# 【第10章-编解码器框架】

Netty 提供了多种组件，简化了为了支持广泛的协议而创建自定义的编解码器的过程。

## 10.1 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。

## 10.2 解码器

在这一节中，我们将研究Netty 所提供的解码器类，并提供关于何时以及如何使用它们的具体示例。这些类覆盖了两个不同的用例：

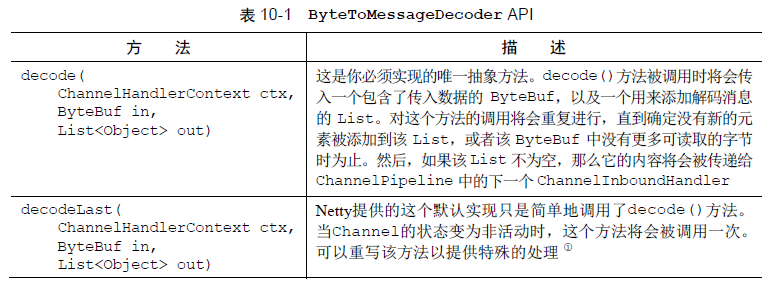
* 将字节解码为消息——ByteToMessageDecoder 和ReplayingDecoder；
* 将一种消息类型解码为另一种——MessageToMessageDecoder。

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，所以知道Netty 的解码器实现了ChannelInboundHandler 也不会让你感到意外。

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，这也是Netty 是如何支持代码的模块化以及复用的一个很好的例子。

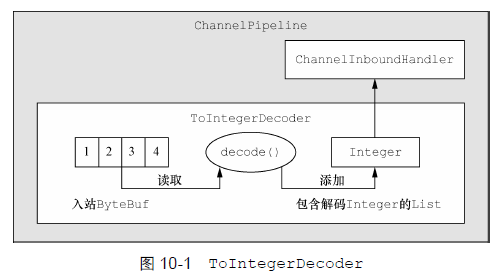
### 10.2.1 抽象类ByteToMessageDecoder

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。表10-1 解释了它最重要的两个方法。



下面举一个如何使用这个类的示例，假设你接收了一个包含简单int 的字节流，每个int都需要被单独处理。在这种情况下，你需要从入站ByteBuf 中读取每个int，并将它传递给ChannelPipeline 中的下一个ChannelInboundHandler。为了解码这个字节流，你要扩展ByteToMessageDecoder 类。（需要注意的是，原子类型的int 在被添加到List 中时，会被自动装箱为Integer。）该设计如图10-1 所示。

每次从入站ByteBuf 中读取4 字节，将其解码为一个int，然后将它添加到一个List 中。当没有更多的元素可以被添加到该List 中时，它的内容将会被发送给下一个ChannelInboundHandler。



|  |
| --- |
| *// 代码清单10-1 ToIntegerDecoder 类扩展了ByteToMessageDecoder*  *// 扩展ByteToMessageDecoder 类，以将字节解码为特定的格式* **public class** ToIntegerDecoder **extends** ByteToMessageDecoder {  @Override  **public void** decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) **throws** Exception {  *// 检查是否至少有4字节可读（一个int的字节长度）* **if** (in.readableBytes() >= **4**) {  *// 从入站ByteBuf 中读取一个int，并将其添加到解码消息的List 中* out.add(in.readInt());  }  } } |

虽然ByteToMessageDecoder 使得可以很简单地实现这种模式，但是你可能会发现，在调用readInt()方法前不得不验证所输入的ByteBuf 是否具有足够的数据有点繁琐。在下一节中，我们将讨论ReplayingDecoder，它是一个特殊的解码器，以少量的开销消除了这个步骤。

**编解码器中的引用计数**

引用计数需要特别的注意。对于编码器和解码器来说，其过程也是相当的简单：一旦消息被编码或者解码，它就会被ReferenceCountUtil.release(message)调用自动释放。如果你需要保留引用以便稍后使用，那么你可以调用ReferenceCountUtil.retain(message)方法。这将会增加该引用计数，从而防止该消息被释放。

## 10.2.2 抽象类ReplayingDecoder

ReplayingDecoder扩展了ByteToMessageDecoder类（如代码清单10-1 所示），使得我们不必调用readableBytes()方法。它通过使用一个自定义的ByteBuf实现，ReplayingDecoderByteBuf，包装传入的ByteBuf实现了这一点，其将在内部执行该调用，指调用readableBytes()方法。

