3

第 3 篇

Netty 核心篇

11

第11章

Netty编解码的艺术

课程目标

1、。

2、。

3、。

内容定位

1.。

2.。

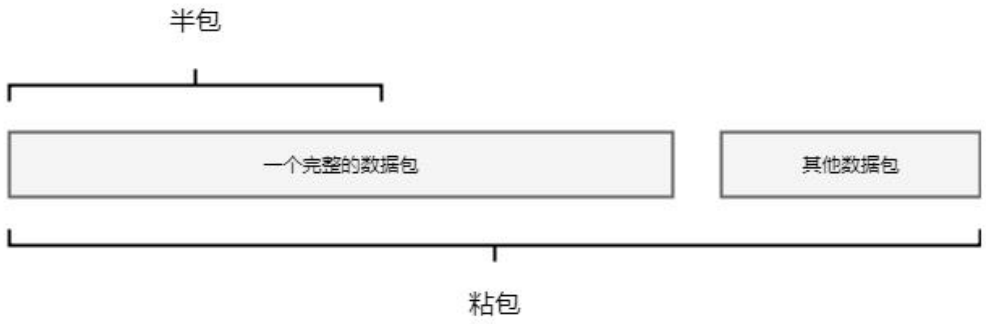
在前面的章节有一个遗留问题，就是如果 Server 在读取客户端的数据的时候，如果一次读取不完整，就触发 channelRead 事件，那么 Netty 是如何处理这类问题的，本章会对此做详细剖析。

11.1 什么是拆包/粘包

11.1.1 TCP 粘包/拆包

TCP 是一个"流"协议，所谓流，就是没有界限的一长串二进制数据。TCP 作为传输层协议并不不了解上层业务数据的具体含义，它会根据 TCP 缓冲区的实际情况进行数据包的划分，所以在业务上认为是一个完整的包，可能会被 TCP 拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的 TCP 粘包和拆包问题。

同样，在 Netty 的编码器中，也会对半包和粘包问题做相应的处理。什么是半包，顾名思义，就是不完整的数据包，因为 Betty 在轮询读事件的时候，每次将 channel 中读取的数据，不一定是一个完整的数据包，这种情况，就叫半包。粘包同样也不难理解，如果 Client 往 Server 发送数据包，如果发送频繁很有可能会将多个数据包的数据都发送到通道中，如果在 server 在读取的时候可能会读取到超过一个完整数据包的长度，这种情况叫粘包。有关半包和粘包，如下图所示：



11.1.2 粘包问题的解决策略

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。业界的主流协议的解决方案，可以归纳如下：

1. 消息定长，报文大小固定长度，例如每个报文的长度固定为 200 字节，如果不够空位补空格；

2. 包尾添加特殊分隔符，例如每条报文结束都添加回车换行符（例如 FTP 协议）或者指定特殊字符作为报文分隔符，接收方通过特殊分隔符切分报文区分；

3. 将消息分为消息头和消息体，消息头中包含表示信息的总长度（或者消息体长度）的字段；

4. 更复杂的自定义应用层协议。

Netty 对半包的或者粘包的处理其实也很简单，通过之前的学习，我们知道，每个 handler 是和 channel 唯一绑定的，一个 handler 只对应一个 channel，所以将 channel 中的数据读取时候经过解析，如果不是一个完整的数据包，则解析失败，将这块数据包进行保存，等下次解析时再和这个数据包进行组装解析，直到解析到完整的数据包，才会将数据包进行向下传递。

11.2 什么是编码和解码

11.2.1 编、解码技术

通常我们也习惯将编码（Encode）称为序列化（serialization），它将对象序列化为字节数组，用于网络传输、数据持久化或者其它用途。反之，解码（Decode）/反序列化（deserialization）把从网络、磁盘等读取的字节数组还原成原始对象（通常是原始对象的拷贝），以方便后续的业务逻辑操作。进行远程跨进程服务调用时（例如 RPC 调用），需要使用特定的编解码技术，对需要进行网络传输的对象做编码或者解码，以便完成远程调用。

11.2.2 Netty 为什么要提供编解码框架？

作为一个高性能的异步、NIO 通信框架，编解码框架是 Netty 的重要组成部分。尽管站在微内核的角度看，编解码框架并不是 Netty 微内核的组成部分，但是通过 ChannelHandler 定制扩展出的编解码框架却是不可或缺的。

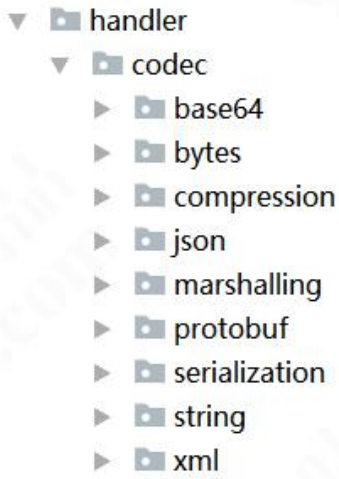
然而，我们已经知道在 Netty 中，从网络读取的 Inbound 消息，需要经过解码，将二进制的数据报转换成应用层协议消息或者业务消息，才能够被上层的应用逻辑识别和处理；同理，用户发送到网络的 Outbound 业务消息，需要经过

编码转换成二进制字节数组（对于 Netty 就是 ByteBuf）才能够发送到网络对端。编码和解码功能是 NIO 框架的有机组成部分，无论是由业务定制扩展实现，还是 NIO 框架内置编解码能力，该功能是必不可少的。

为了降低用户的开发难度，Netty 对常用的功能和 API 做了装饰，以屏蔽底层的实现细节。编解码功能的定制，对于熟悉 Netty 底层实现的开发者而言，直接基于 ChannelHandler 扩展开发，难度并不是很大。但是对于大多数初学者或者不愿意去了解底层实现细节的用户，需要提供给他们更简单的类库和 API，而不是 ChannelHandler。

Netty 在这方面做得非常出色，针对编解码功能，它既提供了通用的编解码框架供用户扩展，又提供了常用的编解码类库供用户直接使用。在保证定制扩展性的基础之上，尽量降低用户的开发工作量和开发门槛，提升开发效率。

Netty 预置的编解码功能列表如下：Base64、Protobuf、JBoss Marshalling、Spdy 等。



11.3 Netty 中常用的解码器

Netty 默认提供了多个解码器，可以进行分包的操作，满足 99%的编码需求。

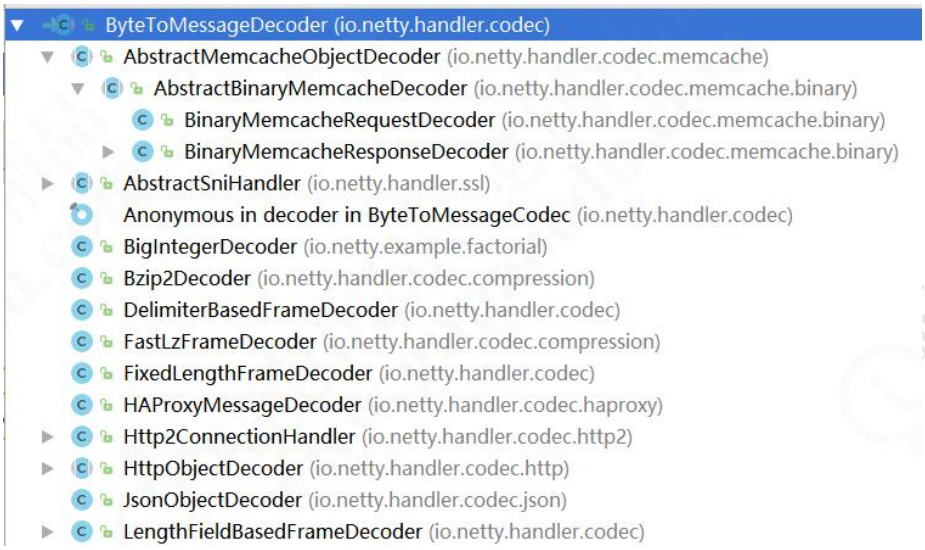
11.3.1 ByteToMessageDecoder 抽象解码器

使用 NIO 进行网络编程时，往往需要将读取到的字节数组或者字节缓冲区解码为业务可以使用的 POJO 对象。为了方便业务将 ByteBuf 解码成业务 POJO 对象，Netty 提供了 ByteToMessageDecoder 抽象工具解码类。

用户自定义解码器继承 ByteToMessageDecoder，只需要实现 void decode（ChannelHandler Context ctx, ByteBuf in,

List<Object> out）抽象方法即可完成 ByteBuf 到 POJO 对象的解码。

由于 ByteToMessageDecoder 并没有考虑 TCP 粘包和拆包等场景，用户自定义解码器需要自己处理"读半包"问题。正因为如此，大多数场景不会直接继承 ByteToMessageDecoder，而是继承另外一些更高级的解码器来屏蔽半包的处理。实际项目中，通常将 LengthFieldBasedFrameDecoder 和 ByteToMessageDecoder 组合使用，前者负责将网络读取的数据报解码为整包消息，后者负责将整包消息解码为最终的业务对象。除了和其它解码器组合形成新的解码器之外，ByteToMessageDecoder 也是很多基础解码器的父类，它的继承关系如下图所示：



下面我们来看源码，ByteToMessageDecoder 类的定义：

```
public abstract class ByteToMessageDecoder extends ChannelInboundHandlerAdapter{
    //类体省略
}
```

从源码中可以看出，ByteToMessageDecoder 继承了 ChannelInboundHandlerAdapter，根据之前的学习，我们知道，这是个 inbound 类型的 handler，也就是处理流向自身事件的 handler。其次，该类通过 abstract 关键字修饰，说明是个抽象类，在我们实际使用的时候，并不是直接使用这个类，而是使用其子类，类定义了解码器的骨架方法，具体实现逻辑交给子类，同样，在半包处理中也是由该类进行实现的。Netty 中很多解码器都实现了这个类，并且，我们也可以通过实现该类进行自定义解码器。

我们重点关注一下该类的 `cumulation` 这个属性，它就是有关半包处理的关键属性，从概述中我们知道，Netty 会将不完整的数据包进行保存，这个数据包就是保存在这个属性中。之前的学习我们知道，ByteBuf 读取完数据会

传递 channelRead 事件，传播过程中会调用 handler 的 channelRead 方法，ByteToMessageDecoder 的 channelRead 方法，就是编码的关键部分。我们来看其 channelRead()方法：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    //如果 message 是 byteBuf 类型
    if (msg instanceof ByteBuf) {
        //简单当成一个 arrayList，用于盛放解析到的对象
        CodecOutputList out = CodecOutputList.newInstance();
        try {
            ByteBuf data = (ByteBuf) msg;
            //当前累加器为空，说明这是第一次从 io 流里面读取数据
            first = cumulation == null;
            if (first) {
                //如果是第一次，则将累加器赋值为刚读进来的对象
                cumulation = data;
            } else {
                //如果不是第一次，则把当前累加的数据和读进来的数据进行累加
                cumulation = cumulator.cumulate(ctx.alloc(), cumulation, data);
            }
            //调用子类的方法进行解析
            callDecode(ctx, cumulation, out);
        } catch (DecoderException e) {
            throw e;
        } catch (Throwable t) {
            throw new DecoderException(t);
        } finally {
            if (cumulation != null && !cumulation.isReadable()) {
                numReads = 0;
                cumulation.release();
                cumulation = null;
            } else if (++ numReads >= discardAfterReads) {
                numReads = 0;
                discardSomeReadBytes();
            }
            //记录 list 长度
            int size = out.size();
            decodeWasNull = !out.insertSinceRecycled();
            //向下传播
            fireChannelRead(ctx, out, size);
            out.recycle();
        }
    } else {
        //不是 byteBuf 类型则向下传播
        ctx.fireChannelRead(msg);
    }
}
```

这方法比较长，我带大家一步步剖析。首先判断如果传来的数据是 ByteBuf，则进入 if 块中，CodecOutputList out = CodecOutputList.newInstance() 这里就当成一个 ArrayList 就好，用于保存解码完成的数据 ByteBuf data = (ByteBuf) msg 这步将数据转化成 ByteBuf ;first = cumulation == null 表示如果 cumulation == null，说明没有存储板半包数据，则将当前的数据保存在属性 cumulation 中；如果 cumulation != null，说明存储了半包数据，则通过

cumulator.cumulate(ctx.alloc(), cumulation, data)将读取到的数据和原来的数据进行累加，保存在属性 cumulation 中，

我们看 cumulator 属性的定义：

```
private Cumulator cumulator = MERGE_CUMULATOR;
```

这里调用了其静态属性 MERGE_CUMULATOR，我们跟进去：

```
public static final Cumulator MERGE_CUMULATOR = new Cumulator() {
    @Override
    public ByteBuf cumulate(ByteBufAllocator alloc, ByteBuf cumulation, ByteBuf in) {
        ByteBuf buffer;
        //不能到过最大内存
        if (cumulation.writerIndex() > cumulation.maxCapacity() - in.readableBytes()
                || cumulation.refCnt() > 1) {
            buffer = expandCumulation(alloc, cumulation, in.readableBytes());
        } else {
            buffer = cumulation;
        }
        //将当前数据 buffer
        buffer.writeBytes(in);
        in.release();
        return buffer;
    }
};
```

这里创建了 Cumulator 类型的静态对象，并重写了 cumulate()方法，这个 cumulate()方法，就是用于将 ByteBuf 进行拼接的方法。在方法中，首先判断 cumulation 的写指针+in 的可读字节数是否超过了 cumulation 的最大长度，如果超过了，将对 cumulation 进行扩容，如果没超过，则将其赋值到局部变量 buffer 中。然后，将 in 的数据写到 buffer 中，将 in 进行释放，返回写入数据后的 ByteBuf。回到 channelRead()方法：最后调用 callDecode(ctx, cumulation, out)方法进行解码，这里传入了 Context 对象，缓冲区 cumulation 和集合 out。我们跟进到 callDecode(ctx, cumulation, out)方法：

```
protected void callDecode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {
    try {
        //只要累加器里面有数据
        while (in.isReadable()) {
            int outSize = out.size();
            //判断当前 List 是否有对象
            if (outSize > 0) {
                //如果有对象，则向下传播事件
                fireChannelRead(ctx, out, outSize);
                //清空当前 list
                out.clear();
                //解码过程中如 ctx 被 removed 掉就 break
                if (ctx.isRemoved()) {
                    break;
                }
                outSize = 0;
            }
            //当前可读数据长度
            int oldInputLength = in.readableBytes();
            //子类实现
            //子类解析，解析玩对象放到 out 里面
            decode(ctx, in, out);
```

