Netty In Action中文版 - 第七章：编解码器Codec

目录 [-]

1. 编解码器Codec
2. 解码器
    1. ByteToMessageDecoder
    2. ReplayingDecoder
    3. MessageToMessageDecoder
    4. 解码器总结
3. 编码器
    1. MessageToByteEncoder
    2. MessageToMessageEncoder
4. 编解码器
    1. byte-to-byte编解码器
    2. ByteToMessageCodec
    3. MessageToMessageCodec
5. 其他编解码方式
    1. CombinedChannelDuplexHandler

本章介绍

- Codec，编解码器
- Decoder，解码器
- Encoder，编码器

Netty提供了编解码器框架，使得编写自定义的编解码器很容易，并且也很容易重用和封装。本章讨论Netty的编解码器框架以及使用。

7.1 编解码器Codec

编写一个网络应用程序需要实现某种编解码器，编解码器的作用就是讲原始字节数据与自定义的消息对象进行互转。网络中都是以字节码的数据形式来传输数据的，服务器编码数据后发送到客户端，客户端需要对数据进行解码，因为编解码器由两部分组成：

- Decoder(解码器)
- Encoder(编码器)

解码器负责将消息从字节或其他序列形式转成指定的消息对象，编码器则相反；解码器负责处理“入站”数据，编码器负责处理“出站”数据。编码器和解码器的结构很简单，消息被编码后解码后会自动通过ReferenceCountUtil.release(message)释放，如果不想释放消息可以使用ReferenceCountUtil.retain(message)，这将会使引用数量增加而没有消息发布，大多数时候不需要这么做。

7.2 解码器

Netty提供了丰富的解码器抽象基类，我们可以很容易的实现这些基类来自定义解码器。下面是解码器的一个类型：

- 解码字节到消息
- 解码消息到消息
- 解码消息到字节

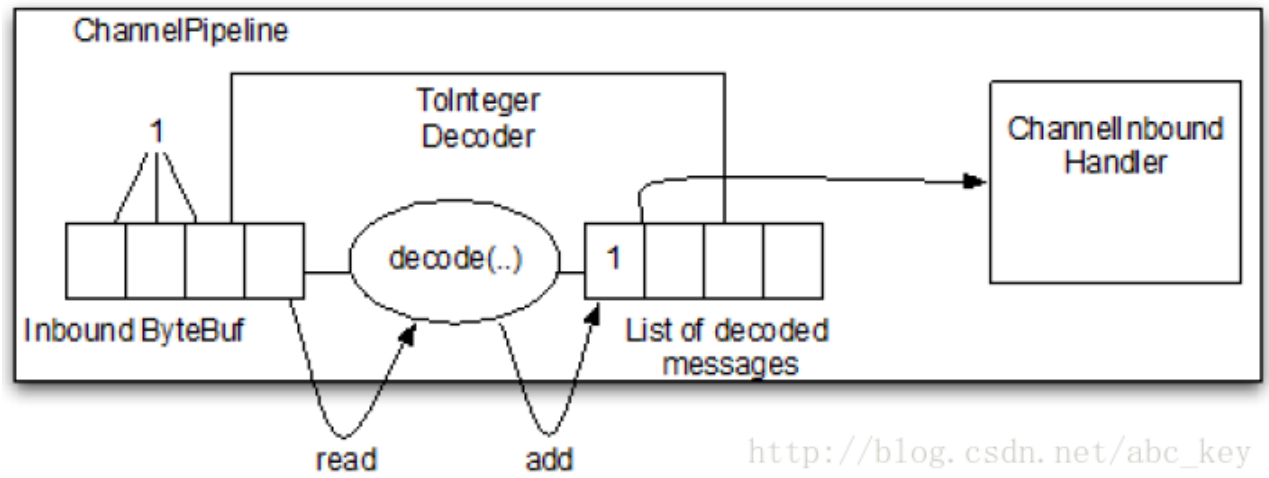
本章将概述不同的抽象基类，来帮助了解解码器的实现。深入了解Netty提供的解码器之前先了解解码器的作用是什么？解码器负责解码“入站”数据从一种格式到另一种格式，解码器处理入站数据是抽象ChannelInboundHandler的实现。实践中使用解码器很简单，就是将入站数据转换格式后传递到ChannelPipeline中的下一个ChannelInboundHandler进行处理；这样的处理时很灵活的，我们可以将解码器放在ChannelPipeline中，重用逻辑。