这个类的完整声明是：

public abstract class ReplayingDecoder<S> extends ByteToMessageDecoder

类型参数S 指定了用于状态管理的类型，其中Void 代表不需要状态管理。代码清单10-2展示了基于ReplayingDecoder 重新实现的ToIntegerDecoder。

|  |
| --- |
| *// 代码清单10-2 ToIntegerDecoder2 类扩展了ReplayingDecoder*  *// 扩展ReplayingDecoder<Void>以将字节解码为消息* **public class** ToIntegerDecoder2 **extends** ReplayingDecoder<Void> {   *// 传入的ByteBuf 是ReplayingDecoderByteBuf* @Override  **public void** decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) **throws** Exception {  *// 从入站ByteBuf 中读取一个int，并将其添加到解码消息的List 中* out.add(in.readInt());  } } |

和之前一样，从ByteBuf中提取的int将会被添加到List中。如果没有足够的字节可用，这个readInt()方法的实现将会抛出一个Error，这里实际上抛出的是一个Signal，详见io.netty.util.Signal 类。

其将在基类中被捕获并处理。当有更多的数据可供读取时，该decode()方法将会被再次调用。（参见表10-1 中关于decode()方法的描述。）请注意ReplayingDecoderByteBuf 的下面这些方面：

* 并不是所有的ByteBuf 操作都被支持，如果调用了一个不被支持的方法，将会抛出一个UnsupportedOperationException；
* ReplayingDecoder 稍慢于ByteToMessageDecoder。

如果对比代码清单10-1 和代码清单10-2，你会发现后者明显更简单。示例本身是很基本的，所以请记住，在真实的、更加复杂的情况下，使用一种或者另一种作为基类所带来的差异可能是很显著的。这里有一个简单的准则：如果使用ByteToMessageDecoder 不会引入太多的复杂性，那么请使用它；否则，请使用ReplayingDecoder。

**更多的解码器**

下面的这些类处理更加复杂的用例：

* io.netty.handler.codec.LineBasedFrameDecoder—这个类在Netty 内部也有使用，它使用了行尾控制字符（\n 或者\r\n）来解析消息数据；
* io.netty.handler.codec.http.HttpObjectDecoder—一个HTTP 数据的解码器。

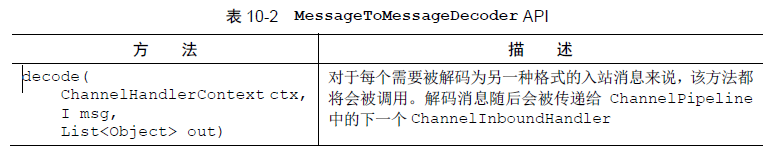
在io.netty.handler.codec 子包下面，你将会发现更多用于特定用例的编码器和解码器实现。更多有关信息参见Netty 的Javadoc。

### 10.2.3 抽象类MessageToMessageDecoder

在这一节中，我们将解释如何使用下面的抽象基类在两个消息格式之间进行转换（例如，从一种POJO 类型转换为另一种）：

public abstract class MessageToMessageDecoder<I> extends ChannelInboundHandlerAdapter

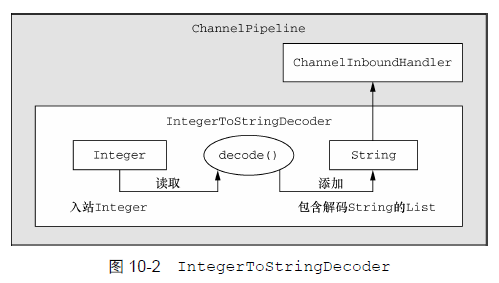
类型参数I 指定了decode()方法的输入参数msg 的类型，它是你必须实现的唯一方法。表10-2 展示了这个方法的详细信息。



在这个示例中，我们将编写一个IntegerToStringDecoder 解码器来扩展MessageToMessageDecoder<Integer>。它的decode()方法会把Integer 参数转换为它的String表示，并将拥有下列签名：

public void decode( ChannelHandlerContext ctx, Int eger msg, List<Object> out ) throws Exception