```
            if (ctx.isRemoved()) {
                break;
            }
            //List 解析前大小 和解析后长度一样(什么没有解析出来)
            if (outSize == out.size()) {
                //原来可读的长度==解析后可读长度
                //说明没有读取数据(当前累加的数据并没有拼成一个完整的数据包)
                if (oldInputLength == in.readableBytes()) {
                    //跳出循环(下次在读取数据才能进行后续的解析)
                    break;
                } else {
                    //没有解析到数据，但是进行读取了
                    continue;
                }
            }
            //out 里面有数据，但是没有从累加器读取数据
            if (oldInputLength == in.readableBytes()) {
                throw new DecoderException(
                        StringUtil.simpleClassName(getClass()) +
                        ".decode() did not read anything but decoded a message.");
            }

            if (isSingleDecode()) {
                break;
            }
        }
    } catch (DecoderException e) {
        throw e;
    } catch (Throwable cause) {
        throw new DecoderException(cause);
    }
}
```

首先循环判断传入的 ByteBuf 是否有可读字节，如果还有可读字节说明没有解码完成，则循环继续解码。然后判断集合 out 的大小，如果大小大于 1，说明 out 中盛放了解码完成之后的数据，然后将事件向下传播，并清空 out。因为我们第一次解码 out 是空的，所以这里不会进入 if 块，这部分我们稍后分析，所以继续往下看，通过 int oldInputLength = in.readableBytes() 获取当前 ByteBuf，其实也就是属性 cumulation 的可读字节数，这里就是一个备份用于比较。我们继续往下看，decode(ctx, in, out)方法是最终的解码操作，这部会读取 cumulation 并且将解码后的数据放入到集合 out 中，在 ByteToMessageDecoder 中的该方法是一个抽象方法，让子类进行实现，我们使用的 netty 很多的解码都是继承了 ByteToMessageDecoder 并实现了 decode 方法从而完成了解码操作，同样我们也可以遵循相应的规则进行自定义解码器，在之后的小节中会讲解 netty 定义的解码器，并剖析相关的实现细节。继续往下看，if (outSize == out.size()) 这个判断表示解析之前的 out 大小和解析之后 out 大小进行比较，如果相同，说明并没有解析出数据，我们进入到 if 块中。if (oldInputLength == in.readableBytes()) 表示 cumulation 的可读字节数在解析之前和解析之后是相同的，说明解码方法中并没有解析数据，也就是当前的数据并不是一个完整的数据包，则跳出循环，留给下次解析，否则，说明没有解

析到数据，但是读取了，所以跳过该次循环进入下次循环。最后判断 if (oldInputLength == in.readableBytes())，这里代表 out 中有数据，但是并没有从 cumulation 读数据，说明这个 out 的内容是非法的，直接抛出异常。现在回到 channRead()方法，我们关注 finally 代码块中的内容：

```
finally {
    if (cumulation != null && !cumulation.isReadable()) {
        numReads = 0;
        cumulation.release();
        cumulation = null;
    } else if (++ numReads >= discardAfterReads) {
        numReads = 0;
        discardSomeReadBytes();
    }
    //记录 list 长度
    int size = out.size();
    decodeWasNull = !out.insertSinceRecycled();
    //向下传播
    fireChannelRead(ctx, out, size);
    out.recycle();
}
```

首先判断 cumulation 不为 null，并且没有可读字节，则将累加器进行释放，并设置为 null，之后记录 out 的长度，通过 fireChannelRead(ctx, out, size)将 channelRead 事件进行向下传播，并回收 out 对象。我们跟到 fireChannelRead(ctx, out, size)方法来看代码：

```
static void fireChannelRead(ChannelHandlerContext ctx, CodecOutputList msgs, int numElements) {
    //遍历 List
    for (int i = 0; i < numElements; i ++) {
        //逐个向下传递
        ctx.fireChannelRead(msgs.getUnsafe(i));
    }
}
```

这里遍历 out 集合，并将里面的元素逐个向下传递，以上就是有关解码的骨架逻辑。

11.3.2 LineBasedFrameDecoder 行解码器

LineBasedFrameDecoder 是回车换行解码器，如果用户发送的消息以回车换行符（以\r\n 或者直接以\n 结尾）作为消息结束的标识，则可以直接使用 Netty 的 LineBasedFrameDecoder 对消息进行解码，只需要在初始化 Netty 服务端或者客户端时将 LineBasedFrameDecoder 正确的添加到 ChannelPipeline 中即可，不需要自己重新实现一套换行解码器。LineBasedFrameDecoder 的工作原理是它依次遍历 ByteBuf 中的可读字节，判断看是否有"\n"或者"\r\n"，如果有，就以此位置为结束位置，从可读索引到结束位置区间的字节就组成了一行。它是以换行符为结束标志的解码器，支持携

带结束符或者不携带结束符两种解码方式，同时支持配置单行的最大长度。如果连续读取到最大长度后仍然没有发现换行符，就会抛出异常，同时忽略掉之前读到的异常码流。防止由于数据报没有携带换行符导致接收到 ByteBuf 无限制积压，引起系统内存溢出。它的使用效果如下：

解码之前：

```
+--------------------------------------------------------------------+
                        接收到的数据报
“This is a netty example for using the nio framework.\r\n When you“
+--------------------------------------------------------------------+
```

解码之后的 ChannelHandler 接收到的 Object 如下：

```
+--------------------------------------------------------------------+
                        解码之后的文本消息
“This is a netty example for using the nio framework.“
+--------------------------------------------------------------------+
```

通常情况下，LineBasedFrameDecoder 会和 StringDecoder 配合使用，组合成按行切换的文本解码器，对于文本类协议的解析，文本换行解码器非常实用，例如对 HTTP 消息头的解析、FTP 协议消息的解析等。

下面我们简单给出文本换行解码器的使用示例：

```
pipeline.addLast(new LineBasedFrameDecoder(1024));
pipeline.addLast(new StringDecoder());
```

初始化 Channel 的时候，首先将 LineBasedFrameDecoder 添加到 ChannelPipeline 中，然后再依次添加字符串解码器 StringDecoder，业务 Handler。

接下来，我们来看 LineBasedFrameDecoder 的源码，LineBasedFrameDecoder 也继承了 ByteToMessageDecoder。

首先看其参数定义：

```
//数据包的最大长度，超过该长度会进行丢弃模式
private final int maxLength;
//超出最大长度是否要抛出异常
private final boolean failFast;
//最终解析的数据包是否带有换行符
private final boolean stripDelimiter;
//为 true 说明当前解码过程为丢弃模式
private boolean discarding;
//丢弃了多少字节
private int discardedBytes;
```

其中的丢弃模式，我们会在源码中看到其中的含义，我们看其 decode()方法：

```
protected final void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
    Object decoded = decode(ctx, in);
    if (decoded != null) {
        out.add(decoded);
    }
}
```

这里的 decode()方法调用重载的 decode()方法，并将解码后的内容放到 out 集合中。我们跟到重载的 decode()方法中：

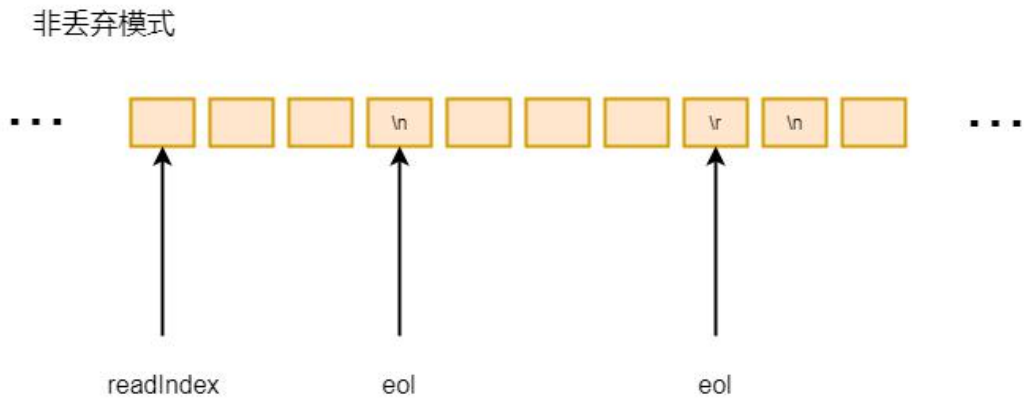
```
protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws Exception {
    //找这行的结尾
    final int eol = findEndOfLine(buffer);
    if (!discarding) {
        if (eol >= 0) {
            final ByteBuf frame;
            //计算从换行符到可读字节之间的长度
            final int length = eol - buffer.readerIndex();
            //拿到分隔符长度，如果是\r\n 结尾，分隔符长度为 2
            final int delimLength = buffer.getByte(eol) == '\r'? 2 : 1;

            //如果长度大于最大长度
            if (length > maxLength) {
                //指向换行符之后的可读字节(这段数据完全丢弃)
                buffer.readerIndex(eol + delimLength);
                //传播异常事件
                fail(ctx, length);
                return null;
            }
            //如果这次解析的数据是有效的
            //分隔符是否算在完整数据包里
            //true 为丢弃分隔符
            if (stripDelimiter) {
                //截取有效长度
                frame = buffer.readRetainedSlice(length);
                //跳过分隔符的字节
                buffer.skipBytes(delimLength);
            } else {
                //包含分隔符
                frame = buffer.readRetainedSlice(length + delimLength);
            }

            return frame;
        } else {
            //如果没找到分隔符(非丢弃模式)
            //可读字节长度
            final int length = buffer.readableBytes();
            //如果朝超过能解析的最大长度
            if (length > maxLength) {
                //将当前长度标记为可丢弃的
                discardedBytes = length;
                //直接将读指针移动到写指针
                buffer.readerIndex(buffer.writerIndex());
                //标记为丢弃模式
                discarding = true;
                //超过最大长度抛出异常
                if (failFast) {
                    fail(ctx, "over " + discardedBytes);
                }
            }
            //没有超过，则直接返回
            return null;
        }
    } else {
        //丢弃模式
        if (eol >= 0) {
            //找到分隔符
            //当前丢弃的字节(前面已经丢弃的+现在丢弃的位置-写指针)
            final int length = discardedBytes + eol - buffer.readerIndex();
            //当前换行符长度为多少
            final int delimLength = buffer.getByte(eol) == '\r'? 2 : 1;
            //读指针直接移到换行符+换行符的长度
            buffer.readerIndex(eol + delimLength);
```

```
            //当前丢弃的字节为0
            discardedBytes = 0;
            //设置为未丢弃模式
            discarding = false;
            //丢弃完字节之后触发异常
            if (!failFast) {
                fail(ctx, length);
            }
        } else {
            //累计已丢弃的字节个数+当前可读的长度
            discardedBytes += buffer.readableBytes();
            //移动
            buffer.readerIndex(buffer.writerIndex());
        }
        return null;
    }
}
```

final int eol = findEndOfLine(buffer) 这里是找当前行的结尾的索引值，也就是\r\n 或者是\n：



从上图中不难看出，如果是以\n 结尾的，返回的索引值是\n 的索引值，如果是\r\n 结尾的，返回的索引值是\r 的索引值

我们看 findEndOfLine(buffer)方法：

```
private static int findEndOfLine(final ByteBuf buffer) {
    //找到/n 这个字节
    int i = buffer.forEachByte(ByteProcessor.FIND_LF);
    //如果找到了，并且前面的字符是-r，则指向/r 字节
    if (i > 0 && buffer.getByte(i - 1) == '\r') {
        i--;
    }
    return i;
}
```

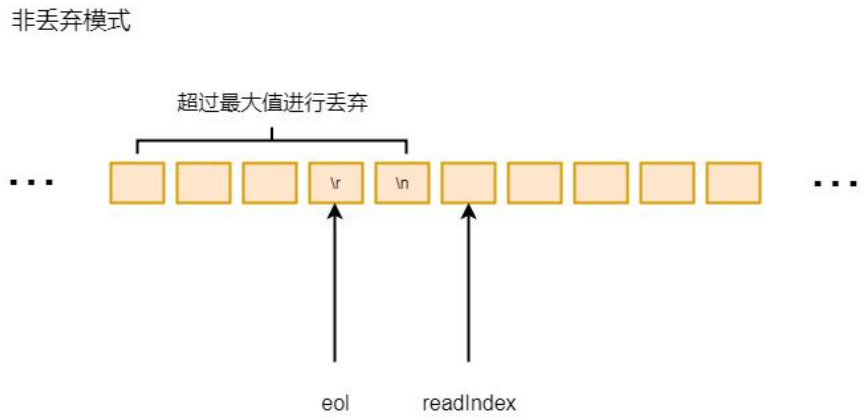
从上面代码看到，通过一个 forEachByte()方法找\n 这个字节，如果找到了，并且前面是\r，则返回\r 的索引，否则返回\n 的索引。回到重载的 decode()方法，if (!discarding) 判断是否为非丢弃模式，默认是就是非丢弃模式，所以进入 if

中；if (eol >= 0) 如果找到了换行符，我们看非丢弃模式下找到换行符的相关逻辑：

```
final ByteBuf frame;
final int length = eol - buffer.readerIndex();
final int delimLength = buffer.getByte(eol) == '\r'? 2 : 1;
if (length > maxLength) {
    buffer.readerIndex(eol + delimLength);
    fail(ctx, length);
    return null;
}
if (stripDelimiter) {
    frame = buffer.readRetainedSlice(length);
    buffer.skipBytes(delimLength);
} else {
    frame = buffer.readRetainedSlice(length + delimLength);
}

return frame;
```

首先获得换行符到可读字节之间的长度，然后拿到换行符的长度，如果是\n 结尾，那么长度为 1，如果是\r 结尾，长度为 2。if (length > maxLength) 带表如果长度超过最大长度，则直接通过 readerIndex(eol + delimLength) 这种方式，将读指针指向换行符之后的字节，说明换行符之前的字节需要完全丢弃。



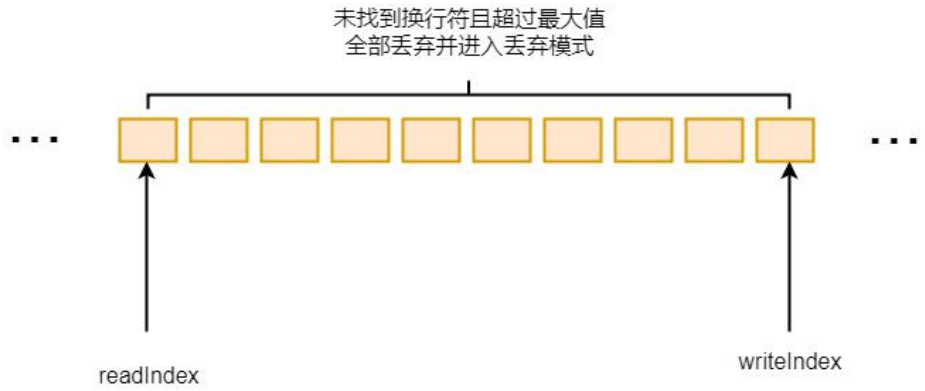
丢弃之后通过 fail 方法传播异常，并返回 null。继续往下看，走到下一步，说明解析出来的数据长度没有超过最大长度，说明是有效数据包。if (stripDelimiter) 表示是否要将分隔符放在完整数据包里面，如果是 true，则说明要丢弃分隔符，然后截取有效长度，并跳过分隔符长度，将包含分隔符进行截取。

以上就是非丢弃模式下找到换行符的相关逻辑，我们再看非丢弃模式下没有找到换行符的相关逻辑，也就是非丢弃模式下， if (eol >= 0) 中的 else 块：

