08 开箱即用：Netty 支持哪些常用的解码器?

在前两节课我们介绍了 TCP 拆包/粘包的问题，以及如何使用 Netty 实现自定义协议的编解码。可以看到，网络通信的底层实现，Netty 都已经帮我们封装好了，我们只需要扩展 ChannelHandler 实现自定义的编解码逻辑即可。更加人性化的是，Netty 提供了很多开箱即用的解码器，这些解码器基本覆盖了 TCP 拆包/粘包的通用解决方案。本节课我们将对 Netty 常用的解码器进行讲解，一起探索下它们有哪些用法和技巧。

在本节课开始之前，我们首先回顾一下 TCP 拆包/粘包的主流解决方案。并梳理出 Netty 对应的编码器类。

固定长度解码器 FixedLengthFrameDecoder

固定长度解码器 FixedLengthFrameDecoder 非常简单，直接通过构造函数设置固定长度的大小 frameLength，无论接收方一次获取多大的数据，都会严格按照 frameLength 进行解码。如果累积读取到长度大小为 frameLength 的消息，那么解码器认为已经获取到了一个完整的消息。如果消息长度小于 frameLength，FixedLengthFrameDecoder 解码器会一直等后续数据包的到达，直至获得完整的消息。下面我们通过一个例子感受一下使用 Netty 实现固定长度解码是多么简单。

```
public class EchoServer {

    public void startEchoServer(int port) throws Exception {

        EventLoopGroup bossGroup = new NioEventLoopGroup();

        EventLoopGroup workerGroup = new NioEventLoopGroup();

        try {

            ServerBootstrap b = new ServerBootstrap();

            b.group(bossGroup, workerGroup)

                    .channel(NioServerSocketChannel.class)

                    .childHandler(new ChannelInitializer<SocketChannel>() {

                        @Override
```

```
                    public void initChannel(SocketChannel ch) {
                        ch.pipeline().addLast(new FixedLengthFrameDecoder(10));
                        ch.pipeline().addLast(new EchoServerHandler());
                    }
                });
            ChannelFuture f = b.bind(port).sync();
            f.channel().closeFuture().sync();
        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
    public static void main(String[] args) throws Exception {
        new EchoServer().startEchoServer(8088);
    }
}
@Sharable
public class EchoServerHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println("Receive client : [" + ((ByteBuf) msg).toString(CharsetU
    }
}
```

在上述服务端的代码中使用了固定 10 字节的解码器，并在解码之后通过 EchoServerHandler 打印结果。我们可以启动服务端，通过 telnet 命令像服务端发送数据，观察代码输出的结果。

客户端输入：

```
telnet localhost 8088

Trying ::1...

Connected to localhost.

Escape character is '^]'.

1234567890123

456789012
```

服务端输出：

```
Receive client : [1234567890]

Receive client : [123

45678]
```

特殊分隔符解码器 DelimiterBasedFrameDecoder

使用特殊分隔符解码器 DelimiterBasedFrameDecoder 之前我们需要了解以下几个属性的作用。

- **delimiters**

delimiters 指定特殊分隔符，通过写入 ByteBuf 作为**参数**传入。delimiters 的类型是 ByteBuf 数组，所以我们可以同时指定多个分隔符，但是最终会选择长度最短的分隔符进行消息拆分。

例如接收方收到的数据为：

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

| ABC\nDEF\r\n |

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

如果指定的多个分隔符为 \n 和 \r\n，DelimiterBasedFrameDecoder 会退化成使用 LineBasedFrameDecoder 进行解析，那么会解码出两个消息。

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

| ABC | DEF |
```

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

如果指定的特定分隔符只有 \r\n，那么只会解码出一个消息：

```
+----------+
| ABC\nDEF |
+----------+
```

- **maxLength**

maxLength 是报文最大长度的限制。如果超过 maxLength 还没有检测到指定分隔符，将会抛出 TooLongFrameException。可以说 maxLength 是对程序在极端情况下的一种**保护措施**。

- **failFast**

failFast 与 maxLength 需要搭配使用，通过设置 failFast 可以控制抛出 TooLongFrameException 的时机，可以说 Netty 在细节上考虑得面面俱到。如果 failFast=true，那么在超出 maxLength 会立即抛出 TooLongFrameException，不再继续进行解码。如果 failFast=false，那么会等到解码出一个完整的消息后才会抛出 TooLongFrameException。

- **stripDelimiter**

stripDelimiter 的作用是判断解码后得到的消息是否去除分隔符。如果 stripDelimiter=false，特定分隔符为 \n，那么上述数据包解码出的结果为：

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

下面我们还是结合代码示例学习 DelimiterBasedFrameDecoder 的用法，依然以固定编码器小节中使用的代码为基础稍做改动，引入特殊分隔符解码器 DelimiterBasedFrameDecoder：

