Netty 使用和常用组件

简述

本次课程以 Netty 4.1.42.Final 版本进行讲解

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId
    <version>4.1.42.Final </version>
    <scope>compile</scope>
</dependency>
```

Netty 的优势

1、API 使用简单，开发门槛低；

2、功能强大，预置了多种编解码功能，支持多种主流协议；

3、定制能力强，可以通过 ChannelHandler 对通信框架进行灵活地扩展；

4、性能高，通过与其他业界主流的 NIO 框架对比，Netty 的综合性能最优；

5、成熟、稳定，Netty 修复了已经发现的所有 JDK NIO BUG，业务开发人员不需要再为 NIO 的 BUG 而烦恼；

6、社区活跃，版本迭代周期短，发现的 BUG 可以被及时修复，同时，更多的新功能会加入；

7、经历了大规模的商业应用考验，质量得到验证。

为什么不用 Netty5

Netty5 已经停止开发了。

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。

而且 Linux 上 AIO 不够成熟，处理回调结果速度跟不上处理需求。

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)

为什么不用 Mina

简单来说，Mina 几乎不再更新了，Netty 本来就是因为 Mina 不够好所以开发出来的。

第一个 Netty 程序

Bootstrap、EventLoop(Group) 、Channel

Bootstrap 是 Netty 框架的启动类和主入口类，分为客户端类 Bootstrap 和服务器类 ServerBootstrap 两种。

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。

EventLoop 暂时可以看成一个线程、EventLoopGroup 自然就可以看成线程组。

事件和 ChannelHandler、ChannelPipeline

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。

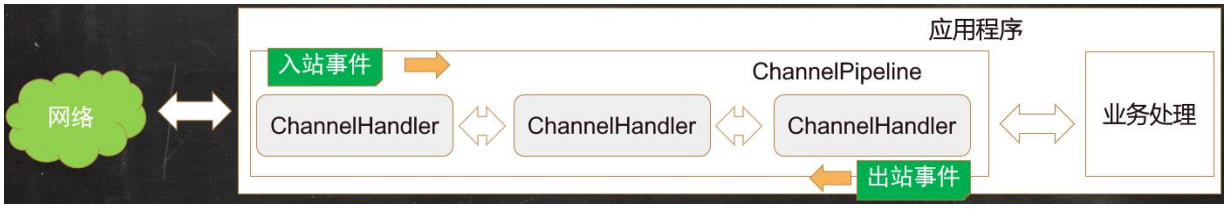
可能由入站数据或者相关的状态更改而触发的事件包括： 连接已被激活或者连接失活；数据读取；用户事件；错误事件。

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，既然事件分为入站和出站，用来处理事件的 ChannelHandler 也被分为可以处理入站事件的 Handler 和出站事件的 Handler，当然有些 Handler 既可以处理入站也可以处理出站。

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。

基于 Netty 的网络应用程序中根据业务需求会使用 Netty 已经提供的 ChannelHandler 或者自行开发 ChannelHandler，这些 ChannelHandler 都放在 ChannelPipeline 中统一管理，事件就会在 ChannelPipeline 中流动，并被其中一个或者多个 ChannelHandler 处理。



ChannelFuture

Netty 中所有的 I/O 操作都是异步的，我们知道“异步的意思就是不需要主动等待结果的返回，而是通过其他手段比如，状态通知，回调函数等”，那就是说至少我们需要一种获得异步执行结果的手段。

JDK 预置了 interface java.util.concurrent.Future，Future 提供了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。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。

一般来说，每个 Netty 的出站 I/O 操作都将返回一个 ChannelFuture。

Hello，Netty！第一个 Netty 程序

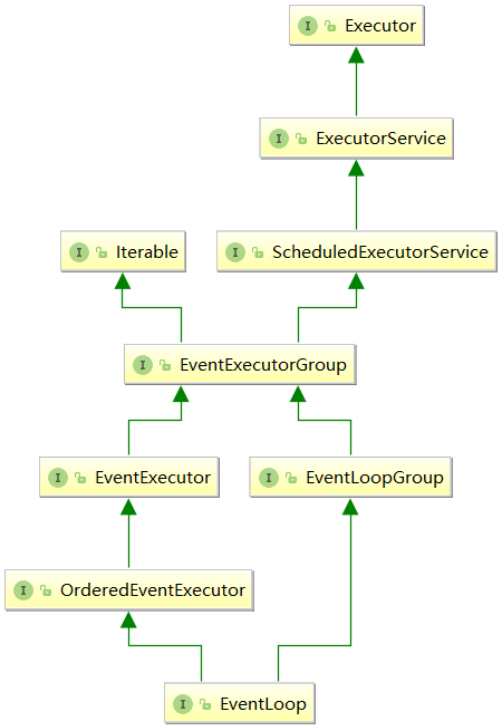
参见模块 netty-basic 中包 cn.tuling.nettybasic.echo

Netty 组件再了解

EventLoop 和 EventLoopGroup

回想一下我们在 NIO 中是如何处理我们关心的事件的？在一个 while 循环中 select 出事件，然后依次处理每种事件。我们可以把它称为事件循环，这就是 EventLoop。interface io.netty.channel. EventLoop 定义了 Netty 的核心抽象，用于处理网络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。而 io.netty.channel 包中的类，为了与 Channel 的事件进行交互，扩展了这些接口/类。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。

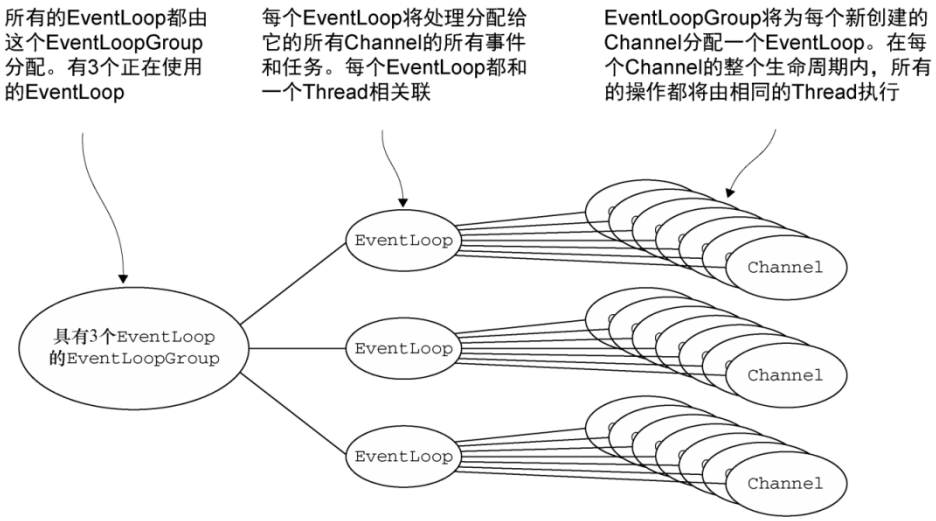


线程的分配

服务于 Channel 的 I/O 和事件的 EventLoop 包含在 EventLoopGroup 中。

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）。

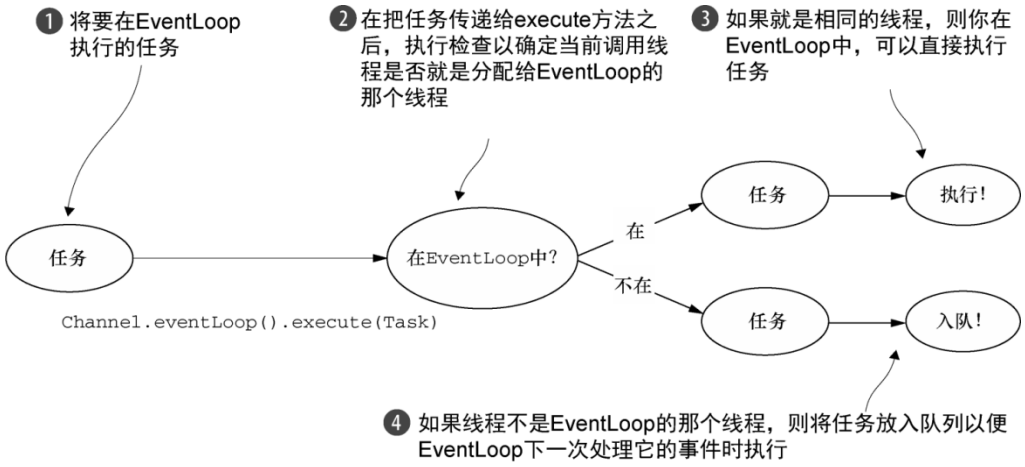


需要注意，EventLoop 的分配方式对 ThreadLocal 的使用的影响。因为一个 EventLoop 通常会被用于支撑多个 Channel，所以对于所有相关联的 Channel 来说，ThreadLocal 都将是一样的。这使得它对于实现状态追踪等功能来说是个糟糕的选择。然而，在一些无状态的上

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。

