死磕 Netty —— 大明哥带你彻底搞定 Netty 的编解码器

> 大家好，我是大明哥，一个专注「[死磕 Java](http://cxyroad.com/”https://www.skjava.com/sike-java”)」系列创作的硬核程序员。
>
> 本文已收录到我的技术网站：[www.skjava.com](http://cxyroad.com/”https://skjava.com/”)。有全网最优质的系列文章、Java 全栈技术文档以及大厂完整面经

要搞定 Netty 的编解码器，我们首先需要先明白什么是拆包/粘包。

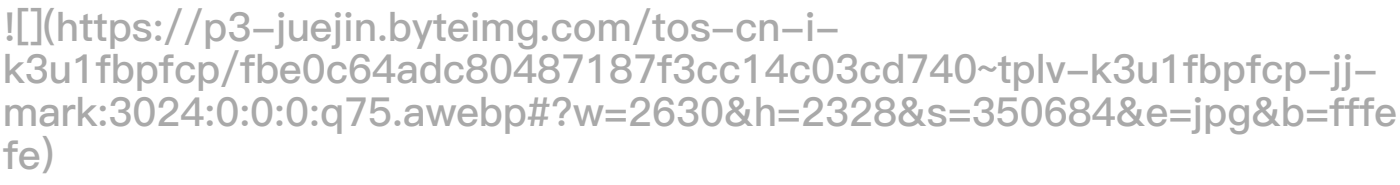
拆包/粘包

现象演示

首先我们需要知道什么是拆包/粘包现象。

假设客户端向服务端发送两个数据包，分别为 package1 和 package2，这个时候服务端接收到客户端的数据有可能有如下四种情况。

* 情况 1：服务端正常接收 package1 和 package 2，这种属于正常情况。
* 情况 2：服务端只接收到了一个 package，由于 TCP 保证送达的特性，所以这个 package 包含了客户端发送的两个 package ，这种现象属于粘包现象。如果客户端和服务端没有对应的协议来明确 package1 和 package2 的界限，那么服务端是无法区分 package1 和 package2 的。
* 情况 3：服务端可能会接收到 3 个 package，package1 可能会被拆分为 package1.1 和 package1.2 ，这种现象属于拆包现象。
* 情况 4：服务端接收到 2 个 package，但是这两个 package 都不为完整的，比如 package1 拆分成了 package1.1 和 package1.2 ，但是服务端接收的两个包为 package1.1 和 package1.2 + package2，这种情况是拆包和粘包的综合体。



下面大明哥通过两个例子来分别阐述 Netty 中的拆包/粘包现象。

粘包现象

从上面图中可以看出，拆包其实就是多个数据包合并成一个。所以我们只需要在客户端发送多个消息给服务端，看服务端是否是接收多次就可以了。

* 服务端代码

```
public class StickyServer {
    public static void main(String[] args) {
        new ServerBootstrap()
                .group(new NioEventLoopGroup(),new NioEventLoopGroup())
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new
LoggingHandler(LogLevel.DEBUG));
                    }
                })
                .bind(8081);
```

```
    }
}
```

服务端没有多余的代码，只有一个 LoggingHandler 的 ChannelHandler，该 Handler 主要是用于打印服务端的日志情况。

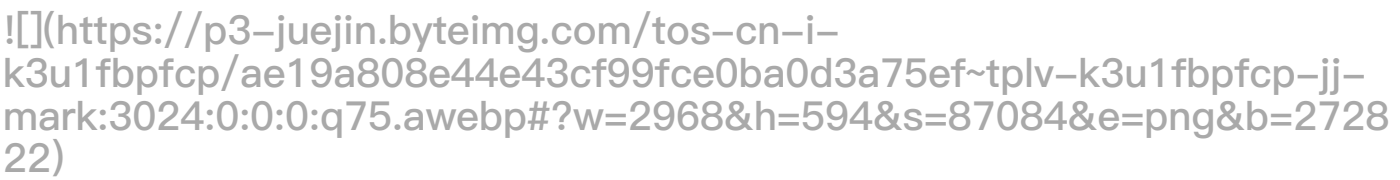
* 客户端代码

```
public class StickyClient {
    public static void main(String[] args) {
        new Bootstrap()
                .group(new NioEventLoopGroup())
                .channel(NioSocketChannel.class)
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new
ChannelInboundHandlerAdapter() {
                            @Override
                            public void channelActive(ChannelHandlerContext
ctx) throws Exception {
                                log.info("客户端连接成功，开始发送数据");
                                for (int i = 0 ; i < 10 ; i++) {
                                    ByteBuf byteBuf = ctx.alloc().buffer(16);
                                    byteBuf.writeBytes(new
byte[]{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15});
                                    ctx.channel().writeAndFlush(byteBuf);
                                }
                                log.info("数据已发送完成");
                            }
                        });
                    }
                }).connect("127.0.0.1",8081);

    }
}
```