和之前一样，解码的String将被添加到传出的List中，并转发给下一个ChannelInboundHandler。该设计如图10-2 所示。



代码清单10-3 给出了IntegerToStringDecoder 的实现。

|  |
| --- |
| *// 代码清单10-3 IntegerToStringDecoder 类*  *// 扩展了MessageToMessageDecoder<Integer>* **public class** IntegerToStringDecoder **extends** MessageToMessageDecoder<Integer> {  @Override  **public void** decode(ChannelHandlerContext ctx, Integer msg, List<Object> out) **throws** Exception {  *// 将Integer 消息转换为它的String 表示，并将其添 加到输出的List 中* out.add(String.*valueOf*(msg));  } } |

**HttpObjectAggregator**

有关更加复杂的例子，请研究io.netty.handler.codec.http.HttpObjectAggregator 类，它扩展了MessageToMessageDecoder<HttpObject>。

### 10.2.4 TooLongFrameException 类

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。

代码清单10-4 展示了ByteToMessageDecoder 是如何使用TooLongFrameException来通知ChannelPipeline 中的其他ChannelHandler 发生了帧大小溢出的。需要注意的是，如果你正在使用一个可变帧大小的协议，那么这种保护措施将是尤为重要的。

|  |
| --- |
| *// 代码清单10-4 TooLongFrameException*  *// 扩展ByteToMessageDecoder以将字节解码为消息* **public class** SafeByteToMessageDecoder **extends** ByteToMessageDecoder {  **private static final int *MAX\_FRAME\_SIZE*** = **1024**;   @Override  **public void** decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) **throws** Exception {  **int** readable = in.readableBytes();  *// 检查缓冲区中是否有超过MAX\_FRAME\_SIZE个字节* **if** (readable > ***MAX\_FRAME\_SIZE***) {  *// 跳过所有的可读字节，抛出TooLongFrameException 并通知ChannelHandler* in.skipBytes(readable);  **throw new** TooLongFrameException(**"Frame too big!"**);  }  *// do something  // ...* } } |

到目前为止，我们已经探讨了解码器的常规用例，以及Netty 所提供的用于构建它们的抽象基类。但是解码器只是硬币的一面。硬币的另一面是编码器，它将消息转换为适合于传出传输的格式。这些编码器完备了编解码器API，它们将是我们的下一个主题。

## 10.3 编码器

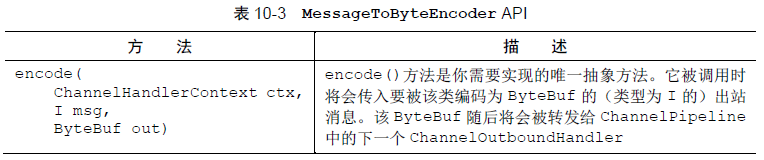
回顾一下我们先前的定义，编码器实现了ChannelOutboundHandler，并将出站数据从一种格式转换为另一种格式，和我们方才学习的解码器的功能正好相反。Netty 提供了一组类，用于帮助你编写具有以下功能的编码器：

* 将消息编码为字节；
* 将消息编码为消息

我们将首先从抽象基类MessageToByteEncoder 开始来对这些类进行考察。

### 10.3.1 抽象类MessageToByteEncoder

前面我们看到了如何使用ByteToMessageDecoder 来将字节转换为消息。现在我们将使用MessageToByteEncoder 来做逆向的事情。表10-3 展示了该API。

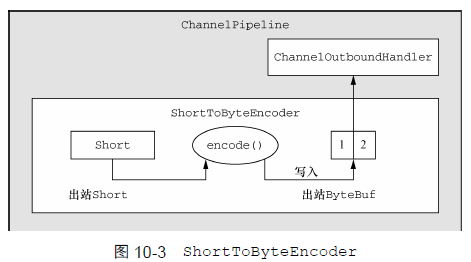


你可能已经注意到了，这个类只有一个方法，而解码器有两个。原因是解码器通常需要在Channel 关闭之后产生最后一个消息（因此也就有了decodeLast()方法）。这显然不适用于编码器的场景——在连接被关闭之后仍然产生一个消息是毫无意义的。

