了javax.net.ssl 包，它的SSLContext 和SSLEngine类使得实现解密和加密相当简单直接。Netty 通过一个名为SslHandler 的ChannelHandler实现利用了这个API，其中SslHandler 在内部使用SSLEngine 来完成实际的工作。

Netty 还提供了使用OpenSSL 工具包（www.openssl.org）的SSLEngine 实现。这个OpenSsl-Engine 类提供了比JDK 提供的SSLEngine 实现更好的性能。

如果OpenSSL库可用，可以将Netty 应用程序（客户端和服务器）配置为默认使用OpenSslEngine。如果不可用，Netty 将会回退到JDK 实现。

在大多数情况下，SslHandler 将是ChannelPipeline 中的第一个ChannelHandler。

#### HTTP 系列

HTTP 是基于请求/响应模式的：客户端向服务器发送一个HTTP 请求，然后服务器将会返回一个HTTP 响应。Netty 提供了多种编码器和解码器以简化对这个协议的使用。

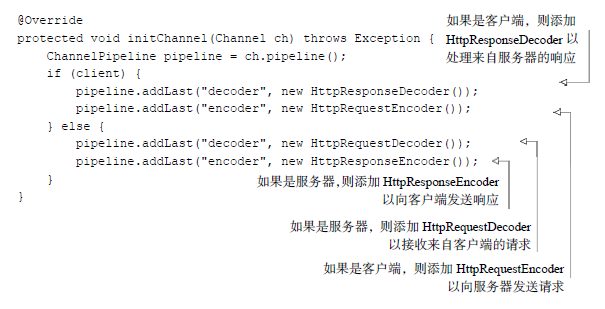
一个HTTP 请求/响应可能由多个数据部分组成，并且它总是以一个LastHttpContent 部分作为结束。FullHttpRequest 和FullHttpResponse 消息是特殊的子类型，分别代表了完整的请求和响应。所有类型的HTTP 消息（FullHttpRequest、LastHttpContent等等）都实现了HttpObject 接口。

HttpRequestEncoder 将HttpRequest、HttpContent 和LastHttpContent 消息编码为字节

HttpResponseEncoder 将HttpResponse、HttpContent 和LastHttpContent 消息编码为字节

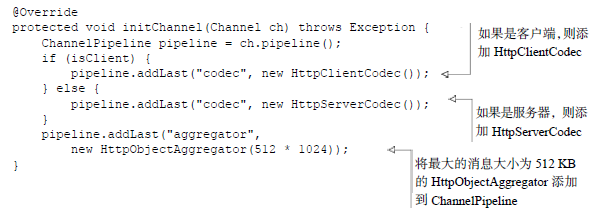
HttpRequestDecoder 将字节解码为HttpRequest、HttpContent 和LastHttpContent 消息

HttpResponseDecoder 将字节解码为HttpResponse、HttpContent 和LastHttpContent 消息



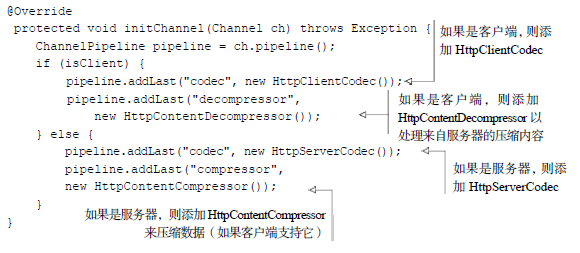
**聚合HTTP 消息**

由于HTTP 的请求和响应可能由许多部分组成，因此你需要聚合它们以形成完整的消息。为了消除这项繁琐的任务，Netty 提供了一个聚合器，它可以将多个消息部分合并为FullHttpRequest 或者FullHttpResponse 消息。通过这样的方式，你将总是看到完整的消息内容。



**HTTP 压缩**

当使用HTTP 时，建议开启压缩功能以尽可能多地减小传输数据的大小。虽然压缩会带来一些CPU 时钟周期上的开销，但是通常来说它都是一个好主意，特别是对于文本数据来说。Netty 为压缩和解压缩提供了ChannelHandler 实现，它们同时支持gzip 和deflate 编码。



**使用HTTPS**

启用HTTPS 只需要将SslHandler 添加到ChannelPipeline 的ChannelHandler 组合中。

#### WebSocket

后面的课会细讲，这里略过。

#### 空闲的连接和超时

检测空闲连接以及超时对于及时释放资源来说是至关重要的。由于这是一项常见的任务，Netty 特地为它提供了几个ChannelHandler 实现。

IdleStateHandler 当连接空闲时间太长时，将会触发一个IdleStateEvent 事件。然后，你可以通过在你的ChannelInboundHandler 中重写userEventTriggered()方法来处理该IdleStateEvent 事件。