线程管理

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。



Channel、EventLoop(Group)和 ChannelFuture

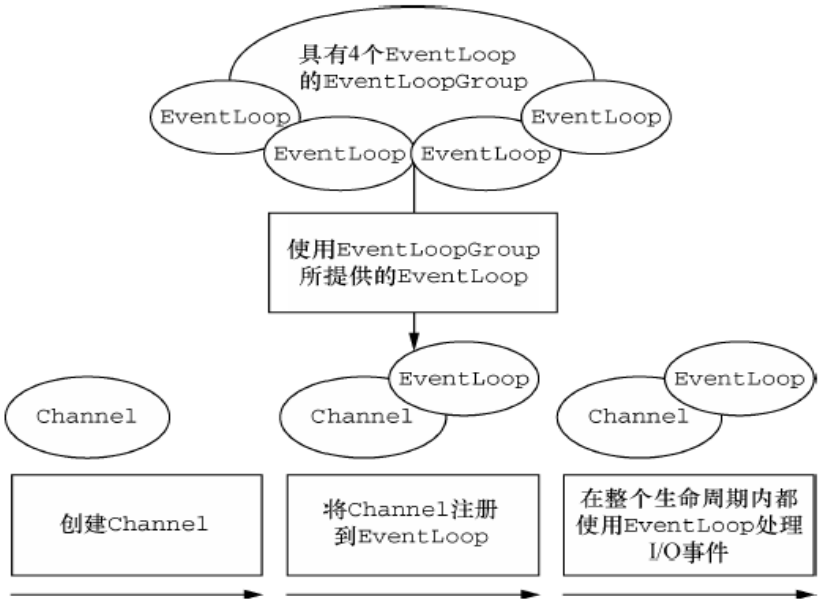
Netty 网络抽象的代表：

Channel—Socket；

EventLoop—控制流、多线程处理、并发；

ChannelFuture—异步通知。

Channel 和 EventLoop 关系如图：



从图上我们可以看出 Channel 需要被注册到某个 EventLoop 上，在 Channel 整个生命周期内都由这个 EventLoop 处理 IO 事件，也就是说一个 Channel 和一个 EventLoop 进行了绑定，但是一个 EventLoop 可以同时被多个 Channel 绑定。这一点在“EventLoop 和 EventLoopGroup”节里也提及过。

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，其基本的构造是类 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。

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

当这些状态发生改变时，将会生成对应的事件。这些事件将会被转发给 ChannelPipeline 中的 ChannelHandler，其可以随后对它们做出响应。在我们的编程中，关注 ChannelActive 和 ChannelInactive 会更多一些。

重要 Channel 的方法

eventLoop： 返回分配给 Channel 的 EventLoop

pipeline： 返回 Channel 的 ChannelPipeline，也就是说每个 Channel 都有自己的 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，注意，这个写只是写往 Netty 内部的缓存，还没有真正写往 socket。

flush： 将之前已写的数据冲刷到底层 socket 进行传输。

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

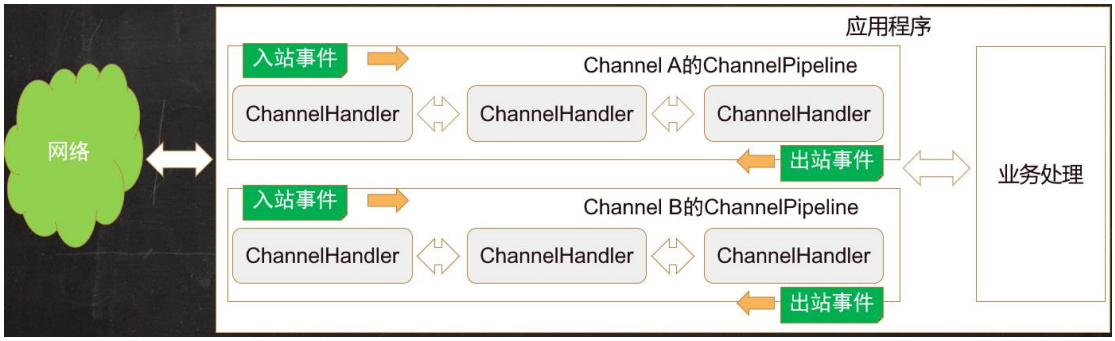
ChannelPipeline 和 ChannelHandlerContext

ChannelPipeline 接口

当 Channel 被创建时，它将会被自动地分配一个新的 ChannelPipeline，每个 Channel 都有自己的 ChannelPipeline。这项关联是永久性的。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。

ChannelPipeline 提供了 ChannelHandler 链的容器，并定义了用于在该链上传播**入站（也就是从网络到业务处理）**和 出站（也就是从业务处理到网络），各种事件流的 API，我们代码中的 ChannelHandler 都是放在 ChannelPipeline 中的。

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 ChannelHandler 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，而且 ChannelHandler 对象也完全可以拦截事件不让事件继续传递。它们的执行顺序是由它们被添加的顺序所决定的。



ChannelHandler 的生命周期

在 ChannelHandler 被添加到 ChannelPipeline 中或者被从 ChannelPipeline 中移除时会调用下面这些方法。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

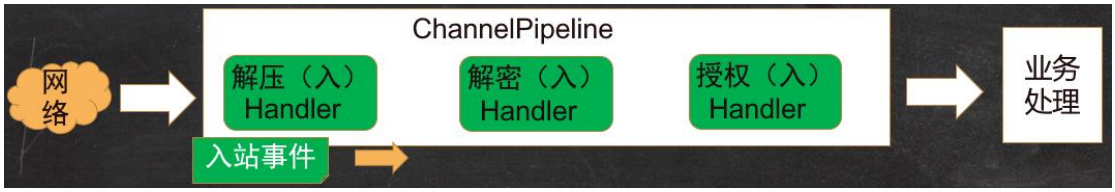
handlerRemoved 当从 ChannelPipeline 中移除 ChannelHandler 时被调用

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

ChannelPipeline 中的 ChannelHandler

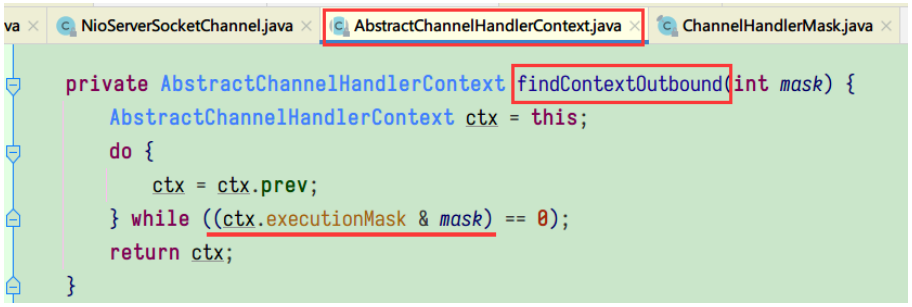
入站和出站 ChannelHandler 被安装到同一个 ChannelPipeline 中，ChannelPipeline 以双向链表的形式进行维护管理。比如下图，我们在网络上传递的数据，要求加密，但是加密后密文比较大，需要压缩后再传输，而且按照业务要求，需要检查报文中携带的用户信息是否合法，于是我们实现了 5 个 Handler：解压（入）Handler、压缩（出）handler、解密（入）Handler、加密（出） Handler、授权（入） Handler。

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，但是只被处理入站事件的 Handler 处理，也就是解压（入）Handler、解密（入） Handler、授权（入） Handler，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，数据将从链的尾端开始流动，但是只被处理出站事件的 Handler 处理，也就是加密（出） Handler、压缩（出）handler，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，也就是我们的 Socket。

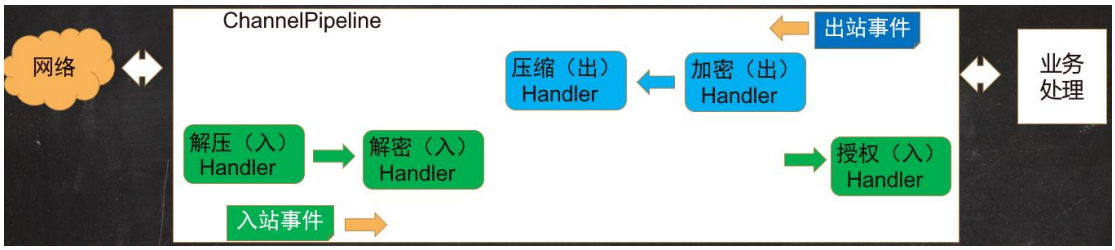
Netty 能区分入站事件的 Handler 和出站事件的 Handler，并确保数据只会在具有相同定向类型的两个 ChannelHandler 之间传递。



