- NETTY

精尽 Netty 源码解析 —— Codec 之 ByteToMessageDecoder（二）FrameDecoder

1. 概述

在《精尽 Netty 源码解析 —— Codec 之 ByteToMessageDecoder（一）》中，我们看到 ByteToMessageDecoder 有四个 FrameDecoder 实现类：

- ① FixedLengthFrameDecoder，基于**固定长度**消息进行粘包拆包处理的。
- ② LengthFieldBasedFrameDecoder，基于**消息头指定消息长度**进行粘包拆包处理的。
- ③ LineBasedFrameDecoder，基于**换行**来进行消息粘包拆包处理的。
- ④ DelimiterBasedFrameDecoder，基于**指定消息边界方式**进行粘包拆包处理的。

实际上，上述四个 FrameDecoder 实现可以进行规整：

- ① 是 ② 的特例，**固定长度**是**消息头指定消息长度**的一种形式。
- **式**的一种形式。

会提供相关的文章。

文章目录

ecoder

ameDecoder ，继承 ByteToMessageDecoder 抽象类，基于**固定长度**消息进行

```
---+-----+
EF | GHI |
---+-----+
```

2.1 构造方法

```
/**
 * 固定长度
 */
private final int frameLength;

/**
 * Creates a new instance.
```

```
     *
     * @param frameLength the length of the frame
     */
    public FixedLengthFrameDecoder(int frameLength) {
        if (frameLength <= 0) {
            throw new IllegalArgumentException("frameLength must be a positive integer: " + frameLength);
        }
        this.frameLength = frameLength;
    }
```

- `frameLength` 属性，固定长度。

2.2 decode

`#decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)` 方法，执行解码。代码如下：

```
1: @Override
2: protected final void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) throws Excepti
3:     // 解码消息
4:     Object decoded = decode(ctx, in);
5:     // 添加到 out 结果中
6:     if (decoded != null) {
7:         out.add(decoded);
8:     }
9: }
```


`Context ctx, ByteBuf in)` 方法，解码消息。代码如下：

```
k ByteBuf} and return it.

link ChannelHandlerContext} which this {@link ByteToMessageDecoc
link ByteBuf} from which to read data
link ByteBuf} which represent the frame or {@code null} if no fr
ated.

arnings("UnusedParameters") ChannelHandlerContext ctx, ByteBuf i
，无法解码出消息。
Length) {

，解码出一条消息。
    } else {
        return in.readRetainedSlice(frameLength);
    }
}
```

- 当可读字节足够 `frameLength` 长度时，调用 `ByteBuf#readRetainedSlice(int length)` 方法，读取一个 Slice ByteBuf 对象，并增加引用计数。并且该 Slice ByteBuf 作为解码的一条消息。另外，`ByteBuf#readRetainedSlice(int length)` 的过程，因为是共享原有 ByteBuf `in` 数组，所以不存在数据拷贝。
- 第 5 至 8 行：若解码到消息，添加到 `out` 结果中。

3. LineBasedFrameDecoder

`io.netty.handler.codec.LineBasedFrameDecoder` ，继承 ByteToMessageDecoder 抽象类，基于**换行**来进行消息粘包拆包处理的。

它会处理 `"\n"` 和 `"\r\n"` 两种换行符。

3.1 构造方法

```
/**
 * 一条消息的最大长度
 *
 * Maximum length of a frame we're willing to decode.
 */
private final int maxLength;
/**
 * 是否快速失败
 *
 * 当 true 时，未找到消息，但是超过最大长度，则马上触发 Exception 到下一个节点
 * 当 false 时，未找到消息，但是超过最大长度，需要匹配到一条消息后，再触发 Exception 到下一个节点
 *
 * Whether or not to throw an exception as soon as we exceed maxLength.
 */
private final boolean failFast;
/**
 * 是否过滤掉换行分隔符。
 *
 * 如果为 true ，解码的消息不包含换行符。
```



```
maxLength )，结果还是找不到换行符

ause we're already over maxLength.
```

```
/**
 * 最后扫描的位置
 *
 * Last scan position.
 */
private int offset;

/**
 * Creates a new decoder.
```

```
     * @param maxLength  the maximum length of the decoded frame.
     *                   A {@link TooLongFrameException} is thrown if
     *                   the length of the frame exceeds this value.
     */
    public LineBasedFrameDecoder(final int maxLength) {
        this(maxLength, true, false);
    }

    /**
     * Creates a new decoder.
     * @param maxLength  the maximum length of the decoded frame.
     *                   A {@link TooLongFrameException} is thrown if
     *                   the length of the frame exceeds this value.
     * @param stripDelimiter  whether the decoded frame should strip out the
     *                        delimiter or not
     * @param failFast  If <tt>true</tt>, a {@link TooLongFrameException} is
     *                  thrown as soon as the decoder notices the length of the
     *                  frame will exceed <tt>maxFrameLength</tt> regardless of
     *                  whether the entire frame has been read.
     *                  If <tt>false</tt>, a {@link TooLongFrameException} is
     *                  thrown after the entire frame that exceeds
     *                  <tt>maxFrameLength</tt> has been read.
     */
    public LineBasedFrameDecoder(final int maxLength, final boolean stripDelimiter, final boolean failFast
        this.maxLength = maxLength;
        this.failFast = failFast;
        this.stripDelimiter = stripDelimiter;
    }
```

