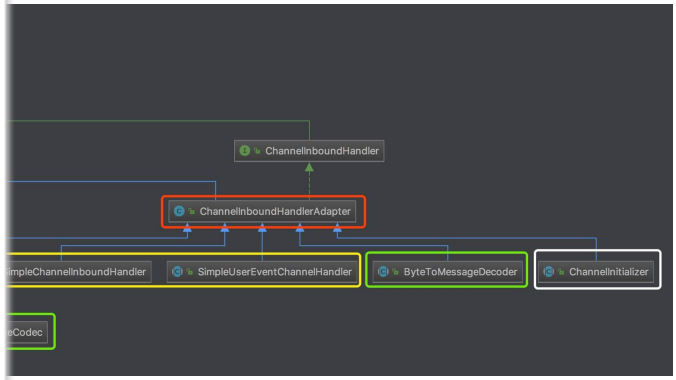
我是一段不羁的公告！

文章目录

ec 之 ByteToMessageDecoder

中，我们看了 ChannelHandler 的核心类图，如下：



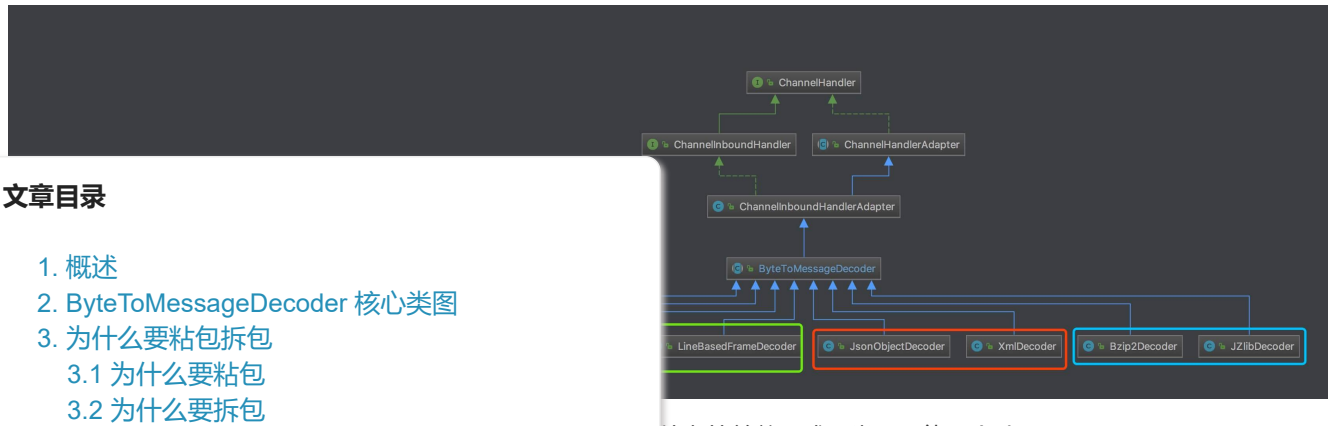
- **绿框**部分，我们可以看到，Netty 基于 ChannelHandler 实现了读写的数据(消息)的编解码。

> Codec(编解码) = Encode(编码) + Decode(解码)。

- 图中有五个和 Codec 相关的类，整理如下：
  - 😈 ，实际应该是六个，漏画了 MessageToMessageDecoder 类。
  - ByteToMessageCodec ， ByteToMessageDecoder + MessageByteEncoder 的**组合**。
    - ByteToMessageDecoder ， 将字节**解码**成消息。
    - MessageByteEncoder ， 将消息**编码**成字节。
  - MessageToMessageCodec ， MessageToMessageDecoder + MessageToMessageEncoder 的**组合**。
    - MessageToMessageDecoder ， 将消息**解码**成另一种消息。
    - MessageToMessageEncoder ， 将消息**编码**成另一种消息。

而本文，我们来分享 ByteToMessageDecoder 部分的内容。

2. ByteToMessageDecoder 核心类图



文章目录

笔者简单整理成三类，可能不全哈：

说该类解码器，用于处理 TCP 的**粘包**现象，将网络发送的
息帧解码为实际的 POJO 对象。 如下图所示：decode
例如：XML、JSON 等等。
配 FrameDecoder 一起使用。
GZip、BZip 等等。
配 FrameDecoder 一起使用。