ReadTimeoutHandler 如果在指定的时间间隔内没有收到任何的入站数据，则抛出一个Read-TimeoutException 并关闭对应的Channel。可以通过重写你的ChannelHandler 中的exceptionCaught()方法来检测该Read-TimeoutException。

WriteTimeoutHandler 如果在指定的时间间隔内没有任何出站数据写入，则抛出一个Write-TimeoutException 并关闭对应的Channel 。可以通过重写你的ChannelHandler 的exceptionCaught()方法检测该WriteTimeout-Exception。

后面的课会细讲如何使用。

# 序列化问题

Java序列化的目的主要有两个：

1.网络传输

2.对象持久化

当选行远程跨迸程服务调用时，需要把被传输的Java对象编码为字节数组或者ByteBuffer对象。而当远程服务读取到ByteBuffer对象或者字节数组时，需要将其解码为发送时的Java 对象。这被称为Java对象编解码技术。

Java序列化仅仅是Java编解码技术的一种，由于它的种种缺陷，衍生出了多种编解码技术和框架

## Java序列化的缺点

Java序列化从JDK1.1版本就已经提供，它不需要添加额外的类库，只需实现java.io.Serializable并生成序列ID即可，因此，它从诞生之初就得到了广泛的应用。

但是在远程服务调用（RPC）时，很少直接使用Java序列化进行消息的编解码和传输，这又是什么原因呢？下面通过分析.Tava序列化的缺点来找出答案。

1  无法跨语言

对于跨进程的服务调用，服务提供者可能会使用C十＋或者其他语言开发，当我们需要和异构语言进程交互时Java序列化就难以胜任。由于Java序列化技术是Java语言内部的私有协议，其他语言并不支持，对于用户来说它完全是黑盒。对于Java序列化后的字节数组，别的语言无法进行反序列化，这就严重阻碍了它的应用。

2  序列化后的码流太大

通过一个实例看下Java序列化后的字节数组大小。

3序列化性能太低

无论是序列化后的码流大小，还是序列化的性能，JDK默认的序列化机制表现得都很差。因此，我们边常不会选择Java序列化作为远程跨节点调用的编解码框架。

## 序列化 – 内置和第三方的MessagePack实战

#### 内置

Netty内置了对JBoss Marshalling和Protocol Buffers的支持

#### 集成第三方MessagePack实战（LengthFieldBasedFrame详解）

##### LengthFieldBasedFrame详解

maxFrameLength：表示的是包的最大长度，

lengthFieldOffset：指的是长度域的偏移量，表示跳过指定个数字节之后的才是长度域；

lengthFieldLength：记录该帧数据长度的字段本身的长度；

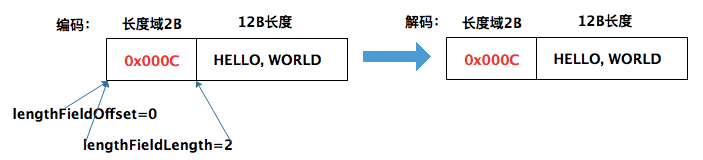
lengthAdjustment：长度的一个修正值，可正可负；

initialBytesToStrip：从数据帧中跳过的字节数，表示得到一个完整的数据包之后，忽略多少字节，开始读取实际我要的数据

failFast：如果为true，则表示读取到长度域，TA的值的超过maxFrameLength，就抛出一个 TooLongFrameException，而为false表示只有当真正读取完长度域的值表示的字节之后，才会抛出 TooLongFrameException，默认情况下设置为true，建议不要修改，否则可能会造成内存溢出。

**公式:** **实际数据包长度 = 长度域中记录的数据长度 + lengthFieldOffset + lengthFieldLength + lengthAdjustment**

**数据包大小: 14B = 长度域2B + "HELLO, WORLD"（单词HELLO+一个逗号+一个空格+单词WORLD）**

****

长度域的值为12B(0x000c)。希望解码后保持一样，根据上面的公式,参数应该为：

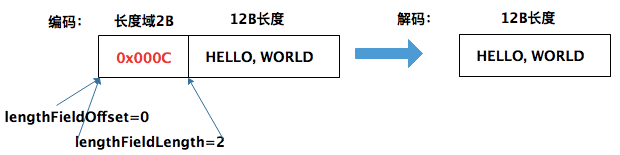
1. lengthFieldOffset = 0

2. lengthFieldLength = 2

3. lengthAdjustment = 0 = 数据包长度(14) - lengthFieldOffset - lengthFieldLength - 长度域的值(12)