文章目录

原本以为 LineBasedFrameDecoder 会比较简单，但是因为多了

为 "abcd\nEF\n" (直接以字符串举例，为了可阅读性)，那么
度为 4 ，超过最大长度 maxLength 。
收到的是 "abc" ，那么无法匹配到 \n 换行符。但是呢， "abc" 的
需要等待读取到 "d\n" 部分，然后抛弃 "abcd" 整条。再之后，继

长度，则马上触发 Exception 到下一个节点。
最大长度，需要匹配到一条消息后，再触发 Exception 到下一个节点。

分隔符。如果为 true ，解码的消息不包含换行符。
discarding 属性，是否处于废弃模式。如果为 true ，说明解析超过最大长度(maxLength)，结果还是找不到换行符。

- 😈 也有点绕，等下结合代码具体理解。
- discardedBytes 属性，废弃的字节数。
- offset 属性，最后扫描的位置。

3.2 decode

#decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) 方法，执行解码。代码如下：

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

- 这段代码，和 `FixedLengthFrameDecoder#decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)` 方法，是一样的。

`#decode(ChannelHandlerContext ctx, ByteBuf buffer)` 方法，执行解码。代码如下：

```
protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws Exception {
    // 获得换行符的位置
    final int eol = findEndOfLine(buffer);
    if (!discarding) { // 未处于废弃模式
        if (eol >= 0) { // 找到
            final ByteBuf frame;
            final int length = eol - buffer.readerIndex(); // 读取长度
            final int delimLength = buffer.getByte(eol) == '\r' ? 2 : 1; // 分隔符的长度。2 为 `\r\r

            // 超过最大长度
            if (length > maxLength) {
                // 设置新的读取位置
                buffer.readerIndex(eol + delimLength);
                                       到下一个节点
                                      ;
                                      未解码到消息

                                      adRetainedSlice(length);
                                      (delimLength); // 忽略换行符

                                      adRetainedSlice(length + delimLength);

            return frame;
        } else { // 未找到
            final int length = buffer.readableBytes();
            // 超过最大长度
            if (length > maxLength) {
                // 记录 discardedBytes
                discardedBytes = length;
                // 跳到写入位置
                buffer.readerIndex(buffer.writerIndex());
                // 标记 discarding 为废弃模式
```

文章目录

```
39:                 discarding = true;
40:                 // 重置 offset
41:                 offset = 0;
42:                 // 如果快速失败，则触发 Exception 到下一个节点
43:                 if (failFast) {
44:                     fail(ctx, "over " + discardedBytes);
45:                 }
46:             }
47:             return null;
48:         }
49:     } else { // 处于废弃模式
50:         if (eol >= 0) { // 找到
51:             final int length = discardedBytes + eol - buffer.readerIndex(); // 读取长度
52:             final int delimLength = buffer.getByte(eol) == '\r' ? 2 : 1; // 分隔符的长度。2 为 `\r\r
53:             // 设置新的读取位置
54:             buffer.readerIndex(eol + delimLength);
55:             // 重置 discardedBytes
56:             discardedBytes = 0;
57:             // 设置 discarding 不为废弃模式
58:             discarding = false;
59:             // 如果不为快速失败，则触发 Exception 到下一个节点
60:             if (!failFast) {
61:                 fail(ctx, length);
62:             }
63:         } else { // 未找到
64:             // 增加 discardedBytes
65:             discardedBytes += buffer.readableBytes();
66:             // 跳到写入位置
                uffer.writerIndex());
```

文章目录

yteBuf buffer) 方法，获得换行符的位置。详细解析，这里胖友先跳到

模式 ==========

的长度。

则**丢弃**该消息。

- 第 15 行：调用 `#fail(...)` 方法，触发 Exception 到下一个节点。详细解析，见「3.4 fail」。😈 注意，此处和 `failFast` 没有关系。
- 【失败】第 17 行：返回 `null` ，即未解码到消息。

