实战：拆包粘包理论与解决方案

> 本小节我们来学习一下 Netty 里面拆包和粘包的概念，并且如何选择适合我们应用程序的拆包器

在开始本小节之前，我们首先来看一个例子，本小节的例子我们选择[客户端与服务端双向通信(https://juejin.im/book/5b4bc28bf265da0f60130116/sectior](https://juejin.im/book/5b4bc28bf265da0f60130116/sectior)这小节的代码，然后做适当修改

拆包粘包例子

> 客户端 FirstClientHandler

```
public class FirstClientHandler extends
ChannelInboundHandlerAdapter {
    @Override
    public void
channelActive(ChannelHandlerContext ctx) {
        for (int i = 0; i < 1000; i++) {
            ByteBuf buffer = getByteBuf(ctx);
            ctx.channel().writeAndFlush(buffer);
        }
    }

    private ByteBuf
getByteBuf(ChannelHandlerContext ctx) {
        byte[] bytes = "你好，欢迎关注我的微信公众号，
《闪电侠的博客》!".getBytes(Charset.forName("utf-
8"));
        ByteBuf buffer = ctx.alloc().buffer();
        buffer.writeBytes(bytes);

        return buffer;
    }
}
```

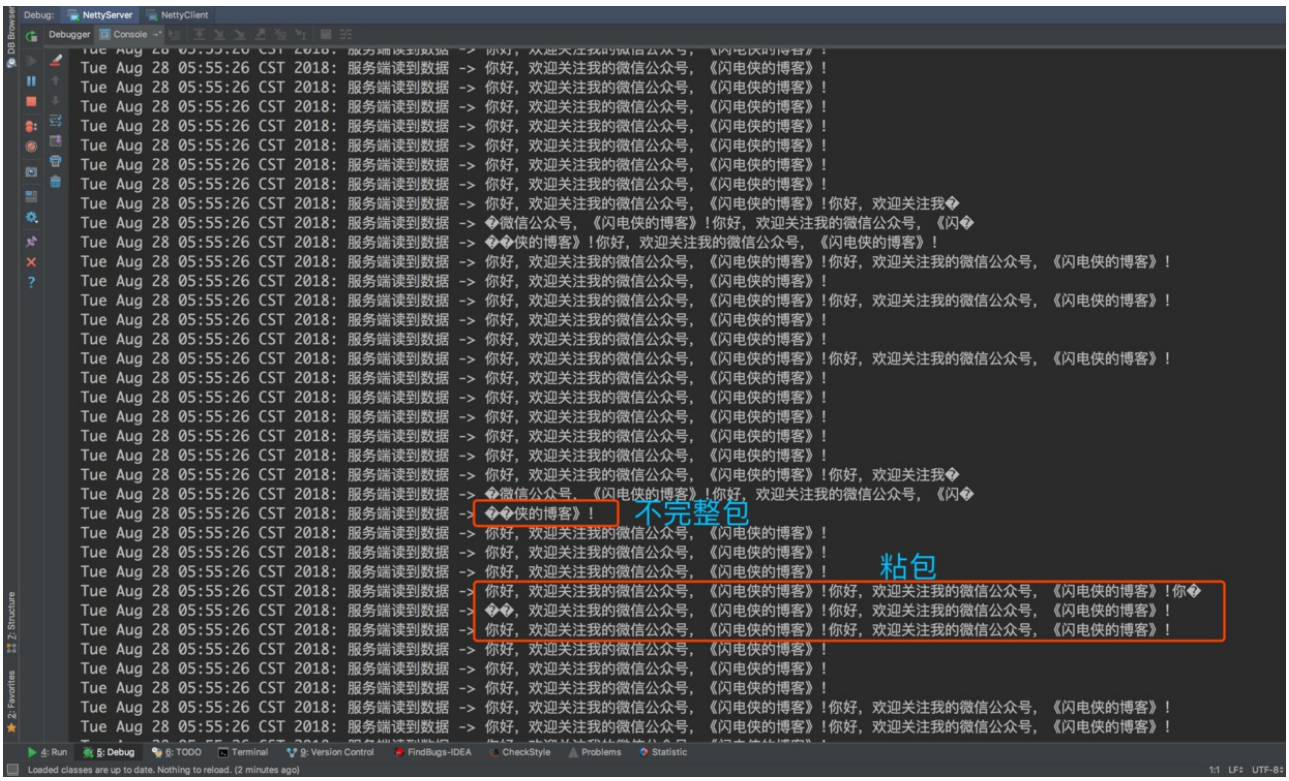
客户端在连接建立成功之后，使用一个 for 循环，不断向服务端写一串数据

服务端 FirstServerHandler

```
public class FirstServerHandler extends
ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext
ctx, Object msg) {
        ByteBuf byteBuf = (ByteBuf) msg;

        System.out.println(new Date() + ": 服务端读
到数据 -> " +
byteBuf.toString(Charset.forName("utf-8")));
    }
}
```