客户端与服务端建立连接后，就向服务端发送 10 次消息，每次 16 byte。

* 服务端日志



从服务端的运行日志中我们可以看出，客户端虽然发了 10 次，但是服务端只接收了一次，一次 160 byte，充分展示了粘包情况。

拆包现象

拆包就和粘包相反，它是将一个数据包拆分为多个数据包。

* 服务端

```
public class UnpackServer {
    public static void main(String[] args) {
        new ServerBootstrap()
                .group(new NioEventLoopGroup(),new NioEventLoopGroup())
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new StringDecoder());
                        ch.pipeline().addLast(new
ChannelInboundHandlerAdapter(){

                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                                System.out.println("服务端接收的内容：" +
msg.toString());
                            }
                        });
                    }
                })
                .bind(8081);
```

```
    }
}
```

服务端就单纯地将客户端发送过来的消息打印即可。

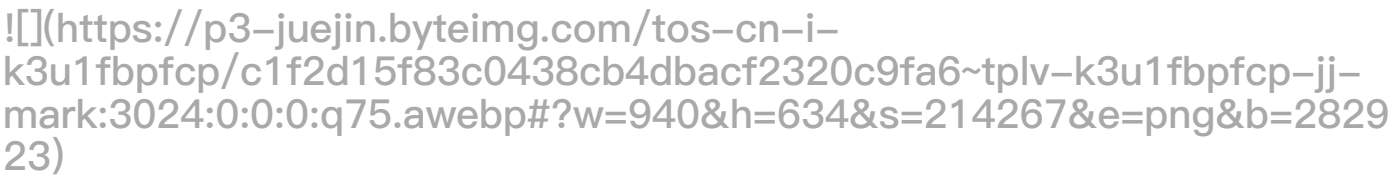
* 客户端

```
public class UnpackClient {
    public static void main(String[] args) {
        new Bootstrap()
                .group(new NioEventLoopGroup())
                .channel(NioSocketChannel.class)
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new StringEncoder());
                        ch.pipeline().addLast(new
ChannelInboundHandlerAdapter() {
                            @Override
                            public void channelActive(ChannelHandlerContext
ctx) throws Exception {
                                for (int i = 0 ; i < 500 ; i++) {
                                    ctx.channel().writeAndFlush("大家好，我是大明哥
，一个专注[死磕 Java] 的男人!!!\r\n");
                                }
                            }
                        });
                    }
                }).connect("127.0.0.1",8081);

    }
}
```

客户端建立连接后，向服务端发送 500 次消息“大家好，我是大明哥，一个专注[死磕 Java] 的男人!!!\r\n”

* 运行结果



运行结果中有一段是乱码，消息也不是完整的，所以这里一定是不完整的，发生了拆包现象。

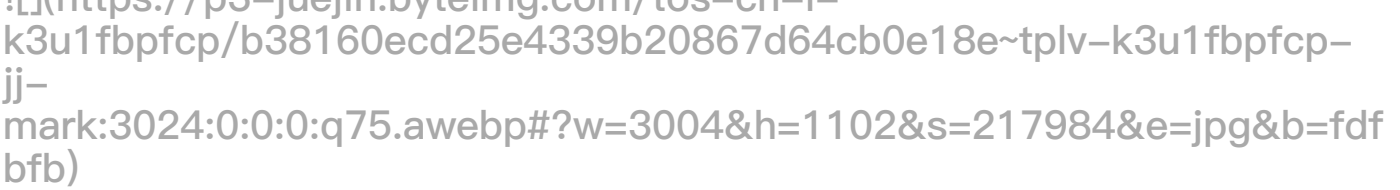
为什么会有拆包/粘包

客户端消息明明是一条一条地发，为什么会有拆包/粘包情况呢？TCP 是传输层协议，它并不了解我们应用层业务数据的含义，它会根据实际情况对数据包进行划分。所以在业务上我们认为是一个完整的数据包，可能会被 TCP 拆分为多个数据库报进行发送，也有可能会将多个数据包合并成一个数据包发送，这就会出现拆包/粘包的问题。

