Netty

1 Netty编解码器

1.1 java的编解码

先了解Java的编解码:

- 编码（Encode）称为序列化， 它将对象序列化为字节数组，用于网络传输、数据持久化或者其它用途。
- 解码（Decode）称为反序列化，它把从网络、磁盘等读取的字节数组还原成原始对象（通常是原始对象的拷贝），以方便后续的业务逻辑操作。

java序列化对象只需要实现java.io.Serializable接口并生成序列化ID，这个类就能够通过java.io.ObjectInput和java.io.ObjectOutput序列化和反序列化。

- java序列化目的：1.网络传输 2.对象持久化。

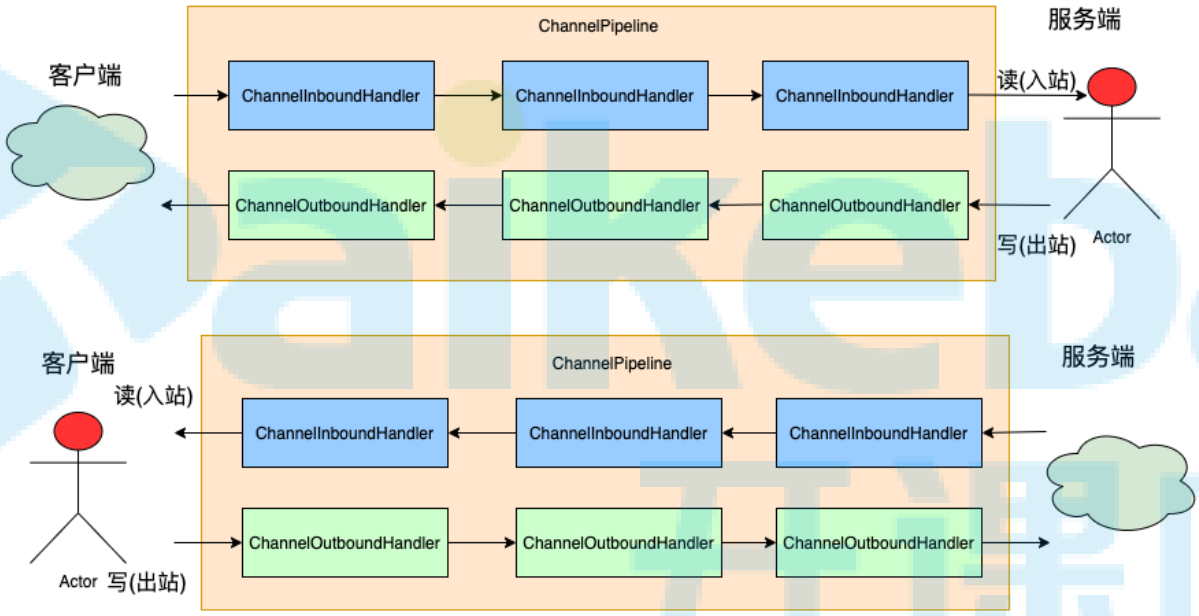
- java序列化缺点：1.无法跨语言 2.序列化后码流太大 3.序列化性能太低。

java序列化仅仅是Java编解码技术的一种，由于它的种种缺陷，衍生出了多种编解码技术和框架，这些编解码框架实现消息的高效序列化。

1.2 基本说明

ChannelHandler 充当了处理入站和出站数据的应用程序逻辑的容器。例如，实现 ChannelInboundHandler 接口（或 ChannelInboundHandlerAdapter），你就可以接收入站事件和数据，这些数据会被业务逻辑处理。当要给客户端发送响应时，也可以从 ChannelInboundHandler 冲刷数据。业务逻辑通常写在一个或者多个 ChannelInboundHandler 中。ChannelOutboundHandler 原理一样，只不过它是用来处理出站数据的

ChannelPipeline 提供了 ChannelHandler 链的容器。以客户端应用程序为例，如果事件的运动方向是从客户端到服务端的，那么我们称这些事件为出站的，即客户端发送给服务端的数据会通过 pipeline 中的一系列 ChannelOutboundHandler，并被这些 Handler 处理，反之则称为入站的