者的基友【闪电侠】在 《netty 源码分析之拆包器的奥秘》 对

> 首先你得了解一下 TCP/IP 协议，在用户数据量非常小的情况下，极端情况下，一个字节，该 TCP 数据包的有效载荷非常低，传递 100 字节的数据，需要 100 次TCP传送， 100 次ACK，在应用及时性要求不高的情况下，将这 100 个有效数据拼接成一个数据包，那会缩短到一个TCP数据包，以及一个 ack ，有效载荷提高了，带宽也节省了。
>
> 非极端情况，有可能两个数据包拼接成一个数据包，也有可能一个半的数据包拼接成一个数据包，也有可能两个半的数据包拼接成一个数据包。

3.2 为什么要拆包

> 拆包和粘包是相对的，一端粘了包，另外一端就需要将粘过的包拆开。举个栗子，发送端将三个数据包粘成两个TCP数据包发送到接收端，接收端就需要根据应用协议将两个数据包重新组装成三个数据包。
>
> 还有一种情况就是用户数据包超过了 mss(最大报文长度)，那么这个数据包在发送的时候必须拆分成几个数据包，接收端收到之后需要将这些数据包粘合起来之

> 后，再拆开。

文章目录

个完整的数据包：

个完整的业务数据包，那就保留该数
到一个完整的数据包。
数据足够拼接成一个数据包，那就将已
够成一个完整的业务数据包传递到业
和下次读到的数据尝试拼接。

译为“累加器”，用于将读取到的数据进行累加到一起，然后再尝

，从而完整。当然，累加的过程，没准又进入了一个不完整的

Cumulator 接口，代码如下：

```
/**
 * ByteBuf 累积器接口
 *
 * Cumulate {@link ByteBuf}s.
 */
public interface Cumulator {

    /**
     * Cumulate the given {@link ByteBuf}s and return the {@link ByteBuf} that holds the cumulated byt
     * The implementation is responsible to correctly handle the life-cycle of the given {@link ByteBu
     * call {@link ByteBuf#release()} if a {@link ByteBuf} is fully consumed.
     *
     * @param alloc ByteBuf 分配器
     * @param cumulation ByteBuf 当前累积结果
     * @param in 当前读取( 输入 ) ByteBuf
     * @return ByteBuf 新的累积结果
     */
    ByteBuf cumulate(ByteBufAllocator alloc, ByteBuf cumulation, ByteBuf in);

}
```

- 对于 `Cumulator#cumulate(ByteBufAllocator alloc, ByteBuf cumulation, ByteBuf in)` 方法，将**原有** `cumulation` 累加上**新的** `in` ，返回“新”的 ByteBuf 对象。
- 如果 `in` 过大，超过 `cumulation` 的空间上限，使用 `alloc` 进行扩容后再累加。

文章目录

ew Cumulator() {

= new Cumulator() {

果空间不够，扩容出**新的** ByteBuf ，再继续进行累积。代码如

o one {@link ByteBuf}'s, using memory copies.

R = new Cumulator() {

```
3:     @Override
4:     public ByteBuf cumulate(ByteBufAllocator alloc, ByteBuf cumulation, ByteBuf in) {
5:         final ByteBuf buffer;
6:         if (cumulation.writerIndex() > cumulation.maxCapacity() - in.readableBytes() // 超过空间大
7:                 || cumulation.refCnt() > 1 // 引用大于 1 ，说明用户使用了 slice().retain() 或 duplic
8:                                            // 此时扩容返回一个新的累积区ByteBuf，方便用户对老的累积区B
9:                 || cumulation.isReadOnly()) { // 只读，不可累加，所以需要改成可写
10:            // Expand cumulation (by replace it) when either there is not more room in the buffer
11:            // or if the refCnt is greater then 1 which may happen when the user use slice().reta
12:            // duplicate().retain() or if its read-only.
13:            //
14:            // See:
15:            // - https://github.com/netty/netty/issues/2327
16:            // - https://github.com/netty/netty/issues/1764
17:            // 扩容，返回新的 buffer
18:            buffer = expandCumulation(alloc, cumulation, in.readableBytes());
19:        } else {
20:            // 使用老的 buffer
21:            buffer = cumulation;
22:        }
23:        // 写入 in 到 buffer 中
24:        buffer.writeBytes(in);
25:        // 释放输入 in
26:        in.release();
27:        // 返回 buffer
```

```
28:         return buffer;
29:     }
30:
31: };
```