```
b.group(bossGroup, workerGroup)
    .channel(NioServerSocketChannel.class)
```

```
    .childHandler(new ChannelInitializer<SocketChannel>() {
        @Override
        public void initChannel(SocketChannel ch) {
            ByteBuf delimiter = Unpooled.copiedBuffer("&".getBytes());
            ch.pipeline().addLast(new DelimiterBasedFrameDecoder(10, true, true, de
            ch.pipeline().addLast(new EchoServerHandler());
        }
    });
```

我们依然通过 telnet 模拟客户端发送数据，观察代码输出的结果，可以发现由于 maxLength 设置的只有 10，所以在解析到第三个消息时抛出异常。

客户端输入：

```
telnet localhost 8088
Trying ::1...
Connected to localhost.
Escape character is '^]'.
hello&world&1234567890ab
```

服务端输出：

```
Receive client : [hello]
Receive client : [world]
九月 25, 2020 8:46:01 下午 io.netty.channel.DefaultChannelPipeline onUnhandledInbour
警告: An exceptionCaught() event was fired, and it reached at the tail of the pipel
io.netty.handler.codec.TooLongFrameException: frame length exceeds 10: 13 - discard
        at io.netty.handler.codec.DelimiterBasedFrameDecoder.fail(DelimiterBasedFra
        at io.netty.handler.codec.DelimiterBasedFrameDecoder.decode(DelimiterBasedF
        at io.netty.handler.codec.DelimiterBasedFrameDecoder.decode(DelimiterBasedF
```

长度域解码器 LengthFieldBasedFrameDecoder

长度域解码器 LengthFieldBasedFrameDecoder 是解决 TCP 拆包/粘包问题最常用的**解码器。**它基本上可以覆盖大部分基于长度拆包场景，开源消息中间件 RocketMQ 就是使用 LengthFieldBasedFrameDecoder 进行解码的。LengthFieldBasedFrameDecoder 相比 FixedLengthFrameDecoder 和 DelimiterBasedFrameDecoder 要复杂一些，接下来我们就一起学习下这个强大的解码器。

首先我们同样先了解 LengthFieldBasedFrameDecoder 中的几个重要属性，这里我主要把它们分为两个部分：**长度域解码器特有属性**以及**与其他解码器（如特定分隔符解码器）的相似的属性。**

- **长度域解码器特有属性。**

```
// 长度字段的偏移量，也就是存放长度数据的起始位置
private final int lengthFieldOffset;
// 长度字段所占用的字节数
private final int lengthFieldLength;
/*
 * 消息长度的修正值
 *
 * 在很多较为复杂一些的协议设计中，长度域不仅仅包含消息的长度，而且包含其他的数据，如版本号
 *
 * lengthAdjustment = 包体的长度值 - 长度域的值
 *
 */
private final int lengthAdjustment;
// 解码后需要跳过的初始字节数，也就是消息内容字段的起始位置
private final int initialBytesToStrip;
// 长度字段结束的偏移量，lengthFieldEndOffset = lengthFieldOffset + lengthFieldLength
private final int lengthFieldEndOffset;
```

- **与固定长度解码器和特定分隔符解码器相似的属性。**

```
private final int maxFrameLength; // 报文最大限制长度

private final boolean failFast; // 是否立即抛出 TooLongFrameException，与 maxFrameLe

private boolean discardingTooLongFrame; // 是否处于丢弃模式

private long tooLongFrameLength; // 需要丢弃的字节数

private long bytesToDiscard; // 累计丢弃的字节数
```

下面我们结合具体的示例来解释下每种参数的组合，其实在 Netty LengthFieldBasedFrameDecoder 源码的注释中已经描述得非常详细，一共给出了 7 个场景示例，理解了这些示例基本上可以真正掌握 LengthFieldBasedFrameDecoder 的参数用法。

示例 1：典型的基于消息长度 + 消息内容的解码。

```
BEFORE DECODE (14 bytes)         AFTER DECODE (14 bytes)

+--------+----------------+      +--------+----------------+

| Length | Actual Content |----->| Length | Actual Content |

| 0x000C | "HELLO, WORLD" |      | 0x000C | "HELLO, WORLD" |

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

上述协议是最基本的格式，报文只包含消息长度 Length 和消息内容 Content 字段，其中 Length 为 16 进制表示，共占用 2 字节，Length 的值 0x000C 代表 Content 占用 12 字节。该协议对应的解码器参数组合如下：

- lengthFieldOffset = 0，因为 Length 字段就在报文的开始位置。
- lengthFieldLength = 2，协议设计的固定长度。
- lengthAdjustment = 0，Length 字段只包含消息长度，不需要做任何修正。
- initialBytesToStrip = 0，解码后内容依然是 Length + Content，不需要跳过任何初始字节。