```
final int length = buffer.readableBytes();
if (length > maxLength) {
    discardedBytes = length;
```

```
    buffer.readerIndex(buffer.writerIndex());
    discarding = true;
    if (failFast) {
        fail(ctx, "over " + discardedBytes);
    }
}
return null;
```

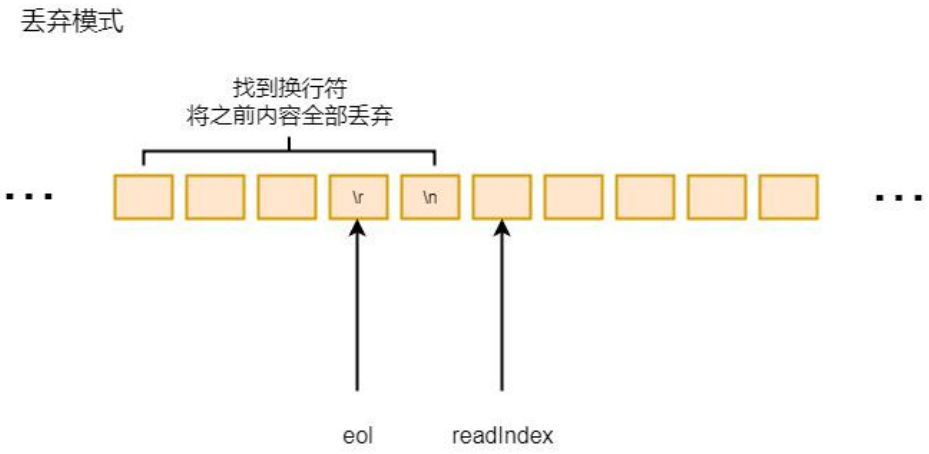
首先通过 final int length = buffer.readableBytes() 获取所有的可读字节数。然后判断可读字节数是否超过了最大值，如果超过最大值，则属性 discardedBytes 标记为这个长度，代表这段内容要进行丢弃。



buffer.readerIndex(buffer.writerIndex()) 这里直接将读指针移动到写指针，并且将 discarding 设置为 true，就是丢弃模式。如果可读字节没有超过最大长度，则返回 null，表示什么都没解析出来，等着下次解析。我们再看丢弃模式的处理逻辑，也就是 if (!discarding) 中的 else 块。首先这里也分两种情况，根据 if (eol >= 0) 判断是否找到了分隔符，我们首先看找到分隔符的解码逻辑：

```
final int length = discardedBytes + eol - buffer.readerIndex();
final int delimLength = buffer.getByte(eol) == '\r'? 2 : 1;
buffer.readerIndex(eol + delimLength);
discardedBytes = 0;
discarding = false;
if (!failFast) {
    fail(ctx, length);
}
```

如果找到换行符，则需要将换行符之前的数据全部丢弃掉。



final int length = discardedBytes + eol - buffer.readerIndex() 这里获得丢弃的字节总数，也就是之前丢弃的字节数+现在需要丢弃的字节数。然后计算换行符的长度，如果是\n 则是 1, \r\n 就是 2。buffer.readerIndex(eol + delimLength) 这里将读指针移动到换行符之后的位置，然后将 discarding 设置为 false，表示当前是非丢弃状态。我们再看丢弃模式未找到换行符的情况，也就是丢弃模式下， if (eol >= 0) 中的 else 块：

```
discardedBytes += buffer.readableBytes();
buffer.readerIndex(buffer.writerIndex());
```

这里做的事情非常简单，就是累计丢弃的字节数，并将读指针移动到写指针，也就是将数据全部丢弃。最后在丢弃模式下, decode()方法返回 null，代表本次没有解析出任何数据。以上就是行解码器的相关逻辑。

11.3.3 DelimiterBasedFrameDecoder 分隔符解码器

DelimiterBasedFrameDecoder 分隔符解码器，是按照指定分隔符进行解码的解码器，通过分隔符，可以将二进制流拆分成完整的数据包。回车换行解码器实际上是一种特殊的 DelimiterBasedFrameDecoder 解码器。

分隔符解码器在实际工作中也有很广泛的应用，笔者所从事的电信行业，很多简单的文本私有协议，都是以特殊的分隔符作为消息结束的标识，特别是对于那些使用长连接的基于文本的私有协议。

分隔符的指定：与大家的习惯不同，分隔符并非以 char 或者 string 作为构造参数，而是 ByteBuf，下面我们就结合实际例子给出它的用法。假如消息以"$_"作为分隔符，服务端或者客户端初始化 ChannelPipeline 的代码实例如下：

```
ByteBuf delimiter = Unpooled.copiedBuffer("$_".getBytes());
pipeline.addLast(new DelimiterBasedFrameDecoder(1024,delimiter));
pipeline.addLast(new StringDecoder());
```

首先将"$_"转换成 ByteBuf 对象，作为参数构造 DelimiterBasedFrameDecoder，将其添加到 ChannelPipeline 中，然后依次添加字符串解码器（通常用于文本解码）和用户 Handler，请注意解码器和 Handler 的添加顺序，如果顺序颠倒，会导致消息解码失败。

DelimiterBasedFrameDecoder 同样继承了 ByteToMessageDecoder 并重写了 decode()方法，我们来看其中的一个构造方法：

```
public DelimiterBasedFrameDecoder(int maxFrameLength, ByteBuf... delimiters) {
    this(maxFrameLength, true, delimiters);
}
```

这里参数 maxFrameLength 代表最大长度, delimiters 是个可变参数, 可以说可以支持多个分隔符进行解码。我们进入 decode()方法:

```
protected final void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
    Object decoded = decode(ctx, in);
    if (decoded != null) {
        out.add(decoded);
    }
}
```

这里同样调用了其重载的 decode()方法并将解析好的数据添加到集合 list 中, 其父类就可以遍历 out, 并将内容传播。

我们跟到重载 decode()方法里面：

```
protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws Exception {
    //行处理器(1)
    if (lineBasedDecoder != null) {
        return lineBasedDecoder.decode(ctx, buffer);
    }
    int minFrameLength = Integer.MAX_VALUE;
    ByteBuf minDelim = null;

    //找到最小长度的分隔符(2)
    for (ByteBuf delim: delimiters) {
        //每个分隔符分隔的数据包长度
        int frameLength = indexOf(buffer, delim);
        if (frameLength >= 0 && frameLength < minFrameLength) {
            minFrameLength = frameLength;
            minDelim = delim;
        }
    }
    //解码(3)
    //已经找到分隔符
    if (minDelim != null) {
        int minDelimLength = minDelim.capacity();
        ByteBuf frame;

        //当前分隔符否处于丢弃模式
        if (discardingTooLongFrame) {
```

```
            //首先设置为非丢弃模式
            discardingTooLongFrame = false;
            //丢弃
            buffer.skipBytes(minFrameLength + minDelimLength);

            int tooLongFrameLength = this.tooLongFrameLength;
            this.tooLongFrameLength = 0;
            if (!failFast) {
                fail(tooLongFrameLength);
            }
            return null;
        }
        //处于非丢弃模式
        //当前找到的数据包，大于允许的数据包
        if (minFrameLength > maxFrameLength) {
            //当前数据包+最小分隔符长度 全部丢弃
            buffer.skipBytes(minFrameLength + minDelimLength);
            //传递异常事件
            fail(minFrameLength);
            return null;
        }
        //如果是正常的长度
        //解析出来的数据包是否忽略分隔符
        if (stripDelimiter) {
            //如果不包含分隔符
            //截取
            frame = buffer.readRetainedSlice(minFrameLength);
            //跳过分隔符
            buffer.skipBytes(minDelimLength);
        } else {
            //截取包含分隔符的长度
            frame = buffer.readRetainedSlice(minFrameLength + minDelimLength);
        }

        return frame;
    } else {
        //如果没有找到分隔符
        //非丢弃模式
        if (!discardingTooLongFrame) {
            //可读字节大于允许的解析出来的长度
            if (buffer.readableBytes() > maxFrameLength) {
                //将这个长度记录下
                tooLongFrameLength = buffer.readableBytes();
                //跳过这段长度
                buffer.skipBytes(buffer.readableBytes());
                //标记当前处于丢弃状态
                discardingTooLongFrame = true;
                if (failFast) {
                    fail(tooLongFrameLength);
                }
            }
        } else {
            tooLongFrameLength += buffer.readableBytes();
            buffer.skipBytes(buffer.readableBytes());
        }
        return null;
    }
}
```

这里的方法也比较长，这里也通过拆分进行剖析：1、行处理器；2、找到最小长度分隔符；3、解码。首先看第 1 步行处理器：

```
if (lineBasedDecoder != null) {
    return lineBasedDecoder.decode(ctx, buffer);
}
```

这里首先判断成员变量 lineBasedDecoder 是否为空，如果不为空则直接调用 lineBasedDecoder 的 decode 的方法进行解码，lineBasedDecoder 实际上就是上一小节剖析的 LineBasedFrameDecoder 解码器。这个成员变量，会在分隔符是\n 和\r\n 的时候进行初始化。我们看初始化该属性的构造方法：

```
public DelimiterBasedFrameDecoder(
        int maxFrameLength, boolean stripDelimiter, boolean failFast, ByteBuf... delimiters) {
    //代码省略
    //如果是基于行的分隔
    if (isLineBased(delimiters) && !isSubclass()) {
        //初始化行处理器
        lineBasedDecoder = new LineBasedFrameDecoder(maxFrameLength, stripDelimiter, failFast);
        this.delimiters = null;
    } else {
        //代码省略
    }
    //代码省略
}
```

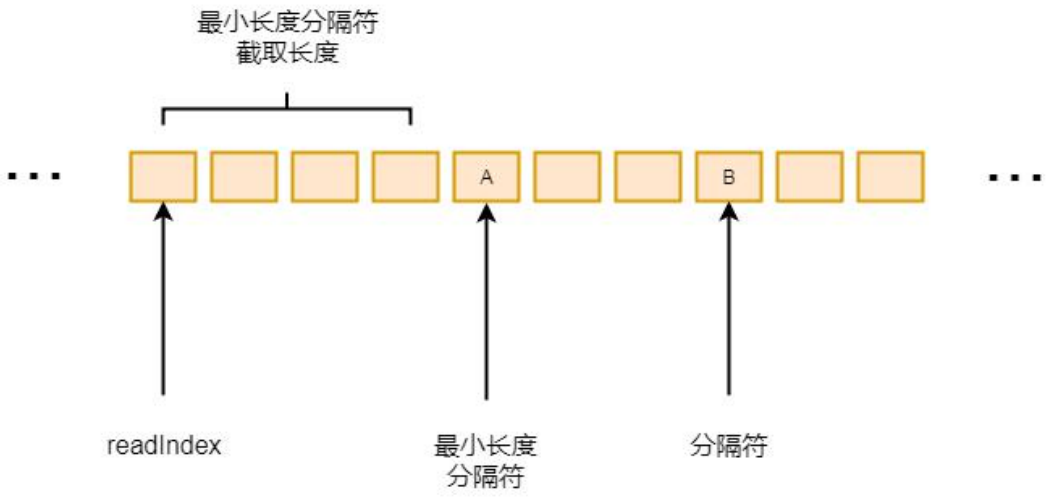
这里 isLineBased(delimiters)会判断是否是基于行的分隔，跟到 isLineBased()方法中：

```
private static boolean isLineBased(final ByteBuf[] delimiters) {
    //分隔符长度不为 2
    if (delimiters.length != 2) {
        return false;
    }
    //拿到第一个分隔符
    ByteBuf a = delimiters[0];
    //拿到第二个分隔符
    ByteBuf b = delimiters[1];
    if (a.capacity() < b.capacity()) {
        a = delimiters[1];
        b = delimiters[0];
    }
    //确保 a 是/r/n 分隔符，确保 b 是/n 分隔符
    return a.capacity() == 2 && b.capacity() == 1
            && a.getByte(0) == '\r' && a.getByte(1) == '\n'
            && b.getByte(0) == '\n';
}
```

首先判断长度等于 2，直接返回 false。然后拿到第一个分隔符 a 和第二个分隔符 b，然后判断 a 的第一个分隔符是不是\r，a 的第二个分隔符是不是\n，b 的第一个分隔符是不是\n，如果都为 true，则条件成立。我们回到 decode()方法中，看第 2 步，找到最小长度的分隔符。这里最小长度的分隔符，意思就是从读指针开始，找到最近的分隔符：

```
for (ByteBuf delim: delimiters) {
    //每个分隔符分隔的数据包长度
    int frameLength = indexOf(buffer, delim);
    if (frameLength >= 0 && frameLength < minFrameLength) {
        minFrameLength = frameLength;
        minDelim = delim;
    }
}
```

这里会遍历所有的分隔符，然后找到每个分隔符到读指针到数据包长度。然后通过 if 判断，找到长度最小的数据包的长度，然后保存当前数据包的的分隔符，如下图：



这里假设 A 和 B 同为分隔符，A 分隔符到读指针的长度小于 B 分隔符到读指针的长度，这里会找到最小的分隔符 A，分隔符的最小长度，就 readIndex 到 A 的长度。我们继续看第 3 步，解码。if (minDelim != null) 表示已经找到最小长度分隔符，我们继续看 if 块中的逻辑：