文章目录

ation.maxCapacity() - in.readableBytes() ，

于 1，说明用户使用了
cate()#retain() 方法，使 refCnt 增加并且大于

到。

expandCumulation(ByteBufAllocator alloc,
容，并返回新的，并赋值给 buffer 。代码如下：

```
tor alloc, ByteBuf cumulation, int readable) {

eadableBytes() + readable);

    // 释放老的 ByteBuf 对象
    oldCumulation.release();
    // 返回新的 ByteBuf 对象
    return cumulation;
}
```

- 标准的扩容，并复制老数据的过程。胖友自己看下注释噢。
- 【无需扩容】第 21 行： buffer 直接使用的 cumulation 对象。

- 第 24 行：写入 in 到 buffer 中，进行累积。
- 第 26 行：释放 in 。
- 第 28 行：返回 buffer 。

4.2 COMPOSITE_CUMULATOR

COMPOSITE_CUMULATOR 思路是，使用 CompositeByteBuf ，组合新输入的 ByteBuf 对象，从而避免内存拷贝。代码如下：

```
// ByteToMessageDecoder.java

    /**
     * Cumulate {@link ByteBuf}s by add them to a {@link CompositeByteBuf} and so do no memory copy wh
     * Be aware that {@link CompositeByteBuf} use a more complex indexing implementation so depending
     * and the decoder implementation this may be slower then just use the {@link #MERGE_CUMULATOR}.
     *
     * 相比 MERGE_CUMULATOR 来说：
     *
```

```
     * 好处是，内存零拷贝
     * 坏处是，因为维护复杂索引，所以某些使用场景下，慢于 MERGE_CUMULATOR
     */
  1: public static final Cumulator COMPOSITE_CUMULATOR = new Cumulator() {
```

文章目录


```
                alloc, ByteBuf cumulation, ByteBuf in) {

                it) when the refCnt is greater then 1 which may happ
                cate().retain().

                ty/issues/2327
                ty/issues/1764
                cumulation, in.readableBytes());

                直接使用
                iteByteBuf) {
                ) cumulation;
                创建，并添加到其中

                uffer(Integer.MAX_VALUE);
                cumulation);
 27:             // 添加 in 到 composite 中
 28:             composite.addComponent(true, in);
 29:             // 赋值给 buffer
 30:             buffer = composite;
 31:         }
 32:         // 返回 buffer
 33:         return buffer;
 34:     }
 35:
 36: };
```

- 第 7 至 16 行： `cumulation.refCnt() > 1` 成立，和 `MERGE_CUMULATOR` 的情况一致，创建一个新的 ByteBuf 对象。这样，再下一次 `#cumulate(...)` 时，就会走【第 22 至 26 行】的情况。
- 获得 `composite` 对象
  - 第 19 至 21 行：如果原来**就是** CompositeByteBuf 类型，直接使用。
  - 第 22 至 26 行：如果原来**不是** CompositeByteBuf 类型，创建 CompositeByteBuf 对象，并添加 `cumulation` 到其中。
- 第 28 行：添加 `in` 到 `composite` 中，避免内存拷贝。

4.3 对比

关于 `MERGE_CUMULATOR` 和 `COMPOSITE_CUMULATOR` 的对比，已经写在 `COMPOSITE_CUMULATOR` 的**头上**的注释。

默认情况下，ByteToMessageDecoder 使用 `MERGE_CUMULATOR` 作为累加器。

5. ByteToMessageDecoder

`io.netty.handler.codec.ByteToMessageDecoder` ，继承 ChannelInboundHandlerAdapter 类，**抽象基类**，负责将 Byte

文章目录

节比较多，建议胖友理解如下小节即可：

```
DE = 1;
ENDING = 2;
```

```
ByteBuf cumulation;
/**
 * 累计器
 */
private Cumulator cumulator = MERGE_CUMULATOR;
/**
 * 是否每次只解码一条消息，默认为 false 。
 *
 * 部分解码器为 true ，例如：Socks4ClientDecoder
 *
 * @see #callDecode(ChannelHandlerContext, ByteBuf, List)
 */
private boolean singleDecode;
/**
 * 是否解码到消息。
 *
 * WasNull ，说明就是没解码到消息
 *
 * @see #channelReadComplete(ChannelHandlerContext)
 */
private boolean decodeWasNull;
/**
 * 是否首次读取，即 {@link #cumulation} 为空
 */
private boolean first;
/**
```

```
 * A bitmask where the bits are defined as
 * <ul>
 *     <li>{@link #STATE_INIT}</li>
 *     <li>{@link #STATE_CALLING_CHILD_DECODE}</li>
                                                  </li>
```

