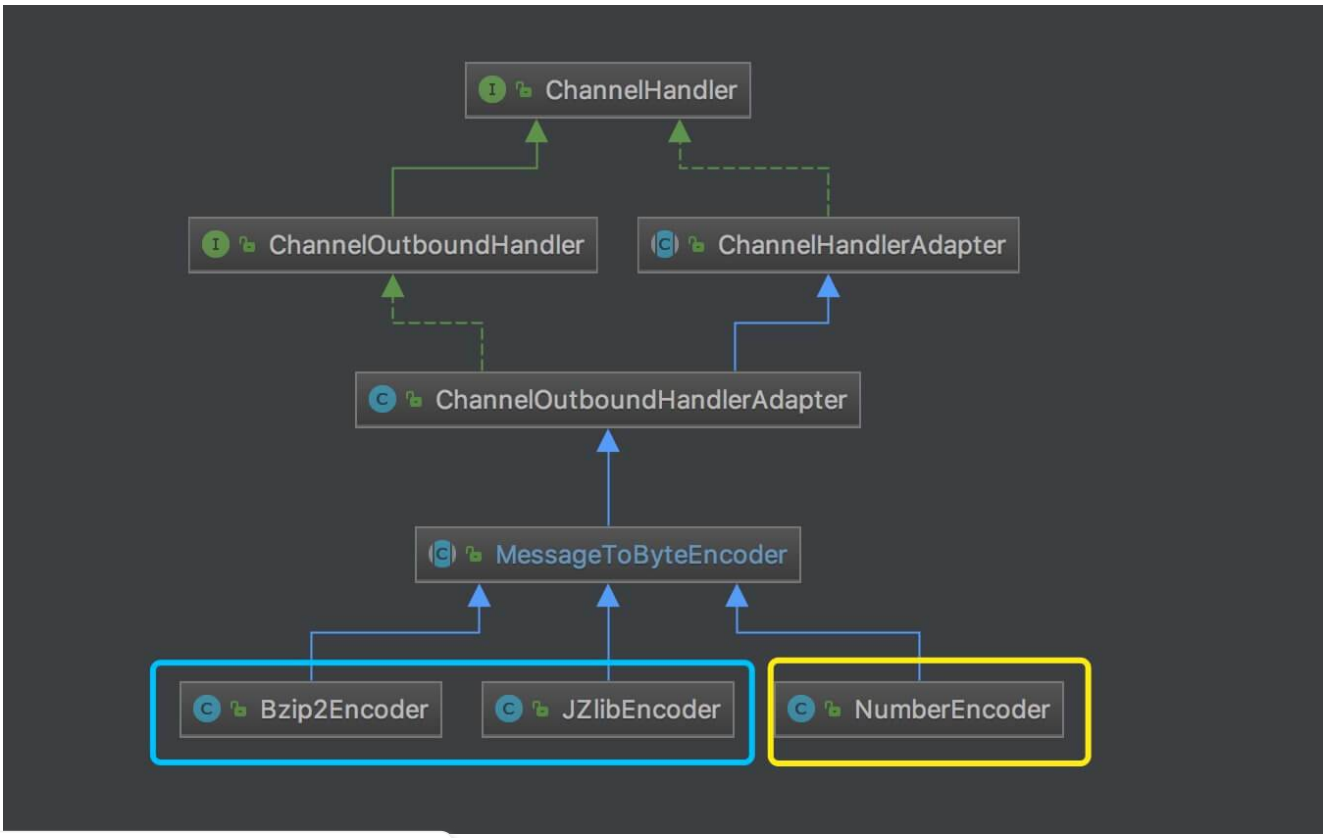
- NETTY

精尽 Netty 源码解析 —— Codec 之 MessageToByteEncoder

1. 概述

本文，我们来分享 MessageToByteEncoder 部分的内容。

MessageToByteEncoder 负责将消息**编码**成字节。核心类图如下：



文章目录

其下有多个子类，笔者简单整理成两类，可能不全哈：

缩算法，例如：GZip、BZip 等等。
转化好的字节流，进一步压缩。
列化成字节。例如：JSON、XML 等等。
等相关的类库，所以不好提供类似 JSONEncoder 或 XMLEncoder ，所以图
的 NumberEncoder 。

yteToMessageDecoder（一）Cumulator》 中，我们提到**粘包拆包**的现象，所以在实际使用 Netty 编码消息的，还需要有为了解决**粘包拆包**的 Encoder 实现类，例如：换行、定长等等方式。关于这块内容，胖友可以看看 《netty使用MessageToByteEncoder 自定义协议》 。