图10-3 展示了ShortToByteEncoder，其接受一个Short 类型的实例作为消息，将它编码为Short 的原子类型值，并将它写入ByteBuf 中，其将随后被转发给ChannelPipeline 中的下一个ChannelOutboundHandler。每个传出的Short 值都将会占用ByteBuf 中的2 字节。ShortToByteEncoder 的实现如代码清单10-5 所示。

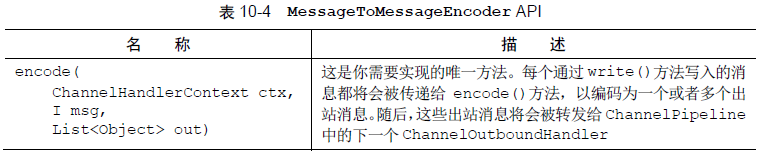
|  |
| --- |
| *// 代码清单10-5 ShortToByteEncoder 类*  *// 扩展了MessageToByteEncoder* **public class** ShortToByteEncoder **extends** MessageToByteEncoder<Short> {  @Override  **public void** encode(ChannelHandlerContext ctx, Short msg, ByteBuf out) **throws** Exception {  *// 将Short 写入ByteBuf 中* out.writeShort(msg);  } } |

Netty 提供了一些专门化的MessageToByteEncoder，你可以基于它们实现自己的编码器。WebSocket08FrameEncoder 类提供了一个很好的实例。你可以在io.netty.handler.codec.http.websocketx 包中找到它。

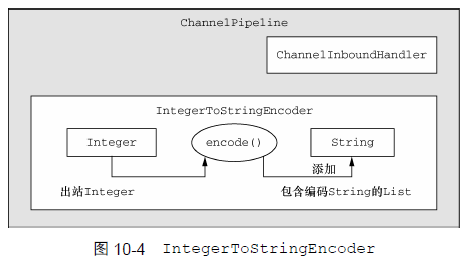


### 10.3.2 抽象类MessageToMessageEncoder

你已经看到了如何将入站数据从一种消息格式解码为另一种。为了完善这幅图，我们将展示对于出站数据将如何从一种消息编码为另一种。MessageToMessageEncoder 类的encode()方法提供了这种能力，如表10-4 所示。



为了演示，代码清单10-6 使用IntegerToStringEncoder 扩展了MessageToMessageEncoder。其设计如图10-4 所示。如代码清单10-6 所示，编码器将每个出站Integer 的String 表示添加到了该List 中。



|  |
| --- |
| *// 代码清单10-6 IntegerToStringEncoder 类*  *// 扩展了MessageToMessageEncoder* **public class** IntegerToStringEncoder **extends** MessageToMessageEncoder<Integer> {  @Override  **public void** encode(ChannelHandlerContext ctx, Integer msg, List<Object> out) **throws** Exception {  *// 将Integer 转换为String，并将其添加到List 中* out.add(String.*valueOf*(msg));  } } |

关于有趣的MessageToMessageEncoder 的专业用法，请查看io.netty.handler.codec.protobuf.ProtobufEncoder 类，它处理了由Google 的Protocol Buffers 规范所定的数据格式。

## 10.4 抽象的编解码器类

虽然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。正如同你可能已经猜想到的，这些类同时实现了ChannelInboundHandler 和ChannelOutboundHandler 接口。

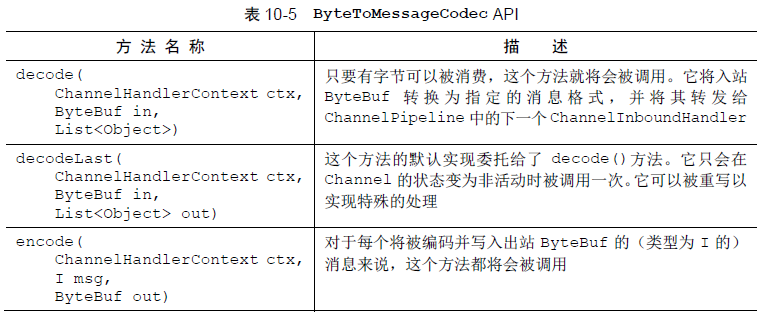
为什么我们并没有一直优先于单独的解码器和编码器使用这些复合类呢？因为通过尽可能地将这两种功能分开，最大化了代码的可重用性和可扩展性，这是Netty 设计的一个基本原则。