```
NioServerSocketChannel.java | AbstractChannelHandlerContext.java | ChannelHandlerMask.java
Reader
private static final int MASK_ALL_INBOUND = MASK_EXCEPTION_CAUGHT | MASK_CHANNEL_REGISTERED |
        MASK_CHANNEL_UNREGISTERED | MASK_CHANNEL_ACTIVE | MASK_CHANNEL_INACTIVE | MASK_CHANNEL_READ |
        MASK_CHANNEL_READ_COMPLETE | MASK_USER_EVENT_TRIGGERED | MASK_CHANNEL_WRITABILITY_CHANGED;
private static final int MASK_ALL_OUTBOUND = MASK_EXCEPTION_CAUGHT | MASK_BIND | MASK_CONNECT | MASK_D
        MASK_CLOSE | MASK_DEREGISTER | MASK_READ | MASK_WRITE | MASK_FLUSH;
```

所以在我们编写 Netty 应用程序时要注意，分属出站和入站不同的 Handler ，**在业务没特殊要求的情况下**是无所谓顺序的，正如我们下面的图所示，比如‘压缩（出）handler‘可以放在‘解压（入）handler‘和‘解密（入） Handler‘中间，也可以放在‘解密（入） Handler‘和‘授权（入） Handler‘之间。

而同属一个方向的 Handler 则是有顺序的，因为上一个 Handler 处理的结果往往是下一个 Handler 的要求的输入。比如入站处理，对于收到的数据，只有先解压才能得到密文，才能解密，只有解密后才能拿到明文中的用户信息进行授权检查，所以解压->解密->授权这个三个入站 Handler 的顺序就不能乱。



ChannelPipeline 上的方法

既然 ChannelPipeline 以双向链表的形式进行维护管理 Handler，自然也提供了对应的方法在 ChannelPipeline 中增加或者删除、替换 Handler。

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

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 ChannelHandlerContext，为什么需要这个 ChannelHandlerContext ？前面我们已经说过，ChannelPipeline 以双向链表的形式进

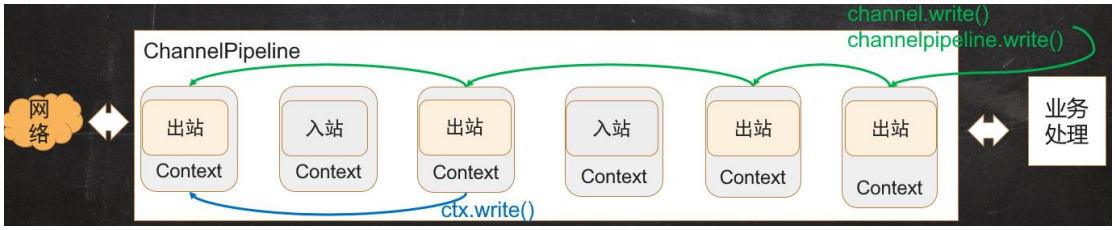
行维护管理 Handler，毫无疑问，Handler 在放入 ChannelPipeline 的时候必须要有两个指针 pre 和 next 来说明它的前一个元素和后一个元素，但是 Handler 本身来维护这两个指针合适吗？想想我们在使用 JDK 的 LinkedList 的时候，我们放入 LinkedList 的数据是不会带这两个指针的，LinkedList 内部会用类 Node 对我们的数据进行包装，而类 Node 则带有两个指针 pre 和 next。

所以，ChannelHandlerContext 的主要作用就和 LinkedList 内部的类 Node 类似。

不过 ChannelHandlerContext 不仅仅只是个包装类，它还提供了很多的方法，比如让事件从当前 ChannelHandler 传递给链中的下一个 ChannelHandler，还可以被用于获取底层的 Channel，还可以用于写出站数据。

Channel、ChannelPipeline 和 ChannelHandlerContext 上的事件传播

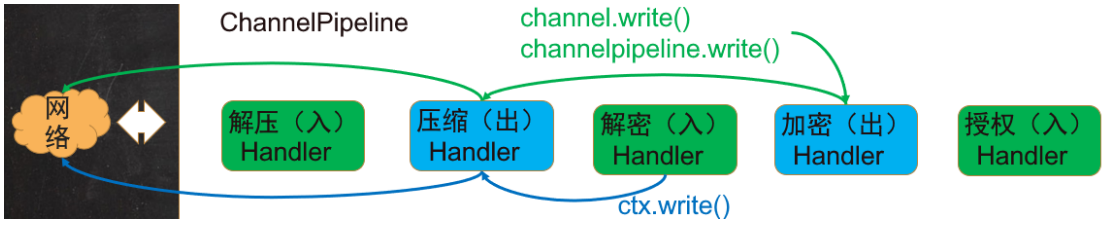
ChannelHandlerContext 有很多的方法，其中一些方法也存在于 Channel 和 Channel-Pipeline 本身上，**但是有一点重要的不同。**如果调用 Channel 或者 ChannelPipeline 上的这些方法，它们将沿着整个 ChannelPipeline 进行传播。而调用位于 ChannelHandlerContext 上的相同方法，则将从当前所关联的 ChannelHandler 开始，并且只会传播给位于该 ChannelPipeline 中的下一个（入站下一个，出站上一个）能够处理该事件的 ChannelHandler。



我们用一个实际例子来说明，比如服务器收到对端发过来的报文，解压后需要进行解密，结果解密失败，要给对端一个应答。

如果发现解密失败原因是服务器和对端的加密算法不一致，应答报文只能以明文的压缩格式发送，就可以在解密 handler 中直接使用 ctx.write 给对端应答，这样应答报文就只经过压缩 Handler 就发往了对端；

其他情况下，应答报文要以加密和压缩格式发送，就可以在解密 handler 中使用 channel.write()或者 channelpipeline.write()给对端应答，这样应答报文就会流经整个出站处理过程。



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(ctx)方法

write 通过这个实例写入消息并经过 ChannelPipeline

writeAndFlush 通过这个实例写入并冲刷消息并经过 ChannelPipeline

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；
- 相对于其他类的同名方法，ChannelHandlerContext 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。

ChannelHandler

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 直接冲刷数据然后输出到对端。应用程序的业务逻辑通常实现在一个或者多个 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应用程序的业务逻辑所处理。

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

下面列出了接口 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

channelRead 当从 Channel 读取数据时被调用

ChannelWritabilityChanged

当 Channel 的可写状态发生改变时被调用。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().setWriteLowWaterMark()方法来设置

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

注意：channelReadComplete 和 channelRead 这两个方法非常让人搞不清两者的区别是什么，我们先放下这个疑问，后面会有解释。

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

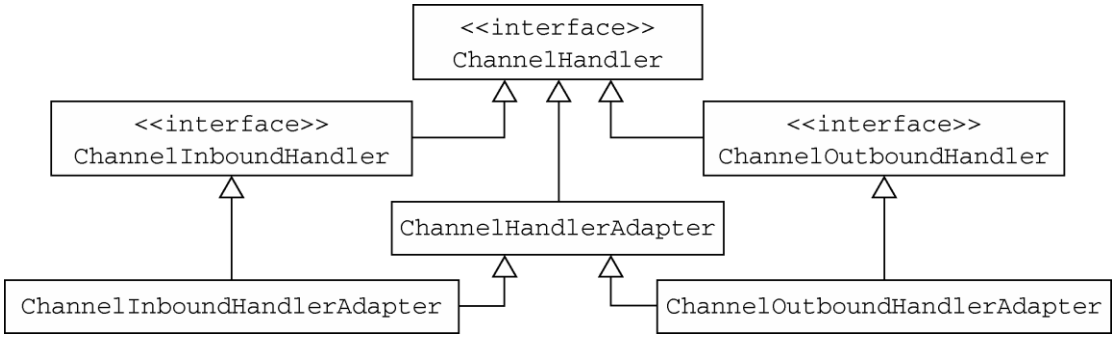
flush(ChannelHandlerContext) 当请求通过 Channel 将入队数据冲刷到远程节点时被调用

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 所需要的工作降到最低限度，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（处理入站） 和 ChannelOutboundHandlerAdapter（处理出站）。

我们可以使用 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。



不过 ChannelOutboundHandler 有个非常让人迷惑的 read 方法，ChannelOutboundHandler 不是处理出站事件的吗？怎么会有 read 方法呢？其实这个 read 方法不是表示读数据，而是表示业务发出了读（read）数据的要求，这个要求也会封装为一个事件进行传播，这个事件因为是业务发出到网络的，自然就是个出站事件，而且这个事件触发的就是 ChannelOutboundHandler 中 read 方法。

如果我们的 Handler 既要处理入站又要处理出站怎么办呢？这个时候就可以使用类 ChannelDuplexHandler，当然也可以同时实现 ChannelOutboundHandler, ChannelInboundHandler 这两个接口，自然就要麻烦很多了。

Handler 的共享和并发安全性

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。

这就牵涉到了我们实现的 Handler 的共享性和线程安全性。回顾我们的 Netty 代码，在往 pipeline 安装 Handler 的时候，我们基本上是 new 出 Handler 的实例