文章目录


```
                                                  ByteBuf, List)} 方法中，正在进行解码
                                                  无法全部解码完，则进行释放，避免 OOM
```

属性比较简单，胖友自己看注释。

5.2 channelRead

`#channelRead(ChannelHandlerContext ctx, Object msg)` 方法，读取到新的数据，进行解码。代码如下：

```
 1: @Override
 2: public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
 3:     if (msg instanceof ByteBuf) {
 4:         // 创建 CodecOutputList 对象
 5:         CodecOutputList out = CodecOutputList.newInstance();
 6:         try {
 7:             ByteBuf data = (ByteBuf) msg;
 8:             // 判断是否首次
 9:             first = cumulation == null;
10:             // 若首次，直接使用读取的 data
11:             if (first) {
12:                 cumulation = data;
13:             // 若非首次，将读取的 data ，累积到 cumulation 中
14:             } else {
15:                 cumulation = cumulator.cumulate(ctx.alloc(), cumulation, data);
16:             }
17:             // 执行解码
```

```
18:             callDecode(ctx, cumulation, out);
19:         } catch (DecoderException e) {
20:             throw e; // 抛出异常
21:         } catch (Exception e) {
                封装成 DecoderException 异常，抛出

                接释放全部
                tion.isReadable()) {
                 次数
                cumulation
                ulation
                上限，释放部分的已读
                AfterReads) {
                try to discard some bytes so we not risk to see a O
                cy/netty/issues/4275
                 次数
                放部分的已读

                ecycled();

                条消息
47:         }
48:     } else {
49:         // 触发 Channel Read 事件到下一个节点
50:         ctx.fireChannelRead(msg);
51:     }
52: }
```


- 第 48 至 51 行：消息的类型**不是** ByteBuf 类，直接触发 Channel Read 事件到下一个节点。也就说，不进行解码。
- 第 3 行：消息的类型**是** ByteBuf 类，开始解码。
- 第 5 行：创建 CodecOutputList 对象。CodecOutputList 的简化代码如下：

```
/**
 * Special {@link AbstractList} implementation which is used within our codec base classes.
 */
final class CodecOutputList extends AbstractList<Object> implements RandomAccess {

    // ... 省略代码
}
```

- 如下内容，引用自 《自顶向下深入分析Netty（八）–CodecHandler》

> 解码结果列表 CodecOutputList 是 Netty 定制的一个特殊列表，该列表在线程中被缓存，可循环使用来存储解码结果，减少不必要的列表实例
> 果需要频繁存储，普通的 ArrayList
> 特殊列表，由此可见 Netty 对优化的

文章目录

，是否为首次 `first` 。

`= (ByteBuf) msg`)。

中。在 「4. Cumulator」 中，我们已经详细解析。

`ctx, ByteBuf in, List<Object> out)` 方法，执行解

，「5.3 callDecode」 。

。

ation 。

接**释放全部。**

fterReads 上限，重置计数，并调用

😈 如果一直不去释放，等到满足【第 24 至 28 行】的条

```
                <1> 如果用户使用了 slice().retain() 和 duplicate(
                make more room in the
                 as otherwise the user may have
                // used slice().retain() or duplicate().retain().
                //
                // See:
                // - https://github.com/netty/netty/issues/2327
                // - https://github.com/netty/netty/issues/1764
                // <2> 释放部分
                cumulation.discardSomeReadBytes();
            }
        }
```

- `<1>` 处，原因见中文注释。
- `<2>` 处，释放**部分**已读字节区。注意，是“部分”，而不是“全部”，避免一次性释放全部，时间过长。并且，能够读取到这么“大”，往往字节数容量不小。如果直接释放掉“全部”，那么后续还需要再重复扩容，反倒不好。