服务端收到数据之后，仅仅把数据打印出来，读者可以花几分钟时间思考一下，服务端的输出会是什么样子的?

可能很多读者觉得服务端会输出 1000 次 “你好，欢迎关注我的微信公众号，《闪电侠的博客》!”，然而实际上服务端却是如下输出：



从服务端的控制台输出可以看出，存在三种类型的输出

1. 一种是正常的字符串输出。
2. 一种是多个字符串“粘”在了一起，我们定义这种 ByteBuf 为粘包。
3. 一种是一个字符串被“拆”开，形成一个破碎的包，我们定义这种 ByteBuf 为半包。

为什么会有粘包半包现象?

我们需要知道，尽管我们在应用层面使用了 Netty，但是对于操作系统来说，只认 TCP 协议，尽管我们的应用层是按照 ByteBuf 为 单位来发送数据，但是到了底层操作系统仍然是按照字节流发送数据，因此，数据到了服务端，也是按照字节流的方式读入，然后到了 Netty 应用层面，重新拼装成 ByteBuf，而这里的 ByteBuf 与客户端按顺序发送的 ByteBuf 可能是不对等的。因此，我们需要在客户端根据自定义协议来组装我们应用层的数据包，然后在服务端根据我们的应用层的协议来组装数据包，这个过程通常在服务端称为拆包，而在客户端称为粘包。

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。

拆包的原理

在没有 Netty 的情况下，用户如果自己需要拆包，基本原理就是不断从 TCP 缓冲区中读取数据，每次读取完都需要判断是否是一个完整的数据包

1. 如果当前读取的数据不足以拼接成一个完整的业务数据包，那就保留该数据，继续从 TCP 缓冲区中读取，直到得到一个完整

的数据包。
2. 如果当前读到的数据加上已经读取的数据足够拼接成一个数据包，那就将已经读取的数据拼接上本次读取的数据，构成一个完整的业务数据包传递到业务逻辑，多余的数据仍然保留，以便和下次读到的数据尝试拼接。

如果我们自己实现拆包，这个过程将会非常麻烦，我们的每一种自定义协议，都需要自己实现，还需要考虑各种异常，而 Netty 自带的一些开箱即用的拆包器已经完全满足我们的需求了，下面我们来介绍一下 Netty 有哪些自带的拆包器。

Netty 自带的拆包器

1. 固定长度的拆包器 FixedLengthFrameDecoder

如果你的应用层协议非常简单，每个数据包的长度都是固定的，比如 100，那么只需要把这个拆包器加到 pipeline 中，Netty 会把一个个长度为 100 的数据包 (ByteBuf) 传递到下一个 channelHandler。

2. 行拆包器 LineBasedFrameDecoder

从字面意思来看，发送端发送数据包的时候，每个数据包之间以换行符作为分隔，接收端通过 LineBasedFrameDecoder 将粘过的 ByteBuf 拆分成一个个完整的应用层数据包。