示例 2：解码结果需要截断。

```
BEFORE DECODE (14 bytes)         AFTER DECODE (12 bytes)

+--------+----------------+      +----------------+

| Length | Actual Content |----->| Actual Content |
```

```
| 0x000C | "HELLO, WORLD" |      | "HELLO, WORLD" |
+--------+----------------+      +----------------+
```

示例 2 和示例 1 的区别在于解码后的结果只包含消息内容，其他的部分是不变的。该协议对应的解码器参数组合如下：

- lengthFieldOffset = 0，因为 Length 字段就在报文的开始位置。
- lengthFieldLength = 2，协议设计的固定长度。
- lengthAdjustment = 0，Length 字段只包含消息长度，不需要做任何修正。
- initialBytesToStrip = 2，跳过 Length 字段的字节长度，解码后 ByteBuf 中只包含 Content字段。

示例 3：长度字段包含消息长度和消息内容所占的字节。

```
BEFORE DECODE (14 bytes)         AFTER DECODE (14 bytes)
+--------+----------------+      +--------+----------------+
| Length | Actual Content |----->| Length | Actual Content |
| 0x000E | "HELLO, WORLD" |      | 0x000E | "HELLO, WORLD" |
+--------+----------------+      +--------+----------------+
```

与前两个示例不同的是，示例 3 的 Length 字段包含 Length 字段自身的固定长度以及 Content 字段所占用的字节数，Length 的值为 0x000E（2 + 12 = 14 字节），在 Length 字段值（14 字节）的基础上做 lengthAdjustment（-2）的修正，才能得到真实的 Content 字段长度，所以对应的解码器参数组合如下：

- lengthFieldOffset = 0，因为 Length 字段就在报文的开始位置。
- lengthFieldLength = 2，协议设计的固定长度。
- lengthAdjustment = -2，长度字段为 14 字节，需要减 2 才是拆包所需要的长度。
- initialBytesToStrip = 0，解码后内容依然是 Length + Content，不需要跳过任何初始字节。

示例 4：基于长度字段偏移的解码。

```
BEFORE DECODE (17 bytes)                      AFTER DECODE (17 bytes)
+----------+----------+----------------+      +----------+----------+--------------
```

```
| Header 1 |  Length  | Actual Content |----->| Header 1 |  Length  | Actual Conten
|  0xCAFE  | 0x00000C | "HELLO, WORLD" |      |  0xCAFE  | 0x00000C | "HELLO, WORLD
+----------+----------+----------------+      +----------+----------+-------------
```

示例 4 中 Length 字段不再是报文的起始位置，Length 字段的值为 0x00000C，表示 Content 字段占用 12 字节，该协议对应的解码器参数组合如下：

- lengthFieldOffset = 2，需要跳过 Header 1 所占用的 2 字节，才是 Length 的起始位置。
- lengthFieldLength = 3，协议设计的固定长度。
- lengthAdjustment = 0，Length 字段只包含消息长度，不需要做任何修正。
- initialBytesToStrip = 0，解码后内容依然是完整的报文，不需要跳过任何初始字节。

示例 5：长度字段与内容字段不再相邻。

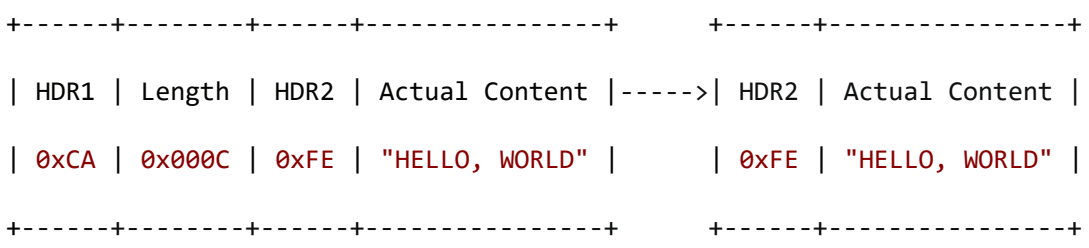
```
BEFORE DECODE (17 bytes)                      AFTER DECODE (17 bytes)
+----------+----------+----------------+      +----------+----------+-------------
|  Length  | Header 1 | Actual Content |----->|  Length  | Header 1 | Actual Conten
| 0x00000C |  0xCAFE  | "HELLO, WORLD" |      | 0x00000C |  0xCAFE  | "HELLO, WORLD
+----------+----------+----------------+      +----------+----------+-------------
```

示例 5 中的 Length 字段之后是 Header 1，Length 与 Content 字段不再相邻。Length 字段所表示的内容略过了 Header 1 字段，所以也需要通过 lengthAdjustment 修正才能得到 Header + Content 的内容。示例 5 所对应的解码器参数组合如下：