在我们查看这些抽象的编解码器类时，我们将会把它们与相应的单独的解码器和编码器进行比较和参照。

### 10.4.1 抽象类ByteToMessageCodec

让我们来研究这样的一个场景：我们需要将字节解码为某种形式的消息，可能是POJO，随后再次对它进行编码。ByteToMessageCodec 将为我们处理好这一切，因为它结合了ByteToMessageDecoder 以及它的逆向——MessageToByteEncoder。表10-5 列出了其中重要的方法。

任何的请求/响应协议都可以作为使用ByteToMessageCodec的理想选择。例如，在某个SMTP的实现中，编解码器将读取传入字节，并将它们解码为一个自定义的消息类型，如SmtpRequest。而在接收端，当一个响应被创建时，将会产生一个SmtpResponse，其将被编码回字节以便进行传输。

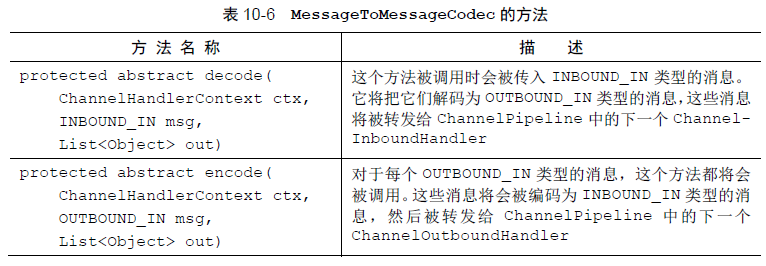


### 10.4.2 抽象类MessageToMessageCodec

在10.3.1 节中，你看到了一个扩展了MessageToMessageEncoder 以将一种消息格式转换为另外一种消息格式的例子。通过使用MessageToMessageCodec，我们可以在一个单个的类中实现该转换的往返过程。MessageToMessageCodec 是一个参数化的类，定义如下：

public abstract class MessageToMessageCodec<INBOUND\_IN,OUTBOUND\_IN>

表10-6 列出了其中重要的方法。



decode() 方法是将INBOUND\_IN 类型的消息转换为OUTBOUND\_IN 类型的消息， 而encode()方法则进行它的逆向操作。将INBOUND\_IN类型的消息看作是通过网络发送的类型，而将OUTBOUND\_IN类型的消息看作是应用程序所处理的类型，将可能有所裨益。

虽然这个编解码器可能看起来有点高深，但是它所处理的用例却是相当常见的：在两种不同的消息API 之间来回转换数据。当我们不得不和使用遗留或者专有消息格式的API 进行互操作时，我们经常会遇到这种模式。

**WebSocket 协议**

下面关于MessageToMessageCodec 的示例引用了一个新出的WebSocket 协议，这个协议能实现Web 浏览器和服务器之间的全双向通信。我们将在第12 章中详细地讨论Netty 对于WebSocket 的支持。

代码清单10-7 展示了这样的对话可能的实现方式。我们的WebSocketConvertHandler在参数化MessageToMessageCodec时将使用INBOUND\_IN类型的WebSocketFrame，以及OUTBOUND\_IN类型的MyWebSocketFrame，后者是WebSocketConvertHandler本身的一个静态嵌套类。

