# Java NIO 框架 Netty 之美：粘包与半包问题

## 一、前言

Netty 是一个可以快速开发网络应用程序的 NIO 框架，它大大简化了 TCP 或者 UDP 服务器的网络编程。Netty 的简易和快速开发并不意味着由它开发的程序将失去可维护性或者存在性能问题，它的设计参考了许多协议的实现，比如 FTP、SMTP、HTTP 和各种二进制和基于文本的传统协议，因此 Netty 成功的实现了兼顾快速开发、性能、稳定性、灵活性为一体，不需要为了考虑一方面原因而妥协其他方面。Netty 的应用还是比较广泛的，比如阿里巴巴开源的 Dubbo 和 Sofa-Bolt 框架底层网络通讯都是基于 Netty 来实现的。

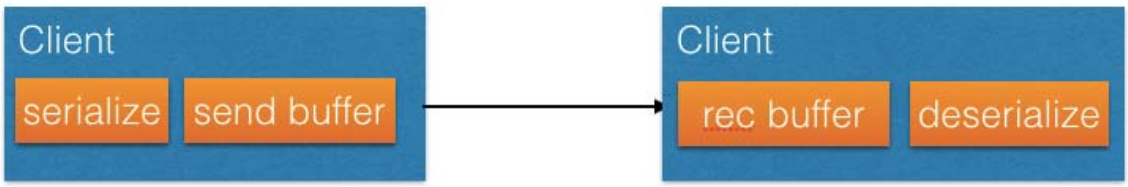
本 Chat 主要讲解 Netty 中的粘包与半包问题，主要包含下面内容：

- 什么是粘包与半包问题，为何会出现，如何避免？
- 如何使用包定长 FixedLengthFrameDecoder 解决粘包与半包问题？原理是什么？
- 如何使用包分隔符 DelimiterBasedFrameDecoder 解决粘包与半包问题？原理是什么？
- Dubbo 在使用 Netty 作为网络通讯时候是如何避免粘包与半包问题？
- Netty 框架本身存在粘包半包问题？什么时候需要考虑粘包与半包问题？



## 二、粘包与半包问题

大家都知道在客户端与服务端进行网络通信时候，客户端会通过socket 把需要发送的内容进行序列化为二进制流后发送出去，然后二进制流通过网络流向服务器端，服务端接受到该请求后会解析该请求包，然后反序列化后对请求进行处理。这看似是一个很简单的过程，但是细细想来却会发现没有那么简单。




如上图首先在客户端发送数据时，实际是把数据写入到了 TCP 发送缓存里面的，如果发送的包的大小比 TCP 发送缓存的容量大，那么这个数据包就会被分成多个包，通过 socket 多次发送到服务端。而服务端获取数据是从接受缓存里面获取的，假设服务端第

一次从接受缓存里面获取的数据是整个包的一部分，这时候就产生了半包现象，半包不是说只收到了全包的一半，是说收到了全包的一部分。

服务器读取到半包数据后，会对读取的二进制流进行解析，一般的会把二进制流反序列化为对象，这里服务器由于只读取了客户端序列化对象后的一部分，所以反序列会报错。

同理如果发送的数据包大小比 TCP 发送缓存容量小，并且假设 TCP 缓存可以存放多个包，那么客户端和服务端的一次通信就可能传递了多个包，这时候服务端从接受缓存就可能一下读取了多个包，这时候就出现了粘包现象，由于服务端从接受缓存获取的二进制流是多个对象转换来的，所以在后续的反序列化时候肯定也会出错。

其实出现粘包和半包的原因是 TCP 层不知道上层业务的包的概念，它只是简单的传递流，所以需要上层应用层协议来识别读取的数据是不是一个完整的包。

**那么如何避免粘包与半包的出现?**

- 比较常见方案是在应用层设计协议时候把协议包分为 header 和 body 两部分，header 里面记录 body 长度。当服务端从接受缓冲区读取数据后，如果发现数据大小小于包的长度则说明出现了半包，这时候就回退读取缓存的指针，等待下次读事件到来的时候再次测试。如果发现包长度大于了包长度则看如果长度是包大小整数倍则说明了出现了粘包，则循环读取多个包，否者就是出现了多个整包+半包，这时候读取整数个包然后回退半包的指针。如下图是 Dubbo 协议帧个数：




其中 header 格式如下：

header格式

| Bit offset | 0-7 | 8-15 | 16-23 | 24-31 | 32-95 | 96-127 |
|---|---|---|---|---|---|---|
| 0 | magic high | magic low | request and serialization flag | response status | request id | body length |