```
.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new EchoClientHandler());

    }
});
```

因为每个 socketChannel 有自己的 pipeline 而且每个 socketChannel 又是和线程绑定的，所以这些 Handler 的实例之间完全独立的，只要 Handler 的实例之间不是共享了全局变量，Handler 的实例是线程安全的。

但是如果业务需要我们在多个 socketChannel 之间共享一个 Handler 的实例怎么办呢？比如统计服务器接受到和发出的业务报文总数，我们就需要用一个 Handler 的实例来横跨所有的 socketChannel 来统计所有 socketChannel 业务报文数。

为了实现这一点，我们可以实现一个 MessageCountHandler，并且在 MessageCountHandler 上使用 Netty 的@Sharable 注解，然后在安装 MessageCountHandler 实例到 pipeline 时，共用一个即可。当然，因为 MessageCountHandler 实例是共享的，所以在实现 MessageCountHandler 的统计功能时，请务必注意线程安全，我们在具体实现时就使用了 Java 并发编程里的 Atomic 类来保证这一点。

具体代码请参考包 cn.tuling.nettybasic.sharehndler。

资源管理和 SimpleChannelInboundHandler

回想一下我们在 NIO 中是如何接收和发送网络数据的？都是首先创建了一个 Buffer，应用程序中的业务部分和 Channel 之间通过 Buffer 进行数据的交换：

```
SocketChannel sc = (SocketChannel) key.channel();
/*创建ByteBuffer，开辟一个缓冲区*/
ByteBuffer buffer = ByteBuffer.allocate(1024);
/*从通道里读取数据，然后写入buffer*/
int readBytes = sc.read(buffer);
if(readBytes>0){
```

Netty 在处理网络数据时，同样也需要 Buffer，在 Read 网络数据时由 Netty 创建 Buffer，Write 网络数据时 Buffer 往往是由业务方创建的。不管是读和写，Buffer 用完后都必须进行释放，否则可能会造成内存泄露。

在 Write 网络数据时，可以确保数据被写往网络了，Netty 会自动进行 Buffer 的释放，但是如果 Write 网络数据时，我们有 outBoundHandler 处理了 write()操作并丢弃了数据，没有继续往下写，要由我们负责释放这个 Buffer，就必须调用 ReferenceCountUtil.release 方法，否则就可能会造成内存泄露。

在 Read 网络数据时，如果我们可以确保每个 InboundHandler 都把数据往后传递了，也就是调用了相关的 fireChannelRead 方法，Netty 也会帮我们释放，同样的，如果我们有 InboundHandler 处理了数据，又不继续往后传递，又不调用负责释放的 ReferenceCountUtil.release 方法，就可能会造成内存泄露。

但是由于消费入站数据是一项常规任务，所以 Netty 提供了一个特殊的被

称为 SimpleChannelInboundHandler 的 ChannelInboundHandler 实现。这个实现会在数据被 channelRead0()方法消费之后自动释放数据。

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    boolean release = true;
    try {
        if (acceptInboundMessage(msg)) {
            @SuppressWarnings("unchecked")
            I imsg = (I) msg;
            channelRead0(ctx, imsg);
        } else {
            release = false;
            ctx.fireChannelRead(msg);
        }
    } finally {
        if (autoRelease && release) {
            ReferenceCountUtil.release(msg);
        }
    }
}
```

同时系统为我们提供的各种预定义 Handler 实现，都实现了数据的正确处理，所以我们自行在编写业务 Handler 时，也需要注意这一点：**要么继续传递，要么自行释放**。

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

网络编程里，“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。

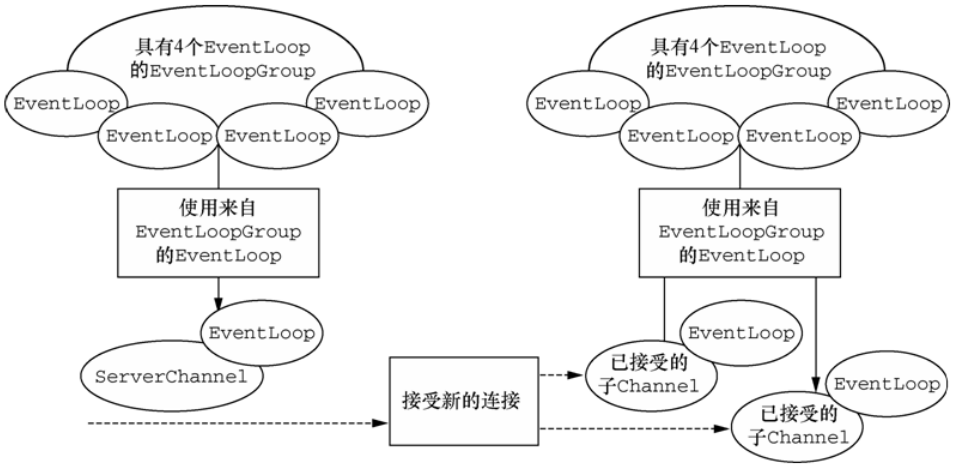
比较 **Bootstrap** 类

	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。

ChannelInitializer

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。

所以，在我们自己的应用程序中，如果存在着某个 handler 只使用一次的情况，也可以仿造 ChannelInitializer，用完以后将自己从 ChannelPipeline 中移除自己，比如授权 handler，某客户端第一次连接登录以后，进行授权检查，检查通过后就可以把这个授权 handler 移除了。如果客户端关闭连接下线，下次再连接的时候，就是一个新的连接，授权 handler 依然会被安装到 ChannelPipeline ，依然会进行授权检查。