7.2.1 ByteToMessageDecoder

通常你需要将消息从字节解码成消息或者从字节解码成其他的序列化字节。这是一个常见的任务，Netty提供了抽象基类，我们可以使用它们来实现。Netty中提供的ByteToMessageDecoder可以将字节消息解码成POJO对象，下面列出了ByteToMessageDecoder两个主要方法：

- decode(ChannelHandlerContext, ByteBuf, List<Object>)，这个方法是唯一的一个需要自己实现的抽象方法，作用是将ByteBuf数据解码成其他形式的数据。
- decodeLast(ChannelHandlerContext, ByteBuf, List<Object>)，实际上调用的是decode(...)。

例如服务器从某个客户端接收到一个整数值的字节码，服务器将数据读入ByteBuf并经过ChannelPipeline中的每个ChannelInboundHandler进行处理，看下图：



上图显示了从“入站”ByteBuf读取bytes后由ToIntegerDecoder进行解码，然后向解码后的消息传递到ChannelPipeline中的下一个ChannelInboundHandler。看下面ToIntegerDecoder的实现代码：

```java
/**
 * Integer解码器,ByteToMessageDecoder实现
 * @author c.k
 *
 */
public class ToIntegerDecoder extends ByteToMessageDecoder {

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
        if(in.readableBytes() >= 4){
            out.add(in.readInt());
        }
    }
}
```

从上面的代码可能会发现，我们需要检查ByteBuf读之前是否有足够的字节，若没有这个检查岂不更好？是的，Netty提供了这样的处理允许byte-to-message解码,在下一节讲解。除了ByteToMessageDecoder之外，Netty还提供了许多其他的解码接口。

7.2.2 ReplayingDecoder

ReplayingDecoder是byte-to-message解码的一种特殊的抽象基类，读取缓冲区的数据之前需要检查缓冲区是否有足够的字节，使用ReplayingDecoder就无需自己检查；若ByteBuf中有足够的字节，则会正常读取；若没有足够的字节则会停止解码。也正因为这样的包装使得ReplayingDecoder带有一定的局限性。

- 不是所有的操作都被ByteBuf支持，如果调用一个不支持的操作会抛出DecoderException。
- ByteBuf.readableBytes()大部分时间不会返回期望值

如果你能忍受上面列出的限制，相比ByteToMessageDecoder，你可能更喜欢ReplayingDecoder。在满足需求的情况下推荐使用ByteToMessageDecoder，因为它的处理比较简单，没有ReplayingDecoder实现的那么复杂。ReplayingDecoder继承与ByteToMessageDecoder，所以他们提供的接口是相同的。下面代码是ReplayingDecoder的实现：

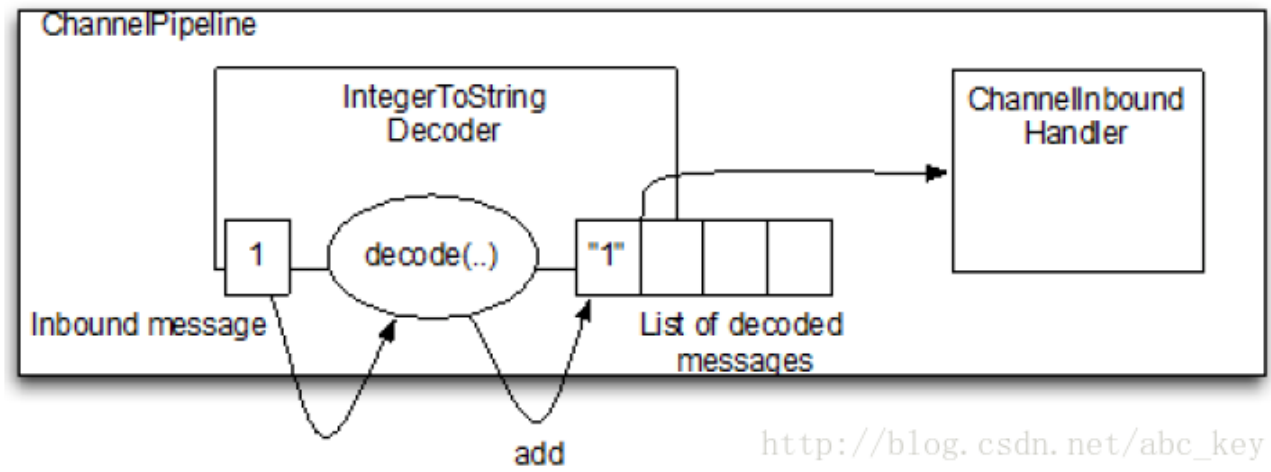
```java
/**
 * Integer解码器,ReplayingDecoder实现
 * @author c.k
 *
 */
public class ToIntegerReplayingDecoder extends ReplayingDecoder<Void> {

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
        out.add(in.readInt());
    }
}
```