- 【成功】第 20 至 26 行：解码出一条消息。调用 `ByteBuf#readRetainedSlice(int length)` 方法，读取一个 Slice ByteBuf 对象，并增加引用计数。并且该 Slice ByteBuf 作为解码的一条消息。另外，`ByteBuf#readRetainedSlice(int length)` 的过程，因为是共享原有 ByteBuf `in` 数组，所以不存在数据拷贝。
- ② 第 30 行：**未找到**换行符，说明当前 `buffer` **不存在**完整的消息。需要继续读取新的数据，再次解码拆包。
- 第 33 行：可读字节，超过最大长度，那么即使后续找到换行符，消息也**一定**超过最大长度。
- 第 35 行：记录 `discardedBytes` 。因为【第 37 行】的代码， `buffer` 跳到写入位置，也就是抛弃了 `discardedBytes` 字节数。

- 第 39 行：标记 `discarding` 为 `true` ，进入废弃模式。那么，后续就会执行【第 49 至 70 行】的代码逻辑，寻找到换行符，解码拆包出该消息，并**抛弃**它。
  - 😈 这段，好好理解下。
- 第 41 行：重置 `offset` 为 0 。
- 第 42 至 45 行：如果快速失败(`failFast = true`)，调用 `#fail(...)` 方法，触发 Exception 到下一个节点。那么，不快速失败(`failFast = false`)呢？继续往下走，答案在【第 59 至 61 行】的代码，见分晓。
- 第 47 行：【失败】第 17 行：返回 `null` ，即未解码到消息。
- `==========` 正处于 `discarding` 模式 `==========`
- `discarding` 模式是什么呢？在【第 33 至 46 行】的代码，如果已读取的字节数，超过最大长度，那么进入 `discarding` 模式，继续寻找到换行符，解码拆包出该消息，并**抛弃**它。😈 实际上，它的效果是【第 30 至 48 行】+【第 49 至 69 行】和【第 10 至 18 行】的代码的效果是**等价的**，只是说【第 30 至 48 行】的代码，因为数据包是**不完整**(找不到换行符)的，所以进入【第 49 至 69 行】的代码。
- 根据是否找到换行符，分成 ① ② 两种情况。
- ① 第 50 行：**找到**换行符。
- 第 51 行：读取长度。此处的长度，算上了 `discardedBytes` 的部分。
  - 第 52 行：获得换行符的长度。
- 第 54 行：设置新的读取位置。因为，**找到**换行符。
- 第 56 行：重置 `discardedBytes` 为 0 。因为，**找到**换行符。
- 第 58 行：重置 `offset` 为 0 。因为，**找到**换行符。
- 第 59 至 62 行：如果不为快速失败(`failFast = false`)，调用 `#fail(...)` 方法，触发 Exception 到下一个节点。
  - 和【第 42 至 45 行】的代码，相对。
  - 也就说， `failFast = false` 的情况下，只有在解析到完整的消息，**才**触发 Exception 到下一个节点。😈 是不是很绕，哈哈哈哈。
- 【失败】第 69 行：返回 `null` ，虽然解码到消息，但是因为消息长度超过最大长度，所以进行**丢失**。和【第 17 行】的代码，是一个目的。
- ② 第 63 行：**未找到**换行符，说明当前 `buffer` **不存在**完整的消息。需要继续读取新的数据，再次解码拆包。
- 第 65 行：增加 `discardedBytes` 。
- 第 67 行：`buffer.readerIndex(buffer.writerIndex())` 跳到写入位置。

方法，获得换行符的位置。代码如下：

```
 of the end of line found.
s found in the buffer.

yteBuf buffer) {
ableBytes();
uffer.readerIndex() + offset, totalLength - offset, ByteProcessor.
    // 找到
    if (i >= 0) {
        // 重置 offset
        offset = 0;
        // 如果前一个字节位 `\r` ，说明找到的是 `\n` ，所以需要 -1 ，因为寻找的是首个换行符的位置
        if (i > 0 && buffer.getByte(i - 1) == '\r') {
            i--;
        }
    // 未找到，记录 offset
    } else {
```

```
14:         offset = totalLength;
15:     }
16:     return i;
17: }
```

- 关于 `offset` 的逻辑，笔者觉得有点问题。在这里，胖友先无视掉它。稍后，我们在统一分享。
- 第 3 行：在 `buffer` 的 `[readerIndex, readerIndex + readableBytes)` 位置范围内，查找 `\n` 换行符的位置。😈 在忽略 `offset` 的前提下。
- 【有找到】
  - 第 7 行：重置 `offset` 。
  - 第 8 至 11 行：如果前一个字节位 `\r` ，说明找到的是 `\n` ，所以需要 -1 ，因为寻找的是首个换行符的位置。
- 【没找到】
  - 第 14 行：记录 `offset` 。
- 第 16 行：返回位置 `i` 。