可知 header 记录 body 长度。

- 还有一种方式是在多个包之间添加分隔符，使用分隔符来判断一个包的结束


可知这种方式时候每个包大小可以不固定，当服务器端读取时候每次遇到分隔符就知道当前包结束了。

- 还有一种是包定长，就是每个包大小固定长度。




可知这种方式每个包的大小必须一致。

# 三、Netty 与粘包半包

首先我们来看一个简单的 Netty 搭建的客户与服务端通信的例子，这个例子比较简单，需要详细讲解的童鞋可以移步：

> http://gitbook.cn/gitchat/activity/5b01714ca0810c23901c55ac

## 3.1 客户端代码

```
public final class NettyClient {

    static final String HOST = System.getProperty("host",
"127.0.0.1");
    static final int PORT =
Integer.parseInt(System.getProperty("port", "8007"));

    public static void main(String[] args) throws Exception {
```

```
        //1.1 创建Reactor线程池
        EventLoopGroup group = new NioEventLoopGroup();
        try {//1.2 创建启动类Bootstrap实例，用来设置客户端相关参数
            Bootstrap b = new Bootstrap();
            b.group(group)//1.2.1设置线程池
            .channel(NioSocketChannel.class)//1.2.2指定用于创建客户
端NIO通道的Class对象
            .option(ChannelOption.TCP_NODELAY, true)//1.2.3设置客
户端套接字参数
                    .handler(new
ChannelInitializer<SocketChannel>() {//1.2.4设置用户自定义handler
                        @Override
                        public void initChannel(SocketChannel ch)
throws Exception {
                            ChannelPipeline p = ch.pipeline();

                            p.addLast(new NettyClientHandler());

                        }
                    });

            // 1.3启动链接
            ChannelFuture f = b.connect(HOST, PORT).sync();

            //1.4 同步等待链接断开
            f.channel().closeFuture().sync();
        } finally {
            // 1.5优雅关闭线程池
            group.shutdownGracefully();
        }
    }
}
```



其中自定义 NettyClientHandler 的代码如下：

```
public class NettyClientHandler extends
ChannelInboundHandlerAdapter {

    private final byte[] request;

    private AtomicInteger atomicInteger = new AtomicInteger(0);

    /**
     * 创建一个客户端 handler.
     */
    public NettyClientHandler() {
        request = "hello server,im a client".getBytes();
    }
```

```
    @Override
    public void channelActive(ChannelHandlerContext ctx) {

        System.out.println("--- client already connected----");

        ByteBuf message = null;
        for (int i = 0; i < 1; ++i) {
            message = Unpooled.buffer(request.length);
            message.writeBytes(request);
            ctx.writeAndFlush(message);
        }
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object
msg) {
        ByteBuf message = (ByteBuf) msg;
        byte[] response = new byte[message.readableBytes()];
        message.readBytes(response);

        System.out.println(atomicInteger.getAndIncrement() +
"receive from server:" + new String(response));
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) {
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```



这个 handler 是当客户端连接到服务端后给服务端发送一个 `hello server,im a client` 的二进制流。并接受服务端写回的数据。

## 3.2 服务端代码

```
public final class NettyServer {

    static final int PORT =
Integer.parseInt(System.getProperty("port", "8007"));

    public static void main(String[] args) throws Exception {
        //（1.1）创建主从Reactor线程池
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
```

```
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            //1.2创建启动类ServerBootstrap实例，用来设置客户端相关参数
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)//1.2.1设置主从线程池组
             .channel(NioServerSocketChannel.class)//1.2.2指定用于
创建客户端NIO通道的Class对象
             .option(ChannelOption.SO_BACKLOG, 100)//1.2.3设置客户
端套接字参数
             .handler(new LoggingHandler(LogLevel.INFO))//1.2.4设
置日志handler
             .childHandler(new ChannelInitializer<SocketChannel>
() {//1.2.5设置用户自定义handler
                 @Override
                 public void initChannel(SocketChannel ch) throws
Exception {
                     ChannelPipeline p = ch.pipeline();

                     //p.addLast(new
LoggingHandler(LogLevel.INFO));
                     p.addLast(new NettyServerHandler());
                 }
             });

            //1.3 启动服务器
            ChannelFuture f = b.bind(PORT).sync();
            System.out.println("----Server Started----");

            //1.4 同步等待服务socket关闭
            f.channel().closeFuture().sync();
        } finally {
            // 1.5优雅关闭线程池组
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```



其中用户自定义的 NettyServerHandler 对应的代码如下：

```
public class NettyServerHandler extends
ChannelInboundHandlerAdapter {
    private AtomicInteger atomicInteger = new AtomicInteger(0);

    // 2.1
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object
msg) {
    ByteBuf message = (ByteBuf) msg;
        byte[] response = new byte[message.readableBytes()];
```

```
            message.readBytes(response);
            System.out.println(atomicInteger.getAndIncrement() +
"receive client info: " + new String(response));

            String sendContent = "hello client ,im server";
            ByteBuf seneMsg = Unpooled.buffer(sendContent.length());
            seneMsg.writeBytes(sendContent.getBytes());

            ctx.writeAndFlush(Unpooled.copiedBuffer(seneMsg));
            System.out.println("send info to client:" + sendContent);

        }

        // 2.2
        @Override
        public void channelActive(ChannelHandlerContext ctx) throws
Exception {
            System.out.println("--- accepted client---");
            ctx.fireChannelActive();
        }

        @Override
        public void channelReadComplete(ChannelHandlerContext ctx) {
        }

        @Override
        public void exceptionCaught(ChannelHandlerContext ctx,
Throwable cause) {
            cause.printStackTrace();
            ctx.close();
        }
}
```