4. initialBytesToStrip = 0 - 解码过程中，没有丢弃任何数据

**数据包大小: 14B = 长度域2B + "HELLO, WORLD"**

****

解码后，希望丢弃长度域2B字段，所以，只要initialBytesToStrip = 2即可。

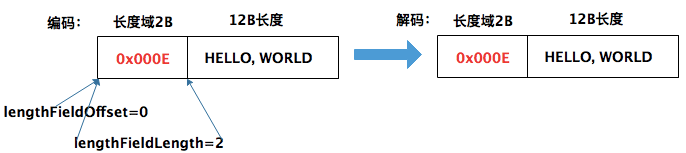
1. lengthFieldOffset = 0

2. lengthFieldLength = 2

3. lengthAdjustment = 0 = 数据包长度(14) - lengthFieldOffset - lengthFieldLength - 长度域的值(12)

4. initialBytesToStrip = 2 解码过程中，没有丢弃2个字节的数据

**数据包大小: 14B = 长度域2B + "HELLO, WORLD"。长度域的值为14(0x000E)**

****

长度域的值为14(0x000E)。希望解码后保持一样，根据上面的公式，参数应该为：

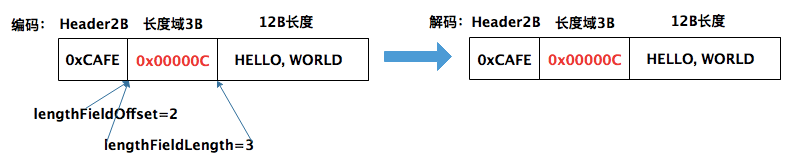
1. lengthFieldOffset = 0

2. lengthFieldLength = 2

3. lengthAdjustment = -2 = 数据包长度(14) - lengthFieldOffset - lengthFieldLength - 长度域的值(14)

4. initialBytesToStrip = 0 - 解码过程中，没有丢弃任何数据

**在长度域前添加2个字节的Header。长度域的值(0x00000C) = 12。总数据包长度: 17=Header(2B) + 长度域(3B) + "HELLO, WORLD"**

****

编码解码后，长度保持一致，所以initialBytesToStrip = 0。参数应该为:

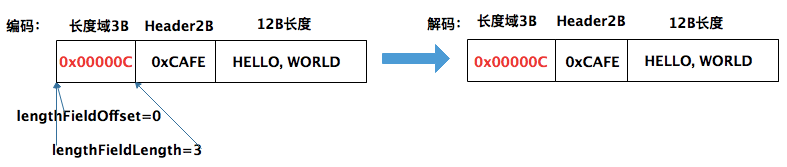
1. lengthFieldOffset = 2

2. lengthFieldLength = 3

3. lengthAdjustment = 0 = 数据包长度(17) - lengthFieldOffset(2) - lengthFieldLength(3) - 长度域的值(12)

4. initialBytesToStrip = 0 - 解码过程中，没有丢弃任何数据

**Header与长度域的位置换了。总数据包长度: 17=长度域(3B) + Header(2B) + "HELLO, WORLD"**

****

编码解码后，长度保持一致，所以initialBytesToStrip = 0。参数应该为:

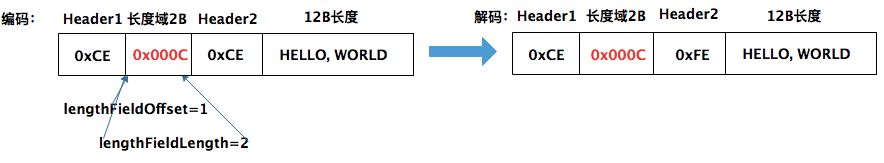
1. lengthFieldOffset = 0

2. lengthFieldLength = 3

3. lengthAdjustment = 2 = 数据包长度(17) - lengthFieldOffset(0) - lengthFieldLength(3) - 长度域的值(12)

4. initialBytesToStrip = 0 - 解码过程中，没有丢弃任何数据

**"HELLO, WORLD"域前有多个字段。总数据长度: 16 = HEADER1(1) + 长度域(2) + HEADER2(1) + "HELLO, WORLD"**

****

1. lengthFieldOffset = 1

2. lengthFieldLength = 2

3. lengthAdjustment = 1 = 数据包长度(16) - lengthFieldOffset(1) - lengthFieldLength(2) - 长度域的值(12)