在网络通信的过程中，影响可以发送的数据包大小受很多因素限制，比如 MTU 传输单元大小、MSS 最大报文长度、滑动窗口。同时 TCP 也采用了 Nagle 算法对网络数据包进行了优化，所以要了解 TCP 为什么会有拆包/粘包问题，就需要了解这些概念。

MTU 最大传输单元

> MTU（Maximum Transmission Unit），最大传输单元，是指网络能够传输的最大数据包大小，它决定了发送端一次能够发送报文的最大字节数。它是链路层协议，其最大值默认为 1500 byte。



MTU 是数据链路层对网络层的限制，最小为 46 byte，最大为 1500 byte，意思就是说网络层必须将发给网卡 API 的数据包大小控制在 1500 byte 以下，否则失败。

那为什么要有一个这样的限制呢？我们都知道网络中通常是以数据包为单位进行信息传输的，那么一次传输多大的数据包就决定了整个传输过程中的效率了，理论上数据包的大小设置为尽可能大，因为着有效的数据量就越大，传输的效率也就越高，但是传输一个数据包的延迟就会很大，而且数据包中 bite 位发送错误的概率也就越大，如果这个数据包丢失了，那么重传的代价就会很大。但是如果我们将数据包大小设置较小，那么我们传输的有效数据就会很小，传输效率就会比较低。所以我们就需要 MTU 来控制网络上传输数据包的大小，如果数据包大，我们就将其拆分，如果小，我们就把几个数据包进行合并，从而提供传输效率。

MSS 最大报文长度

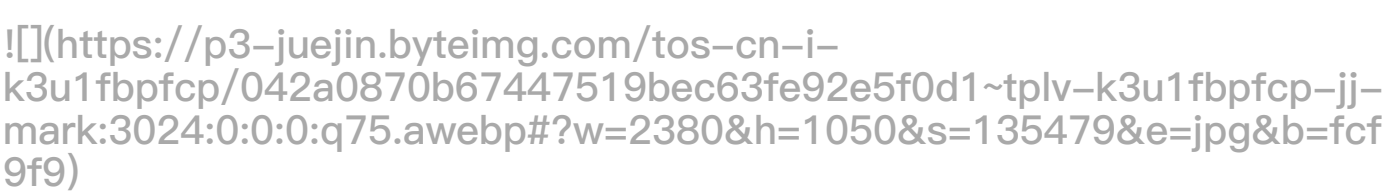
> MSS（Maximum Segment Size），最大报文长度，它表示 TCP payload 的最大值，它是 TCP 用来限制应用层发送的最大字节数。

我们知道了 MTU 限定了网络层往数据链路层发送数据包的大小，如果网络层发现一个数据包大于 MTU，那么它需要将其进行分片，切割成小于 MTU 的数据包，再将其发送到数据链路层。

一台主机上的所有应用都将数据包发往网络层，如果这些数据包太大了，则需要对其进行分片，但是这么多数据包都交给网络层来分片，是不是降低了效率？作为网络层，它的理想状态是，让 TCP 来的每一个数据包，大小都小于 MTU，这样它就不需要分片了。

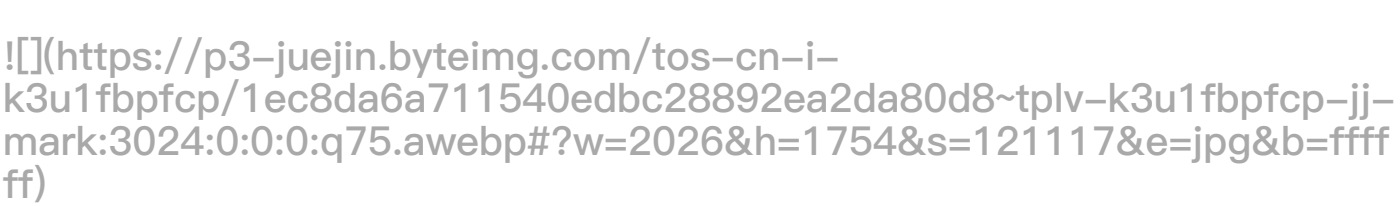
MSS 是 TCP 协议定义的一个选项，是 TCP 用来限定应用层最大的发送字节数。它是在 TCP 连接建立时，双方进行约定的。当一个 TCP 连接建立时，连接的双方都需要通告各自的 MSS，以避免分片。