3. 分隔符拆包器 DelimiterBasedFrameDecoder

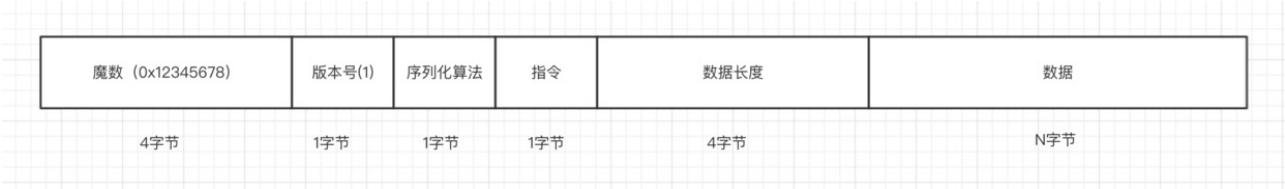
DelimiterBasedFrameDecoder 是行拆包器的通用版本，只不过我们可以自定义分隔符。

4. 基于长度域拆包器 LengthFieldBasedFrameDecoder

最后一种拆包器是最通用的一种拆包器，只要你的自定义协议中包含长度域字段，均可以使用这个拆包器来实现应用层拆包。由于上面三种拆包器比较简单，读者可以自行写出 demo，接下来，我们就结合我们小册的自定义协议，来学习一下如何使用基于长度域的拆包器来拆解我们的数据包。

如何使用 LengthFieldBasedFrameDecoder

首先，我们来回顾一下我们的自定义协议



详细的协议分析参考 客户端与服务端通信协议编解码 (https://juejin.im/book/5b4bc28bf265da0f60130116/sectior 这小节，这里不再赘述。
关于拆包，我们只需要关注

1. 在我们的自定义协议中，我们的长度域在整个数据包的哪个地方，专业术语来说就是长度域相对整个数据包的偏移量是多少，这里显然是 4+1+1+1=7。
2. 另外需要关注的就是，我们长度域的长度是多少，这里显然是 4。
   有了长度域偏移量和长度域的长度，我们就可以构造一个拆包器。

```
new LengthFieldBasedFrameDecoder(Integer.MAX_VALUE, 7, 4);
```

其中，第一个参数指的是数据包的最大长度，第二个参数指的是长度域的偏移量，第三个参数指的是长度域的长度，这样一个拆包器写好之后，只需要在 pipeline 的最前面加上这个拆包器。

由于这类拆包器使用最为广泛，想深入学习的读者可以参考我的这篇文章 [netty源码分析之LengthFieldBasedFrameDecoder (https://www.jianshu.com/p/a0a51fd79f62)](https://www.jianshu.com/p/a0a51fd79f62)

服务端

```
ch.pipeline().addLast(new
LengthFieldBasedFrameDecoder(Integer.MAX_VALUE,
7, 4));
ch.pipeline().addLast(new PacketDecoder());
ch.pipeline().addLast(new LoginRequestHandler());
ch.pipeline().addLast(new
MessageRequestHandler());
ch.pipeline().addLast(new PacketEncoder());
```

客户端

```
ch.pipeline().addLast(new
LengthFieldBasedFrameDecoder(Integer.MAX_VALUE,
7, 4));
ch.pipeline().addLast(new PacketDecoder());
ch.pipeline().addLast(new
LoginResponseHandler());
ch.pipeline().addLast(new
MessageResponseHandler());
ch.pipeline().addLast(new PacketEncoder());
```

这样，在后续 PacketDecoder 进行 decode 操作的时候，ByteBuf 就是一个完整的自定义协议数据包。

LengthFieldBasedFrameDecoder 有很多重载的构造参数，由于篇幅原因，这里不再展开， 但是没关系，关于 LengthFieldBasedFrameDecoder 的详细使用可参考[我的简书](https://www.jianshu.com/p/a0a51fd79f62)，对原理感兴趣的同学可以参考[我的视频](https://coding.imooc.com/class/230.html)，了解了详细的使用方法之后，就可以有针对性地根据你的自定义协议来构造 LengthFieldBasedFrameDecoder。

拒绝非本协议连接

不知道大家还记不记得，我们在设计协议的时候为什么在数据包的开头加上一个魔数，遗忘的同学可以参考[客户端与服务端通信协议编解码](https://juejin.im/book/5b4bc28bf265da0f60130116/sectior
回顾一下。我们设计魔数的原因是为了尽早屏蔽非本协议的客户端，通常在第一个 handler 处理这段逻辑。我们接下来的做法是每个客户端发过来的数据包都做一次快速判断，判断当前发来的数据包是否是满足我的自定义协议，
我们只需要继承自 LengthFieldBasedFrameDecoder 的 decode() 方法，然后在 decode 之前判断前四个字节是否是等于我们定义的魔数 0x12345678