|  |
| --- |
| // 代码清单10-7 使用MessageToMessageCodec  @ChannelHandler.Sharable **public class** WebSocketConvertHandler **extends** MessageToMessageCodec<WebSocketFrame,  WebSocketConvertHandler.MyWebSocketFrame> {  *// 将MyWebSocketFrame 编码为指定的WebSocketFrame子类型* @Override  **protected void** encode(ChannelHandlerContext ctx, WebSocketConvertHandler.MyWebSocketFrame msg,  List<Object> out) **throws** Exception {  *// 实例化一个指定子类型的WebSocketFrame* ByteBuf payload = msg.getData().duplicate().retain();  **switch** (msg.getType()) {  **case *BINARY***:  out.add(**new** BinaryWebSocketFrame(payload));  **break**;  **case *TEXT***:  out.add(**new** TextWebSocketFrame(payload));  **break**;  **case *CLOSE***:  out.add(**new** CloseWebSocketFrame(**true**, **0**, payload));  **break**;  **case *CONTINUATION***:  out.add(**new** ContinuationWebSocketFrame(payload));  **break**;  **case *PONG***:  out.add(**new** PongWebSocketFrame(payload));  **break**;  **case *PING***:  out.add(**new** PingWebSocketFrame(payload));  **break**;  **default**:  **throw new** IllegalStateException(**"Unsupported websocket msg "** + msg);  }  }   *// 将WebSocketFrame 解码为MyWebSocketFrame，并设置FrameType* @Override  **protected void** decode(ChannelHandlerContext ctx, WebSocketFrame msg, List<Object> out) **throws** Exception {  ByteBuf payload = msg.content().duplicate().retain();  **if** (msg **instanceof** BinaryWebSocketFrame) {  out.add(**new** MyWebSocketFrame(  MyWebSocketFrame.FrameType.***BINARY***, payload));  } **else if** (msg **instanceof** CloseWebSocketFrame) {  out.add(**new** MyWebSocketFrame(  MyWebSocketFrame.FrameType.***CLOSE***, payload));  } **else if** (msg **instanceof** PingWebSocketFrame) {  out.add(**new** MyWebSocketFrame(  MyWebSocketFrame.FrameType.***PING***, payload));  } **else if** (msg **instanceof** PongWebSocketFrame) {  out.add(**new** MyWebSocketFrame(  MyWebSocketFrame.FrameType.***PONG***, payload));  } **else if** (msg **instanceof** TextWebSocketFrame) {  out.add(**new** MyWebSocketFrame(  MyWebSocketFrame.FrameType.***TEXT***, payload));  } **else if** (msg **instanceof** ContinuationWebSocketFrame) {  out.add(**new** MyWebSocketFrame(  MyWebSocketFrame.FrameType.***CONTINUATION***, payload));  } **else** {  **throw new** IllegalStateException(  **"Unsupported websocket msg "** + msg);  }  }   *// 声明WebSocketConvertHandler所使用的OUTBOUND\_IN 类型* **public static final class** MyWebSocketFrame {  *// 定义拥有被包装的有效负载的WebSocketFrame的类型* **public enum** FrameType {  ***BINARY***,  ***CLOSE***,  ***PING***,  ***PONG***,  ***TEXT***,  ***CONTINUATION*** }   **private final** FrameType **type**;  **private final** ByteBuf **data**;   **public** MyWebSocketFrame(FrameType type, ByteBuf data) {  **this**.**type** = type;  **this**.**data** = data;  }   **public** FrameType getType() {  **return type**;  }   **public** ByteBuf getData() {  **return data**;  }  } } |

### 10.4.3 CombinedChannelDuplexHandler 类

正如我们前面所提到的，结合一个解码器和编码器可能会对可重用性造成影响。但是，有一种方法既能够避免这种惩罚，又不会牺牲将一个解码器和一个编码器作为一个单独的单元部署所

带来的便利性。CombinedChannelDuplexHandler 提供了这个解决方案，其声明为：

public class CombinedChannelDuplexHandler <I extends ChannelInboundHandler, O extends ChannelOutboundHandler>

这个类充当了ChannelInboundHandler 和ChannelOutboundHandler（该类的类型参数I 和O）的容器。通过提供分别继承了解码器类和编码器类的类型，我们可以实现一个编解码器，而又不必直接扩展抽象的编解码器类。我们将在下面的示例中说明这一点。

首先，让我们研究代码清单10-8 中的ByteToCharDecoder。注意，该实现扩展了ByteToMessageDecoder，因为它要从ByteBuf 中读取字符。

|  |
| --- |
| *// 代码清单10-8 ByteToCharDecoder 类*  *// 扩展了ByteToMessageDecoder* **public class** ByteToCharDecoder **extends** ByteToMessageDecoder {  @Override  **public void** decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) **throws** Exception {  *// 将一个或者多个Character对象添加到传出的List 中* **if** (in.readableBytes() >= **2**) {  out.add(in.readChar());  }  } } |

这里的decode()方法一次将从ByteBuf 中提取2 字节，并将它们作为char 写入到List中，其将会被自动装箱为Character 对象。

代码清单10-9 包含了CharToByteEncoder，它能将Character 转换回字节。这个类扩展了MessageToByteEncoder，因为它需要将char 消息编码到ByteBuf 中。这是通过直接写入ByteBuf 做到的。