TCP 建立连接时，双方都需要根据 MTU 来计算各自的 MSS，计算规则如下：



MTU = IP Header(20) + TCP Header(20) + Data，MTU 默认最大值为 1500，所以 TCP 的有效数据 Data 的最大值为 1500 − 20 − 20 = 1460 ，这个值就是 MSS 的值。

MSS 的值是通过三次握手的方式告知对方的，互相确认对方的 MSS 值大小，取较小的那个作为 MSS。



1. 客户端在发送的 SYN 报文中携带自己的 MSS（1300）。
2. 服务端接收该报文后，取客户端的 MSS（1300） 和自己本地的 MSS（1200）中较小的那个值作为自己的 MSS（1200）。在回复的 SYN-ACK 中也携带自己的 MSS（1200）。
3. 客户端收到该 SYN-ACK 后，取服务端的 MSS（1200）和自己本地的 MSS（1200）中较小的那个值作为客户端的 MSS（1200）。

Nagle 算法

> **Nagle 算法**于 1984 年被福特航空和通信公司定义为 TCP/IP 拥塞控制方法。它主要用于解决频繁发送小数据包而带来的网络拥塞问题。

为了尽可能地利用网络带宽，TCP 总是希望能够发送足够大的数据包，由于有 MSS 的控制，所以它总是希望每次都能够以 MSS 的尺寸来发送数据。但是我们需要发送的数据并不会每次都有那么多字节，怎么办？攒着。Nagle 算法会在数据为得到确认之前会先将其写入到缓冲区中，等待数据确认或者缓冲区积攒到一定大小再把数据包发送出去。

Nagle 算法就是为了尽可能发送大块数据,避免网络中充斥着许多小数据块。

Nagle 能够有效地降低网络开销，但是它会有一定的延时性，如果我们的业务系统对时延要求比较高的话，希望发出去的消息都能够尽快地响应，这个时候我们就需要关闭 Nagle 算法了。Netty 为了使数据传输延迟最小化，所以就默认禁用了 Nagle 算法。

在 Netty 中可以通过参数 `ChannelOption.TCP_NODELAY` 来开启和关闭 Nagle 算法。

Netty 的编解码器
==========

拆包/粘包解决方案

TCP 是面向字节流的协议，它是无法区分数据包界限的。既然底层的 TCP 协议无法区分，那我们就只能在应用层下功夫了 。目前在应用层主流的解决方案有三种：

* 固定长度

双方约定一个固定的长度，比如 100 个字节，那么发送端在发送消息时，每个报文都是 100 个字节，不足的补空格或者 0 等其他特殊字符。接收端则每次读取 100 个字节当做一个完整的报文。

* 特殊字符

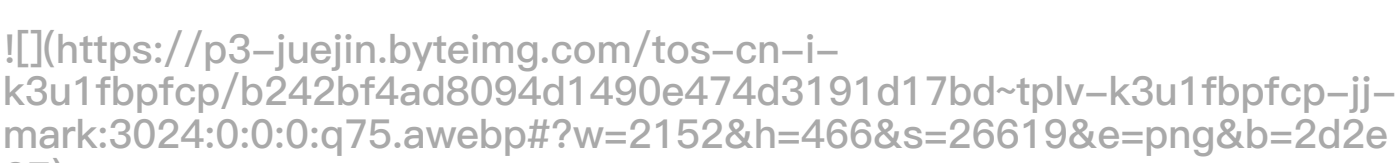
在报文尾部增加一个特殊字符（比如换行符）来作为分割符，接收端接受到消息后可以根据这个分隔符来判断这个消息是否 完整 。