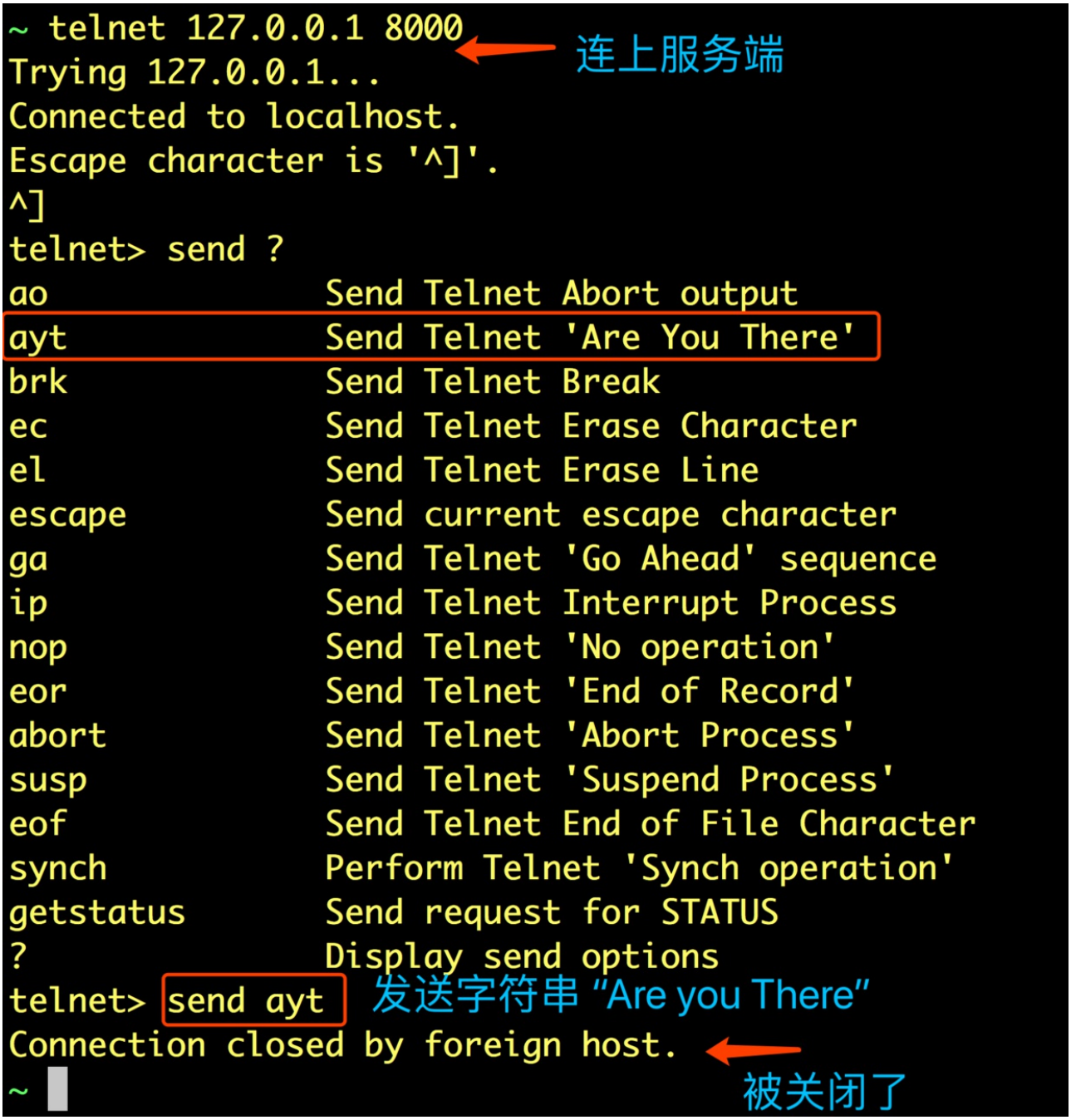
```
public class Spliter extends
LengthFieldBasedFrameDecoder {
    private static final int LENGTH_FIELD_OFFSET
= 7;
    private static final int LENGTH_FIELD_LENGTH
= 4;

    public Spliter() {
        super(Integer.MAX_VALUE,
LENGTH_FIELD_OFFSET, LENGTH_FIELD_LENGTH);
    }

    @Override
    protected Object decode(ChannelHandlerContext
ctx, ByteBuf in) throws Exception {
        // 屏蔽非本协议的客户端
        if (in.getInt(in.readerIndex()) !=
PacketCodeC.MAGIC_NUMBER) {
            ctx.channel().close();
            return null;
        }

        return super.decode(ctx, in);
    }
}
```