```
int minDelimLength = minDelim.capacity();
ByteBuf frame;
if (discardingTooLongFrame) {
    discardingTooLongFrame = false;
    buffer.skipBytes(minFrameLength + minDelimLength);
    int tooLongFrameLength = this.tooLongFrameLength;
    this.tooLongFrameLength = 0;
    if (!failFast) {
        fail(tooLongFrameLength);
    }
    return null;
}
if (minFrameLength > maxFrameLength) {
    buffer.skipBytes(minFrameLength + minDelimLength);
    fail(minFrameLength);
    return null;
}
if (stripDelimiter) {
    frame = buffer.readRetainedSlice(minFrameLength);
    buffer.skipBytes(minDelimLength);
} else {
    frame = buffer.readRetainedSlice(minFrameLength + minDelimLength);
}
return frame;
```

if (discardingTooLongFrame) 表示当前是否处于非丢弃模式，如果是丢弃模式，则进入 if 块。因为第一个不是丢弃模式，所以这里先分析 if 块后面的逻辑。if (minFrameLength > maxFrameLength) 这里是判断当前找到的数据包长度大于最大长度，这里的最大长度使我们创建解码器的时候设置的，如果超过了最大长度，就通过 buffer.skipBytes(minFrameLength + minDelimLength) 方式，跳过数据包+分隔符的长度，也就是将这部分数据进行完全丢弃。继续往下看，如果长度不大最大允许长度，则通过 if (stripDelimiter) 判断解析的出来的数据包是否包含分隔符，如果不包含分隔符，则截取数据包的长度之后，跳过分隔符。我们再回头看 if (discardingTooLongFrame) 中的 if 块中的逻辑，也就是丢弃模式。首先将 discardingTooLongFrame 设置为 false，标记非丢弃模式，然后通过 buffer.skipBytes(minFrameLength + minDelimLength) 将数据包+分隔符长度的字节数跳过，也就是进行丢弃，之后再进行抛出异常。分析完成了找到分隔符之后的丢弃模式非丢弃模式的逻辑处理，我们在分析没找到分隔符的逻辑处理，也就是 if (minDelim != null) 中的 else 块：

```
if (!discardingTooLongFrame) {
    if (buffer.readableBytes() > maxFrameLength) {
        tooLongFrameLength = buffer.readableBytes();
        buffer.skipBytes(buffer.readableBytes());
        discardingTooLongFrame = true;
        if (failFast) {
            fail(tooLongFrameLength);
        }
    }
} else {
    tooLongFrameLength += buffer.readableBytes();
    buffer.skipBytes(buffer.readableBytes());
}
return null;
```

首先通过 if (!discardingTooLongFrame) 判断是否为非丢弃模式，如果是，则进入 if 块:。在 if 块中，首先通过 if (buffer.readableBytes() > maxFrameLength) 判断当前可读字节数是否大于最大允许的长度，如果大于最大允许的长度，则将可读字节数设置到 tooLongFrameLength 的属性中，代表丢弃的字节数，然后通过 buffer.skipBytes(buffer.readableBytes()) 将累计器中所有的可读字节进行丢弃，最后将 discardingTooLongFrame 设置为 true，也就是丢弃模式，之后抛出异常。如果 if (!discardingTooLongFrame) 为 false，也就是当前处于丢弃模式，则追加 tooLongFrameLength 也就是丢弃的字节数的长度，并通过 buffer.skipBytes(buffer.readableBytes()) 将所有的字节继续进行丢弃。以上就是分隔符解码器的相关逻辑。

11.3.4 FixedLengthFrameDecoder 固定长度解码器

FixedLengthFrameDecoder 固定长度解码器，它能够按照指定的长度对消息进行自动解码，开发者不需要考虑 TCP 的粘包/拆包等问题，非常实用。

对于定长消息，如果消息实际长度小于定长，则往往会进行补位操作，它在一定程度上导致了空间和资源的浪费。但是它的优点也是非常明显的，编解码比较简单，因此在实际项目中仍然有一定的应用场景。

利用 FixedLengthFrameDecoder 解码器，无论一次接收到多少数据报，它都会按照构造函数中设置的固定长度进行解码，如果是半包消息，FixedLengthFrameDecoder 会缓存半包消息并等待下个包到达后进行拼包，直到读取到一个完整的包。假如单条消息的长度是 20 字节，使用 FixedLengthFrameDecoder 解码器的效果如下：

```
解码前：
+------------------------------------------------------------------+
                        接收到的数据报
“HELLO NETTY FOR USER DEVELOPER“
+------------------------------------------------------------------+
解码后：
+------------------------------------------------------------------+
                        解码后的数据报
“HELLO NETTY FOR USER“
+------------------------------------------------------------------+
```

来看其类的定义：

```
public class FixedLengthFrameDecoder extends ByteToMessageDecoder {
    //长度大小
    private final int frameLength;
    public FixedLengthFrameDecoder(int frameLength) {
        if (frameLength <= 0) {
            throw new IllegalArgumentException(
                    "frameLength must be a positive integer: " + frameLength);
        }
        //保存当前 frameLength
        this.frameLength = frameLength;
    }
    @Override
    protected final void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exception {
        //通过 ByteBuf 去解码.解码到对象之后添加到 out 上
        Object decoded = decode(ctx, in);
        if (decoded != null) {
            //将解析到 byteBuf 添加到对象里面
            out.add(decoded);
        }
    }
    protected Object decode(
            @SuppressWarnings("UnusedParameters") ChannelHandlerContext ctx, ByteBuf in) throws Exception {
        //字节是否小于这个固定长度
        if (in.readableBytes() < frameLength) {
            return null;
        } else {
```

```
            //当前累加器中截取这个长度的数值
            return in.readRetainedSlice(frameLength);
        }
    }
}
```

我们看到 FixedLengthFrameDecoder 类继承了 ByteToMessageDecoder，重写了 decode()方法，这个类只有一个属性叫 frameLength，并在构造方法中初始化了该属性。再看 decode()方法，在 decode()方法中又调用了自身另一个重载的 decode()方法进行解析，解析出来之后将解析后的数据放在集合 out 中。再看重载的 decode()方法，重载的 decode()方法中首先判断累加器的字节数是否小于固定长度，如果小于固定长度则返回 null，代表不是一个完整的数据包，直接返回 null。如果大于等于固定长度，则直接从累加器中截取这个长度的数值 in.readRetainedSlice(frameLength) 会返回一个新的截取后的 ByteBuf，并将原来的累加器读指针后移 frameLength 个字节。如果累计器中还有数据，则会通过 ByteToMessageDecoder 中 callDecode()方法里 while 循环的方式，继续进行解码。这样，就是实现了固定长度的解码工作。

11.3.5 LengthFieldBasedFrameDecoder 通用解码器

了解 TCP 通信机制的该都知道 TCP 底层的粘包和拆包，当我们在接收消息的时候，显示不能认为读取到的报文就是个整包消息，特别是对于采用非阻塞 I/O 和长连接通信的程序。

如何区分一个整包消息，通常有如下 4 种做法：

1) 固定长度，例如每 120 个字节代表一个整包消息，不足的前面补位。解码器在处理这类定常消息的时候比较简单，每次读到指定长度的字节后再进行解码；

2) 通过回车换行符区分消息，例如 HTTP 协议。这类区分消息的方式多用于文本协议；

3) 通过特定的分隔符区分整包消息；

4) 通过在协议头/消息头中设置长度字段来标识整包消息。

前三种解码器之前的章节已经做了详细介绍，下面让我们来一起学习最后一种通用解码器-LengthFieldBasedFrameDecoder。

大多数的协议（私有或者公有），协议头中会携带长度字段，用于标识消息体或者整包消息的长度，例如 SMPP、HTTP 协议等。由于基于长度解码需求的通用性，以及为了降低用户的协议开发难度，Netty 提供了 LengthFieldBasedFrameDecoder，自动屏蔽 TCP 底层的拆包和粘包问题，只需要传入正确的参数，即可轻松解决"读半包"问题。

下面我们看看如何通过参数组合的不同来实现不同的"半包"读取策略。第一种常用的方式是消息的第一个字段是长度字段，后面是消息体，消息头中只包含一个长度字段。它的消息结构定义如图所示：

```
+---------+-----------------+
|  Length |  Actual Content |
|  0x000C |  “HELLO,WORLD”  |
+---------+-----------------+
```

使用以下参数组合进行解码：

1) lengthFieldOffset = 0；

2) lengthFieldLength = 2；

3) lengthAdjustment = 0；

4) initialBytesToStrip = 0。

解码后的字节缓冲区内容如图所示：

```
+---------+-----------------+
|  Length |  Actual Content |
|  0x000C |  “HELLO,WORLD”  |
+---------+-----------------+
```

通过 ByteBuf.readableBytes()方法我们可以获取当前消息的长度，所以解码后的字节缓冲区可以不携带长度字段，由于长度字段在起始位置并且长度为 2，所以将 initialBytesToStrip 设置为 2，参数组合修改为：

1) lengthFieldOffset = 0；

2) lengthFieldLength = 2；

3) lengthAdjustment = 0；

4) initialBytesToStrip = 2。

解码后的字节缓冲区内容如图所示：

```
+-----------------+
```

```
| Actual Content  |
| “HELLO,WORLD”   |
+-----------------+
```

解码后的字节缓冲区丢弃了长度字段，仅仅包含消息体，对于大多数的协议，解码之后消息长度没有用处，因此可以丢弃。在大多数的应用场景中，长度字段仅用来标识消息体的长度，这类协议通常由消息长度字段+消息体组成，如上图所示的几个例子。但是，对于某些协议，长度字段还包含了消息头的长度。在这种应用场景中，往往需要使用 lengthAdjustment 进行修正。由于整个消息（包含消息头）的长度往往大于消息体的长度，所以，lengthAdjustment 为负数。下图展示了通过指定 lengthAdjustment 字段来包含消息头的长度：

1) lengthFieldOffset = 0；

2) lengthFieldLength = 2；

3) lengthAdjustment = -2；

4) initialBytesToStrip = 0。

解码之前的码流：

```
+---------+-----------------+
| Length |  Actual Content |
| 0x000E | “HELLO,WORLD”  |
+---------+-----------------+
```

解码之后的码流：

```
+---------+-----------------+
| Length |  Actual Content |
| 0x000E | “HELLO,WORLD”  |
+---------+-----------------+
```

由于协议种类繁多，并不是所有的协议都将长度字段放在消息头的首位，当标识消息长度的字段位于消息头的中间或者尾部时，需要使用 lengthFieldOffset 字段进行标识，下面的参数组合给出了如何解决消息长度字段不在首位的问题：

1) lengthFieldOffset = 2；

2) lengthFieldLength = 3；

3) lengthAdjustment = 0；

4) initialBytesToStrip = 0。

其中 lengthFieldOffset 表示长度字段在消息头中偏移的字节数，lengthFieldLength 表示长度字段自身的长度，解码效

果如下：

解码之前：

```
+------------+---------+-----------------+
|  Header 1  |  Length  |  Actual Content |
|   0xCAFE   | 0x00000C |  “HELLO,WORLD”  |
+------------+---------+-----------------+
```

解码之后：

```
+------------+---------+-----------------+
|  Header 1  |  Length  |  Actual Content |
|   0xCAFE   | 0x00000C |  “HELLO,WORLD”  |
+------------+---------+-----------------+
```

由于消息头 1 的长度为 2，所以长度字段的偏移量为 2；消息长度字段 Length 为 3，所以 lengthFieldLength 值为 3。

由于长度字段仅仅标识消息体的长度，所以 lengthAdjustment 和 initialBytesToStrip 都为 0。

最后一种场景是长度字段夹在两个消息头之间或者长度字段位于消息头的中间，前后都有其它消息头字段，在这种场景下如果想忽略长度字段以及其前面的其它消息头字段 ,则可以通过 initialBytesToStrip 参数来跳过要忽略的字节长度，它的组合配置示意如下：

lengthFieldOffset = 1 ;

lengthFieldLength = 2 ;

lengthAdjustment = 1 ;

initialBytesToStrip = 3。

解码之前的码流（16 字节）：

```
+-------+---------+------+----------------+
|  HDR1  | Length | HDR2 | Actual Content  |
|  0xCA  | 0x000C | 0xFE | “HELLO,WORLD”   |
+-------+---------+------+----------------+
```

解码之后的码流（13 字节）：

```
+------+----------------+
| HDR2 | Actual Content |
| 0xFE | “HELLO,WORLD”  |
+------+----------------+
```

由于 HDR1 的长度为 1，所以长度字段的偏移量 lengthFieldOffset 为 1；长度字段为 2 个字节，所以 lengthFieldLength 为 2。由于长度字段是消息体的长度，解码后如果携带消息头中的字段，则需要使用 lengthAdjustment 进行调整，此

处它的值为 1 ,代表的是 HDR2 的长度 ,最后由于解码后的缓冲区要忽略长度字段和 HDR1 部分 ,所以 lengthAdjustment 为 3。解码后的结果为 13 个字节，HDR1 和 Length 字段被忽略。

事实上，通过 4 个参数的不同组合，可以达到不同的解码效果，用户在使用过程中可以根据业务的实际情况进行灵活调整。

由于 TCP 存在粘包和组包问题，所以通常情况下用户需要自己处理半包消息。利用 LengthFieldBasedFrameDecoder 解码器可以自动解决半包问题，它的习惯用法如下：

```
pipeline.addLast("frameDecoder", new LengthFieldBasedFrameDecoder(65536,0,2));
```

在 pipeline 中增加 LengthFieldBasedFrameDecoder 解码器，指定正确的参数组合，它可以将 Netty 的 ByteBuf 解码成整包消息，后面的用户解码器拿到的就是个完整的数据报，按照逻辑正常进行解码即可，不再需要额外考虑"读半包"问题，降低了用户的开发难度。

11.4 Netty 编码器原理和数据输出

Netty 默认提供了丰富的编解码框架供用户集成使用，我们只对较常用的 Java 序列化编码器进行讲解。其它的编码器，实现方式大同小异。其实编码器和解码器比较类似，编码器也是一个 handler，并且属于 outbounfHandle，就是将准备发出去的数据进行拦截，拦截之后进行相应的处理之后再次进发送处理，如果理解了解码器，那么编码器的相关内容理解起来也比较容易。

11.4.1 writeAndFlush 事件传播

我们在前面的章节学习 Pipeline 的时候，讲解了 write 事件的传播过程，但在实际使用的时候，我们通常不会调用 channel 的 write 方法，因为该方法只会写入到发送数据的缓存中，并不会直接写入 channel 中，如果想写入到 channel 中，还需要调用 flush 方法。实际使用过程中，我们用的更多的是 writeAndFlush()方法，这方法既能将数据写到发送缓存中，也能刷新到 channel 中。我们看一个最简单的使用的场景：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
```

```
    ctx.channel().writeAndFlush("test data");
}
```

这个地方小伙伴们肯定不陌生，通过这种方式，可以将数据发送到 channel 中，对方可以收到响应。简单回顾一下跟到 writeAndFlush()方法中，首先会走到 AbstractChannel 的 writeAndFlush()方法：

```
public ChannelFuture writeAndFlush(Object msg) {
    return pipeline.writeAndFlush(msg);
}
```

继续跟到 DefualtChannelPipeline 中的 writeAndFlush()方法中：

```
public final ChannelFuture writeAndFlush(Object msg) {
    return tail.writeAndFlush(msg);
}
```