当从接收的数据ByteBuf读取integer，若没有足够的字节可读，decode(...)会停止解码，若有足够的字节可读，则会读取数据添加到List列表中。使用ReplayingDecoder或ByteToMessageDecoder是个人喜好的问题，Netty提供了这两种实现，选择哪一个都可以。

上面讲了byte-to-message的解码实现方式，那message-to-message该如何实现呢？Netty提供了MessageToMessageDecoder抽象类。

7.2.3 MessageToMessageDecoder

将消息对象转成消息对象可是使用MessageToMessageDecoder，它是一个抽象类，需要我们自己实现其decode(...)。message-to-message同上面讲的byte-to-message的处理机制一样，看下图：



看下面的实现代码：

```java
/**
 * 将接收的Integer消息转成String类型，MessageToMessageDecoder实现
 * @author c.k
 *
 */
public class IntegerToStringDecoder extends MessageToMessageDecoder<Integer> {

    @Override
    protected void decode(ChannelHandlerContext ctx, Integer msg, List<Object> out) throws Exception {
        out.add(String.valueOf(msg));
    }
}
```

7.2.4 解码器总结

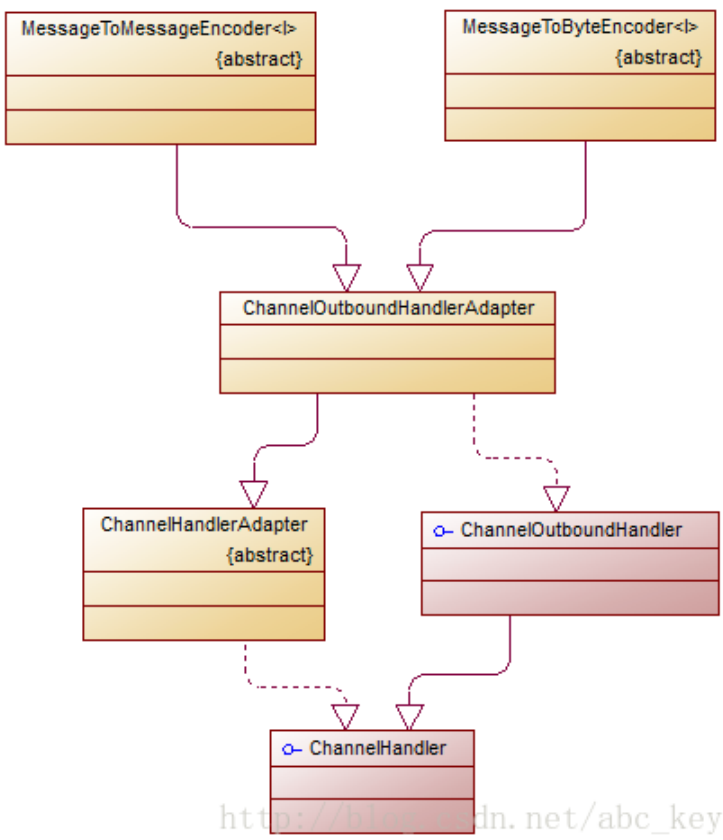
解码器是用来处理入站数据，Netty提供了很多解码器的实现，可以根据需求详细了解。那我们发送数据需要将数据编码，Netty中也提供了编码器的支持。下一节将讲解如何实现编码器。