1.3 编码解码器

在网络应用中需要实现某种编解码器，将原始字节数据与自定义的消息对象进行互相转换。**网络中都是以字节码的数据形式来传输数据的，服务器编码数据后发送到客户端，客户端需要对数据进行解码。**

netty提供了强大的编解码器框架，使得我们编写自定义的编解码器很容易，也容易封装重用。对于Netty而言，编解码器由两部分组成：编码器、解码器。

- 解码器：负责将消息从字节或其他序列形式转成指定的消息对象。
- 编码器：将消息对象转成字节或其他序列形式在网络上传输。

Netty 的编（解）码器实现了 ChannelHandlerAdapter，也是一种特殊的 ChannelHandler，所以依赖于 ChannelPipeline，可以将多个编（解）码器链接在一起，以实现复杂的转换逻辑。

Netty里面的编解码：

- 解码器：负责处理“入站 InboundHandler”数据；
- 编码器：负责处理“出站 OutboundHandler” 数据；

不论解码器 handler 还是编码器 handler 即接收的消息类型必须与待处理的消息类型一致，否则该 handler 不会被执行

Netty提供了很多编解码器，例如：

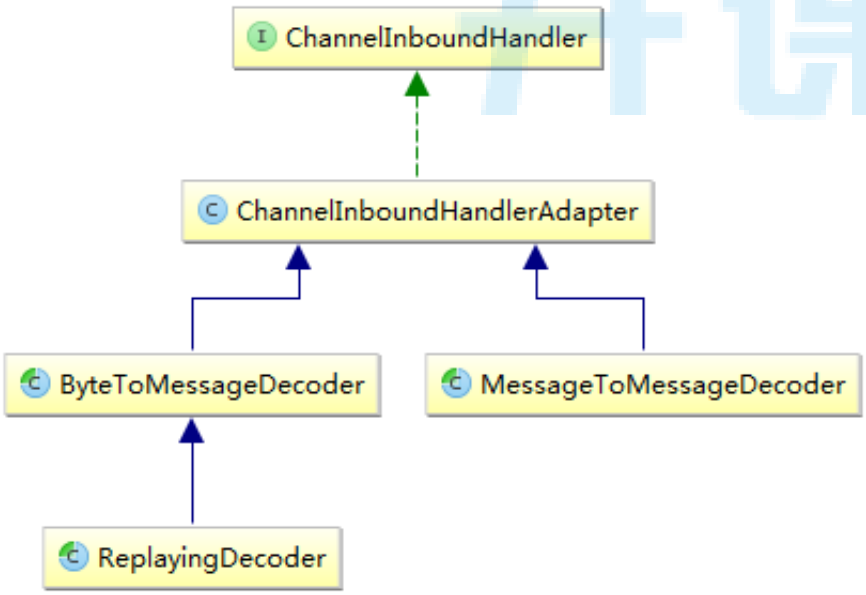
- StringEncoder字符串编码器
- StringDecoder字符串解码器
- `ObjectEncoder`对象编码器
- `ObjectDecoder`对象解码器
- `FixedLengthFrameDecoder`固定长度的解码器
- `LineBasedFrameDecoder`以换行符为结束标识的解码器
- `DelimiterBasedFrameDecoder`指定消息分隔符的解码器
- LengthFieldBasedFrameDecoder基于长度通用解码器

1.4 解码器(Decoder)

解码器负责 解码“入站”数据从一种格式到另一种格式，解码器处理入站数据是抽象ChannelInboundHandler的实现。实践中使用解码器很简单，就是将入站数据转换格式后

传递到ChannelPipeline中的下一个ChannelInboundHandler进行处理；这样的处理时很灵活的，我们可以将解码器放在ChannelPipeline中，重用逻辑。

对于解码器，Netty中主要提供了抽象基类ByteToMessageDecoder和MessageToMessageDecoder



抽象解码器

1) ByteToMessageDecoder: 用于将字节转为消息，需要检查缓冲区是否有足够的字节

2) ReplayingDecoder: 继承ByteToMessageDecoder，不需要检查缓冲区是否有足够的字节,但是ReplayingDecoder速度略慢于ByteToMessageDecoder,同时不是所有的ByteBuf都支持。