2. MessageToByteEncoder

`io.netty.handler.codec.MessageToByteEncoder` ，继承 ChannelOutboundHandlerAdapter 类，负责将消息**编码**成字节，支持**匹配指定类型**的消息。

2.1 构造方法

```
public abstract class MessageToByteEncoder<I> extends ChannelOutboundHandlerAdapter {

    /**
     * 类型匹配器
     */
    private final TypeParameterMatcher matcher;
    /**
     * 是否偏向使用 Direct 内存
     */
    private final boolean preferDirect;

    protected MessageToByteEncoder() {
        this(true);
    }

    protected MessageToByteEncoder(Class<? extends I> outboundMessageType) {
        this(outboundMessageType, true);
    }

    protected MessageToByteEncoder(boolean preferDirect) {
        // <1> 获得 matcher
        matcher = TypeParameterMatcher.find(this, MessageToByteEncoder.class, "I");
        this.preferDirect = preferDirect;
    }

    protected MessageToByteEncoder(Class<? extends I> outboundMessageType, boolean preferDirect) {
        // <2> 获得 matcher
        matcher = TypeParameterMatcher.get(outboundMessageType);
        this.preferDirect = preferDirect;
    }

    // ... 省略其他无关代码
```


型对应的 TypeParameterMatcher 类型匹配器。

pe 参数对应的 TypeParameterMatcher 类型匹配器。

详细的了解 `io.netty.util.internal.TypeParameterMatcher` 的代码

netty 简单Inbound通道处理器（SimpleChannelInboundHandler）》 的

irect 内存。默认为 `true` 。

2.2 acceptInboundMessage

`#acceptInboundMessage(Object msg)` 方法，判断消息是否匹配。代码如下：

```
/**
 * Returns {@code true} if the given message should be handled. If {@code false} it will be passed to
 * {@link ChannelInboundHandler} in the {@link ChannelPipeline}.
 */
public boolean acceptInboundMessage(Object msg) {
    return matcher.match(msg);
}
```

一般情况下， `matcher` 的类型是 ReflectiveMatcher(它是 TypeParameterMatcher 的内部类)。代码如下：

```
private static final class ReflectiveMatcher extends TypeParameterMatcher {

    /**
     * 类型
     */
    private final Class<?> type;

    ReflectiveMatcher(Class<?> type) {
        this.type = type;
    }

    @Override
    public boolean match(Object msg) {
        return type.isInstance(msg); // <1>
    }

}
```

- 匹配逻辑，看 `<1>` 处，使用 `Class#isInstance(Object obj)` 方法。对于这个方法，如果我们定义的 `I` 泛型是个父类，那可以匹配所有的子类。例如 `I` 设置为 Object 类，那么所有消息，都可以被匹配列。

2.3 write

`#write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)` 方法，匹配指定的消息类型，编码消息成 ByteBuf 对象，继续写到下一个节点。代码如下：

```
 1: @Override
 2: public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception
```

文章目录


```
                age(msg)) {
                "unchecked")


                er(ctx, cast, preferDirect);

13:                 encode(ctx, cast, buf);
14:             } finally {
15:                 // 释放 msg
16:                 ReferenceCountUtil.release(cast);
```

```
17:             }
18:
19:             // buf 可读，说明有编码到数据
20:             if (buf.isReadable()) {
21:                 // 写入 buf 到下一个节点
22:                 ctx.write(buf, promise);
23:             } else {
24:                 // 释放 buf
25:                 buf.release();
26:                 // 写入 EMPTY_BUFFER 到下一个节点，为了 promise 的回调
27:                 ctx.write(Unpooled.EMPTY_BUFFER, promise);
28:             }
29:
30:             // 置空 buf
31:             buf = null;
32:         } else {
33:             // 提交 write 事件给下一个节点
34:             ctx.write(msg, promise);
35:         }
36:     } catch (EncoderException e) {
37:         throw e;
38:     } catch (Throwable e) {
39:         throw new EncoderException(e);
40:     } finally {
41:         // 释放 buf
42:         if (buf != null) {
43:             buf.release();
44:         }
45:     }
46: }
```