4. initialBytesToStrip = 0 - 解码过程中，没有丢弃任何数据

##### MessagePack集成

看代码: cn.enjoyedu.ch02.serializable.msgpack包下

# 如何进行单元测试

一种特殊的Channel 实现——EmbeddedChannel，它是Netty 专门为改进针对ChannelHandler 的单元测试而提供的。

将入站数据或者出站数据写入到EmbeddedChannel 中，然后检查是否有任何东西到达了ChannelPipeline 的尾端。以这种方式，你便可以确定消息是否已经被编码或者被解码过了，以及是否触发了任何的ChannelHandler 动作。

**writeInbound(Object... msgs)**

将入站消息写到EmbeddedChannel 中。**如果**可以通过readInbound()方法从EmbeddedChannel 中读取数据，则返回true

**readInbound()**

从EmbeddedChannel 中读取一个入站消息。任何返回的东西都穿越了整个ChannelPipeline。**如果**没有任何可供读取的，则返回null

**writeOutbound(Object... msgs)**

将出站消息写到EmbeddedChannel中。**如果**现在可以通过readOutbound()方法从EmbeddedChannel 中读取到什么东西，则返回true

**readOutbound()**

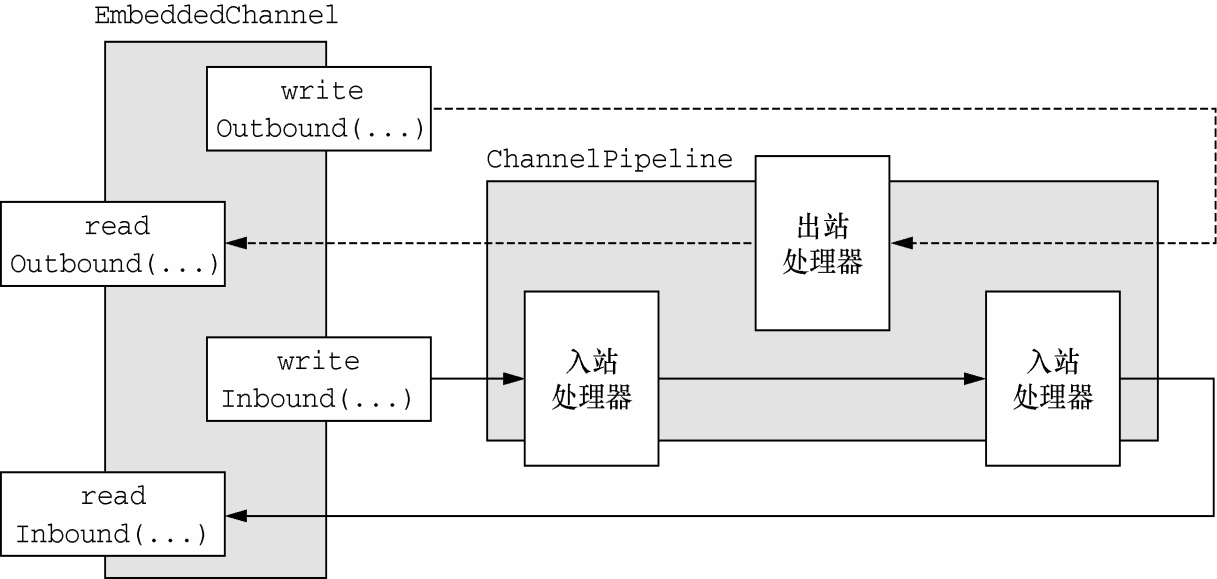
从EmbeddedChannel 中读取一个出站消息。任何返回的东西都穿越了整个ChannelPipeline。如果没有任何可供读取的，则返回null

**finish()** 将EmbeddedChannel 标记为完成，并且如果有可被读取的入站数据或者出站数据，则返回true。这个方法还将会调用EmbeddedChannel 上的close()方法。

入站数据由ChannelInboundHandler 处理，代表从远程节点读取的数据。出站数据由ChannelOutboundHandler 处理，代表将要写到远程节点的数据。

使用writeOutbound()方法将消息写到Channel 中，并通过ChannelPipeline 沿着出站的方向传递。随后，你可以使用readOutbound()方法来读取已被处理过的消息，以确定结果是否和预期一样。 类似地，对于入站数据，你需要使用writeInbound()和readInbound()方法。