选择：项目复杂性高则使用ReplayingDecoder，否则使用 ByteToMessageDecoder

3)MessageToMessageDecoder: 用于从一种消息解码为另外一种消息（例如POJO到POJO）

1.4.1 ByteToMessageDecoder

用于将接收到的二进制数据(Byte)解码，得到完整的请求报文(Message)。

ByteToMessageDecoder是一种ChannelInboundHandler，可以称为解码器，负责将byte字节流(ByteBuf)转换成一种Message，Message是应用可以自己定义的一种java对象。

下面列出了ByteToMessageDecoder两个主要方法：

```
//这个方法是唯一的一个需要自己实现的抽象方法，作用是将ByteBuf数据解码成其他形式的数据。
protected abstract void decode(ChannelHandlerContext ctx, ByteBuf in,
List<Object> out)
decodeLast(ChannelHandlerContext, ByteBuf, List<Object>), //实际上调用的是
decode(...)。
```

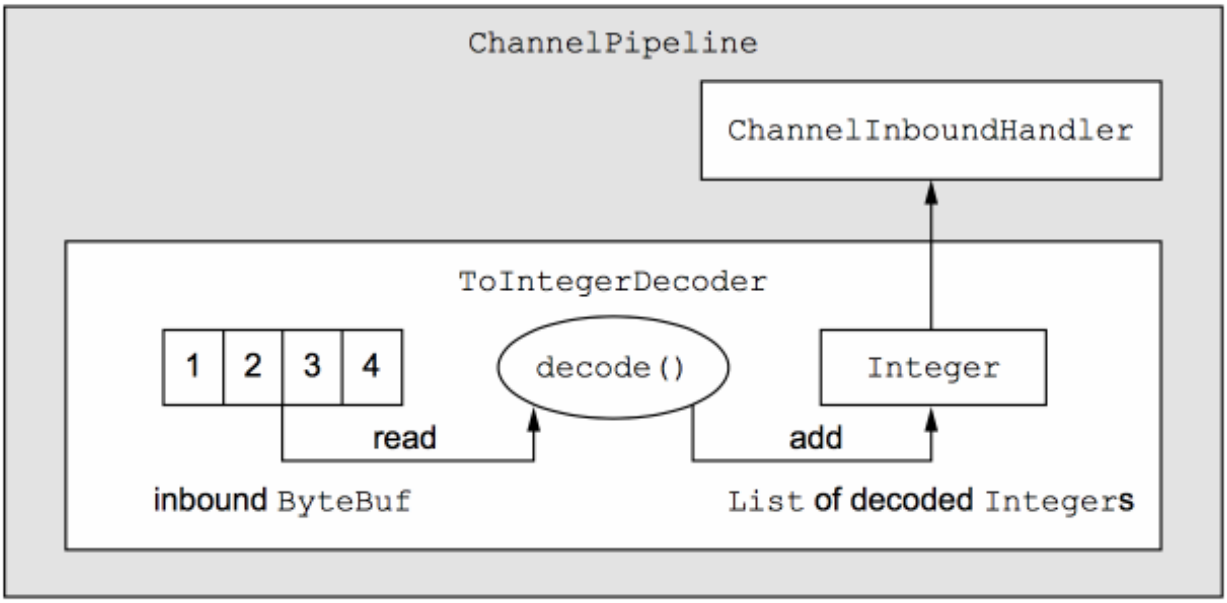
参数的作用如下：

- BytuBuf：需要解码的二进制数据。
- List：解码后的有效报文列表，我们需要将解码后的报文添加到这个List中。之所以使用一个List表示，是因为考虑到粘包问题，因此入参的in中可能包含多个有效报文。