这里我们看到，writeAndFlush 是从 tail 节点进行传播，有关事件传播，我们在 Pipeline 中进行过剖析，相信这个不会陌生。继续跟，会跟到 AbstractChannelHandlerContext 中的 writeAndFlush()方法:

```
public ChannelFuture writeAndFlush(Object msg) {
    return writeAndFlush(msg, newPromise());
}
```

继续跟：

```
public ChannelFuture writeAndFlush(Object msg, ChannelPromise promise) {
    if (msg == null) {
        throw new NullPointerException("msg");
    }
    if (!validatePromise(promise, true)) {
        ReferenceCountUtil.release(msg);
        // cancelled
        return promise;
    }
    write(msg, true, promise);
    return promise;
}
```

继续跟 write()方法：

```
private void write(Object msg, boolean flush, ChannelPromise promise) {
    //findContextOutbound()寻找前一个 outbound 节点
    //最后到 head 节点结束
    AbstractChannelHandlerContext next = findContextOutbound();
    final Object m = pipeline.touch(msg, next);
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) {
        if (flush) {
            next.invokeWriteAndFlush(m, promise);
        } else {
            //没有调 flush
            next.invokeWrite(m, promise);
        }
    } else {
        AbstractWriteTask task;
        if (flush) {
            task = WriteAndFlushTask.newInstance(next, m, promise);
        }  else {
            task = WriteTask.newInstance(next, m, promise);
        }
```

```
        safeExecute(executor, task, promise, m);
    }
}
```

这里的逻辑我们也不陌生，找到下一个节点，因为 writeAndFlush 是从 tail 节点开始的，并且是 outBound 的事件，所以这里会找到 tail 节点的上一个 outBoundHandler，有可能是编码器，也有可能是我们业务处理的 handler。if (executor.inEventLoop()) 判断是否是 eventLoop 线程，如果不是，则封装成 task 通过 nioEventLoop 异步执行，我们这里先按照是 eventLoop 线程分析。首先，这里通过 flush 判断是否调用了 flush，这里显然是 true，因为我们调用的方法是 writeAndFlush()方法，我们跟到 invokeWriteAndFlush 中：

```
private void invokeWriteAndFlush(Object msg, ChannelPromise promise) {
    if (invokeHandler()) {
        //写入
        invokeWrite0(msg, promise);
        //刷新
        invokeFlush0();
    } else {
        writeAndFlush(msg, promise);
    }
}
```

这里就真相大白了，其实在 writeAndFlush()方法中，首先调用 write，write 完成之后再调用 flush 方法进行的刷新。首先跟到 invokeWrite0()方法中：

```
private void invokeWrite0(Object msg, ChannelPromise promise) {
    try {
        //调用当前 handler 的 wirte()方法
        ((ChannelOutboundHandler) handler()).write(this, msg, promise);
    } catch (Throwable t) {
        notifyOutboundHandlerException(t, promise);
    }
}
```

该方法我们在 pipeline 中已经进行过分析，就是调用当前 handler 的 write 方法，如果当前 handler 中 write 方法是继续往下传播，在会继续传播写事件，直到传播到 head 节点，最后会走到 HeadContext 的 write 方法中 ,跟到 HeadContext 的 write 方法中：

```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
    unsafe.write(msg, promise);
}
```

这里通过当前 channel 的 unsafe 对象对将当前消息写到缓存中，回到到 invokeWriteAndFlush()方法中：

```
private void invokeWriteAndFlush(Object msg, ChannelPromise promise) {
    if (invokeHandler()) {
        //写入
        invokeWrite0(msg, promise);
        //刷新
        invokeFlush0();
    } else {
        writeAndFlush(msg, promise);
```

```
    }
}
```

我们再看 invokeFlush0()方法:

```
private void invokeFlush0() {
    try {
        ((ChannelOutboundHandler) handler()).flush(this);
    } catch (Throwable t) {
        notifyHandlerException(t);
    }
}
```

同样，这里会调用当前 handler 的 flush 方法，如果当前 handler 的 flush 方法是继续传播 flush 事件，则 flush 事件会继续往下传播，直到最后会调用 head 节点的 flush 方法，如果我们熟悉 pipeline 的话，对这里的逻辑不会陌生。跟到 HeadContext 的 flush 方法中：

```
public void flush(ChannelHandlerContext ctx) throws Exception {
    unsafe.flush();
}
```

这里同样，会通过当前 channel 的 unsafe 对象通过调用 flush 方法将缓存的数据刷新到 channel 中，有关刷新的逻辑，我们会在以后的小节进行剖析。以上就是 writeAndFlush 的相关逻辑，整体上比较简单，掌握了 Pipeline 的小伙伴应该很容易理解。

11.4.2 MessageToByteEncoder 抽象编码器

同解码器一样，编码器中也有一个抽象类叫 MessageToByteEncoder，其中定义了编码器的骨架方法，具体编码逻辑交给子类实现。解码器同样也是个 handler，将写出的数据进行截取处理，我们在学习 Pipeline 时我们知道，写数据的时候会传递 write 事件，传递过程中会调用 handler 的 write 方法，所以编码器码器可以重写 write 方法，将数据编码成二进制字节流然后再继续传递 write 事件。首先来看 MessageToByteEncoder 的类声明:MessageToByteEncoder 负责将 POJO 对象编码成 ByteBuf，用户的编码器继承 Message ToByteEncoder，实现 void encode(ChannelHandlerContext ctx, I msg, ByteBuf out)接口接口，示例代码如下：

```
    public class IntegerEncoder extends MessageToByteEncoder<Integer> {
        @Override
        public void encode(ChannelHandlerContext ctx, Integer msg,ByteBuf out)
           throws Exception {
               out.writeInt(msg);
        }
    }
```

它的实现原理如下：调用 write 操作时，首先判断当前编码器是否支持需要发送的消息，如果不支持则直接透传；如果支持则判断缓冲区的类型，对于直接内存分配 ioBuffer（堆外内存），对于堆内存通过 heapBuffer 方法分配，源码如下：

```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
        ByteBuf buf = null;
        try {
            if (acceptOutboundMessage(msg)) {
                @SuppressWarnings("unchecked")
                I cast = (I) msg;
                buf = allocateBuffer(ctx, cast, preferDirect);
                try {
                    encode(ctx, cast, buf);
                } finally {
                    ReferenceCountUtil.release(cast);
                }

                if (buf.isReadable()) {
                    ctx.write(buf, promise);
                } else {
                    buf.release();
                    ctx.write(Unpooled.EMPTY_BUFFER, promise);
                }
                buf = null;
            } else {
                ctx.write(msg, promise);
            }
        } catch (EncoderException e) {
            throw e;
        } catch (Throwable e) {
            throw new EncoderException(e);
        } finally {
            if (buf != null) {
                buf.release();
            }
        }
    }
```

编码使用的缓冲区分配完成之后，调用 encode 抽象方法进行编码，方法定义如下：它由子类负责具体实现。

```
protected abstract void encode(ChannelHandlerContext ctx, I msg, ByteBuf out) throws Exception;
```

编码完成之后，调用 ReferenceCountUtil 的 release 方法释放编码对象 msg。对编码后的 ByteBuf 进行以下判断：

1) 如果缓冲区包含可发送的字节，则调用 ChannelHandlerContext 的 write 方法发送 ByteBuf；

2) 如果缓冲区没有包含可写的字节，则需要释放编码后的 ByteBuf，写入一个空的 ByteBuf 到 ChannelHandlerContext 中。

发送操作完成之后，在方法退出之前释放编码缓冲区 ByteBuf 对象。

11.4.3 写入 Buffer 队列

前面的章节我们介绍过, writeAndFlush 方法其实最终会调用 write 和 flush 方法 ,write 方法最终会传递到 head 节点，调用 HeadContext 的 write 方法：

```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
    unsafe.write(msg, promise);
}
```

这里通过 unsafe 对象的 write 方法，将消息写入到缓存中。我们跟到 AbstractUnsafe 的 write 方法中：

```
public final void write(Object msg, ChannelPromise promise) {
    assertEventLoop();
    //负责缓冲写进来的 byteBuf
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        safeSetFailure(promise, WRITE_CLOSED_CHANNEL_EXCEPTION);
        ReferenceCountUtil.release(msg);
        return;
    }
    int size;
    try {
        //非堆外内存转化为堆外内存
        msg = filterOutboundMessage(msg);
        size = pipeline.estimatorHandle().size(msg);
        if (size < 0) {
            size = 0;
        }
    } catch (Throwable t) {
        safeSetFailure(promise, t);
        ReferenceCountUtil.release(msg);
        return;
    }
    //插入写队列
    outboundBuffer.addMessage(msg, size, promise);
}
```

首先看 ChannelOutboundBuffer outboundBuffer = this.outboundBuffer，ChannelOutboundBuffer 的功能就是缓存写入的 ByteBuf。我们继续看 try 块中的 msg = filterOutboundMessage(msg) ，这步的意义就是将非对外内存转化为堆外内存 ,filterOutboundMessage 方法方法最终会调用 AbstractNioByteChannel 中的 filterOutboundMessage 方法：

```
protected final Object filterOutboundMessage(Object msg) {
    if (msg instanceof ByteBuf) {
        ByteBuf buf = (ByteBuf) msg;
        //是堆外内存，直接返回
        if (buf.isDirect()) {
            return msg;
        }
        return newDirectBuffer(buf);
    }
    if (msg instanceof FileRegion) {
        return msg;
    }
    throw new UnsupportedOperationException(
            "unsupported message type: " + StringUtil.simpleClassName(msg) + EXPECTED_TYPES);
}
```

首先判断 msg 是否 byteBuf 对象，如果是，判断是否堆外内存，如果是堆外内存，则直接返回，否则，通过

newDirectBuffer(buf)这种方式转化为堆外内存。回到 write 方法中，outboundBuffer.addMessage(msg, size, promise)将已经转化为堆外内存的 msg 插入到写队列。我们跟到 addMessage()方法当中，这是 ChannelOutboundBuffer 中的方法：

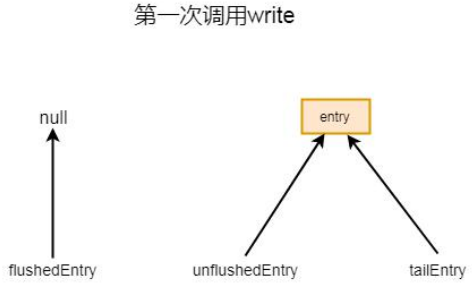
```
public void addMessage(Object msg, int size, ChannelPromise promise) {
    Entry entry = Entry.newInstance(msg, size, total(msg), promise);
    if (tailEntry == null) {
        flushedEntry = null;
        tailEntry = entry;
    } else {
        Entry tail = tailEntry;
        tail.next = entry;
        tailEntry = entry;
    }
    if (unflushedEntry == null) {
        unflushedEntry = entry;
    }
    incrementPendingOutboundBytes(size, false);
}
```

首先通过 Entry.newInstance(msg, size, total(msg), promise) 的方式将 msg 封装成 entry，然后通过调整 tailEntry, flushedEntry, unflushedEntry 三个指针，完成 entry 的添加。这三个指针均是 ChannelOutboundBuffer 的成员变量：

flushedEntry 指向第一个被 flush 的 entry

unflushedEntry 指向第一个未被 flush 的 entry

也就是说，从 flushedEntry 到 unflushedEntry 之间的 entry，都是被已经被 flush 的 entry。tailEntry 指向最后一个 entry，也就是从 unflushedEntry 到 tailEntry 之间的 entry 都是没 flush 的 entry。我们回到代码中，创建了 entry 之后首先判断尾指针是否为空，在第一次添加的时候，均是空，所以会将 flushedEntry 设置为 null，并且将尾指针设置为当前创建的 entry ，最后判断 unflushedEntry 是否为空，如果第一次添加这里也是空，所以这里将 unflushedEntry 设置为新创建的 entry。第一次添加如下图所示：



如果不是第一次调用 write 方法，则会进入 if (tailEntry == null) 中 else 块:

Entry tail = tailEntry 这里 tail 就是当前尾节点

tail.next = entry 代表尾节点的下一个节点指向新创建的 entry

tailEntry = entry 将尾节点也指向 entry

这样就完成了添加操作，其实就是将新创建的节点追加到原来尾节点之后，第二次添加 if (unflushedEntry == null) 会返回 false，所以不会进入 if 块。第二次添加之后指针的指向情况如下图所示：

以后每次调用 write，如果没有调用 flush 的话都会在尾节点之后进行追加。回到代码中，看这一步 incrementPendingOutboundBytes(size, false)，这步时统计当前有多少字节需要被写出，我们跟到这个方法中：

```
private void incrementPendingOutboundBytes(long size, boolean invokeLater) {
    if (size == 0) {
        return;
    }
    //TOTAL_PENDING_SIZE_UPDATER 当前缓冲区里面有多少待写的字节
    long newWriteBufferSize = TOTAL_PENDING_SIZE_UPDATER.addAndGet(this, size);
    //getWriteBufferHighWaterMark() 最高不能超过 64k
    if (newWriteBufferSize > channel.config().getWriteBufferHighWaterMark()) {
        setUnwritable(invokeLater);
    }
}
```

看这一步：long newWriteBufferSize = TOTAL_PENDING_SIZE_UPDATER.addAndGet(this, size)。TOTAL_PENDING_SIZE_UPDATER 表示当前缓冲区还有多少待写的字节，addAndGet 就是将当前的 ByteBuf 的长度进行累加，累加到 newWriteBufferSize 中。再继续看判断 if (newWriteBufferSize >

channel.config().getWriteBufferHighWaterMark()) 。channel.config().getWriteBufferHighWaterMark() 表示写 buffer 的高水位值，默认是 64KB，也就是说写 buffer 的最大长度不能超过 64KB。如果超过了 64KB，则会调用 setUnwritable(invokeLater)方法设置写状态，我们跟到 setUnwritable(invokeLater)方法中：

```
private void setUnwritable(boolean invokeLater) {
    for (;;) {
        final int oldValue = unwritable;
        final int newValue = oldValue | 1;
        if (UNWRITABLE_UPDATER.compareAndSet(this, oldValue, newValue)) {
            if (oldValue == 0 && newValue != 0) {
                fireChannelWritabilityChanged(invokeLater);
            }
            break;
        }
    }
}
```

这里通过自旋和 cas 操作，传播一个 ChannelWritabilityChanged 事件，最终会调用 handler 的 channelWritabilityChanged 方法进行处理，以上就是写 buffer 的相关逻辑。

11.4.4 刷新 Buffer 队列

通过前面的学习我们知道，flush 方法通过事件传递，最终会传递到 HeadContext 的 flush 方法：

```
public void flush(ChannelHandlerContext ctx) throws Exception {
    unsafe.flush();
}
```

这里最终会调用 AbstractUnsafe 的 flush 方法：

```
public final void flush() {
    assertEventLoop();
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        return;
    }
    outboundBuffer.addFlush();
    flush0();
}
```