- 第 38 行：获得解码消息的数量。
  - 第 40 行：是否解码到消息。为什么不直接使用 `size` 来判断呢？因为如果添加了消息，然后又移除该消息，此时 `size` 为 0 ，但是 `!out.insertSinceRecycled()` 为 `true` 。
    - 另外，我们在 「5.3 callDecode」 中，将会看到一个 `out` 的清理操作，到时会更加明白。
- 第 43 行：调用 `#fireChannelRead(ChannelHandlerContext ctx, List<Object> msgs, int numElements)` **静态**方法，触发 Channel Read 事件。可能是多条消息。代码如下：

```
/**
 * Get {@code numElements} out of the {@link List} and forward these through the pipeline.
 */
```

```
static void fireChannelRead(ChannelHandlerContext ctx, List<Object> msgs, int numElements) {
    if (msgs instanceof CodecOutputList) { // 如果是 CodecOutputList 类型，特殊优化
        fireChannelRead(ctx, (CodecOutputList) msgs, numElements);
    } else {
```

文章目录

decOutputList} and forward these through the pipeli

xt ctx, CodecOutputList msgs, int numElements) {

// getUnsafe 是自定义的方法，减少越界判断，效率更高

件。

, List<Object> out) 方法，执行解码。而解码的结果，会

```
protected void callDecode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {
    try {
        // 循环读取，直到不可读
        while (in.isReadable()) {
            // 记录
            int outSize = out.size();
            // out 非空，说明上一次解码有解码到消息
            if (outSize > 0) {
                // 触发 Channel Read 事件。可能是多条消息
                fireChannelRead(ctx, out, outSize);
                // 清空
                out.clear();

                // 用户主动删除该 Handler ，继续操作 in 是不安全的
                // Check if this handler was removed before continuing with decoding.
                // If it was removed, it is not safe to continue to operate on the buffer.
                //
                // See:
                // - https://github.com/netty/netty/issues/4635
                if (ctx.isRemoved()) {
                    break;
                }
                outSize = 0;
            }

            // 记录当前可读字节数
```

```
27:             int oldInputLength = in.readableBytes();
28:
29:             // 执行解码。如果 Handler 准备移除，在解码完成后，进行移除。
30:             decodeRemovalReentryProtection(ctx, in, out);
```



```
56:
57:             // 如果开启 singleDecode ，表示只解析一次，结束循环
58:             if (isSingleDecode()) {
59:                 break;
60:             }
61:         }
62:     } catch (DecoderException e) {
63:         throw e;
64:     } catch (Exception cause) {
65:         throw new DecoderException(cause);
66:     }
67: }
```

- 第 4 行：循环读取 `in` ，直到不可读。
- 第 5 行：记录 `out` 的大小。
  - 第 8 行：如果 `out` 非空，说明上一次解码有解码到消息。
  - 第 10 行：调用 `#fireChannelRead(ChannelHandlerContext ctx, List<Object> msgs, int numElements)` **静态**方法，触发 Channel Read 事件。可能是多条消息。😈 关于该方法，上文已经详细解析。
  - 第 12 行：清空 `out` 。所以，有可能会出现 `#channelRead(ChannelHandlerContext ctx, Object msg)` 方法的【第 40 行】的情况。
  - 第 14 至 22 行：用户主动删除该 Handler ，继续操作 `in` 是不安全的，所以结束循环。
  - 第 23 行：记录 `out` 的大小为**零**。所以，实际上， `outSize` 没有必要记录。因为，一定是为**零**。
- 第 27 行：记录当前可读字节数。

- 第 30 行：调用 `#decodeRemovalReentryProtection(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)` 方法，执行解码。如果 Handler 准备移除，在解码完成后，进行移除。详细解析，见「5.3.1 decodeRemovalReentryProtection」 中。
- 第 32 至 39 行：用户主动删除该 Handler ，继续操作 `in` 是不安全的，所以结束循环。

【第 8 至 24 行】的代码，能够保证 `outSize` 等于零。

环。

新开始循环，即继续读取。

出 DecoderException 异常。说明，有问题。

次， `break` 结束循环。

常。

text ctx, ByteBuf in, List<Object> out) 方法，执行

码如下：

```
nelHandlerContext ctx, ByteBuf in, List<Object> out)

= STATE_HANDLER_REMOVED_PENDING;
```

```
12:         // 移除当前 Handler
13:         if (removePending) {
14:             handlerRemoved(ctx);
15:         }
16:     }
17: }
```