当然，也有可能发生了拆包，in中包含的数据还不足以构成一个有效报文，此时不往List中添加元素即可。

另外特别要注意的是，在解码时，不能直接调用ByteBuf的readXXX方法来读取数据，而是应该首先要判断能否构成一个有效的报文。

案例，假设协议规定传输的数据都是int类型的整数



上图中显式输入的ByteBuf中包含4个字节，每个字节的值分别为：1，2，3，4。我们自定义一个ToIntegerDecoder进行解码，尽管这里我看到了4个字节刚好可以构成一个int类型整数，但是在真正解码之前，我们并不知道ByteBuf包含的字节数能否构成完成的有效报文，因此需要首先判断ByteBuf中剩余可读的字节，是否大于等于4，如下：

```
public class ToIntegerDecoder extends ByteToMessageDecoder {
    @Override
   public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)
throws Exception {
    if (in.readableBytes() >= 4) {
        out.add(in.readInt());
    } }
}
```

只有在可读字节数>=4的情况下，我们才进行解码，即读取一个int，并添加到List中。

在可读字节数小于4的情况下，我们并没有做任何处理，假设剩余可读字节数为3，不足以构成1个int。那么父类ByteToMessageDecoder发现这次解码List中的元素没有变化，则会对in中的剩余3个字节进行缓存，等待下1个字节的到来，之后再回到调用ToIntegerDecoder的decode方法。

另外需要注意: 在ToIntegerDecoder的decode方法中，每次最多只读取一个1个int。如果ByteBuf中的字节数很多，例如为16，那么可以构成4个int，而这里只读取了1个int，那么剩余12字节怎么办？ByteToMessageDecoder再每次回调子类的decode方法之后，都会判断输入的ByteBuf中是否还有剩余字节可读，如果还有，会再次回调子类的decode方法，直到某个回调decode方法List中的元素个数没有变化时才停止，元素个数没有变化，实际上意味着子类已经没有办法从剩余的字节中读取一个有效报文。

由于存在剩余可读字节时，ByteToMessageDecoder会自动再次回调子类decode方法，在实现ByteToMessageDecoder时，decode方法每次只解析一个有效报文即可，没有必要一次全部解析出来。

ByteToMessageDecoder提供的一些常见的实现类:

- FixedLengthFrameDecoder：定长协议解码器，我们可以指定固定的字节数算一个完整的报文
- LineBasedFrameDecoder： 行分隔符解码器，遇到\n或者\r\n，则认为是一个完整的报文
- DelimiterBasedFrameDecoder： 分隔符解码器，与LineBasedFrameDecoder类似，只不过分隔符可以自己指定
- LengthFieldBasedFrameDecoder：长度编码解码器，将报文划分为报文头/报文体，根据报文头中的Length字段确定报文体的长度，因此报文提的长度是可变的
- JsonObjectDecoder：json格式解码器，当检测到匹配数量的"{" 、"}"或"[""]"时，则认为是一个完整的json对象或者json数组。

这些实现类，都只是将接收到的二进制数据，解码成包含完整报文信息的ByteBuf实例后，就直接交给了之后的ChannelInboundHandler处理。

1.4.2 ReplayingDecoder

ReplayingDecoder是byte-to-message解码的一种特殊的抽象基类，byte-to-message解码读取缓冲区的数据之前需要检查缓冲区是否有足够的字节，

使用ReplayingDecoder就无需自己检查；若ByteBuf中有足够的字节，则会正常读取；若没有足够的字节则会停止解码。

ReplayingDecoder 使用方便，但它也有一些局限性：

1) 不是所有的操作都被ByteBuf支持，如果调用一个不支持的操作会抛出DecoderException。

2) ByteBuf.readableBytes()大部分时间不会返回期望值

3)ReplayingDecoder 在某些情况下可能稍慢于 ByteToMessageDecoder，例如网络缓慢并且消息格式复杂时，消息会被拆成了多个碎片，速度变慢

在满足需求的情况下推荐使用ByteToMessageDecoder，因为它的处理比较简单，没有ReplayingDecoder实现的那么复杂。ReplayingDecoder继承于ByteToMessageDecoder，所以他们提供的接口是相同的。下面代码是ReplayingDecoder的实现：

```
/**
 * Integer解码器,ReplayingDecoder实现
 */
public class ToIntegerReplayingDecoder extends ReplayingDecoder<Void> {

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
         System.out.println("ToIntegerReplayingDecoder 被调用");
        //在 ReplayingDecoder 不需要判断数据是否足够读取，内部会进行处理判断
        out.add(in.readInt());
    }
}
```