这个 handler 是读取客户端发来的 `hello server,im a client` 的二进制流并转化为字符串打印输出，并向客户端写入 `hello client ,im server` 。

## 3.3 试验

### 3.3.1 试验一

首先启动服务端，然后启动客户端，然后输出结果如下：

服务端结果：

```
----Server Started----
--- accepted client---
0receive client info: hello server,im a client
send info to client:hello client ,im server
```

客户端结果：

```
--- client already connected----
0receive from server:hello server,im a client
```

这个结果在意料之中。

#### 3.3.2 试验二

修改 NettyClientHandler 的 channelActive 中循环为 10，然后重新启动服务器和客户端，结果如下：

服务器结果：

```
----Server Started----
--- accepted client---
0receive client info: hello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a clienthello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a client
send info to client:hello client ,im server
```



客户端结果：

```
--- client already connected----
0receive from server:hello client ,im server
```

从程序代码我们希望服务器端输出 10 行：

```
0receive client info: hello server,im a client
1receive client info: hello server,im a client
2receive client info: hello server,im a client
3receive client info: hello server,im a client
4receive client info: hello server,im a client
5receive client info: hello server,im a client
6receive client info: hello server,im a client
7receive client info: hello server,im a client
8receive client info: hello server,im a client
9receive client info: hello server,im a client
```

但是结果却如上面只输出了一行，这就是出现了粘包现象。

然后读者可以修改修改 NettyClientHandler的channelActive 中循环为更大的数，比如 50，然后重新启动服务器和客户端，多运行几次会出现下面结果：

客户端结果：

```
--- client already connected----
0receive from server:hello client ,im serverhello client ,im
server
```

服务端结果：

```
--- accepted client---
0receive client info: hello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a clienthello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a clienthello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a clienthello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a clienthello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a clienthello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a clienthello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a clienthello server,im a clienthello server,im a
clienthello server,im
send info to client:hello client ,im server

1receive client info: a clienthello server,im a clienthello
server,im a clienthello server,im a clienthello server,im a
clienthello server,im a clienthello server,im a clienthello
server,im a client

send info to client:hello client ,im server
```

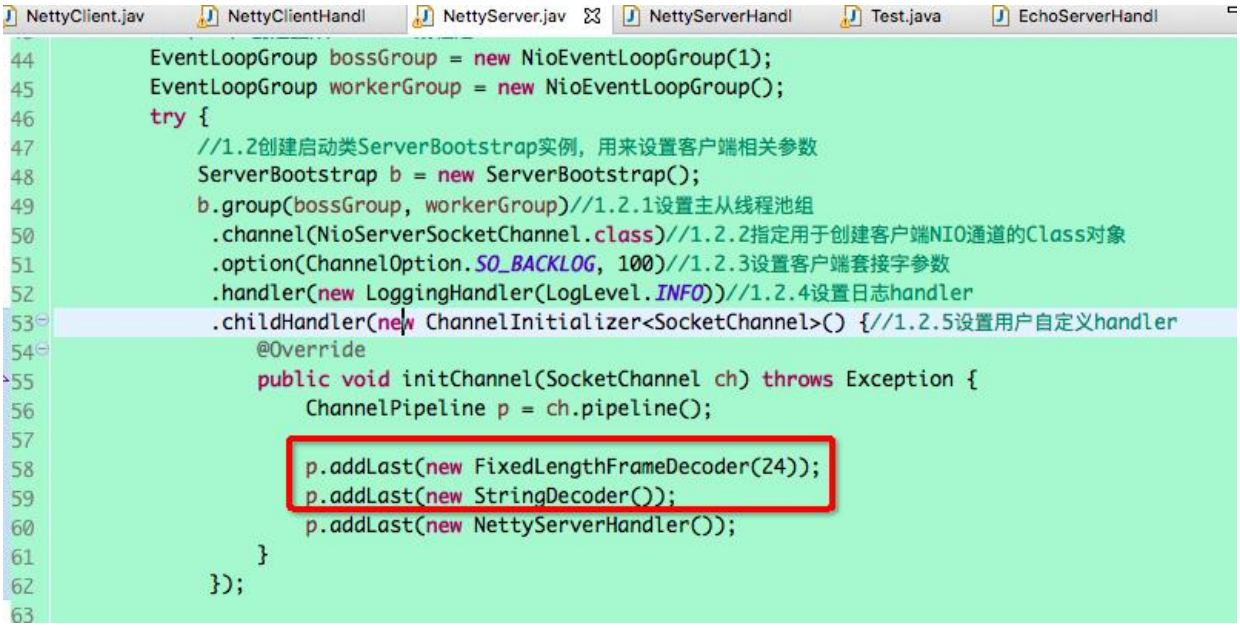
可知出现了粘包和半包。

# 四、使用包定长 FixedLengthFrameDecoder 解决半包粘包

## 4.1 试验

由于客户端发给服务器端的是 `hello server,im a client` 字符串，该字符串占用 24 字节，所以在服务器端 channelpipeline 里面添加一个长度为 24 的定长解码器和二进制转