7.3 编码器

Netty提供了一些基类，我们可以很简单的编码器。同样的，编码器有下面两种类型：

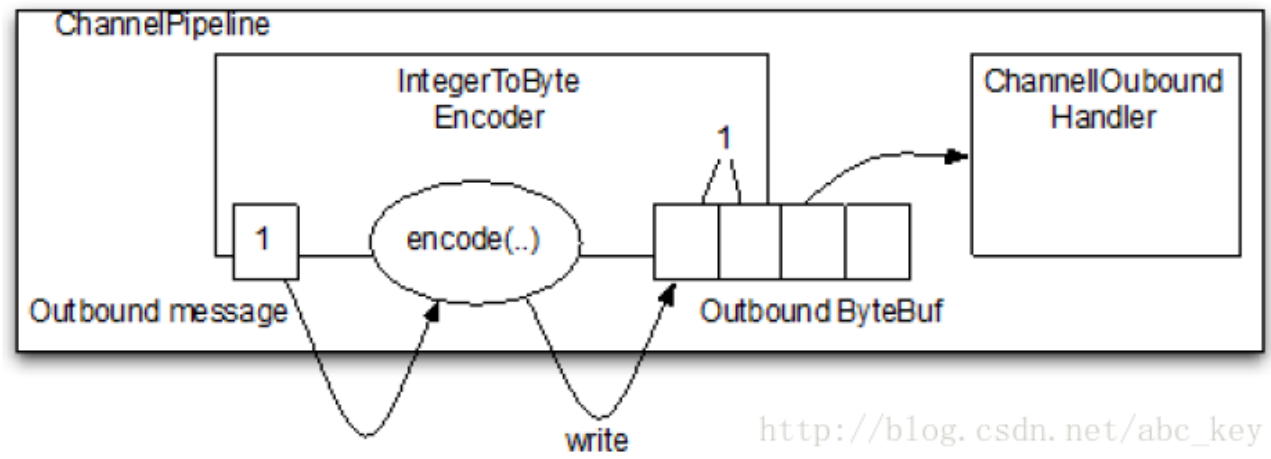
- 消息对象编码成消息对象
- 消息对象编码成字节码

相对解码器，编码器少了一个byte-to-byte的类型，因为出站数据这样做没有意义。编码器的作用就是将处理好的数据转成字节码以便在网络中传输。对照上面列出的两种编码器类型，Netty也分别提供了两个抽象类：MessageToByteEncoder和MessageToMessageEncoder。下面是类关系图：



7.3.1 MessageToByteEncoder

MessageToByteEncoder是抽象类，我们自定义一个继承MessageToByteEncoder的编码器只需要实现其提供的encode(...)方法。其工作流程如下图：

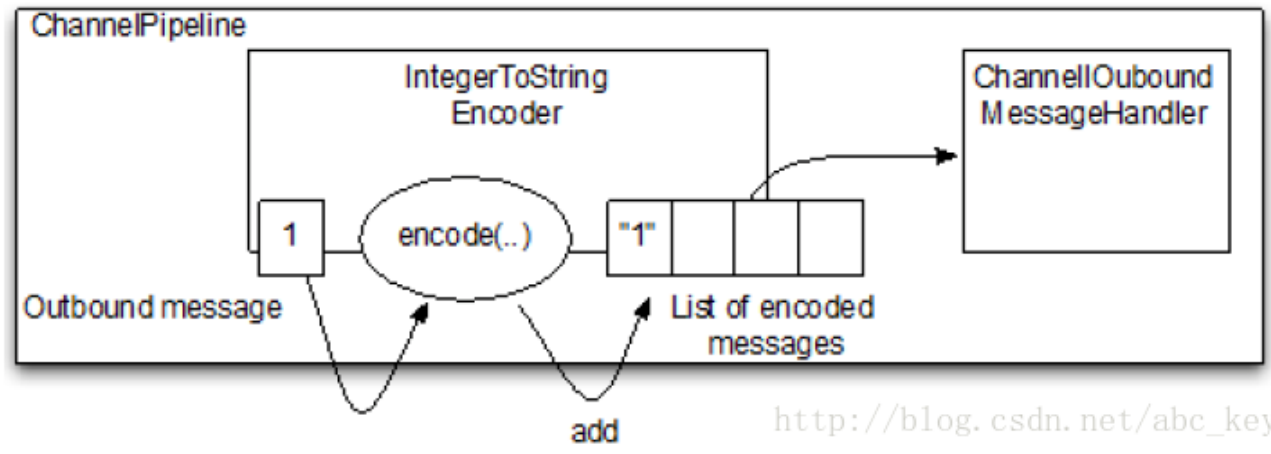


实现代码如下：

```java
/**
 * 编码器，将Integer值编码成byte[]，MessageToByteEncoder实现
 * @author c.k
 *
 */
public class IntegerToByteEncoder extends MessageToByteEncoder<Integer> {
    @Override
    protected void encode(ChannelHandlerContext ctx, Integer msg, ByteBuf out) throws Exception {
        out.writeInt(msg);
    }
}
```