* 消息头携带信息

将消息分为消息头和消息体，消息头中包内含消息的长度，接收端获取消息后，从消息头解析出消息的长度，然后向后读取该长度的内容。

Netty 常用的解码器

Netty 提供了几种开箱即用的解码器，这些解码器基本覆盖了 TCP 拆包/粘包的通用解决方案。



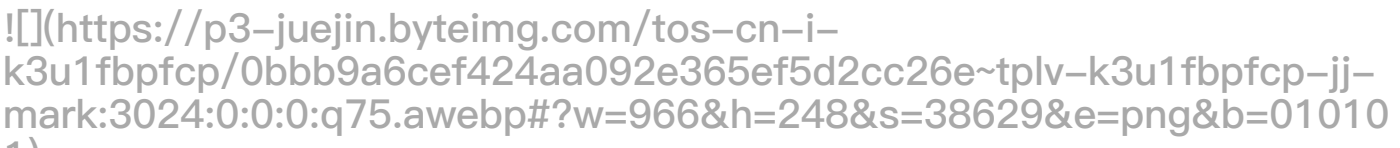
下面我们就来了解这些解码器吧！

FixedLengthFrameDecoder

FixedLengthFrameDecoder 为定长解码器，使用起来非常方便，只需要通过构造函数设定一个长度 frameLength 即可。无论发送方怎么发送数据，它都会严格按照设定的长度 frameLength 来解码。下面就来演示下。

```
public class FixedLengthFrameServer {
    public static void main(String[] args) {
        new ServerBootstrap()
                .group(new NioEventLoopGroup(),new NioEventLoopGroup())
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new
FixedLengthFrameDecoder(8));
                        ch.pipeline().addLast(new
LoggingHandler(LogLevel.DEBUG));
                        ch.pipeline().addLast(new
ChannelInboundHandlerAdapter(){
                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                                System.out.println(”接收内容：” +
((ByteBuf)msg).toString(Charset.defaultCharset()));
                            }
                        });
                    }
                })
                .bind(8081);
    }
}
```

`new FixedLengthFrameDecoder(8)`，固定 8 byte 的解码器。我们通过 telnet 命令向服务端发送数据，发送内容为：



服务端响应结果：

绿色部分是每次读取缓冲区的大小，这里可以看出都是 8 byte。第一次发送 `1234567890`，服务端只读取了 `12345678`，其中 `90\n`还保留这，继续发送 `12345678` ，这个时候够了 8 byte，服务端又读取 `90\n1234`，`5678` 保留。

FixedLengthFrameDecoder 的优势就在于使用起来非常简单，也很容易理解，但是缺点就在于，客户端每次发送报文的时候都要补齐 N 位，不够用特殊字符来补齐，是非常浪费空间的，比如定义固定长度为 1024，但是我们发送的数据为 `1`，客户端在发送这个报文时也需要补齐 1024 位，但是这个时候的有效位只有 1 位，1023 位都是无效数据，太浪费了。

LineBasedFrameDecoder

LineBasedFrameDecoder 是基于回车换行符解码器，它能够按照我们输入的回车换行符（ `\n`or `\r\n`）对接收到的消息进行解码。

LineBasedFrameDecoder 的构造器接受一个 int 类型的参数 `maxLength`，用来限制一次最大的解码长度。如果超过 `maxLength` 还没有检测到回车换行符，就会抛出 `TooLongFrameException`，可以说 `maxLength` 是对程序的一种 保护措施。

我们来演示下。

```
public class LineBasedFrameDecoderServer {
```

```
    public static void main(String[] args) {
        new ServerBootstrap()
                .group(new NioEventLoopGroup(), new NioEventLoopGroup())
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        ch.pipeline().addLast(new LineBasedFrameDecoder(12));
                        ch.pipeline().addLast(new
LoggingHandler(LogLevel.DEBUG));
                        ch.pipeline().addLast(new
ChannelInboundHandlerAdapter() {
                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                                System.out.println("接收内容：" + ((ByteBuf)
msg).toString(Charset.defaultCharset()));
                            }
                        });
                    }
                })
                .bind(8081);
    }
}

```

定义 `LineBasedFrameDecoder(12)`，即解码最大的长度为 12 byte，超过 12 byte 的数据都会丢弃。客户端发送内容为 `123\n45678\r90\r\nabcdf\n\rghijkmnopqrst\n`，服务端解析结果如下：

```
2022-08-09 08:57:29.933 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0x13804585,
L:/127.0.0.1:8081 - R:/127.0.0.1:57368] READ: 3B
         +-------------------------------------------------+
         | 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
+--------+-------------------------------------------------
--+----------------+
|00000000| 31 32 33                                        |123             |
+--------+-------------------------------------------------
--+----------------+
接收内容：123
2022-08-09 08:57:29.934 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0x13804585,
```

```
L:/127.0.0.1:8081 - R:/127.0.0.1:57368] READ: 8B
         +-------------------------------------------------+
         | 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
+--------+-------------------------------------------------+----------------+
|00000000| 34 35 36 37 38 0d 39 30                         |45678.90        |
+--------+-------------------------------------------------+----------------+
90
2022-08-09 08:57:29.934 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0x13804585,
L:/127.0.0.1:8081 - R:/127.0.0.1:57368] READ: 5B
         +-------------------------------------------------+
         | 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
+--------+-------------------------------------------------+----------------+
|00000000| 61 62 63 64 66                                  |abcdf           |
+--------+-------------------------------------------------+----------------+
接收内容：abcdf
2022-08-09 08:57:29.935 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0x13804585,
L:/127.0.0.1:8081 - R:/127.0.0.1:57368] EXCEPTION:
io.netty.handler.codec.TooLongFrameException: frame length (14)
exceeds the allowed maximum (12)
io.netty.handler.codec.TooLongFrameException: frame length (14)
exceeds the allowed maximum (12)
.....
```