ChannelOption

ChannelOption 的各种属性在套接字选项中都有对应。

ChannelOption.SO_BACKLOG

ChannelOption.SO_BACKLOG 对应的是 tcp/ip 协议 listen 函数中的 backlog 参数，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。所以操作系统里一般有两个队列，一个是 ACCEPT 队列，保存着已经完成了 TCP 的三次握手的连接，一个 SYN 队列，服务器正在等待 TCP 的三次握手完成的队列。

BSD 派生系统里 backlog 指的就是 SYN 队列的大小，在 Linux 的实现里 backlog 相对来说，就含糊不清了，有些内核版本指的是 ACCEPT 队列+SYN 队列合起来的大小，有的是指 SYN 队列的大小。

但是从 Linux 2.2 开始，backlog 的参数行为在 Linux 2.2 中发生了变化，现在它指定等待接受的完全建立的套接字的队列长度，而不是不完整的连接请求的数量。 不完整套接字队列的最大长度可以使用 /proc/sys/net/ipv4/tcp_max_syn_backlog 设置，默认值为 128。

如果 backlog 参数大于 /proc/sys/net/core/somaxconn 中的值，那么它会被静默截断为值 128。在 2.4.25 之前的内核中，此限制是硬编码值，后续内核版本也可以通过 vim /etc/sysctl.conf 来修改，包括我们前面所说的 tcp_max_syn_backlog 也可以在此处修改，然后通过命令 sysctl -p 生效。

```
The behavior of the backlog argument on TCP sockets changed with Linux 2.2.  Now  it  specifies  the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests.  The maximum length of the  queue  for  incomplete  sockets can be set using /proc/sys/net/ipv4/tcp_max_syn_backlog.  When syncookies are enabled there is no logical maximum length and this setting is ignored.  See tcp(7) for more information.

If  the  backlog argument is greater than the value in /proc/sys/net/core/somaxconn, then it is silently truncated to that value; the default value in this  file  is  128.   In  kernels before 2.4.25, this limit was a hard coded value, SOMAXCONN, with the value 128.
```

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，多网卡（IP）绑定相同端口，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。

但是注意，这个参数无法做到让应用绑定完全相同 IP + Port 来重复启动。

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。

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。

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

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。）

使用模式

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。

复合缓冲区

复合缓冲区 CompositeByteBuf，它为多个 ByteBuf 提供一个聚合视图。比如 HTTP 协议，分为消息头和消息体，这两部分可能由应用程序的不同模块产生，各有各的 ByteBuf，将会在消息被发送的时候组装为一个 ByteBuf，此时可以将这两个 ByteBuf 聚合为一个 CompositeByteBuf，然后使用统一和通用的 ByteBuf API 来操作。

分配

如何在我们的程序中获得 ByteBuf 的实例，并使用它呢？Netty 提供了两种方式

ByteBufAllocator 接口

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

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 类还可用于 ByteBuf 同样可用于那些并不需要 Netty 的其他组件的非网络项目。

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

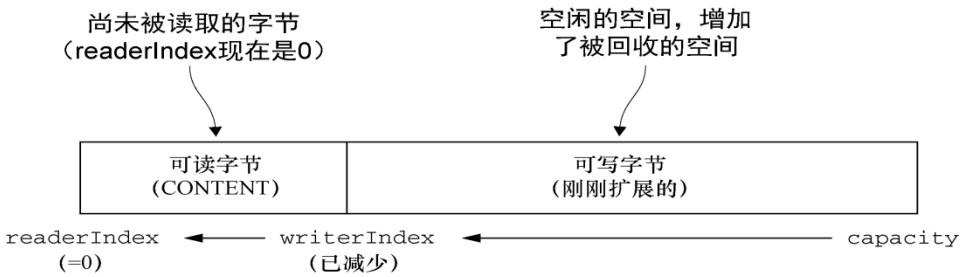
hasArray() 如果 ByteBuf 由一个字节数组支撑，则返回 true

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

可丢弃字节的分段包含了已经被读过的字节。通过调用 discardReadBytes()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。

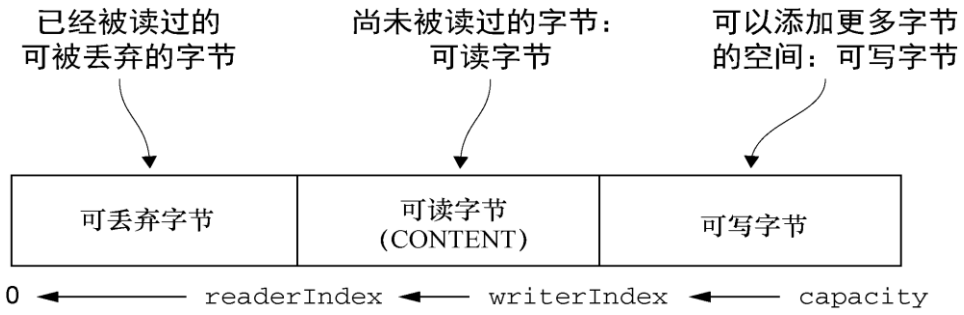


可读字节

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 forEachByte()。

下面的代码展示了一个查找回车符（\r）的例子。

```
ByteBuf buffer = .. .;

int index = buffer.forEachByte(ByteBufProcessor.FIND_CR);
```

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。

资源释放

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 ReferenceCountUtil.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 SimpleChannelInboundHandler，SimpleChannelInboundHandler 会自动释放资源。

1、对于入站请求，Netty 的 EventLoo 在处理 Channel 的读操作时进行分配 ByteBuf，对于这类 ByteBuf，需要我们自行进行释放，有三种方式：

或者使用 SimpleChannelInboundHandler；

或者在重写 channelRead()方法使用 ReferenceCountUtil.release()

或者在重写 channelRead()方法使用使用 ctx.fireChannelRead 继续向后传递；

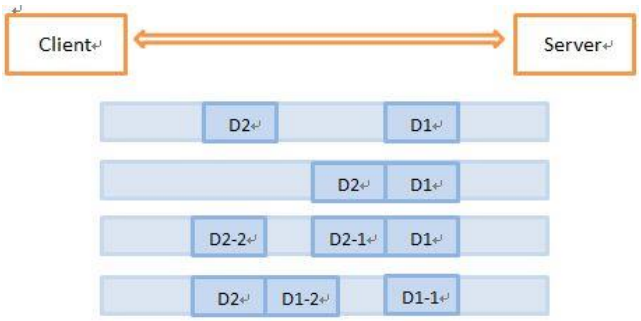
2、对于出站请求，不管 ByteBuf 是否由我们的业务创建的，当调用了 write 或者 writeAndFlush 方法后，Netty 会自动替我们释放，不需要我们业务代码自行释放。

解决粘包/半包

回顾我们的的 Hello,Netty

参见 cn.tuling.nettybasic.splicing.demo 下的代码

什么是 TCP 粘包半包?



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：就是一个数据包被分成了多次接收。

更具体的原因至少包括：

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。

解决粘包半包

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；

参见 cn.tuling.nettybasic.splicing.linebase（回车换行符进行分割）和 cn.tuling.nettybasic.splicing.delimiter（自定义分割符）下的代码

（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；

参见 cn.tuling.nettybasic.splicing.fixed 下的代码

（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，使用 LengthFieldBasedFrameDecoder，后面会有详细说明和使用。

辨析 channelRead 和 channelReadComplete

两者的区别：

Netty 是在读到完整的业务请求报文后才调用一次业务 ChannelHandler 的 channelRead 方法，无论这条报文底层经过了几次 SocketChannel 的 read 调用。

但是 channelReadComplete 方法并不是在业务语义上的读取消息完成后被触发的，而是在每次从 SocketChannel 成功读到消息后，由系统触发，也就是说如果一个业务消息被 TCP 协议栈发送了 N 次，则服务端的 channelReadComplete 方法就会被调用 N 次。

我们用代码来看看 cn.tuling.nettybasic.checkread：

我们在客户端发送 5 次较小报文

```
/*发送较小报文5次*/
String requestSmall = "small" + System.getP
for(int i = 0; i<5; i++ ){
    msg = Unpooled.buffer(requestSmall.lengt
    msg.writeBytes(requestSmall.getBytes());
    ctx.writeAndFlush(msg);
}
LOG.info("client send small message 5.");
```