在每种情况下，消息都将会传递过ChannelPipeline，并且被相关的ChannelInboundHandler 或者ChannelOutboundHandler 处理。

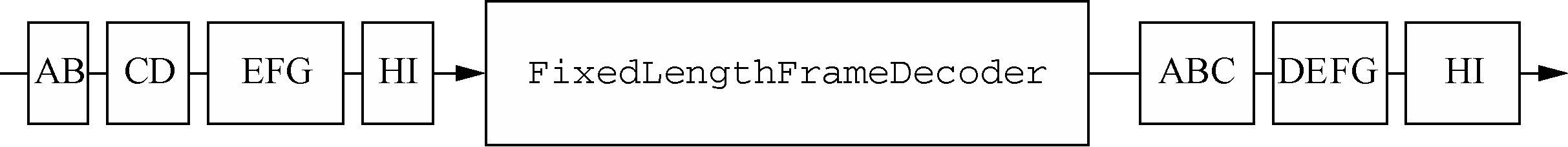


## 测试入站消息

展示了一个简单的ByteToMessageDecoder 实现。给定足够的数据，这个实现将产生固定大小的帧。如果没有足够的数据可供读取，它将等待下一个数据块的到来，并将再次检查是否能够产生一个新的帧。

这个特定的解码器将产生固定为3 字节大小的帧。因此，它可能会需要多个事件来提供足够的字节数以产生一个帧。

该testFramesDecoded()方法验证了：一个包含9 个可读字节的ByteBuf 被解码为3个ByteBuf，每个都包含了3 字节。需要注意的是，仅通过一次对writeInbound()方法的调用，ByteBuf 是如何被填充了9 个可读字节的。在此之后，通过执行finish()方法，将EmbeddedChannel 标记为了已完成状态。最后，通过调用readInbound()方法，从Embedded-Channel 中正好读取了3 个帧和一个null。



testFramesDecoded2()方法也是类似的，只有一处不同：入站ByteBuf 是通过两个步骤写入的。当writeInbound(input.readBytes(2))被调用时，返回了false。为什么呢？正如同表9-1 中所描述的，如果对readInbound()的后续调用将会返回数据，那么write-Inbound()方法将会返回true。但是只有当有3 个或者更多的字节可供读取时，FixedLength-FrameDecoder 才会产生输出。该测试剩下的部分和testFramesDecoded()是相同的。

## 测试出站消息

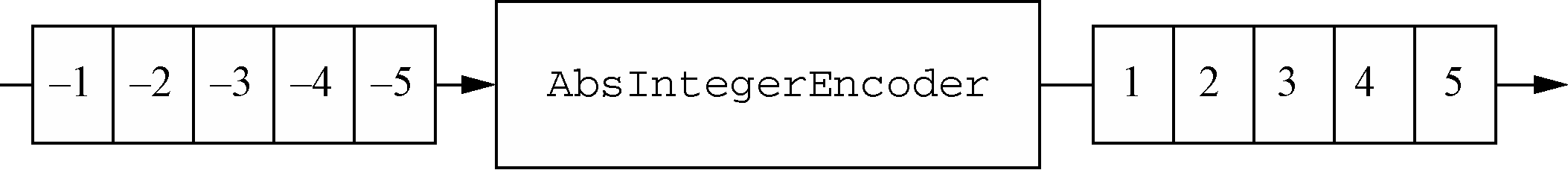
在测试的处理器—AbsIntegerEncoder，它是Netty 的MessageToMessageEncoder 的一个特殊化的实现，用于将负值整数转换为绝对值。

该示例将会按照下列方式工作：

持有AbsIntegerEncoder 的EmbeddedChannel 将会以4 字节的负整数的形式写出站数据；

编码器将从传入的ByteBuf 中读取每个负整数，并将会调用Math.abs()方法来获取其绝对值；

编码器将会把每个负整数的绝对值写到ChannelPipeline 中。



## 测试异常处理

应用程序通常需要执行比转换数据更加复杂的任务。例如，你可能需要处理格式不正确的输入或者过量的数据。在下一个示例中，如果所读取的字节数超出了某个特定的限制，我们将会抛出一个TooLongFrameException。

这是一种经常用来防范资源被耗尽的方法。设定最大的帧大小已经被设置为3 字节。如果一个帧的大小超出了该限制，那么程序将会丢弃它的字节，并抛出一个TooLongFrameException。位于ChannelPipeline 中的其他ChannelHandler 可以选择在exceptionCaught()方法中处理该异常或者忽略它。