从结果可以看到：

* 第一次：读取 3 个字节，内容为 `123`，所以 `\n` 可以解析。
* 第二次：读取 8 个字节，内容为 `45678.90，`所以 `\r` 无法解析，`\r\n` 可以解析。
* 第三次：读取 5 个字节，内容为 `abcdf`
* 第四次：抛出 TooLongFrameException 异常，因为我们输入的内容为 14 个字节超过了 12 个字节。细心的小伙伴可能会发现 `ghijkmnopqrst` 不是只有 12 个字节么，怎么会是 14 呢？因为前面还有一个 `\r`

DelimiterBasedFrameDecoder

DelimiterBasedFrameDecoder 是特殊分隔符解码器，它和 LineBasedFrameDecoder 相似，只不是 LineBasedFrameDecoder 是固定回

车换行符为分割符而已。相比 LineBasedFrameDecoder，DelimiterBasedFrameDecoder 更加通用，允许我们指定任何特殊字符作为分割符，而且提供了更加精细的控制。

DelimiterBasedFrameDecoder 提供了多个构造方法以供我们来使用它，但最终都是调用以下构造方法：

```
    public DelimiterBasedFrameDecoder(
            int maxFrameLength, boolean stripDelimiter, boolean failFast,
ByteBuf... delimiters) {
    }
```

下面我们就来了解这个构造方法每个属性的含义。

* **delimiters**

delimiters 指定分割符。我们可以指定一个或者多个分割符，如果指定多个，那么 DelimiterBasedFrameDecoder 在解码的时候会选择长度最短的分割符进行消息拆分。

比如

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

如果我们指定分隔符为 \n 和 \r\n，那么将会解码出 2 个消息：

```
+-----+-----+
| ABC | DEF |
+-----+-----+
```

```

如果我们指定分割符为 \r\n，那只会解码出来 1 个消息：

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

* **maxLength**

maxLength 为最大报文长度限制，与 LineBasedFrameDecoder 的 maxLength 属性意义一样：如果超过 maxLength 还没有检测到分割符，就会抛出 TooLongFrameException。

* **failFast**

failFast 为是否快速失败开关。它与 maxLength 需要搭配使用，通过设置 failFast 可以控制抛出 TooLongFrameException 的时机。如果 failFast = true，那么就会立刻抛出 TooLongFrameException，不再继续解码。如果 failFast = false，那么会 等到解码出这个完整的消息后才会抛出 TooLongFrameException。

* **stripDelimiter**

用于判断解码后是否需要去掉分割符。如果为 false，那么上面的解码结果为：

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

下面我们来小试牛刀。
```

```
public class DelimiterBasedFrameDecoderServer {
    public static void main(String[] args) {
        new ServerBootstrap()
                .group(new NioEventLoopGroup(), new NioEventLoopGroup())
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws
Exception {
                        ByteBuf delimiter =
Unpooled.copiedBuffer("|".getBytes());
                        ch.pipeline().addLast(new
DelimiterBasedFrameDecoder(10, false, false, delimiter));

                        ch.pipeline().addLast(new
LoggingHandler(LogLevel.DEBUG));
                        ch.pipeline().addLast(new
ChannelInboundHandlerAdapter() {
                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
                                System.out.println("接收内容：" + ((ByteBuf)
msg).toString(Charset.defaultCharset()));
                            }
                        });
                    }
                })
                .bind(8081);
    }
}
```

咱们的解码器为： `new DelimiterBasedFrameDecoder(10, true, true, delimiter)`：

* 分隔符 delimiter：|
* 最大报文长度 maxLength ： 10
* 是否快速失败 failFast：false
* 是否去掉分隔符stripDelimiter：false