为什么可以在 decode() 方法写这段逻辑？是因为这里的 decode() 方法中，第二个参数 in，每次传递进来的时候，均为一个数据包的开头，想了解原理的同学可以参考 [netty 源码分析之拆包器的奥秘 (https://www.jianshu.com/p/dc26e944da95)](https://www.jianshu.com/p/dc26e944da95)。

最后，我们只需要替换一下如下代码即可

```
//ch.pipeline().addLast(new
LengthFieldBasedFrameDecoder(Integer.MAX_VALUE,
7, 4));
// 替换为
ch.pipeline().addLast(new Spliter());
```

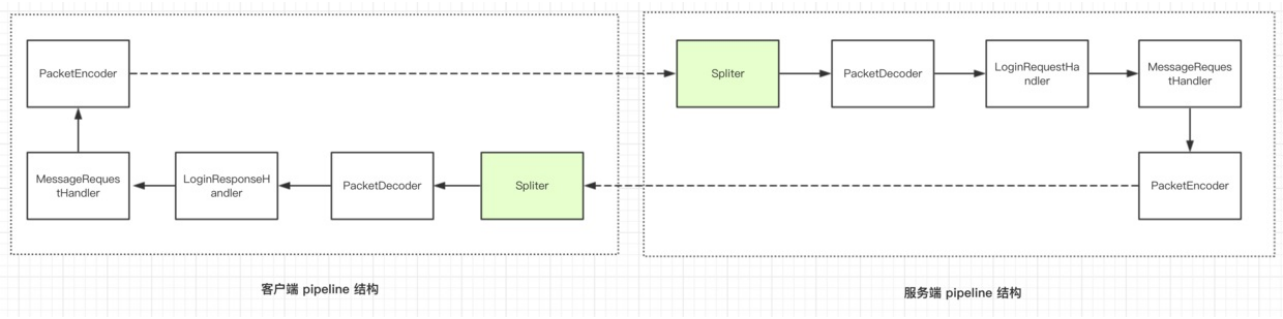
然后，我们再来实验一下



可以看到，我们使用 telnet 连接上服务端之后（与服务端建立了连接），向服务端发送一段字符串，由于这段字符串是不符合我们的自定义协议的，于是在第一时间，我们的服务端就关闭了这条连接。

服务端和客户端的 pipeline 结构

至此，我们服务端和客户端的 pipeline 结构为



最后，我们对本小节内容做一下总结

总结

1. 我们通过一个例子来理解为什么要有拆包器，说白了，拆包器的作用就是根据我们的自定义协议，把数据拼装成一个个符合我们自定义数据包大小的 ByteBuf，然后送到我们的自定义协议解码器去解码。
2. Netty 自带的拆包器包括基于固定长度的拆包器，基于换行符和自定义分隔符的拆包器，还有另外一种最重要的基于长度域的拆包器。通常 Netty 自带的拆包器已完全满足我们的需求，无需重复造轮子。
3. 基于 Netty 自带的拆包器，我们可以在拆包之前判断当前连上来的客户端是否是支持自定义协议的客户端，如果不支持，可尽早关闭，节省资源。

本小节完整代码在 github (https://github.com/lightningMan/flash-netty/tree/%E6%8B%86%E5%8C%85%E7%B2%98%E5%8C%8) 对应本小节的分支。

思考

在我们的 IM 这个 完整的 pipeline 中，如果我们不添加拆包器，客户端连续向服务端发送数据，会有什么现象发生？为什么会发生这种现象?
欢迎留言讨论。