7.3.2 MessageToMessageEncoder

需要将消息编码成其他的消息时可以使用Netty提供的MessageToMessageEncoder抽象类来实现。例如将Integer编码成String，其工作流程如下图：



代码实现如下：

```java
/**
 * 编码器，将Integer编码成String，MessageToMessageEncoder实现
 * @author c.k
 *
 */
public class IntegerToStringEncoder extends MessageToMessageEncoder<Integer> {

    @Override
    protected void encode(ChannelHandlerContext ctx, Integer msg, List<Object> out) throws Exception {
        out.add(String.valueOf(msg));
    }
}
```

7.4 编解码器

实际编码中，一般会将编码和解码操作封装太一个类中，解码处理“入站”数据，编码处理“出站”数据。知道了编码和解码器，对于下面的情况不会感觉惊讶：

- byte-to-message编码和解码
- message-to-message编码和解码

如果确定需要在ChannelPipeline中使用编码器和解码器，需要更好的使用一个抽象的编解码器。同样，使用编解码器的时候，不可能只删除解码器或编码器而离开ChannelPipeline导致某种不一致的状态。使用编解码器将强制性的要么都在ChannelPipeline，要么都不

在ChannelPipeline。

考虑到这一点，我们在下面几节将更深入的分析Netty提供的编解码抽象类。

7.4.1 byte-to-byte编解码器

Netty4较之前的版本，其结构有很大的变化，在Netty4中实现byte-to-byte提供了2个类：ByteArrayEncoder和ByteArrayDecoder。这两个类用来处理字节到字节的编码和解码。下面是这两个类的源码，一看就知道是如何处理的：

```java
public class ByteArrayDecoder extends MessageToMessageDecoder<ByteBuf> {
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) throws Exception {
         // copy the ByteBuf content to a byte array
        byte[] array = new byte[msg.readableBytes()];
        msg.getBytes(0, array);

        out.add(array);
    }
}
```

```java
@Sharable
public class ByteArrayEncoder extends MessageToMessageEncoder<byte[]> {
    @Override
    protected void encode(ChannelHandlerContext ctx, byte[] msg, List<Object> out) throws Exception {
        out.add(Unpooled.wrappedBuffer(msg));
    }
}
```

7.4.2 ByteToMessageCodec

ByteToMessageCodec用来处理byte-to-message和message-to-byte。如果想要解码字节消息成POJO或编码POJO消息成字节，对于这种情况，ByteToMessageCodec<I>是一个不错的选择。ByteToMessageCodec是一种组合，其等同于ByteToMessageDecoder和MessageToByteEncoder的组合。MessageToByteEncoder是个抽象类，其中有2个方法需要我们自己实现：

- encode(ChannelHandlerContext, I, ByteBuf)，编码
- decode(ChannelHandlerContext, ByteBuf, List<Object>)，解码

7.4.3 MessageToMessageCodec

MessageToMessageCodec用于message-to-message的编码和解码，可以看成是MessageToMessageDecoder和MessageToMessageEncoder的组合体。MessageToMessageCodec是抽象类，其中有2个方法需要我们自己实现：

- encode(ChannelHandlerContext, OUTBOUND_IN, List<Object>)
- decode(ChannelHandlerContext, INBOUND_IN, List<Object>)

但是，这种编解码器能有用吗？

有许多用例，最常见的就是需要将消息从一个API转到另一个API。这种情况下需要自定义API或旧的API使用另一种消息类型。下面的代码显示了在WebSocket框架APIs之间转换消息：

```java
package netty.in.action;

import java.util.List;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelHandler.Sharable;
import io.netty.handler.codec.MessageToMessageCodec;
import io.netty.handler.codec.http.websocketx.BinaryWebSocketFrame;
import io.netty.handler.codec.http.websocketx.CloseWebSocketFrame;
import io.netty.handler.codec.http.websocketx.ContinuationWebSocketFrame;
import io.netty.handler.codec.http.websocketx.PingWebSocketFrame;
import io.netty.handler.codec.http.websocketx.PongWebSocketFrame;
import io.netty.handler.codec.http.websocketx.TextWebSocketFrame;
import io.netty.handler.codec.http.websocketx.WebSocketFrame;

@Sharable
public class WebSocketConvertHandler extends
        MessageToMessageCodec<WebSocketFrame, WebSocketConvertHandler.MyWebSocketFrame> {

    public static final WebSocketConvertHandler INSTANCE = new WebSocketConvertHandler();