换为 string 的解码器：



```
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            //1.2创建启动类ServerBootstrap实例，用来设置客户端相关参数
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)//1.2.1设置主从线程池组
             .channel(NioServerSocketChannel.class)//1.2.2指定用于创建客户端NIO通道的Class对象
             .option(ChannelOption.SO_BACKLOG, 100)//1.2.3设置客户端套接字参数
             .handler(new LoggingHandler(LogLevel.INFO))//1.2.4设置日志handler
             .childHandler(new ChannelInitializer<SocketChannel>() {//1.2.5设置用户自定义handler
                 @Override
                 public void initChannel(SocketChannel ch) throws Exception {
                     ChannelPipeline p = ch.pipeline();

                     p.addLast(new FixedLengthFrameDecoder(24));
                     p.addLast(new StringDecoder());
                     p.addLast(new NettyServerHandler());
                 }
             });
```

然后修改 NettyServerHandler 的 channelRead 如下：

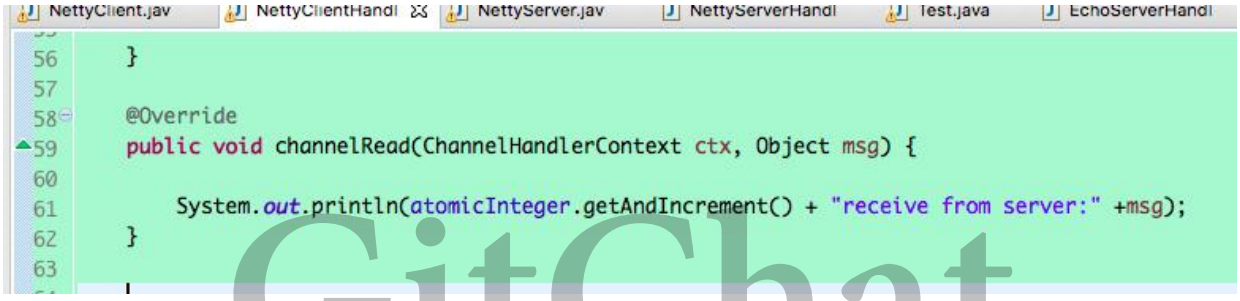
```
@Sharable
public class NettyServerHandler extends ChannelInboundHandlerAdapter {
    private AtomicInteger atomicInteger = new AtomicInteger(0);

    // 2.1
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {

        System.out.println(atomicInteger.getAndIncrement() + "receive client info: " + msg);

        String sendContent = "hello client ,im server";
        ByteBuf seneMsg = Unpooled.buffer(sendContent.length());
        seneMsg.writeBytes(sendContent.getBytes());

        ctx.writeAndFlush(Unpooled.copiedBuffer(seneMsg));
        System.out.println("send info to client:" + sendContent);

    }
```



由于服务器发给客户端的是 `hello client ,im server` 字符串，该字符串占用 23 字节，所以在客户端端 channelpipeline 里面添加一个长度为 23 的定长解码器和二进制转换为 string 的解码器：

```
47
48          // 1.1 创建Reactor线程池
49          EventLoopGroup group = new NioEventLoopGroup();
50          try {// 1.2 创建启动类Bootstrap实例，用来设置客户端相关参数
51              Bootstrap b = new Bootstrap();
52              b.group(group)// 1.2.1设置线程池
53                      .channel(NioSocketChannel.class)// 1.2.2指定用于创建客户端NIO通道的Class对象
54                      .option(ChannelOption.TCP_NODELAY, true)// 1.2.3设置客户端套接字参数
55                      .handler(new ChannelInitializer<SocketChannel>() {// 1.2.4设置用户自定义handler
56                          @Override
57                          public void initChannel(SocketChannel ch) throws Exception {
58                              ChannelPipeline p = ch.pipeline();
59                              p.addLast(new FixedLengthFrameDecoder(23));
60                              p.addLast(new StringDecoder());
61                              p.addLast(new NettyClientHandler());
62
63                          }
64                      });
65
66              // 1.3启动链接
67              ChannelFuture f = b.connect(HOST, PORT).sync();
```

然后修改 NettyClientHandler 的 channelRead 如下：



```
56      }
57
58      @Override
59      public void channelRead(ChannelHandlerContext ctx, Object msg) {
60
61          System.out.println(atomicInteger.getAndIncrement() + "receive from server:" +msg);
62      }
63
```