这里首先也是拿到 ChannelOutboundBuffer 对象，然后我们看这一步：

```
outboundBuffer.addFlush();
```

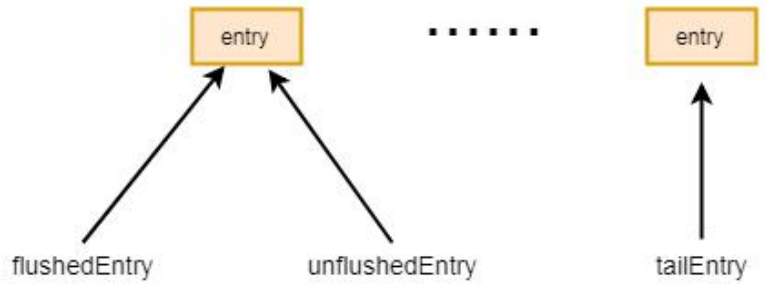
这一步同样也是调整 ChannelOutboundBuffer 的指针，跟进 addFlush 方法：

```
public void addFlush() {
    Entry entry = unflushedEntry;
    if (entry != null) {
        if (flushedEntry == null) {
            flushedEntry = entry;
        }
        do {
```

```
            flushed ++;
            if (!entry.promise.setUncancellable()) {
                int pending = entry.cancel();
                decrementPendingOutboundBytes(pending, false, true);
            }
            entry = entry.next;
        } while (entry != null);
        unflushedEntry = null;
    }
}
```

首先声明一个 entry 指向 unflushedEntry，也就是第一个未 flush 的 entry。通常情况下 unflushedEntry 是不为空的，所以进入 if，再未刷新前 flushedEntry 通常为空，所以会执行到 flushedEntry = entry，也就是 flushedEntry 指向 entry。

经过上述操作，缓冲区的指针情况如图所示：



然后通过 do-while 将，不断寻找 unflushedEntry 后面的节点，直到没有节点为止 ,flushed 自增代表需要刷新多少个节点。循环中我们关注这一步：

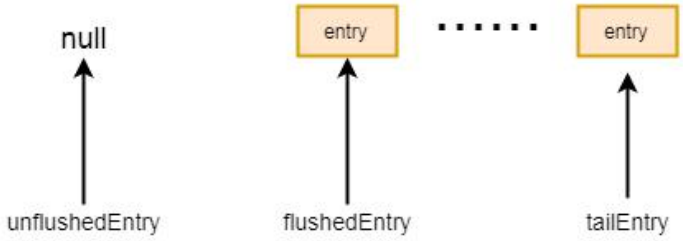
```
decrementPendingOutboundBytes(pending, false, true);
```

这一步也是统计缓冲区中的字节数，但是是和上一小节的 incrementPendingOutboundBytes 正好是相反，因为这里是刷新，所以这里要减掉刷新后的字节数，我们跟到方法中：

```
private void decrementPendingOutboundBytes(long size, boolean invokeLater, boolean notifyWritability) {
    if (size == 0) {
        return;
    }
    //从总的大小减去
    long newWriteBufferSize = TOTAL_PENDING_SIZE_UPDATER.addAndGet(this, -size);
    //直到减到小于某一个阈值 32 个字节
    if (notifyWritability && newWriteBufferSize < channel.config().getWriteBufferLowWaterMark()) {
        //设置写状态
        setWritable(invokeLater);
    }
}
```

同样 TOTAL_PENDING_SIZE_UPDATER 代表缓冲区的字节数，这里的 addAndGet 中参数是-size，也就是减掉 size 的长

度。再看 if (notifyWritability && newWriteBufferSize < channel.config().getWriteBufferLowWaterMark()) 。getWriteBufferLowWaterMark()代表写 buffer 的第水位值，也就是 32k，如果写 buffer 的长度小于这个数，就通过 setWritable 方法设置写状态，也就是通道由原来的不可写改成可写。回到 addFlush 方法，遍历 do-while 循环结束之后，将 unflushedEntry 指为空，代表所有的 entry 都是可写的。经过上述操作，缓冲区的指针情况如下图所示：



回到 AbstractUnsafe 的 flush 方法，指针调整完之后，我们跟到 flush0()方法中：

```
protected void flush0() {
    if (inFlush0) {
        return;
    }
    final ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null || outboundBuffer.isEmpty()) {
        return;
    }
    inFlush0 = true;
    if (!isActive()) {
        try {
            if (isOpen()) {
                outboundBuffer.failFlushed(FLUSH0_NOT_YET_CONNECTED_EXCEPTION, true);
            } else {
                outboundBuffer.failFlushed(FLUSH0_CLOSED_CHANNEL_EXCEPTION, false);
            }
        } finally {
            inFlush0 = false;
        }
        return;
    }
    try {
        doWrite(outboundBuffer);
    } catch (Throwable t) {
        if (t instanceof IOException && config().isAutoClose()) {
            close(voidPromise(), t, FLUSH0_CLOSED_CHANNEL_EXCEPTION, false);
        } else {
            outboundBuffer.failFlushed(t, true);
        }
    } finally {
        inFlush0 = false;
    }
}
```

if (inFlush0) 表示判断当前 flush 是否在进行中，如果在进行中，则返回，避免重复进入。我们重点关注 doWrite()方法，跟到 AbstractNioByteChannel 的 doWrite 方法中去：

```
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    int writeSpinCount = -1;
    boolean setOpWrite = false;
    for (;;) {
        //每次拿到当前节点
        Object msg = in.current();
        if (msg == null) {
            clearOpWrite();
            return;
        }
        if (msg instanceof ByteBuf) {
            //转化成 ByteBuf
            ByteBuf buf = (ByteBuf) msg;
            //如果没有可写的值
            int readableBytes = buf.readableBytes();
            if (readableBytes == 0) {
                //移除
                in.remove();
                continue;
            }
            boolean done = false;
            long flushedAmount = 0;
            if (writeSpinCount == -1) {
                writeSpinCount = config().getWriteSpinCount();
            }
            for (int i = writeSpinCount - 1; i >= 0; i --) {
                //将 buf 写入到 socket 里面
                //localFlushedAmount 代表向 jdk 底层写了多少字节
                int localFlushedAmount = doWriteBytes(buf);
                //如果一个字节没写，直接 break
                if (localFlushedAmount == 0) {
                    setOpWrite = true;
                    break;
                }
                //统计总共写了多少字节
                flushedAmount += localFlushedAmount;
                //如果 buffer 全部写到 jdk 底层
                if (!buf.isReadable()) {
                    //标记全写道
                    done = true;
                    break;
                }
            }
            in.progress(flushedAmount);
            if (done) {
                //移除当前对象
                in.remove();
            } else {
                break;
            }
        } else if (msg instanceof FileRegion) {
            //代码省略
        } else {
            throw new Error();
        }
    }
    incompleteWrite(setOpWrite);
```

```
}
```

首先是一个无限 for 循环，Object msg = in.current() 这一步是拿到 flushedEntry 指向的 entry 中的 msg，跟到 current()

方法中：

```
public Object current() {
    Entry entry = flushedEntry;
    if (entry == null) {
        return null;
    }
    return entry.msg;
}
```

这里直接拿到 flushedEntry 指向的 entry 中关联的 msg，也就是一个 ByteBuf。回到 doWrite 方法：

如果 msg 为 null，说明没有可以刷新的 entry，则调用 clearOpWrite()方法清除写标识；

如果 msg 不为 null，则会判断是否是 ByteBuf 类型，如果是 ByteBuf，就进入 if 块中的逻辑。

if 块中首先将 msg 转化为 ByteBuf，然后判断 ByteBuf 是否可读，如果不可读，则通过 in.remove()将当前的 byteBuf 所关联的 entry 移除，然后跳过这次循环进入下次循环。remove 方法稍后分析，这里我们先继续往下看： boolean done = false 这里设置一个标识，标识刷新操作是否执行完成，这里默认值为 false 代表走到这里没有执行完。

writeSpinCount = config().getWriteSpinCount() 这里是获得一个写操作的循环次数，默认是 16，然后根据这个循环次数，进行循环的写操作。在循环中，关注这一步：

```
int localFlushedAmount = doWriteBytes(buf);
```

这一步就是将 buf 的内容写到 channel 中，并返回写的字节数，这里会调用 NioSocketChannel 的 doWriteBytes，我们跟到 doWriteBytes()方法中：

```
protected int doWriteBytes(ByteBuf buf) throws Exception {
    final int expectedWrittenBytes = buf.readableBytes();
    return buf.readBytes(javaChannel(), expectedWrittenBytes);
}
```

这里首先拿到 buf 的可读字节数，然后通过 readBytes 将可读字节写入到 jdk 底层的 channel 中。回到 doWrite 方法，将内容写的 jdk 底层的 channel 之后，如果一个字节都没写，说明现在 channel 可能不可写，将 setOpWrite 设置为 true，用于标识写操作位，并退出循环。如果已经写出字节，则通过 flushedAmount += localFlushedAmount 累加写出的字节数，然后根据是 buf 是否没有可读字节数判断是否 buf 的数据已经写完，如果写完，将 done 设置为 true，说明写操作完成，并退出循环。因为有时候不一定一次就能将 byteBuf 所有的字节写完，所以这里会继续通过循环进行写出，直到循环到 16 次。如果 ByteBuf 内容完全写完，会通过 in.remove()将当前 entry 移除掉，我们跟到 remove 方法中：

```
public boolean remove() {
    //拿到当前第一个 flush 的 entry
    Entry e = flushedEntry;
    if (e == null) {
        clearNioBuffers();
        return false;
    }
    Object msg = e.msg;
    ChannelPromise promise = e.promise;
    int size = e.pendingSize;
    removeEntry(e);
    if (!e.cancelled) {
        ReferenceCountUtil.safeRelease(msg);
        safeSuccess(promise);
        decrementPendingOutboundBytes(size, false, true);
    }
    e.recycle();
    return true;
}
```

首先拿到当前的 flushedEntry，我们重点关注 removeEntry 这步，跟进去：

```
private void removeEntry(Entry e) {
    if (-- flushed == 0) {
        //位置为空
        flushedEntry = null;
        //如果是最后一个节点
        if (e == tailEntry) {
            //全部设置为空
            tailEntry = null;
            unflushedEntry = null;
        }
    } else {
        //移动到下一个节点
        flushedEntry = e.next;
    }
}
```

if (-- flushed == 0) 表示当前节点是否为需要刷新的最后一个节点，如果是，则 flushedEntry 指针设置为空。如果当前节点是 tailEntry 节点，说明当前节点是最后一个节点，将 tailEntry 和 unflushedEntry 两个指针全部设置为空。如果当前节点不是需要刷新的最后的一个节点，则通过 flushedEntry = e.nex t 这步将 flushedEntry 指针移动到下一个节点。

以上就是 flush 操作的相关逻辑。

11.4.5 数据输出回调

首先我们看一段写在 handler 中的业务代码：

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    ChannelFuture future = ctx.writeAndFlush("test data");
    future.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws Exception {
            if (future.isSuccess()){
                System.out.println("写出成功");
```

```
            }else{
                System.out.println("写出失败");
            }
        }
    });
}
```

这种写法小伙伴们应该已经不陌生了，首先调用 writeAndFlush 方法将数据写出，然后返回的 future 进行添加 Listener，并且重写回调函数。这只是一个最简单的示例，在回调函数中判断 future 的状态成功与否，成功的话就打印"写出成功"，否则节打印"写出失败"。这里如果写在 handler 中通常是 NioEventLoop 线程执行的，在 future 返回之后才会执行添加 listener 的操作，如果在用户线程中 writeAndFlush 是异步执行的，在添加监听的时候有可能写出操作没有执行完毕，等写出操作执行完毕之后才会执行回调。以上逻辑在代码中如何体现的呢?我们首先跟到 writeAndFlush 的方法中去，会走到 AbstractChannelHandlerContext 中的 writeAndFlush 方法中：

```
public ChannelFuture writeAndFlush(Object msg) {
    return writeAndFlush(msg, newPromise());
}
```

这里的逻辑在之前的章节中剖析过，想必大家并不陌生，我们重点关注 newPromise()方法，跟进去：

```
public ChannelPromise newPromise() {
    return new DefaultChannelPromise(channel(), executor());
}
```

这里直接创建了 DefaultChannelPromise 这个对象并传入了当前 channel 和当前 channel 绑定 NioEventLoop 对象。在 DefaultChannelPromise 构造方法中，也会将 channel 和 NioEventLoop 对象绑定在自身成员变量中。回到 writeAndFlush()方法继续跟：

```
public ChannelFuture writeAndFlush(Object msg, ChannelPromise promise) {
    if (msg == null) {
        throw new NullPointerException("msg");
    }
    if (!validatePromise(promise, true)) {
        ReferenceCountUtil.release(msg);
        return promise;
    }
    write(msg, true, promise);
    return promise;
}
```

这里的逻辑也不陌生，注意这里最后返回了 promise，其实就是我们上一步创建 DefaultChannelPromise 对象，DefaultChannelPromise 实现了 ChannelFuture 接口，所以方法如果返回该对象可以被 ChannelFuture 类型接收。我们继续跟 write 方法：

```
private void write(Object msg, boolean flush, ChannelPromise promise) {
    AbstractChannelHandlerContext next = findContextOutbound();
    final Object m = pipeline.touch(msg, next);
```

```
    EventExecutor executor = next.executor();
    if (executor.inEventLoop()) {
        if (flush) {
            next.invokeWriteAndFlush(m, promise);
        } else {
            next.invokeWrite(m, promise);
        }
    } else {
        AbstractWriteTask task;
        if (flush) {
            task = WriteAndFlushTask.newInstance(next, m, promise);
        }  else {
            task = WriteTask.newInstance(next, m, promise);
        }
        safeExecute(executor, task, promise, m);
    }
}
```

这里的逻辑我们同样不陌生，如果 nioEventLoop 线程，我们继续调 invokeWriteAndFlush 方法，如果不是 nioEventLoop 线程则将 writeAndFlush 事件封装成 task，交给 eventLoop 线程异步。这里如果是异步执行，则到这一步之后，我们的业务代码中，writeAndFlush 就会返回并添加监听，有关添加监听的逻辑稍后分析。走到这里，无论同步异步，都会执行到 invokeWriteAndFlush 方法：

```
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
    unsafe.write(msg, promise);
}
```

这里最终调用 unsafe 的 write 方法，并传入了 promise 对象，跟到 AbstractUnsafe 的 write 方法中：

```
public final void write(Object msg, ChannelPromise promise) {
    assertEventLoop();