- lengthFieldOffset = 0，因为 Length 字段就在报文的开始位置。
- lengthFieldLength = 3，协议设计的固定长度。
- lengthAdjustment = 2，由于 Header + Content 一共占用 2 + 12 = 14 字节，所以 Length 字段值（12 字节）加上 lengthAdjustment（2 字节）才能得到 Header + Content 的内容（14 字节）。
- initialBytesToStrip = 0，解码后内容依然是完整的报文，不需要跳过任何初始字节。

示例 6：基于长度偏移和长度修正的解码。

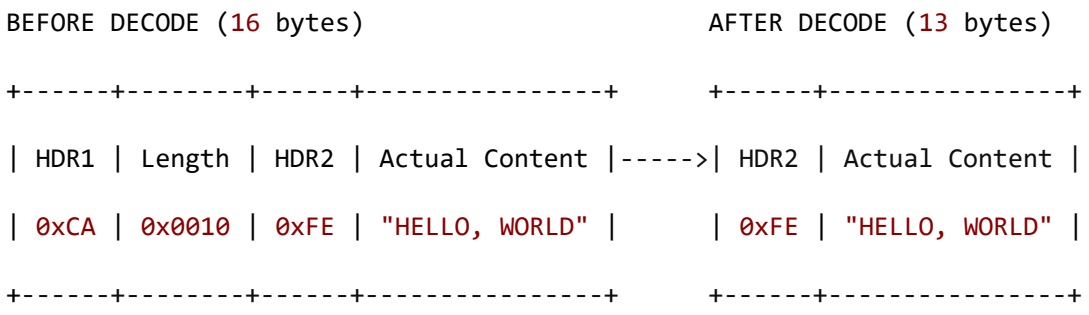
```
BEFORE DECODE (16 bytes)                       AFTER DECODE (13 bytes)
```



示例 6 中 Length 字段前后分为别 HDR1 和 HDR2 字段，各占用 1 字节，所以既需要做长度字段的偏移，也需要做 lengthAdjustment 修正，具体修正的过程与 示例 5 类似。对应的解码器参数组合如下：

- lengthFieldOffset = 1，需要跳过 HDR1 所占用的 1 字节，才是 Length 的起始位置。
- lengthFieldLength = 2，协议设计的固定长度。
- lengthAdjustment = 1，由于 HDR2 + Content 一共占用 1 + 12 = 13 字节，所以 Length 字段值（12 字节）加上 lengthAdjustment（1）才能得到 HDR2 + Content 的内容（13 字节）。
- initialBytesToStrip = 3，解码后跳过 HDR1 和 Length 字段，共占用 3 字节。

示例 7：长度字段包含除 Content 外的多个其他字段。



示例 7 与 示例 6 的区别在于 Length 字段记录了整个报文的长度，包含 Length 自身所占字节、HDR1 、HDR2 以及 Content 字段的长度，解码器需要知道如何进行 lengthAdjustment 调整，才能得到 HDR2 和 Content 的内容。所以我们可以采用如下的解码器参数组合：

- lengthFieldOffset = 1，需要跳过 HDR1 所占用的 1 字节，才是 Length 的起始位置。
- lengthFieldLength = 2，协议设计的固定长度。
- lengthAdjustment = -3，Length 字段值（16 字节）需要减去 HDR1（1 字节） 和 Length 自身所占字节长度（2 字节）才能得到 HDR2 和 Content 的内容（1 + 12 = 13 字节）。
- initialBytesToStrip = 3，解码后跳过 HDR1 和 Length 字段，共占用 3 字节。

以上 7 种示例涵盖了 LengthFieldBasedFrameDecoder 大部分的使用场景，你是否学会了呢？最后留一个小任务，在上一节课程中我们设计了一个较为通用的协议，如下所示。如何使用长度域解码器 LengthFieldBasedFrameDecoder 完成该协议的解码呢？抓紧自己尝试下吧。

```
+---------------------------------------------------------------+
| 魔数 2byte | 协议版本号 1byte | 序列化算法 1byte | 报文类型 1byte  |
+---------------------------------------------------------------+
| 状态 1byte |        保留字段 4byte     |      数据长度 4byte     |
+---------------------------------------------------------------+
|                     数据内容 （长度不定）                       |
+---------------------------------------------------------------+
```

总结

本节课我们介绍了三种常用的解码器，从中我们可以体会到 Netty 在设计上的优雅，只需要调整参数就可以轻松实现各种功能。在健壮性上，Netty 也考虑得非常全面，很多边界情况 Netty 都贴心地增加了保护性措施。实现一个健壮的解码器并不容易，很可能因为一次解析错误就会导致解码器一直处理错乱的状态。如果你使用了基于长度编码的二进制协议，那么推荐你使用 LengthFieldBasedFrameDecoder，它已经可以满足实际项目中的大部分场景，基本不需要再自定义实现了。希望朋友们在项目开发中能够学以致用。

上一页 下一页