然后重新启动服务器客户端，结果如下：

服务器端结果：

```
----Server Started----
--- accepted client---
0receive client info: hello server,im a client
send info to client:hello client ,im server
1receive client info: hello server,im a client
send info to client:hello client ,im server
2receive client info: hello server,im a client
send info to client:hello client ,im server
3receive client info: hello server,im a client
send info to client:hello client ,im server
4receive client info: hello server,im a client
send info to client:hello client ,im server
5receive client info: hello server,im a client
send info to client:hello client ,im server
6receive client info: hello server,im a client
send info to client:hello client ,im server
7receive client info: hello server,im a client
send info to client:hello client ,im server
8receive client info: hello server,im a client
send info to client:hello client ,im server
```

```
9receive client info: hello server,im a client
send info to client:hello client ,im server
```

客户端结果：

```
--- client already connected----
0receive from server:hello client ,im server
1receive from server:hello client ,im server
2receive from server:hello client ,im server
3receive from server:hello client ,im server
4receive from server:hello client ,im server
5receive from server:hello client ,im server
6receive from server:hello client ,im server
7receive from server:hello client ,im server
8receive from server:hello client ,im server
9receive from server:hello client ,im server
```

可知使用 FixedLengthFrameDecoder 已经解决了半包粘包问题。

## 4.2 FixedLengthFrameDecoder 的原理



顾名思义是使用包定长方式来解决粘包半包问题，假设服务端接受到下面四个包分片：

```
+---+----+------+----+
| A | BC | DEFG | HI |
+---+----+------+----+
```

那么使用 FixedLengthFrameDecoder(3) 会将接受 buffer 里面的上面数据解码为下面固定长度为 3 的 3 个包：

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

FixedLengthFrameDecoder 是继承自 ByteToMessageDecoder 类的，当服务器接受 buffer 数据就绪后，会调用 ByteToMessageDecoder 的 channelRead 方法进行读取，下面我们从这个函数开始讲解：

```
public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {
        //4.2.1
        if (msg instanceof ByteBuf) {
            CodecOutputList out = CodecOutputList.newInstance();
            try {
                ...
                //4.2.2
                callDecode(ctx, cumulation, out);
            } catch (DecoderException e) {
```

```
                throw e;
            } catch (Throwable t) {
                throw new DecoderException(t);
            } finally {
                ...
            }
        } else {
            //4.2.3
            ctx.fireChannelRead(msg);
        }
    }
```

如上代码 4.2.2 具体是方法 callDecode 进行数据读取的，其代码如下：

```
    protected void callDecode(ChannelHandlerContext ctx, ByteBuf in,
    List<Object> out) {
            try {
                //4.2.4
                while (in.isReadable()) {
                    int outSize = out.size();
                    //4.2.4.1
                    if (outSize > 0) {
                        fireChannelRead(ctx, out, outSize);
                        out.clear();
                        ...
                    }
                    //4.2.4.2
                    int oldInputLength = in.readableBytes();
                    decodeRemovalReentryProtection(ctx, in, out);

                    ...
                    //4.2.4.3
                    if (outSize == out.size()) {
                        if (oldInputLength == in.readableBytes()) {
                            break;
                        } else {
                            continue;
                        }
                    }
                    ...
                    //4.2.4.4
                    if (isSingleDecode()) {
                        break;
                    }
                }
            } catch (DecoderException e) {
                throw e;
            } catch (Throwable cause) {
                throw new DecoderException(cause);
```

```
        }
    }
```

如上代码 callDecode 中 4.2.4 是使用循环进行读取，这是因为可能出现粘包情况，使用循环可以逐个对单包进行处理。

其中 4.2.4.1 判断如果读取到了全包，则调用 fireChannelRead 激活 channelpipeline 里面的其它 handler 的 channelRead 方法，因为这里，FixedLengthFrameDecoder 只是 channelpipeline 中的一个 handler。

代码 4.2.4.2 的 decodeRemovalReentryProtection 方法作用是调用 FixedLengthFrameDecoder 的 decode 方法具体从接受 buffer 读取数据，后者代码如下：

```
    protected final void decode(ChannelHandlerContext ctx,
ByteBuf in, List<Object> out) throws Exception {
        Object decoded = decode(ctx, in);//4.2.6
        if (decoded != null) {
            out.add(decoded);
        }
    }
```



```
        protected Object decode(
            @SuppressWarnings("UnusedParameters")
ChannelHandlerContext ctx, ByteBuf in) throws Exception {
        if (in.readableBytes() < frameLength) {
            return null;
        } else {
            return in.readRetainedSlice(frameLength);
        }
    }
```

如上代码 4.2.6 如果发现接受 buffer 里面的字节数小于我们设置的固定长度 frameLength 则说明出现了半包情况，则直接返回 null；否者读取固定长度的字节数。

然后执行代码 4.2.4.3，其判断 outSize == out.size() 说明代码 4.2.6 没有读取一个包（说明出现了半包），则看当前 buffer 缓存的字节数是否变化了，如果没有变化则结束循环读取，如果变化了则可能之前的半包已经变成了全包，则需要再次调用 4.2.6 进行读取判断。