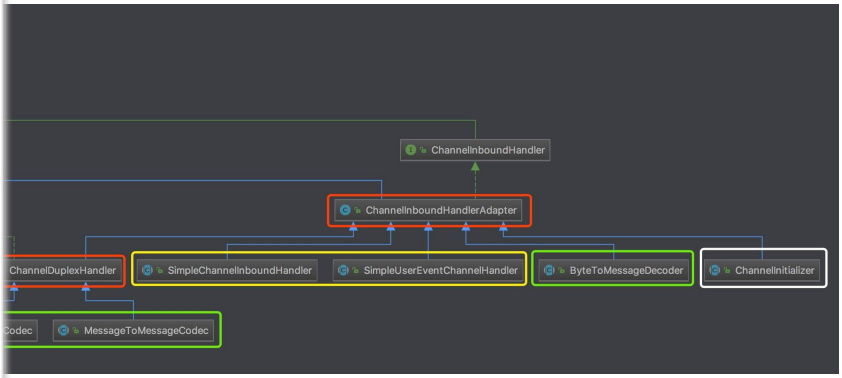
3.4 fail

`#fail(...)` 方法，触发 Exception 到下一个节点。代码如下：

```
private void fail(final ChannelHandlerContext ctx, int length) {
    fail(ctx, String.valueOf(length));
}

private void fail(final ChannelHandlerContext ctx, String length) {
    ctx.fireExceptionCaught(new TooLongFrameException("frame length (" + length + ") exceeds the allow
}
```


笔者的一个推测。下面，我们来推导下。



- 第一根红线，在 `discarding` 模式下，如果读取不到换行符，每次 `buffer` 的读取位置，都会跳到写入位置。
- 第三根红线， `offset` 记录**上一次**读取的字节数。
- 第二根红线，如果查找的范围 `+ offset` ，但是 `buffer` 的读取位置已经跳到写入位置，岂不是和 `offset` 的重复了？？

所以，笔者认为，应该去掉 `offset` 的相关逻辑。

下面，我们以一个实际情况，举个例子。如下图所示：

假设 maxLength = 1					
第一次接收到数据			第 2 次接收到数据		第 3 次接收到数据
0	1	2	3	4	\n
				r = w = 4	
				offset = 2	

- 假设 `maxLength` 等于 1 。
- 第一次接收到数据 `"012"` ，未找到换行符，但是超过最大长度，所以进入 `discarding` 模式。
- 第二次接收到数据 `"34"` ，未找到换行符， `r = w = 4` ，并且 `offset = 2` 。
- 第三次接收到数据 `"\n"` ，但是查找范围是 `buffer.readerIndex() + offset = 4 + 2 > 5` ，超过范围。

因此，笔者觉得，这个可能是 offset 的一个 bug 。

4. LengthFieldBasedFrameDecoder

`io.netty.handler.codec.LengthFieldBasedFrameDecoder` ，继承 ByteToMessageDecoder 抽象类，基于**消息头指定消息长度**进行粘包拆包处理的。

详细解析，见基友【闪电侠】的 《netty源码分析之LengthFieldBasedFrameDecoder》 一文。

或者，【Hypercube】的 《自顶向下深入分析Netty（八）– LengthFieldBasedFrameDecoder》 一文。

5. DelimiterBasedFrameDecoder

`io.netty.handler.codec.DelimiterBasedFrameDecoder` ，继承 ByteToMessageDecoder 抽象类，基于**指定消息边界

文章目录

Netty（八）–CodecHandler》 的 「8.1.2
der」 小节。

未使用引用语法。

个分隔符，每个分隔符可为一个或多个字符。如果定义了多个分隔符，并且可解
例如，使用行分隔符 `\r\n` 和 `\n` 分隔：

```
| ABC\nDEF\r\n |
+--------------+
```

可有两种结果：

```
+-----+-----+             +----------+
| ABC | DEF |  (√)   和   | ABC\nDEF |  (×)
+-----+-----+             +----------+
```

该编码器可配置的变量与 `LineBasedFrameDecoder` 类似，只是多了一个 `ByteBuf[] delimiters` 用于配置具体的分隔符。

Netty在 `Delimiters` 类中定义了两种默认的分隔符，分别是NULL分隔符和行分隔符：

```
public static ByteBuf[] nulDelimiter() {
    return new ByteBuf[] {
            Unpooled.wrappedBuffer(new byte[] { 0 }) };
}

public static ByteBuf[] lineDelimiter() {
    return new ByteBuf[] {
            Unpooled.wrappedBuffer(new byte[] { '\r', '\n' }),
            Unpooled.wrappedBuffer(new byte[] { '\n' }),
    };
}
```

666. 彩蛋

在 FixedLengthFrameDecoder 那里，卡了好长时间，Netty 在细节这块，扣的真给力啊！！！

本文参考如下文章：

- 简书闪电侠 《netty源码分析之LengthFieldBasedFrameDecoder》
- Hypercube 《自顶向下深入分析Netty（八）–CodecHandler》

欠

文章目录