1.4.3 MessageToMessageDecoder

ByteToMessageDecoder是将二进制流进行解码后，得到有效报文。而MessageToMessageDecoder则是将一个本身就包含完整报文信息的对象转换成另一个java对象。

前面介绍了ByteToMessageDecoder的部分子类解码后，会直接将包含了报文完整信息的ByteBuf实例交由之后的ChannelInboundHandler处理，此时，你可以在

ChannelPipeline中，再添加一个MessageToMessageDecoder，将ByteBuf中的信息解析后封装到Java对象中，简化之后的ChannelInboundHandler的操作。

另外：一些场景下，有可能你的报文信息已经封装到了Java对象中，但是还要继续转成另外的Java对象，因此一个MessageToMessageDecoder后面可能还跟着另一个MessageToMessageDecoder。一个比较容易的理解的类比案例是java Web编程，通常客户端浏览器发送过来的二进制数据，已经被web容器(如tomcat)解析成了一个

HttpServletRequest对象，但是我们还是需要将HttpServletRequest中的数据提取出来，封装成我们自己的POJO类，也就是从一个java对象(HttpServletRequest)转换成另一个Java对象(我们的POJO类)。

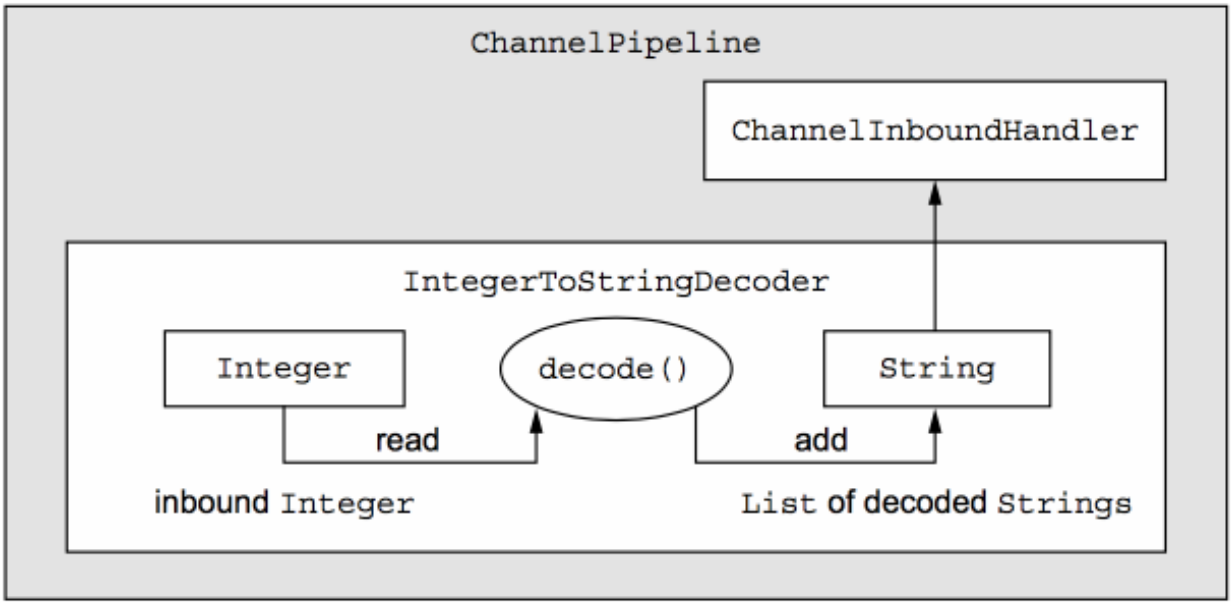
MessageToMessageDecoder的类声明如下：

```
/**
  * 其中泛型参数I表示我们要解码的消息类型。例前面，我们在ToIntegerDecoder中，把二进制字节流转换成了一个int类型的整数。
  */
public abstract class MessageToMessageDecoder<I> extends
ChannelInboundHandlerAdapter
```