- 第 3 行：设置状态(`decodeState`)为 `STATE_CALLING_CHILD_DECODE` 。
- 第 6 行：调用 `#decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)` 方法，执行解码。代码如下：

```
/**
 * Decode the from one {@link ByteBuf} to an other. This method will be called till either the inp
 * {@link ByteBuf} has nothing to read when return from this method or till nothing was read from
 * {@link ByteBuf}.
 *
 * @param ctx           the {@link ChannelHandlerContext} which this {@link ByteToMessageDecoder}
 * @param in            the {@link ByteBuf} from which to read data
 * @param out           the {@link List} to which decoded messages should be added
 * @throws Exception    is thrown if an error occurs
 */
protected abstract void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Exc
```

- 子类实现该方法，就可以愉快的解码消息了，**并且，也只需要实现该方法。**其它的逻辑，ByteToMessageDecoder 已经全部帮忙实现了。

- 第 9 行：判断是否准备移除。那么什么情况下，会出现 `decodeState == STATE_HANDLER_REMOVED_PENDING` 成立呢？详细解析，见 「5.7 handlerRemoved」 。

R_REMOVED_PENDING 。

ved(ChannelHandlerContext ctx) 方法，移除当前

方法，代码如下：

```
Context ctx) throws Exception {

起读取，期望读取到更多数据，以便解码到消息

asNull
()) {

点
```

- 第 4 行：重置 `numReads` 。
- 第 6 行：调用 `#discardSomeReadBytes()` 方法，释放部分的已读。
- 第 7 至 13 行：未解码到消息(`decodeWasNull == true`)，并且未开启自动读取(`ctx.channel().config().isAutoRead() == false`)，则再次发起读取，期望读取到更多数据，以便解码到消息。
- 第 15 行：触发 Channel ReadComplete 事件到下一个节点。

5.5 channelInactive

`#channelInactive(ChannelHandlerContext ctx)` 方法，通道处于未激活(Inactive)，解码完剩余的消息，并释放相关资源。代码如下：

```
@Override
public void channelInactive(ChannelHandlerContext ctx) throws Exception {
    channelInputClosed(ctx, true);
}
```

- 调用 `#channelInputClosed(ChannelHandlerContext ctx, boolean callChannelInactive)` 方法，执行 Channel 读取关闭的逻辑。代码如下：

```
1: private void channelInputClosed(ChannelHandlerContext ctx, boolean callChannelInactive) throws
2:     // 创建 CodecOutputList 对象
3:     CodecOutputList out = CodecOutputList.newInstance();
```

```
4:     try {
5:         // 当 Channel 读取关闭时，执行解码剩余消息的逻辑
6:         channelInputClosed(ctx, out);
7:     } catch (DecoderException e) {
```

文章目录


```
节点。可能是多条消息
;
el ReadComplete 事件到下一个节点。

 fireChannelReadComplete()
();

active(...)` ，则触发 Channel Inactive 事件到下一个节
```

```
33:             out.recycle();
34:         }
35:     }
36: }
```

- 第 3 行：创建 CodecOutputList 对象。
- 第 6 行：调用 `#channelInputClosed(ChannelHandlerContext ctx, List<Object> out)` 方法，当 Channel 读取关闭时，执行解码剩余消息的逻辑。代码如下：

```
/**
 * Called when the input of the channel was closed which may be because it changed to ina
 * {@link ChannelInputShutdownEvent}.
 */
void channelInputClosed(ChannelHandlerContext ctx, List<Object> out) throws Exception {
    if (cumulation != null) {
        // 执行解码
        callDecode(ctx, cumulation, out);
        // 最后一次，执行解码
        decodeLast(ctx, cumulation, out);
    } else {
        // 最后一次，执行解码
        decodeLast(ctx, Unpooled.EMPTY_BUFFER, out);
    }
}
```

```
/**
 * Is called one last time when the {@link ChannelHandlerContext} goes in-active. Which m
 * {@link #channelInactive(ChannelHandlerContext)} was triggered.
```