```

```java
    @Override
    protected void encode(ChannelHandlerContext ctx, MyWebSocketFrame msg, List<Object> out) throws Exception {
        switch (msg.getType()) {
        case BINARY:
            out.add(new BinaryWebSocketFrame(msg.getData()));
            break;
        case CLOSE:
            out.add(new CloseWebSocketFrame(true, 0, msg.getData()));
            break;
        case PING:
            out.add(new PingWebSocketFrame(msg.getData()));
            break;
        case PONG:
            out.add(new PongWebSocketFrame(msg.getData()));
            break;
        case TEXT:
            out.add(new TextWebSocketFrame(msg.getData()));
            break;
        case CONTINUATION:
            out.add(new ContinuationWebSocketFrame(msg.getData()));
            break;
        default:
            throw new IllegalStateException("Unsupported websocket msg " + msg);
        }
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, WebSocketFrame msg, List<Object> out) throws Exception {
        if (msg instanceof BinaryWebSocketFrame) {
            out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.BINARY, msg.content().copy()));
            return;
        }
        if (msg instanceof CloseWebSocketFrame) {
            out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.CLOSE, msg.content().copy()));
            return;
        }
        if (msg instanceof PingWebSocketFrame) {
            out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.PING, msg.content().copy()));
            return;
        }
        if (msg instanceof PongWebSocketFrame) {
            out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.PONG, msg.content().copy()));
            return;
        }
        if (msg instanceof TextWebSocketFrame) {
            out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.TEXT, msg.content().copy()));
            return;
        }
        if (msg instanceof ContinuationWebSocketFrame) {
            out.add(new MyWebSocketFrame(MyWebSocketFrame.FrameType.CONTINUATION, msg.content().copy()));
            return;
        }
        throw new IllegalStateException("Unsupported websocket msg " + msg);
    }

    public static final class MyWebSocketFrame {
        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;
        }

    }
}
```

7.5 其他编解码方式

使用编解码器来充当编码器和解码器的组合失去了单独使用编码器或解码器的灵活性，编解码器是要么都有要么都没有。你可能想知道是否有解决这个僵化问题的方式，还可以让编码器和解码器在ChannelPipeline中作为一个逻辑单元。幸运的是，Netty提供了一种解决方案，使用CombinedChannelDuplexHandler。虽然这个类不是编解码器API的一部分，但是它经常被用来简历一个编解码器。

7.5.1 CombinedChannelDuplexHandler

如何使用CombinedChannelDuplexHandler来结合解码器和编码器呢？下面我们从两个简单的例子看了解。

```java
/**
 * 解码器，将byte转成char
 * @author c.k
 *
 */
public class ByteToCharDecoder extends ByteToMessageDecoder {

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
        while(in.readableBytes() >= 2){
            out.add(Character.valueOf(in.readChar()));
        }
    }

}
```

```java
/**
 * 编码器，将char转成byte
 * @author Administrator
 *
 */
public class CharToByteEncoder extends MessageToByteEncoder<Character> {

    @Override
    protected void encode(ChannelHandlerContext ctx, Character msg, ByteBuf out) throws Exception {
        out.writeChar(msg);
    }
}
```

```java
/**
 * 继承CombinedChannelDuplexHandler，用于绑定解码器和编码器
 * @author c.k
 *
 */
public class CharCodec extends CombinedChannelDuplexHandler<ByteToCharDecoder, CharToByteEncoder> {
    public CharCodec(){
        super(new ByteToCharDecoder(), new CharToByteEncoder());
    }
}
```

从上面代码可以看出，使用CombinedChannelDuplexHandler绑定解码器和编码器很容易实现，比使用*Codec更灵活。Netty还提供了其他的协议支持，放在io.netty.handler.codec包下，如：

- Google的protobuf，在io.netty.handler.codec.protobuf包下
- Google的SPDY协议
- RTSP(Real Time Streaming Protocol，实时流传输协议)，在io.netty.handler.codec.rtsp包下
- SCTP(Stream Control Transmission Protocol，流控制传输协议)，在io.netty.handler.codec.sctp包下
- ......