类似的，MessageToMessageDecoder也有一个decode方法需要覆盖，如下：

```
/**
* 参数msg，需要进行解码的参数。例如ByteToMessageDecoder解码后的得到的包含完整报文信息ByteBuf
* List<Object> out参数：将msg经过解析后得到的java对象，添加到放到List<Object> out中
*/
protected abstract void decode(ChannelHandlerContext ctx, I msg, List<Object>
out) throws Exception;
```

例如，现在我们想编写一个IntegerToStringDecoder，把前面编写的ToIntegerDecoder输出的int参数转换成字符串，此时泛型I就应该是Integer类型。



integerToStringDecoder源码如下所示

```
public class IntegerToStringDecoder extends MessageToMessageDecoder<Integer> {
    @Override
    public void decode(ChannelHandlerContext ctx, Integer msg List<Object>
out) throws Exception {
        out.add(String.valueOf(msg));
    }
}
```

此时我们应该按照如下顺序组织ChannelPipieline中ToIntegerDecoder和IntegerToStringDecoder 的关系：

也就是说，前一个ChannelInboudHandler输出的参数类型，就是后一个ChannelInboudHandler的输入类型。

特别注意，如果我们指定MessageToMessageDecoder的泛型参数为ByteBuf，表示其可以直接针对ByteBuf进行解码，那么其是否能替代ByteToMessageDecoder呢?

答案是不可以的。因为ByteToMessageDecoder除了进行解码，还要会对不足以构成一个完整数据的报文拆包数据(拆包)进行缓存。而MessageToMessageDecoder则没有这样的逻辑。

因此通常的使用建议是，使用一个ByteToMessageDecoder进行粘包、拆包处理，得到完整的有效报文的ByteBuf实例，然后交由之后的一个或者多个MessageToMessageDecoder对ByteBuf实例中的数据进行解析，转换成POJO类。

1.4.4 自定义解码器

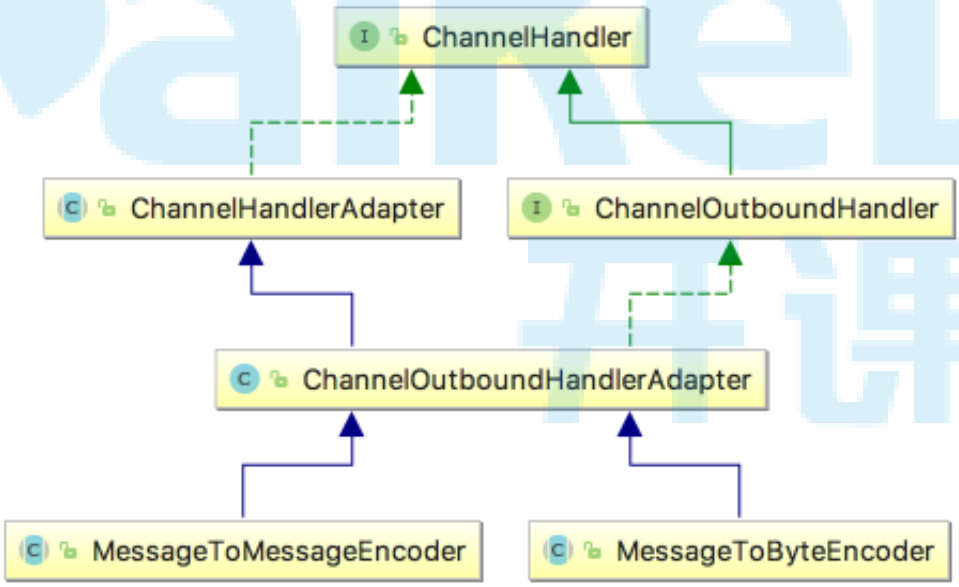
通过继承ByteToMessageDecoder自定义解码器

在解码器进行数据解码时，需判断缓存区（ByteBuf）的数据是否足够，否则收到结果与期望结果可能不一致

```
/**
 * todo 自定义解码器
 */
public class ByteToLongDecoder extends ByteToMessageDecoder {
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        System.out.println("ByteToLongDecoder decode 被调用");
        //todo 因为long占8个字节, 需要判断大于等于8个字节时才能读取一个long
        if(in.readableBytes() >= 8) {
            out.add(in.readLong());
        }
    }
}
```