    //负责缓冲写进来的 byteBuf
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        safeSetFailure(promise, WRITE_CLOSED_CHANNEL_EXCEPTION);
        ReferenceCountUtil.release(msg);
        return;
    }

    int size;
    try {
        msg = filterOutboundMessage(msg);
        size = pipeline.estimatorHandle().size(msg);
        if (size < 0) {
            size = 0;
        }
    } catch (Throwable t) {
        safeSetFailure(promise, t);
        ReferenceCountUtil.release(msg);
        return;
    }

    //插入写队列
    outboundBuffer.addMessage(msg, size, promise);
}
```

这里的逻辑之前剖析过，这里我们首先关注两个部分，首先看在 catch 中 safeSetFailure 这步。因为是 catch 块，说明发

生了异常，写到缓冲区不成功，safeSetFailure 就是设置写出失败的状态。我们跟到 safeSetFailure 方法中：

```
protected final void safeSetFailure(ChannelPromise promise, Throwable cause) {
    if (!(promise instanceof VoidChannelPromise) && !promise.tryFailure(cause)) {
        logger.warn("Failed to mark a promise as failure because it's done already: {}", promise, cause);
    }
}
```

这里看 if 判断，首先我们的 promise 是 DefaultChannelPromise，所以!(promise instanceof VoidChannelPromise)为 true。重点分析 promise.tryFailure(cause)，这里是设置失败状态，这里会调用 DefaultPromise 的 tryFailure 方法，跟进 tryFailure 方法：

```
public boolean tryFailure(Throwable cause) {
    if (setFailure0(cause)) {
        notifyListeners();
        return true;
    }
    return false;
}
```

再跟到 setFailure0(cause)中：

```
private boolean setValue0(Object objResult) {
    if (RESULT_UPDATER.compareAndSet(this, null, objResult) ||
        RESULT_UPDATER.compareAndSet(this, UNCANCELLABLE, objResult)) {
        checkNotifyWaiters();
        return true;
    }
    return false;
}
```

这里在 if 块中的 cas 操作，会将参数 objResult 的值设置到 DefaultPromise 的成员变量 result 中，表示当前操作为异常状态。

回到 tryFailure 方法，我们关注 notifyListeners()这个方法，这个方法是执行添加监听的回调函数，当 writeAndFlush 和 addListener 是异步执行的时候，这里有可能添加已经添加，所以通过这个方法可以调用添加监听后的回调。如果 writeAndFlush 和 addListener 是同步执行的时候，也就是都在 NioEventLoop 线程中执行的时候，那么走到这里 addListener 还没执行，所以这里不能回调添加监听的回调函数，那么回调是什么时候执行的呢?我们在剖析 addListener 步骤的时候会给大家分析。具体执行回调我们再讲解添加监听的时候进行剖析，以上就是记录异常状态的大概逻辑。回到 AbstractUnsafe 的 write 方法，我们再关注这一步：outboundBuffer.addMessage(msg, size, promise);

跟到 addMessage()方法中：

```
public void addMessage(Object msg, int size, ChannelPromise promise) {
    Entry entry = Entry.newInstance(msg, size, total(msg), promise);
    //代码省略
```

```
}
```

我们只需要关注包装 Entry 的 newInstance 方法，该方法传入 promise 对象，跟到 newInstance 中：

```
static Entry newInstance(Object msg, int size, long total, ChannelPromise promise) {
    Entry entry = RECYCLER.get();
    entry.msg = msg;
    entry.pendingSize = size;
    entry.total = total;
    entry.promise = promise;
    return entry;
}
```

这里将 promise 设置到 Entry 的成员变量中了，也就是说，每个 Entry 都关联了唯一的一个 promise，我们回到 AbstractChannelHandlerContext 的 invokeWriteAndFlush 方法中：

```
private void invokeWriteAndFlush(Object msg, ChannelPromise promise) {
    if (invokeHandler()) {
        invokeWrite0(msg, promise);
        invokeFlush0();
    } else {
        writeAndFlush(msg, promise);
    }
}
```

我们刚才分析了 write 操作中 promise 的传递以及状态设置的大概过程，我们继续看在 flush 中 promise 的操作过程。这里 invokeFlush0()并没有传入 promise 对象，是因为我们刚才分析过，promise 对象会绑定在缓冲区中 entry 的成员变量中，可以通过其成员变量拿到 promise 对象。invokeFlush0()我们之前也分析过，通过事件传递，最终会调用 HeadContext 的 flush 方法：

```
public void flush(ChannelHandlerContext ctx) throws Exception {
    unsafe.flush();
}
```

最后跟到 AbstractUnsafe 的 flush 方法：

```
public final void flush() {
    assertEventLoop();
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        return;
    }
    outboundBuffer.addFlush();
    flush0();
}
```

这块逻辑之前已分析过，继续看 flush0 方法：

```
protected void flush0() {
    //代码省略
    try {
        doWrite(outboundBuffer);
    } catch (Throwable t) {
        //代码省略
    } finally {
        inFlush0 = false;
```

```
    }
}
```

篇幅原因我们省略大段代码，我们继续跟进 doWrite 方法：

```
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    int writeSpinCount = -1;
    boolean setOpWrite = false;
    for (;;) {
        Object msg = in.current();
        if (msg == null) {
            clearOpWrite();
            return;
        }
        if (msg instanceof ByteBuf) {
            //代码省略
            boolean done = false;
            //代码省略
            if (done) {
                //移除当前对象
                in.remove();
            } else {
                break;
            }
        } else if (msg instanceof FileRegion) {
            //代码省略
        } else {
            throw new Error();
        }
    }
    incompleteWrite(setOpWrite);
}
```

这里也省略了大段代码，我们重点关注 in.remove()这里，之前介绍过，如果 done 为 true，说明刷新事件已完成，则移除当前 entry 节点，我们跟到 remove()方法中：

```
public boolean remove() {
    Entry e = flushedEntry;
    if (e == null) {
        clearNioBuffers();
        return false;
    }
    Object msg = e.msg;
    ChannelPromise promise = e.promise;
    int size = e.pendingSize;
    removeEntry(e);
    if (!e.cancelled) {
        ReferenceCountUtil.safeRelease(msg);
        safeSuccess(promise);
        decrementPendingOutboundBytes(size, false, true);
    }
    e.recycle();
    return true;
}
```

这里我们看这一步:

```
ChannelPromise promise = e.promise;
```

之前我们剖析过 promise 对象会绑定在 entry 中，而这步就是从 entry 中获取 promise 对象，等 remove 操作完成，会

执行到这一步：

```
safeSuccess(promise);
```

这一步正好和我们刚才分析的 safeSetFailure 相反，这里是设置成功状态，跟到 safeSuccess 方法中：

```
private static void safeSuccess(ChannelPromise promise) {
    if (!(promise instanceof VoidChannelPromise)) {
        PromiseNotificationUtil.trySuccess(promise, null, logger);
    }
}
```

再跟到 trySuccess 方法中：

```
public static <V> void trySuccess(Promise<? super V> p, V result, InternalLogger logger) {
    if (!p.trySuccess(result) && logger != null) {
        //代码省略
    }
}
```

这里再继续跟 if 中的 trySuccess 方法，最后会走到 DefaultPromise 的 trySuccess 方法：

```
public boolean trySuccess(V result) {
    if (setSuccess0(result)) {
        notifyListeners();
        return true;
    }
    return false;
}
```

这里跟到 setSuccess0 方法中：

```
private boolean setSuccess0(V result) {
    return setValue0(result == null ? SUCCESS : result);
}
```

这里的逻辑我们刚才剖析过了，这里参数传入一个信号 SUCCESS，表示设置成功状。再继续跟 setValue 方法：

```
private boolean setValue0(Object objResult) {
    if (RESULT_UPDATER.compareAndSet(this, null, objResult) ||
        RESULT_UPDATER.compareAndSet(this, UNCANCELLABLE, objResult)) {
        checkNotifyWaiters();
        return true;
    }
    return false;
}
```

同样，在 if 判断中，通过 cas 操作将参数传入的 SUCCESS 对象赋值到 DefaultPromise 的属性 result 中，我们看这个属性: private volatile Object result; 这里是 Object 类型，也就是可以赋值成任何类型。SUCCESS 是一个 Signal 类型的对象，这里我们可以简单理解成一种状态，SUCCESS 表示一种成功的状态。通过上述 cas 操作，result 的值将赋值成 SUCCESS，我们回到 trySuccess 方法：

```
public boolean trySuccess(V result) {
    if (setSuccess0(result)) {
        notifyListeners();
        return true;
    }
    return false;
```

```
}
```

设置完成功状态之后，则会通过 notifyListeners()执行监听中的回调。我们看用户代码：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    ChannelFuture future = ctx.writeAndFlush("test data");
    future.addListener(new ChannelFutureListener() {
        @Override
        public void operationComplete(ChannelFuture future) throws Exception {
            if (future.isSuccess()){
                System.out.println("写出成功");
            }else{
                System.out.println("写出失败");
            }
        }
    });
}
```

在回调中会判断 future.isSuccess(), promise 设置为成功状态这里会返回 true，从而打印写出成功"。跟到 isSuccess 方法中，这里会调用 DefaultPromise 的 isSuccess 方法：

```
public boolean isSuccess() {
    Object result = this.result;
    return result != null && result != UNCANCELLABLE && !(result instanceof CauseHolder);
}
```

我们看到首先会拿到 result 对象，然后判断 result 不为空，并且不是 UNCANCELLABLE，并且不属于 CauseHolder 对象.

我们刚才分析如果 promise 设置为成功装载，则 result 为 SUCCESS，所以这里条件成立，可以执行 if (future.isSuccess()) 中 if 块的逻辑。和设置错误状态的逻辑一样，这里也有同样的问题，如果 writeAndFlush 是和 addListener 是异步操作，那么执行到回调的时候，可能 addListener 已经添加完成，所以可以正常的执行回调。那么如果 writeAndFlush 是和 addListener 是同步操作, writeAndFlush 在执行回调的时候, addListener 并没有执行，所以无法执行回调方法，那么回调方法是如何执行的呢, ? 我们看 addListener 这个方法，addListener 传入 ChannelFutureListener 对象，并重写了 operationComplete 方法，也就是执行回调的方法，会执行到 DefaultChannelPromise 的 addListener 方法，跟进去：

```
public ChannelPromise addListener(GenericFutureListener<? extends Future<? super Void>> listener) {
    super.addListener(listener);
    return this;
}
```

跟到父类的 addListener 中:

```
public Promise<V> addListener(GenericFutureListener<? extends Future<? super V>> listener) {
    checkNotNull(listener, "listener");
    synchronized (this) {
        addListener0(listener);
    }
    if (isDone()) {
        notifyListeners();
    }
    return this;
```

```
}
```

这里通过 addListener0 方法添加 listener，因为添加 listener 有可能会在不同的线程中操作，比如用户线程和 NioEventLoop 线程，为了防止并发问题，这里简单粗暴的加了个 synchronized 关键字。跟到 addListener0 方法中：

```
private void addListener0(GenericFutureListener<? extends Future<? super V>> listener) {
    if (listeners == null) {
        listeners = listener;
    } else if (listeners instanceof DefaultFutureListeners) {
        ((DefaultFutureListeners) listeners).add(listener);
    } else {
        listeners = new DefaultFutureListeners((GenericFutureListener<? extends Future<V>>) listeners, listener);
    }
}
```

如果是第一次添加 listener，则成员变量 listeners 为 null，这样就把参数传入的 GenericFutureListener 赋值到成员变量 listeners。如果是第二次添加 listener，listeners 不为空，会走到 else if 判断，因为第一次添加的 listener 是 GenericFutureListener 类型，并不是 DefaultFutureListeners 类型，所以 else if 判断返回 false，进入到 else 块中。else 块中，通过 new 的方式创建一个 DefaultFutureListeners 对象并赋值到成员变量 listeners 中。DefaultFutureListeners 的构造方法中，第一个参数传入 DefaultPromise 中的成员变量 listeners，也就是第一次添加的 GenericFutureListener 对象，第二个参数为第二次添加的 GenericFutureListener 对象，这里通过两个 GenericFutureListener 对象包装成一个 DefaultFutureListeners 对象。我们看 listeners 的定义：

```
private Object listeners;
```

这里是个 Object 类型，所以可以保存任何类型的对象。再看 DefaultFutureListeners 的构造方法：

```
DefaultFutureListeners(
        GenericFutureListener<? extends Future<?>> first, GenericFutureListener<? extends Future<?>> second) {
    listeners = new GenericFutureListener[2];
    //第 0 个
    listeners[0] = first;
    //第 1 个
    listeners[1] = second;
    size = 2;
    //代码省略
}
```

在 DefaultFutureListeners 类中也定义了一个成员变量 listeners，类型为 GenericFutureListener 数组。构造方法中初始化 listeners 这个数组，并且数组中第一个值赋值为我们第一次添加的 GenericFutureListener，第二个赋值为我们第二次添加的 GenericFutureListener。回到 addListener0 方法中：

```
private void addListener0(GenericFutureListener<? extends Future<? super V>> listener) {
    if (listeners == null) {
        listeners = listener;
    } else if (listeners instanceof DefaultFutureListeners) {
        ((DefaultFutureListeners) listeners).add(listener);
```

```
    } else {
        listeners = new DefaultFutureListeners((GenericFutureListener<? extends Future<V>>) listeners, listener);
    }
}
```

经过两次添加 listener，属性 listeners 的值就变成了 DefaultFutureListeners 类型的对象，如果第三次添加 listener，则会走到 else if 块中，DefaultFutureListeners 对象通过调用 add 方法继续添加 listener。跟到 add 方法中：

```
public void add(GenericFutureListener<? extends Future<?>> l) {
    GenericFutureListener<? extends Future<?>>[] listeners = this.listeners;
    final int size = this.size;
    if (size == listeners.length) {
        this.listeners = listeners = Arrays.copyOf(listeners, size << 1);
    }
    listeners[size] = l;
    this.size = size + 1;
    //代码省略
}
```

这里的逻辑也比较简单，就是为当前的数组对象 listeners 中追加新的 GenericFutureListener 对象，如果 listeners 容量不足则进行扩容操作。根据以上逻辑，就完成了 listener 的添加逻辑。那么再看我们刚才遗留的问题，如果 writeAndFlush 和 addListener 是同步进行的，writeAndFlush 执行回调时还没有 addListener 还没有执行回调，那么回调是如何执行的呢?回到 DefaultPromise 的 addListener 中：

```
public Promise<V> addListener(GenericFutureListener<? extends Future<? super V>> listener) {
    checkNotNull(listener, "listener");
    synchronized (this) {
        addListener0(listener);
    }
    if (isDone()) {
        notifyListeners();
    }
    return this;
}
```

我们分析完了 addListener0 方法，再往下看。这个会有 if 判断 isDone(), isDone 方法，就是程序执行到这一步的时候，判断刷新事件是否执行完成，跟到 isDone 方法中：

```
public boolean isDone() {
    return isDone0(result);
}
```