我们发送如下内容：

```
hello,|this is|sikejava.com|wula|
```

运行结果：

```
2022-08-10 21:53:50.706 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0x96ae16a6,
L:/127.0.0.1:8081 - R:/127.0.0.1:63971] READ: 7B
         +-------------------------------------------------+
         | 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
+--------+-------------------------------------------------
--+----------------+
|00000000| 68 65 6c 6c 6f 2c 7c                            |hello,|         |
+--------+-------------------------------------------------
--+----------------+
接收内容：hello,|
2022-08-10 21:53:50.706 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0x96ae16a6,
L:/127.0.0.1:8081 - R:/127.0.0.1:63971] READ: 8B
         +-------------------------------------------------+
         | 0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f |
+--------+-------------------------------------------------
--+----------------+
|00000000| 74 68 69 73 20 69 73 7c                         |this is|        |
+--------+-------------------------------------------------
--+----------------+
接收内容：this is|
2022-08-10 21:53:50.707 [nioEventLoopGroup-3-1] DEBUG
io.netty.handler.logging.LoggingHandler - [id: 0x96ae16a6,
L:/127.0.0.1:8081 - R:/127.0.0.1:63971] EXCEPTION:
io.netty.handler.codec.TooLongFrameException: frame length exceeds
10: 12 - discarded
io.netty.handler.codec.TooLongFrameException: frame length exceeds
10: 12 - discarded
```

LengthFieldBasedFrameDecoder

LengthFieldBasedFrameDecoder 是长度域解码器，它相比前面三个解码器来

说复杂多了，当然功能也更加强大了，它是 Netty 中最常用解决拆包/粘包的解码器了。

要掌握 LengthFieldBasedFrameDecoder 必须要理解它的 4 个属性：

* **lengthFieldOffset**：长度字段的偏移量，也就是存放长度数据的起始位置，即接收的字节数组中下标为 lengthFieldOffset 的地方就是长度域的开始地方。
* **lengthFieldLength**：长度字段所占用的字节数。即接收的字节数组中 `bytes[lengthFieldOffset,(lengthFieldOffset + lengthFieldLength)]` 就是长度字段。
* **lengthAdjustment**：消息修正值。在一些复杂的场景中，长度域不仅仅只是单存地包括消息，还包括了版本号、属性类型、数据状态等等，这个时候我们就需要利用 lengthAdjustment 来进行修正。
* **initialBytesToStrip**：解码后需要跳过的字节数，也就是我们消息内容的起始位置。

这 4 个属性说实在话，这样硬看还是很难理解的，Netty LengthFieldBasedFrameDecoder 注释中一共给了 7 种场景，描述的非常详细了，我们把这 7 中场景明白了，就真正理解了这 4 个属性的含义以及如何使用 LengthFieldBasedFrameDecoder 了。

场景 一：最基本消息长度 + 消息内容

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

+--------+----------------+      +--------+----------------+
| Length | Actual Content |----->| Length | Actual Content |
| 0x000C | "HELLO, WORLD" |      | 0x000C | "HELLO, WORLD" |
+--------+----------------+      +--------+----------------+
```

这个场景是最简单，最基本的，因为解码前后，报文内容没有任何变化，报文只包含消息长度 Length 和消息内容 Actual Content。Length 为 16 进制表示，占 2 Byte，Length 内容为 0x000C = 12，表示 Actual Content 内容占用 12 Byte。所以对应 LengthFieldBasedFrameDecoder 参数为：

* lengthFieldOffset = 0，因为 Length 字段在报文开始的位置。
* lengthFieldLength = 2，0x000C 内容占用 2 B。

* lengthAdjustment = 0，Length 字段只包含了 Actual Content 长度，不需要做任何修正。
* initialBytesToStrip = 0，解码后的内容依然为 Length + Actual Content，不需要跳过任何字节，为 0。

场景二：解码后内容只保留 Conteng

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

+--------+----------------+      +----------------+
| Length | Actual Content |----->| Actual Content |
| 0x000C | "HELLO, WORLD" |      | "HELLO, WORLD" |
+--------+----------------+      +----------------+
```

相比场景一，场景二解码后的内容只保留了 Actual Content，其他部分保持不变，所以对应参数如下：

* lengthFieldOffset = 0，因为 Length 字段在报文开始的位置。
* lengthFieldLength = 2，0x000C 内容占用 2 B。
* lengthAdjustment = 0，Length 字段只包含了 Actual Content 长度，不需要做任何修正。
* initialBytesToStrip = 2，解码后的内容只有 Actual Content，需要跳过 Length，所以为 2