- 第 6 行：调用 `#acceptInboundMessage(Object msg)` 方法，判断是否为匹配的消息。
- ① 第 6 行：**匹配。**
  - 第 8 行：对象类型转化为 `I` 类型的消息。
  - 第 10 行：调用 `#allocateBuffer(ChannelHandlerContext ctx, I msg, boolean preferDirect)` 方法，申请 `buf` 。代码如下：

```
which will be used as argument of {@link #encode(ChannelHandler
is method to return {@link ByteBuf} with a perfect matching {@c

er(ChannelHandlerContext ctx, @SuppressWarnings("unused") I msg

uffer();

pBuffer();

}
```

    - x
  - 第 13 行：调用 `#encode(ChannelHandlerContext ctx, I msg, ByteBuf out)` 方法，编码。代码如下：

文章目录

```
/**
 * Encode a message into a {@link ByteBuf}. This method will be called for each written messa
 * by this encoder.
 *
 * @param ctx           the {@link ChannelHandlerContext} which this {@link MessageToByteEnco
 * @param msg           the message to encode
 * @param out           the {@link ByteBuf} into which the encoded message will be written
 * @throws Exception    is thrown if an error occurs
 */
protected abstract void encode(ChannelHandlerContext ctx, I msg, ByteBuf out) throws Exceptio
```

  - 子类可以实现该方法，实现自定义的编码功能。
- 第 16 行：调用 `ReferenceCountUtil#release(Object msg)` 方法，释放 `msg` 。
- 第 19 至 22 行： `buf` 可读，说明编码消息到 `buf` 中了，所以写入 `buf` 到下一个节点。😈 因为 `buf` 需要继续被下一个节点使用，所以不进行释放。
- 第 23 至 28 行： `buf` 不可读，说明无法编码，所以释放 `buf` ，并写入 `EMPTY_BUFFER` 到下一个节点，为了 promise 的回调。
- 第 31 行：置空 `buf` 为空。这里是为了防止【第 41 至 44 行】的代码，释放 `buf` 。
- ② 第 32 行：**不匹配。**
  - 提交 write 事件给下一个节点。
- 第 36 至 39 行：发生异常，抛出 EncoderException 异常。
- 第 40 至 45 行：如果中间发生异常，导致 `buf` 不为空，所以此处释放 `buf` 。

3. NumberEncoder

`io.netty.example.factorial.NumberEncoder` ，继承 MessageToByteEncoder 抽象类，Number 类型的消息的 Encoder 实现类。代码如下：

> NumberEncoder 是 `netty-example` 模块提供的示例类，实际使用时，需要做调整。

```
public class NumberEncoder extends MessageToByteEncoder<Number> {

    @Override
    protected void encode(ChannelHandlerContext ctx, Number msg, ByteBuf out) {
        // Check the type.
        if (!(msg instanceof BigInteger)) {
            throw new IllegalStringException(
                    msg.getClass().getName() +
                    " is not a " + BigInteger.class.getSimpleName() +
                    " supported by " + getClass().getSimpleName());
        }

        // Convert to a BigInteger first for easier implementation.
        BigInteger v = (BigInteger) msg;

        // Convert the number into a byte array.
        byte[] data = v.toByteArray();
        int dataLength = data.length;

        // Write a message.
        out.writeInt(dataLength);
        out.writeBytes(data);
    }
}
```

```
        byte[] data = v.toByteArray();
        int dataLength = data.length;

        // <3> Write a message.
        out.writeByte((byte) 'F'); // magic number
        out.writeInt(dataLength);  // data length
        out.writeBytes(data);      // data
    }

}
```

- `<1>` 处，转化消息类型为 BigInteger 对象，方便统一处理。
- `<2>` 处，转化为字节数组。
- `<3>` 处
  - 首位，写入 magic number ，方便区分**不同类型**的消息。例如说，后面如果有 Double 类型，可以使用 `D` ；String 类型，可以使用 `S` 。
  - 后两位，写入 data length + data 。如果没有 data length ，那么数组内容，是无法读取的。

实际一般不采用 NumberEncoder 的方式，因为 POJO 类型不好支持。关于这一块，可以参看下：

- Dubbo
- Motan
- Sofa-RPC

对 Encoder 和 Codec 真正实战。hoho

666. 彩蛋

MessageToByteEncoder 相比 ByteToMessageDecoder 来说，简单好多。

推荐阅读文章：

- Hypercube 《自顶向下深入分析Netty（八）–CodecHandler》

另外，可能很多胖友，看完 Encoder 和 Decoder ，还是一脸懵逼，不知道实际如何使用。可以在网络上，再 Google 一些资料，不要方，不要怕。