服务端输出：

```
c.t.n.c.EchoServerCRHandler - channelRead - INFO  channelRead执行了1次
c.t.n.c.EchoServerCRHandler - channelReadComplete - INFO  channelReadComplete执行了1次
c.t.n.c.EchoServerCRHandler - channelRead - INFO  channelRead执行了2次
c.t.n.c.EchoServerCRHandler - channelRead - INFO  channelRead执行了3次
c.t.n.c.EchoServerCRHandler - channelReadComplete - INFO  channelReadComplete执行了2次
c.t.n.c.EchoServerCRHandler - channelRead - INFO  channelRead执行了4次
c.t.n.c.EchoServerCRHandler - channelRead - INFO  channelRead执行了5次
c.t.n.c.EchoServerCRHandler - channelReadComplete - INFO  channelReadComplete执行了3次
```

很明显，channelRead 是一个报文执行一次，执行的执行次数和客户端发送报文数一样，channelReadComplete 虽然也执行了多次，但是和客户端发送报文数没什么关系，而且也没什么规律可寻。

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 ChannelInboundHandler 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。

比如一个实际的业务场景，两端通信，通过 JSON 交换信息，而且 JSON 文本需要加密，接收端就可以：

网络加密报文 -> 经过 ByteToMessageDecoder -> String 类型的 JSON 明文；

String 类型的 JSON 文本-> 经过 MessageToMessageDecoder -> Java 里的对象

所以我们可以把 ByteToMessageDecoder 看成一次解码器，MessageToMessageDecoder 看成二次或者多次解码器

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，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)

是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。

```java
public class TooLongSample extends ByteToMessageDecoder {

    private static final int MAX_SIZE = 1024;

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)
            throws Exception {
        int readable = in.readableBytes();
        if(readable>MAX_SIZE){
            ctx.close();
            throw new TooLongFrameException("传入的数据太多！");
        }
    }
}
```

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<I>

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

还是用我们上面的业务场景，两端通信，通过 JSON 交换信息，而且 JSON 文本需要加密，发送端就可以：

Java 里的对象-> 经过 MessageToMessageEncoder -> String 类型的 JSON 文本

String 类型的 JSON 明文 -> 经过 MessageToByteEncoder-> 网络加密报文；

所以我们可以把 MessageToByteEncoder 看成网络报文编码器，MessageToMessageEncoder 看成业务编码器。

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 的出站消息（类型为 I 的）。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，但是有时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。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

实战 – 实现 SSL/TLS 和 Web 服务

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，这减少了我们花费的时间与精力。

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。

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，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

HttpClientCodec 和 HttpServerCodec 则将请求和响应做了一个组合。

```
public final class HttpClientCodec extends CombinedChannelDuplexHandler<HttpResponseDecoder, HttpRequestEncoder>
```

```
public final class HttpServerCodec extends CombinedChannelDuplexHandler<HttpRequestDecoder, HttpResponseEncoder>
```

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 HttpObjectAggregator，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。

SSL 和 HTTP 的代码参见模块 netty-http

视频中实现步骤：

1、首先实现 Http 服务器并浏览器访问；

2、增加 SSL 控制；

3、实现客户端并访问。

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

当选行远程跨进程服务调用时，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。

代码参见模块 netty-basic 下的包 cn.tuling.nettybasic.serializable.protogenesis

如何选择序列化框架

选择四要点

是否需要跨语言的支持

空间:编码后占用空间

时间:编解码速度

是否追求可读性

如果项目里有跨语言支持的硬性要求，某种序列化框架只支持特定语言，即使它比其他的框架快 1 万倍，也没法选择。

空间和时间其实是对序列化框架的性能要求，这两者其实是存在矛盾的，想要编码后占用空间小，自然就要花费更多的时间去编码，所以这两者往往要追求一种平衡。

有些项目里还要求序列化后的数据是人类可读的，这个时候的选择就不多了，一般是 JSON 格式或者 XML 格式，有些序列化框架也支持通过自带工具观察序列化后的数据，也可以考虑选择。

序列化框架比较

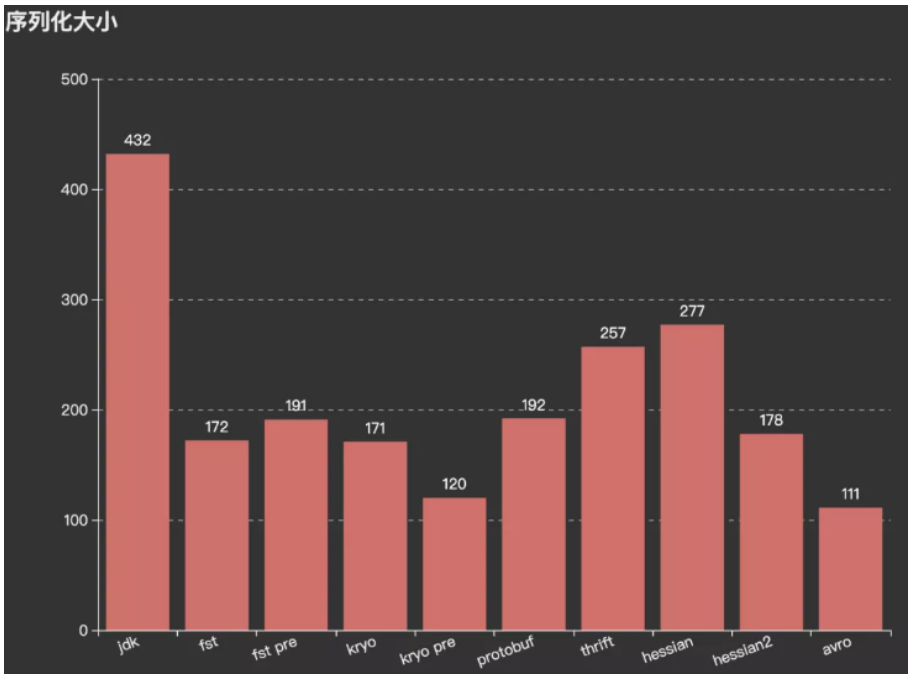
我们可以借鉴阿里技术官方的分析结果（https://developer.aliyun.com/article/783611?utm_content=g_1000268438）

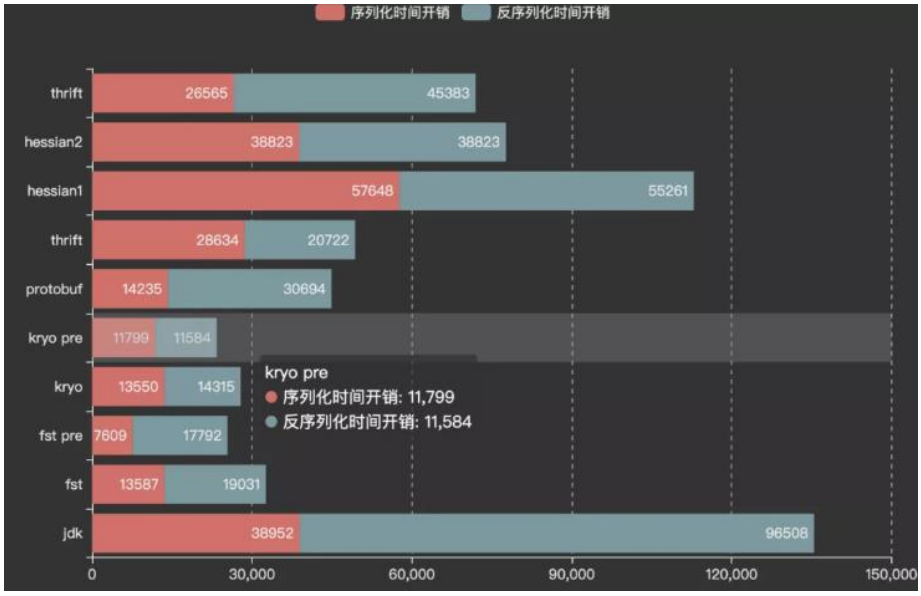
跨语言来说：

序列化框架	通用性
JDK Serializer	只适用于Java
FST	只适用于Java
Kryo	主要适用于Java(可复杂支持跨语言)
Protocol buffer	支持多种语言
Thrift	支持多种语言
Hessian	支持多种语言
Avro	支持多种语言