代码 4.2.4.4 判断是否只需要读取单个包（默认 false），如果是则读取一个包后就跳出循环，也就是如果出现了粘包现象，在一次 channelRead 事件到来后并不会循环读取所有的包，而是读取最先到的一个包，那么 buffer 里面剩余的包要等下一次 channelRead 事件到了时候在读取。

# 五、使用包定长 DelimiterBasedFrameDecoder 解决半包粘包

## 5.1 试验

首先修改客户端代码，首先需要修改 NettyClientHandler 的代码：

```
    public NettyClientHandler() {
        request = "hello server,im a client|".getBytes();
    }

    public void channelRead(ChannelHandlerContext ctx, Object
msg) {

        System.out.println(atomicInteger.getAndIncrement() +
"receive from server:" +msg);
    }
```

可知只是在字符串后面添加了 | 作为包分隔符。

主函数添加 DelimiterBasedFrameDecoder和StringDecoder 解码器：



```
...
ByteBuf delimiter = Unpooled.copiedBuffer("|".getBytes());
                            p.addLast(new
DelimiterBasedFrameDecoder(1000,delimiter));
                            p.addLast(new StringDecoder());
                            p.addLast(new NettyClientHandler());
...
```

其中添加了一个 DelimiterBasedFrameDecoder 解码器，第一个参数是允许的最大包大小，第二个函数是分隔符，用来具体判定一个包的结束。

然后修改服务端代码，首先需要修改 NettyServerHandler 代码如下：

```
    public void channelRead(ChannelHandlerContext ctx, Object
msg) {

        System.out.println(atomicInteger.getAndIncrement() +
"receive client info: " + msg);

        String sendContent = "hello client ,im server|";
        ByteBuf seneMsg = Unpooled.buffer(sendContent.length());
        seneMsg.writeBytes(sendContent.getBytes());
```

```
            ctx.writeAndFlush(Unpooled.copiedBuffer(seneMsg));
            System.out.println("send info to client:" + sendContent);

        }
```

可知只是在 sendContent 的后面添加了 | 作为分隔符。同理主函数添加 DelimiterBasedFrameDecoder 和 StringDecoder：

```
                                ByteBuf delimiter =
    Unpooled.copiedBuffer("|".getBytes());
                                p.addLast(new
    DelimiterBasedFrameDecoder(1000, delimiter));
                                p.addLast(new StringDecoder());
                                p.addLast(new NettyServerHandler());
```

然后重新启动服务器和客户端输出结果如下：

客户端结果如下：

```
--- client already connected----
0receive from server:hello client ,im server
1receive from server:hello client ,im server
2receive from server:hello client ,im server
3receive from server:hello client ,im server
4receive from server:hello client ,im server
5receive from server:hello client ,im server
6receive from server:hello client ,im server
7receive from server:hello client ,im server
8receive from server:hello client ,im server
9receive from server:hello client ,im server
```

服务端结果如下：

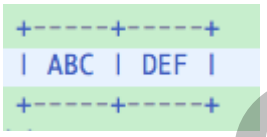
```
----Server Started----
--- accepted client---
0receive client info: hello server,im a client
send info to client:hello client ,im server|
1receive client info: hello server,im a client
send info to client:hello client ,im server|
2receive client info: hello server,im a client
send info to client:hello client ,im server|
3receive client info: hello server,im a client
send info to client:hello client ,im server|
4receive client info: hello server,im a client
send info to client:hello client ,im server|
5receive client info: hello server,im a client
send info to client:hello client ,im server|
```

```
6receive client info: hello server,im a client
send info to client:hello client ,im server|
7receive client info: hello server,im a client
send info to client:hello client ,im server|
8receive client info: hello server,im a client
send info to client:hello client ,im server|
9receive client info: hello server,im a client
send info to client:hello client ,im server|
```