场景三：长度字段包含 Length + Content 的长度

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

+--------+----------------+      +--------+----------------+
| Length | Actual Content |----->| Length | Actual Content |
| 0x000E | "HELLO, WORLD" |      | 0x000E | "HELLO, WORLD" |
+--------+----------------+      +--------+----------------+
```

场景三与场景一的区别在于 Length 为 0x000E（14），它包含了长度字段 Length 所占用的字节和 Actual Content 所占用的字节。所以如果我们要得到

Actual Content 的长度就必须要减去 Length 所占用的字节，参数如下：

* lengthFieldOffset = 0，因为 Length 字段在报文开始的位置。
* lengthFieldLength = 2，0x000E 内容占用 2 B。
* lengthAdjustment = –2，长度字段 0x000E 为 14 ，需要减去长度字段 Length 所占用的字节才能得到 Content 的长度。
* initialBytesToStrip = 0，解码后的内容依然为 Length + Actual Content，不需要跳过任何字节，为 0。

场景四：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 Content |
|  0xCAFE  | 0x00000C | "HELLO, WORLD" |      |  0xCAFE  | 0x00000C | "HELLO, WORLD" |
+----------+----------+----------------+      +----------+----------+----------------+
```

场景四相比前面三个场景它多了一个 Header 1部分，该部分内容 `0xCAFE` 占用 2 个字节，Length 内容为 `0x00000C` 占用三个字节，值为 12 ，所以参数如下：

* lengthFieldOffset = 2，Length 的 offset 不再是起始位置了，它需要跳过 Header 1 所占用的 2 字节，所以为 2。
* lengthFieldLength = 3，0x00000C 占用 3 字节。
* lengthAdjustment = 0，Length 字段只包含有 Content 的长度，不需要做调整，为 0。
* initialBytesToStrip = 0，解码后的内容为完整报文，不需要跳过任何字节，为 0。

场景五：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 Content |
| 0x00000C |  0xCAFE  | "HELLO, WORLD" |      | 0x00000C |  0xCAFE  | "HELLO, WORLD" |
+----------+----------+----------------+      +----------+----------+----------------+
```

场景五与场景四类似，只不过它的 Length 和 Actual Content 不再相邻，这个时候如果我们要取 Actual Content 的内容就要略过 Header 1 字段，参数如下：

* lengthFieldOffset = 0，Length 在报文开始位置。
* lengthFieldLength = 3，0x00000C 占用 3 字节。
* lengthAdjustment = 2，Header 1 + Actual Content 为 14 字节，但是 Length 内容为 12，所以需要加上 lengthAdjustment（2）才能得到 Header 1 + Actual Content 的内容。
* initialBytesToStrip = 0，解码后的内容为完整报文，不需要跳过任何字节，为 0。

场景六：**基于长度偏移和长度修正的解码**

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

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

场景六相比场景五而言，多了一个 HDR1 部分，占用 1 字节，且解码后的内容丢弃了 HDR1 和 Length，所以参数如下：

* lengthFieldOffset = 1，需要跳过 HDR1部分，占用 1 字节，所以为 1。
* lengthFieldLength = 2，0x000C 占用 2 字节。
* lengthAdjustment = 1，HDR2 + Actual Content 为 13 字节，所以 Length 字段值（12）需要加上 lengthAdjustment（1）才能得到 HDR2 + Actual Content 的内容。
* initialBytesToStrip = 3，解码后的内容跳过了 HDR1 + Length，占用 3 字节，所以为 3。

场景七：**长度字段包含除 Content 外的多个其他字段**

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

+------+--------+------+----------------+      +------+----------------+
| HDR1 | Length | HDR2 | Actual Content |----->| HDR2 | Actual Content |
| 0xCA | 0x0010 | 0xFE | "HELLO, WORLD" |      | 0xFE | "HELLO, WORLD" |
+------+--------+------+----------------+      +------+----------------+
```

场景七与场景六的区别在于 Length 字段记录了整个消息报文的长度，所以如果要得到 HDR2 + Actual Content 的内容，我们需要通过 lengthAdjustment 来调整，所以参数如下：

* lengthFieldOffset = 1，需要跳过 HDR1部分，占用 1 字节，所以为 1。
* lengthFieldLength = 2，0x000C 占用 2 字节。
* lengthAdjustment = -3，HDR2 + Actual Content 为 13 字节，但是 Length 字段值为 16 ，需要减去 HDR1 + Length 的字节数（3），才能得到 HDR2 + Actual Content （13）。
* initialBytesToStrip = 3，解码后的内容跳过了 HDR1 + Length，占用 3 字节，所以为 3。

对于 LengthFieldBasedFrameDecoder 而言，上面 7 中场景已经涵盖了大部分的使用场景了，所以如果对那四个属性的含义还不是很理解的话，多看几遍吧。