还有一种没体现在上面的 msgpack（https://msgpack.org/）也是支持跨语言的。

从性能上来说





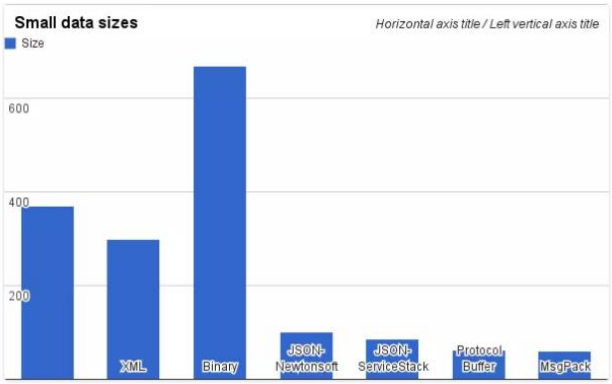
从空间性能来看，avro、kryo、Hessian2、fst、Protocol buffer 表现都不错；

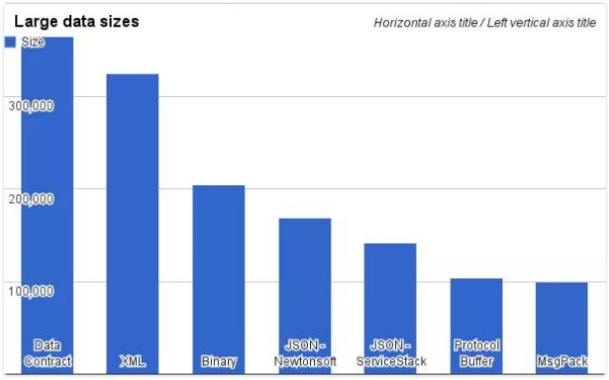
从时间性能来看，kryo、fst、Protocol buffer 表现也很好

Msgpack 也是一种很优秀的序列化框架，性能和 Protocol buffer 不相上下：

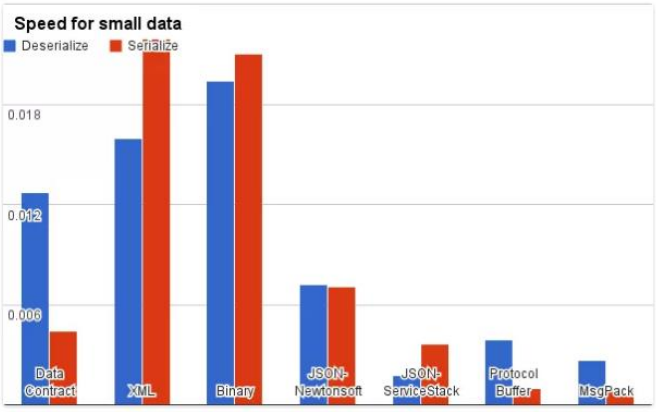
https://www.howtoautomate.in.th/protobuf-101/2017-05-06-10_30_22-serialization-performance-comparisonxmlbinaryjsonp/

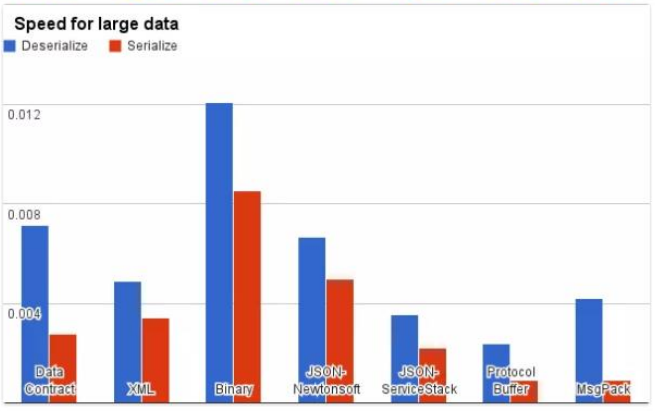
https://www.howtoautomate.in.th/wp-content/uploads/2017/05/2017-05-06-10_30_34-Serialization-Performance-comparisonXMLBinaryJSONP....png





(* smaller is better)





(* smaller is better)

所以接下来，在后续的课程中我们将讲述 Protocol Buffer、MessagePack 和 Kyro 的 Netty 的集成，其他的序列化框架和 Netty 的集成，大家可以参考后自行实现。

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

Protocol Buffers 序列化机制代码参见模块 netty-basic 下的包 cn.tuling.nettybasic.serializable.protobuf

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；

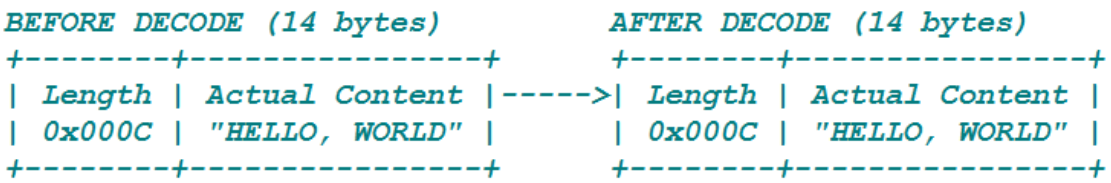
lengthFieldLength：记录该帧数据长度的字段，也就是长度域本身的长度；

lengthAdjustment：长度的一个修正值，可正可负，Netty 在读取到数据包的长度值 N 后，认为接下来的 N 个字节都是需要读取的，但是根据实际情况，有可能需要增加 N 的值，也有可能需要减少 N 的值，具体增加多少，减少多少，写在这个参数里；

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，扔掉这个数据包中多少字节数，才是后续业务实际需要的业务数据。

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。

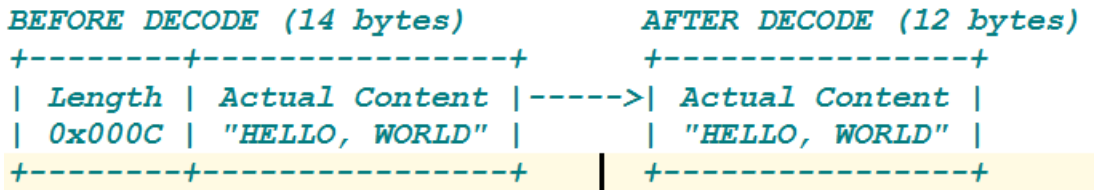
数据包大小: 14B = 长度域 2B + "HELLO, WORLD"（单词 HELLO+一个逗号+一个空格+单词 WORLD）



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　无需调整
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"



长度域的值为 12B(0x000c)。解码后，希望丢弃长度域 2B 字段，所以，只要 initialBytesToStrip = 2 即可。

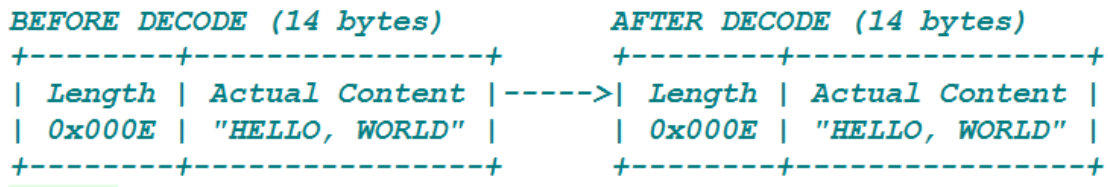
1. lengthFieldOffset = 0

2. lengthFieldLength = 2

3. lengthAdjustment　无需调整

4. initialBytesToStrip = 2 解码过程中，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)，包含了长度域本身的长度。希望解码后保持一样，根据上面的公式，参数应该为：