## 5.2 DelimiterBasedFrameDecoder 的原理

顾名思义 DelimiterBasedFrameDecoder 是使用分隔符来判断一个包的结束，从而可以界定不同的包，假设服务端接受缓存里面内容如下：


然后我们使用 \n 作为分隔符，则解码后会生成两个包。






DelimiterBasedFrameDecoder 也是继承自 ByteToMessageDecoder，与 FixedLengthFrameDecoder 不同在于重写的 decode 方法不同，所以这里我们只看 DelimiterBasedFrameDecoder 重写的 decode 方法：

```
 protected Object decode(ChannelHandlerContext ctx, ByteBuf
buffer) throws Exception {
        ...
        //5.1
        int minFrameLength = Integer.MAX_VALUE;
        ByteBuf minDelim = null;
        for (ByteBuf delim: delimiters) {
            int frameLength = indexOf(buffer, delim);
            if (frameLength >= 0 && frameLength < minFrameLength)
{
                minFrameLength = frameLength;
                minDelim = delim;
            }
        }
        //5.2
        if (minDelim != null) {
            int minDelimLength = minDelim.capacity();
            ByteBuf frame;
            ...
            //5.2.1
```

```
            if (minFrameLength > maxFrameLength) {
                // Discard read frame.
                buffer.skipBytes(minFrameLength +
minDelimLength);
                fail(minFrameLength);
                return null;
            }
            //5.2.2
            if (stripDelimiter) {
                frame = buffer.readRetainedSlice(minFrameLength);
                buffer.skipBytes(minDelimLength);
            } else {
                frame = buffer.readRetainedSlice(minFrameLength +
minDelimLength);
            }

            return frame;
        //5.3
        } else {
            if (!discardingTooLongFrame) {
                if (buffer.readableBytes() > maxFrameLength) {
                    // Discard the content of the buffer until a
delimiter is found.
                    tooLongFrameLength = buffer.readableBytes();
                    buffer.skipBytes(buffer.readableBytes());
                    discardingTooLongFrame = true;
                    if (failFast) {
                        fail(tooLongFrameLength);
                    }
                }
            } else {
                // Still discarding the buffer since a delimiter
is not found.
                tooLongFrameLength += buffer.readableBytes();
                buffer.skipBytes(buffer.readableBytes());
            }
            return null;
        }
    }
```



如上代码 5.1 寻找分隔符集合中能分割出最小包的分隔符，比如接受 buffer 里面内容为 `ABC\nDEF\r\n` ；那么在分隔符为 `\n` 和 `\r\n` 时候，会选择 `\n` 为分隔符，因为使用 `\n` 分隔出来的包大小比用 `\r\n` 分隔出来的包大小小。

代码 5.2 如果找到了分隔符，则看 5.2.1 使用分隔符分隔出来的帧大小是否大于设置的最大帧大小，如果大于则丢弃当前帧。

代码 5.2.2 如果设置了丢弃分隔符（也就是要不要吧分隔符作为业务数据读取出来），则解析出来的帧里面不包含分隔符，否者包含。然后从接受缓存具体读取一个帧包。

代码 5.3 如果发现接受缓存里面不存在我们指定的分隔符则丢弃接受缓存里面的数据。

# 六、Dubbo 在使用 Netty 时候是如何避免粘包与半包的

前面我们讲解了基于包定长和分隔符来解决粘包与半包的问题，本节我们来讲解 Dubbo 中使用的第三种方法，也就是使用自定义包来避免，Dubbo 的包协议前面讲解过了，下面在重复下：




其中 header 格式如下：

header格式

| Bit offset | 0-7 | 8-15 | 16-23 | 24-31 | 32-95 | 96-127 |
|---|---|---|---|---|---|---|
| 0 | magic high | magic low | request and serialization flag | response status | request id | body length |

可知 header 记录 body 长度。

Dubbo 里面具体是 InternalDecoder 类来进行处理粘包问题的，它也是继承了 Netty 的 ByteToMessageDecoder 类，所以我们只需要看 InternalDecoder 重写的 decode 方法：

```
    private class InternalDecoder extends ByteToMessageDecoder {

        protected void decode(ChannelHandlerContext ctx, ByteBuf
input, List<Object> out) throws Exception {
            ...
            try {
                //6.1 解码对象
                do {
```

```
                    //6.1.1
                    saveReaderIndex = message.readerIndex();
                    try {
                        msg = codec.decode(channel, message);
                    } catch (IOException e) {
                        throw e;
                    }
                    //6.1.2
                    if (msg ==
Codec2.DecodeResult.NEED_MORE_INPUT) {
                        message.readerIndex(saveReaderIndex);
                        break;
                    } else {
                    //6.1.3
                        if (saveReaderIndex ==
message.readerIndex()) {
                            throw new IOException("Decode without
read data.");
                        }
                        if (msg != null) {
                            out.add(msg);
                        }
                    }
                } while (message.readable());
            } finally {

NettyChannel.removeChannelIfDisconnected(ctx.channel());
            }
        }
}
```



如上代码 6.1 可知也是使用循环来处理粘包的，其中 6.1.1 记录当前缓存的读指针到 saveReaderIndex，然后具体调用 DubboCountCodec 的方法具体进行解析，代码 6.1.2 判断 6.1.1 返回结果是不是 `NEED_MORE_INPUT`，如果是则说明 6.1.1 由于半包问题没有返回一个完成包，则回退缓存的指针到 saveReaderIndex 然后退出循环，在下次 channelRead 事件到来后在进行尝试读取。

代码走到 6.1.3 理论上说明代码 6.1.2 已经读取了至少一个完成的包了，但是这里还是判断下 `saveReaderIndex == message.readerIndex()` 来看 6.1.2 是否读取了内容，如果这个判断为 true 则说明 6.1.2 没有读取任何内容，则直接抛出异常。