文章目录


```
ecode(ChannelHandlerContext, ByteBuf, List)} b
peration.

text ctx, ByteBuf in, List<Object> out) throws

omething left in the buffer to decode.
ty/issues/4386
 in, out);
```

ctx, ByteBuf in, List<Object> out) 方法，
了该方法。
tion 异常。

ontext ctx, List<Object> msgs, int
可能是多条消息。
发 Channel ReadComplete 事件到下一个节点。
ve(...) ，则触发 Channel Inactive 事件到下一个节

的 cumulation ，在 Channel 变成未激活时。细节好多呀！！！

5.6 userEventTriggered

#userEventTriggered(ChannelHandlerContext ctx, Object evt) 方法，处理 ChannelInputShutdownEvent 事件，即 Channel 关闭读取。代码如下：

```
@Override
public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
    if (evt instanceof ChannelInputShutdownEvent) {
        // The decodeLast method is invoked when a channelInactive event is encountered.
        // This method is responsible for ending requests in some situations and must be called
        // when the input has been shutdown.
        channelInputClosed(ctx, false);
    }
    // 继续传播 evt 到下一个节点
    super.userEventTriggered(ctx, evt);
}
```

- 调用 #channelInputClosed(ChannelHandlerContext ctx, boolean callChannelInactive) 方法，执行 Channel 读取关闭的逻辑。
- 继续传播 evt 到下一个节点。

😈 对于该方法的目的，笔者的理解是，尽可能在解码一次剩余的 cumulation ，在 Channel 关闭读取。细节好多呀！！！

5.7 handlerRemoved

`#handlerRemoved(ChannelHandlerContext ctx)` 方法，代码如下：

文章目录


```
erContext ctx) throws Exception {
 标记状态为 STATE_HANDLER_REMOVED_PENDING
ODE) {
ENDING;
ovalReentryProtection(...)` 方法，一起看。

 sure we not access it in any other method here anym

dable);
```

```
26:         }
27:
28:         // 置空 numReads
29:         numReads = 0;
30:         // 触发 Channel ReadComplete 到下一个节点
31:         ctx.fireChannelReadComplete();
32:     }
33:     // 执行移除逻辑
34:     handlerRemoved0(ctx);
35: }
```

- 第 3 至 7 行：如果状态(`decodeState`)处于 `STATE_CALLING_CHILD_DECODE` 时，说明正在执行 `#decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)` 方法中。如果此时，直接往下执行，`cumulation` 将被直接释放，而 `#decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)` 方法可能正在解码中，很大可能性造成影响，导致错误。所以，此处仅仅标记状态(`decodeState`)为 `STATE_HANDLER_REMOVED_PENDING` 。等到 `#decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)` 方法执行完成后，在进行移除。胖友，此时可以再跳回 「5.3.1 decodeRemovalReentryProtection」 ，进行再次理解。
- 【有可读字节】第 15 至 21 行：读取剩余字节，并释放 `buf` 。然后，触发 Channel Read 到下一个节点。通过这样的方式，避免 `cumulation` 中，有字节被“丢失”，即使当前可能无法解码成一个数据包。
- 【无可读字节】第 22 至 26 行：直接释放 `buf` 。
- 第 29 行：置空 `numReads` 。
- 第 34 行：调用 `#handlerRemoved0(ChannelHandlerContext ctx)` 方法，执行移除逻辑。代码如下：

```
/**
 * Gets called after the {@link ByteToMessageDecoder} was removed from the actual context and it d
 * events anymore.
```

```
ntext ctx) throws Exception { }
```

自定义的资源释放。目前重写该方法的类，例如：

```
 decoder. You usually
tly to write a decoder.
isk.
```

文章目录

`#actualReadableBytes()` 方法，获得可读字节数。代码如下：

```
/**
 * Returns the actual number of readable bytes in the internal cumulative
 * buffer of this decoder. You usually do not need to rely on this value
 * to write a decoder. Use it only when you must use it at your own risk.
 * This method is a shortcut to {@link #internalBuffer() internalBuffer().readableBytes()}.
 */
protected int actualReadableBytes() {
    return internalBuffer().readableBytes();
}
```

666. 彩蛋

细节有点多，可能对如下小节理解不够到位。如有错误，烦请胖友教育。

- 「5.5 channelInactive」
- 「5.6 userEventTriggered」
- 「5.7 handlerRemoved」

本文参考如下文章：

- 简书闪电侠 《netty源码分析之拆包器的奥秘》
- Hypercube 《自顶向下深入分析Netty（八）–CodecHandler》

文章目录