1. lengthFieldOffset = 0

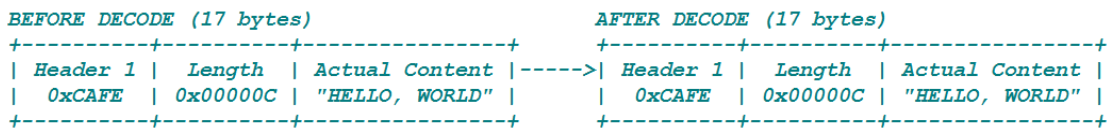
2. lengthFieldLength = 2

3. lengthAdjustment = -2　因为长度域为 14，而报文内容为 12，为了防止读取报文超出报文本体，和将长度字段一起读取进来，需要告诉 netty，实际读取的报文长度比长度域中的要少 2（12-14=-2）

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"



长度域的值为 12B(0x000c)。编码解码后，长度保持一致，所以 initialBytesToStrip = 0。参数应该为:

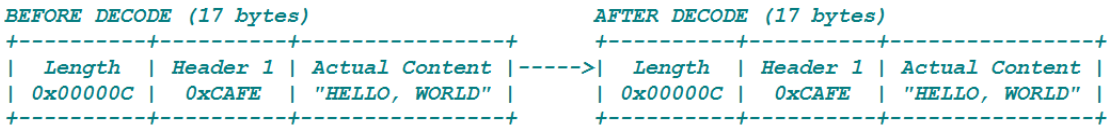
1. lengthFieldOffset = 2

2. lengthFieldLength = 3

3. lengthAdjustment = 0 无需调整

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"



长度域的值为 12B(0x000c)。编码解码后，长度保持一致，所以 initialBytesToStrip = 0。参数应该为:

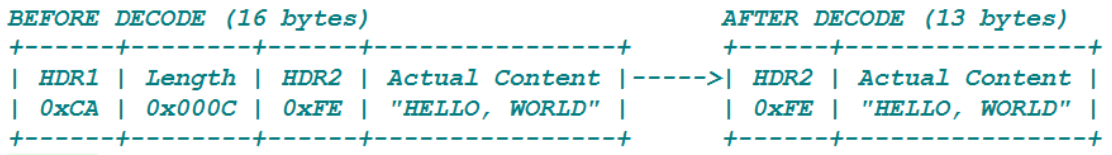
1. lengthFieldOffset = 0

2. lengthFieldLength = 3

3. lengthAdjustment = 2 因为长度域为 12，而报文内容为 12，但是我们需要把 Header 的值一起读取进来，需要告诉 netty，实际读取的报文内容长度比长度域中的要多 2（12+2=14）

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

带有两个 header。HDR1 丢弃，长度域丢弃，只剩下第二个 header 和有效包体，这种协议中，一般 HDR1 可以表示 magicNumber，表示应用只接受以该 magicNumber 开头的二进制数据，rpc 里面用的比较多。总数据包长度: 16=HDR1(1B)+长度域(2B) +HDR2(1B) + "HELLO, WORLD"



长度域的值为 12B(0x000c)

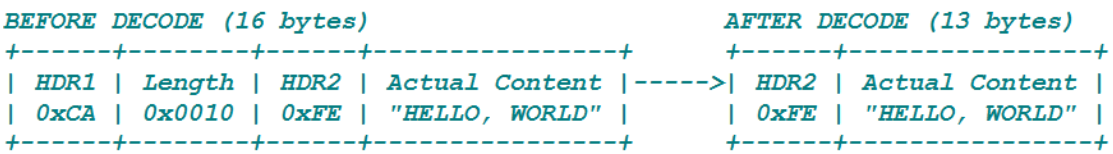
1. lengthFieldOffset = 1 (HDR1 的长度)

2. lengthFieldLength = 2

3. lengthAdjustment =1 因为长度域为 12，而报文内容为 12，但是我们需要把 HDR2 的值一起读取进来，需要告诉 netty，实际读取的报文内容长度比长度域中的要多 1（12+1=13）

4. initialBytesToStrip = 3 丢弃了 HDR1 和长度字段

带有两个 header，HDR1 丢弃，长度域丢弃，只剩下第二个 header 和有效包体。总数据包长度: 16=HDR1(1B)+长度域(2B) +HDR2(1B) + "HELLO, WORLD"



长度域的值为 16B(0x0010)，长度为 2，HDR1 的长度为 1，HDR2 的长度为 1，包体的长度为 12，1+1+2+12=16。

1. lengthFieldOffset = 1

2. lengthFieldLength = 2

3. lengthAdjustment = -3 因为长度域为 16，需要告诉 netty，实际读取的报文内容长度比长度域中的要 少 3（13-16= -3）

4. initialBytesToStrip = 3 丢弃了 HDR1 和长度字段

MessagePack 集成

代码参见模块 netty-basic 下的: cn.tuling.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()

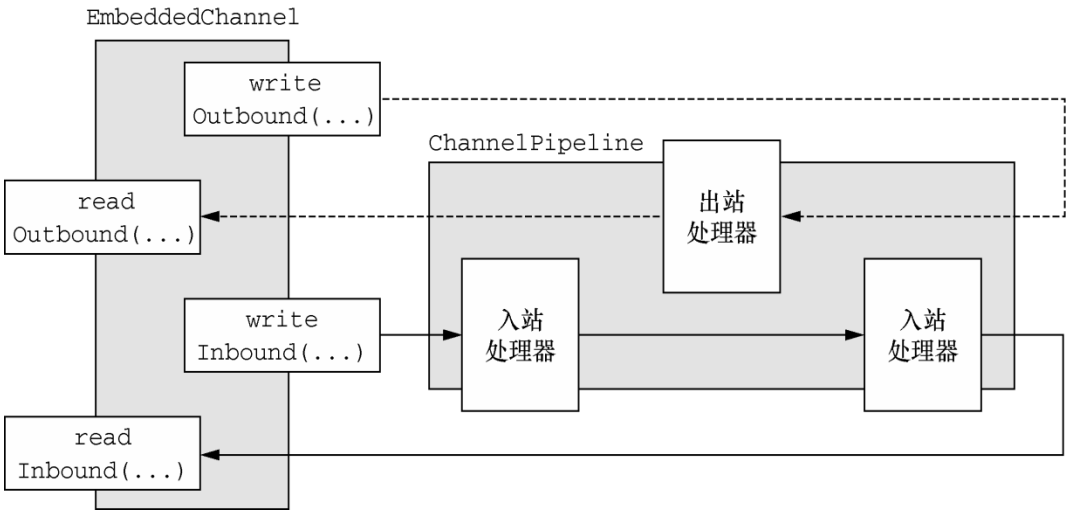
从 EmbeddedChannel 中读取一个出站消息。任何返回的东西都穿越了整个 ChannelPipeline。如果没有任何可供读取的，则返回 null

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

我们有一个简单的 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。

测试出站消息

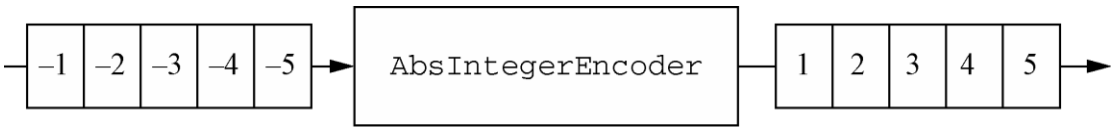
在测试的处理器—AbsIntegerEncoder，它是 Netty 的 MessageToMessageEncoder 的一个特殊化的实现，用于将负值整数转换为绝对值。

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。

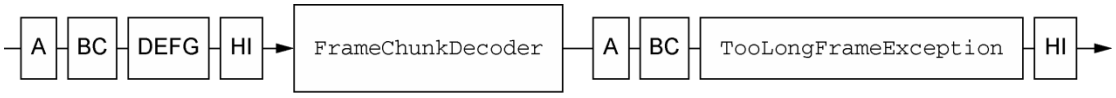


测试异常处理

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。



本文档分享地址：

http://note.youdao.com/noteshare?id=f71d324d40f6c4bfb67e419a8fb42862&sub=28DC5500D0B549329967B06BA6D2DEA5