1.5 编码器(Encoder)

Netty提供了对应的编码器实现MessageToByteEncoder和MessageToMessageEncoder，二者都实现ChannelOutboundHandler接口。



相对来说，**编码器比解码器的实现要更加简单，原因在于解码器除了要按照协议解析数据，还要要处理粘包、拆包问题；而编码器只要将数据转换成协议规定的二进制格式发送即可。**

1.5.1 抽象类MessageToByteEncoder

MessageToByteEncoder也是一个泛型类，泛型参数I表示将需要编码的对象的类型，编码的结果是将信息转换成二进制流放入ByteBuf中。子类通过覆写其抽象方法encode来实现编码，如下所示：

```
public abstract class MessageToByteEncoder<I> extends
ChannelOutboundHandlerAdapter {
....
     protected abstract void encode(ChannelHandlerContext ctx, I msg, ByteBuf
out) throws Exception;
}
```

可以看到，MessageToByteEncoder的输出对象out是一个ByteBuf实例，我们应该将泛型参数msg包含的信息写入到这个out对象中。

```
public class IntegerToByteEncoder extends MessageToByteEncoder<Integer> {
    @Override
    protected void encode(ChannelHandlerContext ctx, Integer msg, ByteBuf out)
throws Exception {
        out.writeInt(msg);//将Integer转成二进制字节流写入ByteBuf中
    }
}
```

1.5.2 抽象类MessageToMessageEncoder

MessageToMessageEncoder同样是一个泛型类，泛型参数I表示将需要编码的对象的类型，编码的结果是将信息放到一个List中。子类通过覆写其抽象方法encode，来实现编码，如下所示：

```
public abstract class MessageToMessageEncoder<I> extends
ChannelOutboundHandlerAdapter {
    ...
    protected abstract void encode(ChannelHandlerContext ctx, I msg,
List<Object> out) throws Exception;
    ...
}
```

与MessageToByteEncoder不同的，MessageToMessageEncoder编码后的结果放到的out参数类型是一个List中。例如，你一次发送2个报文，因此msg参数中实际上包含了2个报文，因此应该解码出两个报文对象放到List中。

MessageToMessageEncoder提供的常见子类包括：

- LineEncoder：按行编码，给定一个CharSequence(如String)，在其之后添加换行符\n或者\r\n，并封装到ByteBuf进行输出，与LineBasedFrameDecoder相对应。
- Base64Encoder：给定一个ByteBuf，得到对其包含的二进制数据进行Base64编码后的新的ByteBuf进行输出，与Base64Decoder相对应。
- LengthFieldPrepender：给定一个ByteBuf，为其添加报文头Length字段，得到一个新的ByteBuf进行输出。Length字段表示报文长度，与LengthFieldBasedFrameDecoder相对应。
- StringEncoder：给定一个CharSequence(如：StringBuilder、StringBuffer、String等)，将其转换成ByteBuf进行输出，与StringDecoder对应。