继续跟 isDone0，这里传入了成员变量 result。

```
private static boolean isDone0(Object result) {
    return result != null && result != UNCANCELLABLE;
}
```

这里判断 result 不为 null 并且不为 UNCANCELLABLE，则就表示完成。因为成功的状态是 SUCCESS，所以 flush 成功这里会返回 true。回到 addListener 中 ,如果执行完成，就通过 notifyListeners()方法执行回调，这也解释刚才的问题，在同步操作中，writeAndFlush 在执行回调时并没有添加 listener，所以添加 listener 的时候会判断 writeAndFlush 的执行状

态，如果状态时完成，则会这里执行回调。同样，在异步操作中，走到这里 writeAndFlush 可能还没完成，所以这里不会执行回调，由 writeAndFlush 执行回调。所以，无论 writeAndFlush 和 addListener 谁先完成，都可以执行到回调方法。

跟到 notifyListeners()方法中：

```
private void notifyListeners() {
    EventExecutor executor = executor();
    if (executor.inEventLoop()) {
        final InternalThreadLocalMap threadLocals = InternalThreadLocalMap.get();
        final int stackDepth = threadLocals.futureListenerStackDepth();
        if (stackDepth < MAX_LISTENER_STACK_DEPTH) {
            threadLocals.setFutureListenerStackDepth(stackDepth + 1);
            try {
                notifyListenersNow();
            } finally {
                threadLocals.setFutureListenerStackDepth(stackDepth);
            }
            return;
        }
    }
    safeExecute(executor, new Runnable() {
        @Override
        public void run() {
            notifyListenersNow();
        }
    });
}
```

这里首先判断是否是 eventLoop 线程，如果是 eventLoop 线程则执行 if 块中的逻辑，如果不是 eventLoop 线程，则把执行回调的逻辑封装成 task 丢到 EventLoop 的任务队列中异步执行。我们重点关注 notifyListenersNow()方法，跟进去：

```
private void notifyListenersNow() {
    Object listeners;
    synchronized (this) {
        if (notifyingListeners || this.listeners == null) {
            return;
        }
        notifyingListeners = true;
        listeners = this.listeners;
        this.listeners = null;
    }
    for (;;) {
        if (listeners instanceof DefaultFutureListeners) {
            notifyListeners0((DefaultFutureListeners) listeners);
        } else {
            notifyListener0(this, (GenericFutureListener<? extends Future<V>>) listeners);
        }
        //代码省略
    }
}
```

在无限 for 循环中，首先首先判断 listeners 是不是 DefaultFutureListeners 类型，根据我们之前的逻辑，如果只添加了一个 listener，则 listeners 是 GenericFutureListener 类型。通常在添加的时候只会添加一个 listener，所以我们跟到 else 块中的 notifyListener0 方法：

```
private static void notifyListener0(Future future, GenericFutureListener l) {
    try {
        l.operationComplete(future);
    } catch (Throwable t) {
        logger.warn("An exception was thrown by " + l.getClass().getName() + ".operationComplete()", t);
    }
}
```

我们看到，这里执行了 GenericFutureListener 的中我们重写的回调函数 operationComplete。以上就是执行回调的相关逻辑。

11.5 自定义编、解码

尽管 Netty 预置了丰富的编解码类库功能，但是在实际的业务开发过程中，总是需要对编解码功能做一些定制。使用 Netty 的编解码框架，可以非常方便的进行协议定制。本章节将对常用的支持定制的编解码类库进行讲解，以期让读者能够尽快熟悉和掌握编解码框架。

11.5.1 MessageToMessageDecoder 抽象解码器

MessageToMessageDecoder 实际上是 Netty 的二次解码器，它的职责是将一个对象二次解码为其它对象。

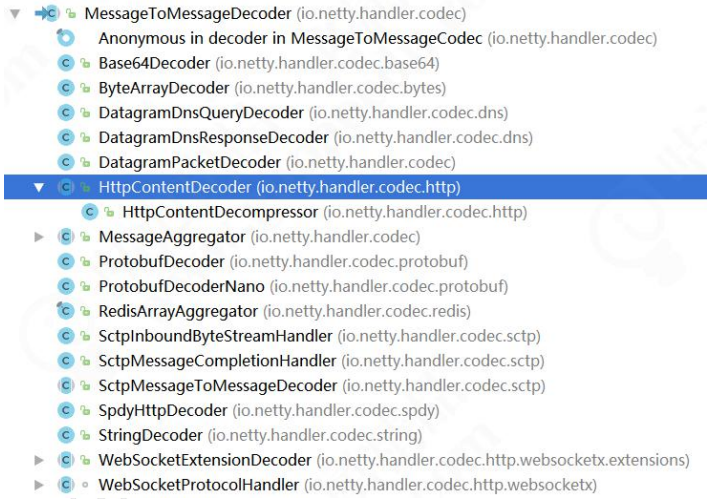
为什么称它为二次解码器呢？我们知道，从 SocketChannel 读取到的 TCP 数据报是 ByteBuffer，实际就是字节数组。我们首先需要将 ByteBuffer 缓冲区中的数据报读取出来，并将其解码为 Java 对象；然后对 Java 对象根据某些规则做二次解码，将其解码为另一个 POJO 对象。因为 MessageToMessageDecoder 在 ByteToMessageDecoder 之后，所以称之为二次解码器。

二次解码器在实际的商业项目中非常有用，以 HTTP+XML 协议栈为例，第一次解码往往是将字节数组解码成 HttpRequest 对象，然后对 HttpRequest 消息中的消息体字符串进行二次解码，将 XML 格式的字符串解码为 POJO 对象，这就用到了二次解码器。类似这样的场景还有很多，不再一一枚举。

事实上，做一个超级复杂的解码器将多个解码器组合成一个大而全的 MessageToMessageDecoder 解码器似乎也能解决多次解码的问题，但是采用这种方式的代码可维护性会非常差。例如，如果我们打算在 HTTP+XML 协议栈中增加一个打印码流的功能，即首次解码获取 HttpRequest 对象之后打印 XML 格式的码流。如果采用多个解码器组合，在中间

插入一个打印消息体的 Handler 即可，不需要修改原有的代码；如果做一个大而全的解码器，就需要在解码的方法中增加打印码流的代码，可扩展性和可维护性都会变差。

用户的解码器只需要实现 void decode(ChannelHandlerContext ctx, I msg, List<Object> out)抽象方法即可，由于它是将一个 POJO 解码为另一个 POJO，所以一般不会涉及到半包的处理，相对于 ByteToMessageDecoder 更加简单些。它的继承关系图如下所示：



11.5.2 MessageToMessageEncoder 抽象编码器

将一个 POJO 对象编码成另一个对象，以 HTTP+XML 协议为例，它的一种实现方式是：先将 POJO 对象编码成 XML 字符串，再将字符串编码为 HTTP 请求或者应答消息。对于复杂协议，往往需要经历多次编码，为了便于功能扩展，可以通过多个编码器组合来实现相关功能。

用户的解码器继承 MessageToMessageEncoder 解码器，实现 void encode(Channel HandlerContext ctx, I msg, List<Object> out)方法即可。注意，它与 MessageToByteEncoder 的区别是输出是对象列表而不是 ByteBuf，示例代码如下：

```
public class IntegerToStringEncoder extends MessageToMessageEncoder <Integer> {
        @Override
        public void encode(ChannelHandlerContext ctx, Integer message,
           List<Object> out)
                throws Exception
        {
            out.add(message.toString());
        }
```

```
    }
```

MessageToMessageEncoder 编码器的实现原理与之前分析的 MessageToByteEncoder 相似，唯一的差别是它编码后的输出是个中间对象，并非最终可传输的 ByteBuf。

简单看下它的源码实现：创建 RecyclableArrayList 对象，判断当前需要编码的对象是否是编码器可处理的类型，如果不是，则忽略，执行下一个 ChannelHandler 的 write 方法。

具体的编码方法实现由用户子类编码器负责完成，如果编码后的 RecyclableArrayList 为空，说明编码没有成功，释放 RecyclableArrayList 引用。

如果编码成功，则通过遍历 RecyclableArrayList，循环发送编码后的 POJO 对象，代码如下所示：

```
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
           CodecOutputList out = null;
           try {
               if (acceptOutboundMessage(msg)) {
                   out = CodecOutputList.newInstance();
                   @SuppressWarnings("unchecked")
                   I cast = (I) msg;
                   try {
                       encode(ctx, cast, out);
                   } finally {
                       ReferenceCountUtil.release(cast);
                   }

                   if (out.isEmpty()) {
                       out.recycle();
                       out = null;

                       throw new EncoderException(
                              StringUtil.simpleClassName(this) + " must produce at least one message.");
                   }
               } else {
                   ctx.write(msg, promise);
               }
           }
           // 省略异常处理代码
       }
```

11.5.3 ObjectEncoder 序列化编码器

ObjectEncoder 是 Java 序列化编码器，它负责将实现 Serializable 接口的对象序列化为 byte []，然后写入到 ByteBuf 中用于消息的跨网络传输。下面我们一起分析下它的实现，首先，我们发现它继承自 MessageToByteEncoder，它的作用就是将对象编码成 ByteBuf：

```
public class ObjectEncoder extends MessageToByteEncoder<Serializable>
```

如果要使用 Java 序列化，对象必须实现 Serializable 接口，因此，它的泛型类型为 Serializable。

MessageToByteEncoder 的子类只需要实现 encode(ChannelHandlerContext ctx, I msg, ByteBuf out)方法即可，下面我们重点关注 encode 方法的实现：

```
protected void encode(ChannelHandlerContext ctx, Serializable msg, ByteBuf out) throws Exception {
     int startIdx = out.writerIndex();

     ByteBufOutputStream bout = new ByteBufOutputStream(out);
     bout.write(LENGTH_PLACEHOLDER);
     ObjectOutputStream oout = new CompactObjectOutputStream(bout);
     oout.writeObject(msg);
     oout.flush();
     oout.close();

     int endIdx = out.writerIndex();

     out.setInt(startIdx, endIdx - startIdx - 4);
  }
```

首先创建 ByteBufOutputStream 和 ObjectOutputStream，用于将 Object 对象序列化到 ByteBuf 中，值得注意的是在 writeObject 之前需要先将长度字段（4 个字节）预留，用于后续长度字段的更新。

依次写入长度占位符（4 字节）、序列化之后的 Object 对象，之后根据 ByteBuf 的 writeIndex 计算序列化之后的码流长度，最后调用 ByteBuf 的 setInt(int index, int value)更新长度占位符为实际的码流长度。

有个细节需要注意，更新码流长度字段使用了 setInt 方法而不是 writeInt，原因就是 setInt 方法只更新内容，并不修改 readerIndex 和 writerIndex。

11.5.4 LengthFieldPrepender 通用编码器

如果协议中的第一个字段为长度字段，Netty 提供了 LengthFieldPrepender 编码器，它可以计算当前待发送消息的二进制字节长度，将该长度添加到 ByteBuf 的缓冲区头中，如图所示：

```
编码前(12 bytes)                编码后(14 bytes)
+----------------+              +--------+----------------+
| “HELLO,WORLD” |    ----->    | 0x000C |  “HELLO,WORLD”  |
+----------------+              +--------+----------------+
```

通过 LengthFieldPrepender 可以将待发送消息的长度写入到 ByteBuf 的前 2 个字节，编码后的消息组成为长度字段+原消息的方式。

通过设置 LengthFieldPrepender 为 true，消息长度将包含长度本身占用的字节数，打开 LengthFieldPrepender 后，上图示例中的编码结果如下图所示：

```
   编码前(12 bytes)                编码后(14 bytes)
   +----------------+              +--------+----------------+
   | “HELLO,WORLD” |    ----->    | 0x000E | “HELLO,WORLD”  |
   +----------------+              +--------+----------------+
```

LengthFieldPrepender 工作原理分析如下：首先对长度字段进行设置，如果需要包含消息长度自身，则在原来长度的基础之上再加上 lengthFieldLength 的长度。

如果调整后的消息长度小于 0，则抛出参数非法异常。对消息长度自身所占的字节数进行判断，以便采用正确的方法将长度字段写入到 ByteBuf 中，共有以下 6 种可能：

1) 长度字段所占字节为 1：如果使用 1 个 Byte 字节代表消息长度，则最大长度需要小于 256 个字节。对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的 ByteBuf 并通过 writeByte 将长度值写入到 ByteBuf 中；

2) 长度字段所占字节为 2：如果使用 2 个 Byte 字节代表消息长度，则最大长度需要小于 65536 个字节，对长度进行校验 ,如果校验失败 ,则抛出参数非法异常 ;若校验通过 ,则创建新的 ByteBuf 并通过 writeShort 将长度值写入到 ByteBuf 中；

3) 长度字段所占字节为 3：如果使用 3 个 Byte 字节代表消息长度，则最大长度需要小于 16777216 个字节，对长度进行校验，如果校验失败，则抛出参数非法异常；若校验通过，则创建新的 ByteBuf 并通过 writeMedium 将长度值写入到 ByteBuf 中；

4) 长度字段所占字节为 4：创建新的 ByteBuf，并通过 writeInt 将长度值写入到 ByteBuf 中；

5) 长度字段所占字节为 8：创建新的 ByteBuf，并通过 writeLong 将长度值写入到 ByteBuf 中；

6) 其它长度值：直接抛出 Error。

相关代码如下：

```
    protected void encode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) throws Exception {
          int length = msg.readableBytes() + lengthAdjustment;
          if (lengthIncludesLengthFieldLength) {
              length += lengthFieldLength;
          }

          if (length < 0) {
              throw new IllegalArgumentException(
```

```
                "Adjusted frame length (" + length + ") is less than zero");
        }

        switch (lengthFieldLength) {
        case 1:
            if (length >= 256) {
                throw new IllegalArgumentException(
                        "length does not fit into a byte: " + length);
            }
            out.add(ctx.alloc().buffer(1).order(byteOrder).writeByte((byte) length));
            break;
        case 2:
            if (length >= 65536) {
                throw new IllegalArgumentException(
                        "length does not fit into a short integer: " + length);
            }
            out.add(ctx.alloc().buffer(2).order(byteOrder).writeShort((short) length));
            break;
        case 3:
            if (length >= 16777216) {
                throw new IllegalArgumentException(
                        "length does not fit into a medium integer: " + length);
            }
            out.add(ctx.alloc().buffer(3).order(byteOrder).writeMedium(length));
            break;
        case 4:
            out.add(ctx.alloc().buffer(4).order(byteOrder).writeInt(length));
            break;
        case 8:
            out.add(ctx.alloc().buffer(8).order(byteOrder).writeLong(length));
            break;
        default:
            throw new Error("should not reach here");
        }
        out.add(msg.retain());
    }
```