下面我们重点看代码 6.1.2，代码 DubboCountCodec 的 decode 方法：

```
public Object decode(Channel channel, ChannelBuffer buffer)
throws IOException {
        //6.1.2.1
        int save = buffer.readerIndex();
        MultiMessage result = MultiMessage.create();
        do {
```

```
                //6.1.2.2
                Object obj = codec.decode(channel, buffer);
                if (Codec2.DecodeResult.NEED_MORE_INPUT == obj) {
                    buffer.readerIndex(save);
                    break;
                } else {
                    result.addMessage(obj);
                    save = buffer.readerIndex();
                }
            } while (true);
            //6.1.2.3
            if (result.isEmpty()) {
                return Codec2.DecodeResult.NEED_MORE_INPUT;
            }

            //6.1.2.4
            if (result.size() == 1) {
                return result.get(0);
            }
            return result;
        }
```

如上代码 6.1.2.1 首先保存了缓存的读取指针位置，然后创建了一个 result 对象，Result 内部是一个列表。



代码 6.1.2.2 具体进行读取数据，如果返回结果为 `NEED_MORE_INPUT` 则重置缓存读取指针，然后退出循环，这时候返回 `NEED_MORE_INPUT` 到上层；如果 6.1.2.2 返回结果不为 `NEED_MORE_INPUT` 则说明返回了完成包，则添加到 Result 列表里面。

下面我们重点看代码 6.1.2.2，也就是 ExchangeCodec 的 decode 方法的代码：

```
    public Object decode(Channel channel, ChannelBuffer buffer)
throws IOException {
        int readable = buffer.readableBytes();
        byte[] header = new byte[Math.min(readable,
HEADER_LENGTH)];
        buffer.readBytes(header);
        return decode(channel, buffer, readable, header);
    }
    protected static final int HEADER_LENGTH = 16;
```

可知首先保存总共可读字节数保存到 readable，然后读取协议的头部到 header 数组，dubbo 协议的头部固定长度为 16 字节。

然后我们看 decode(channel, buffer, readable, header) 的逻辑：

```
 protected Object decode(Channel channel, ChannelBuffer buffer,
int readable, byte[] header) throws IOException {
```

```
            // 6.1.2.2.1 检查魔数
            if (readable > 0 && header[0] != MAGIC_HIGH
                    || readable > 1 && header[1] != MAGIC_LOW) {
                ...
            }
            // 6.1.2.2.2检查包长
            if (readable < HEADER_LENGTH) {
                return DecodeResult.NEED_MORE_INPUT;
            }

            // 6.1.2.2.3 从header里面获取body的大小
            int len = Bytes.bytes2int(header, 12);
            checkPayload(channel, len);
            // 6.1.2.2.4 检查长度
            int tt = len + HEADER_LENGTH;
            if (readable < tt) {
                return DecodeResult.NEED_MORE_INPUT;
            }


            ChannelBufferInputStream is = new
    ChannelBufferInputStream(buffer, len);
            // 6.1.2.2.5解析body
            try {
                return decodeBody(channel, is, header);
            } finally {
                ...
            }
        }
```



如上代码 6.1.2.2.1 检查魔数，类似 Java Class 文件，Dubbo 协议规定了一个协议包必须使用数字 0xdabb 开头。

代码 6.1.2.2.2 看可读取的字节数是否小于 Dubbo 协议包的头部固定长度，如果小于则出现了半包，则直接返回 `NEED_MORE_INPUT` 。

代码 6.1.2.2.3 从 header 里面获取 body 的大小，然后检查 body 的大小是否超过了设置的数据包大小，如果超过了则抛出 ExceedPayloadLimitException 异常。

代码 6.1.2.2.4 检查 header+body 的长度是否大于当前可读的字节数，如果是则出现半包情况，则直接返回 `NEED_MORE_INPUT` ，否者执行代码 6.1.2.2.5 具体解析 body 内容。

> 注：Dubbo 通过自定义协议使用 header+body 的方式来解决粘包半包问题，其中 header 中记录了 body 的大小，这种方式便于协议的升级，比如 header 里面新增了几个字段。另外需要注意在读取 header 后，buffer 的读取指针已经后移了，如果后面发现出现了半包现象，需要把读指针重置。

# 七、总结

本文首先讲解粘包与半包的概念以及出粘包半包的原因和解决方法，然后讲解了如何使用对应三种方法来具体解决粘包半包问题。

其中使用包定长方式需要保证每个包都是一样大小，而使用分隔符则不需要每个包大小一致，在框架使用上偏向使用 header+body 这种方式，因为这种方式比较灵活，并且扩展比较好。

Netty 框架本身仅仅是做二进制流的网络传送，本身是不知道包的概念的，所以如果你在使用 Netty 的时候为了避免粘包和半包出现，可以使用 Netty 提供的这些现有的解码器来避免，当然你可以类似 Dubbo 自定义解码器来实现自己的功能。

需要注意的时候在使用 Netty 的时候是一直需要注意粘包和半包的问题的，只是 Netty 提供了一些避免的 handler 可以供我们使用，但是还是需要用户自己手动去设置这些 handler 到 ChannelPipeline 的。