这些MessageToMessageEncoder实现类最终输出的都是ByteBuf，因为最终在网络上传输的都要是二进制数据。

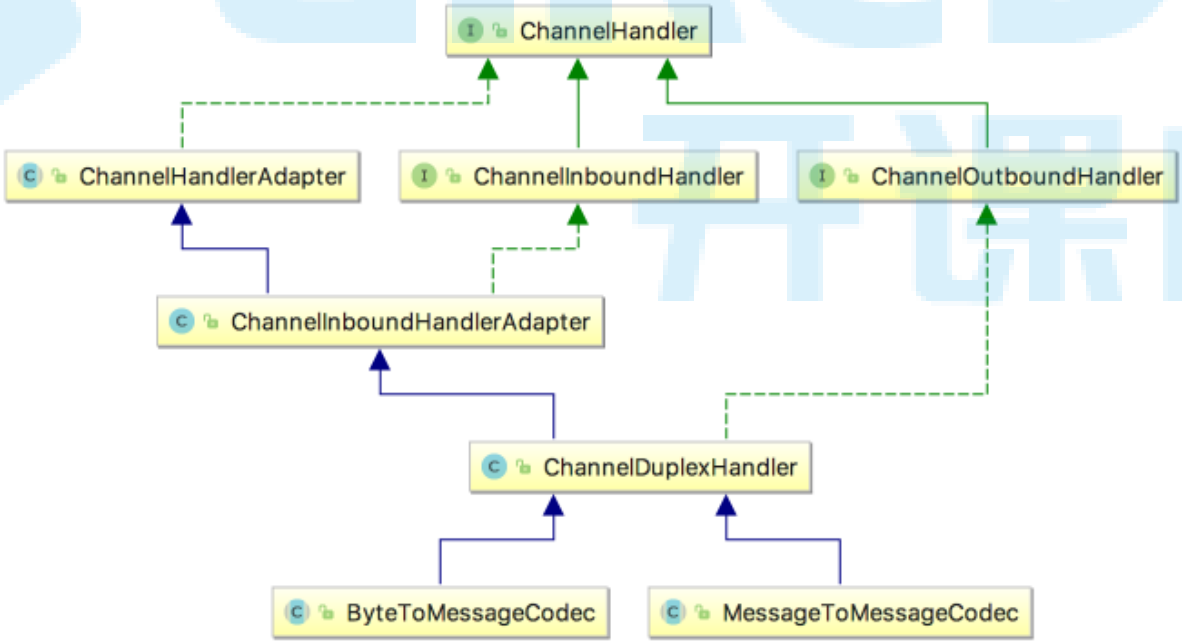
1.5.3 自定义编码器

- 通过继承MessageToByteEncoder自定义编码器

```
 * 自定义编码器
 */
public class LongToByteEncoder extends MessageToByteEncoder<Long> {
    @Override
    protected void encode(ChannelHandlerContext ctx, Long msg, ByteBuf out)
throws Exception {
        System.out.println("LongToByteEncoder encode被调用");
        System.out.println("msg=" + msg);
        out.writeLong(msg);
    }
}
```

1.6 编码解码器Codec

编码解码器: 同时具有编码与解码功能，特点同时实现了ChannelInboundHandler和ChannelOutboundHandler接口，因此在数据输入和输出时都能进行处理。



Netty提供提供了一个ChannelDuplexHandler适配器类，编码解码器的抽象基类ByteToMessageCodec 、MessageToMessageCodec都继承与此类

ByteToMessageCodec内部维护了一个ByteToMessageDecoder和一个MessageToByteEncoder实例，可以认为是二者的功集合，泛型参数I是接受的编码类型：

```
public abstract class ByteToMessageCodec<I> extends ChannelDuplexHandler {
    private final TypeParameterMatcher outboundMsgMatcher;
    private final MessageToByteEncoder<I> encoder;
    private final ByteToMessageDecoder decoder = new ByteToMessageDecoder(){…}

    ...
    protected abstract void encode(ChannelHandlerContext ctx, I msg, ByteBuf
out) throws Exception;
    protected abstract void decode(ChannelHandlerContext ctx, ByteBuf in,
List<Object> out) throws Exception;
    ...
}
```

MessageToMessageCodec内部维护了一个MessageToMessageDecoder和一个MessageToMessageEncoder实例，泛型参数

INBOUND_IN和OUTBOUND_IN分别表示需要解码和编码的数据类型。

```
public abstract class MessageToMessageCodec<INBOUND_IN, OUTBOUND_IN> extends
ChannelDuplexHandler {
   private final MessageToMessageEncoder<Object> encoder= ...
   private final MessageToMessageDecoder<Object> decoder =…
   ...
   protected abstract void encode(ChannelHandlerContext ctx, OUTBOUND_IN msg,
List<Object> out) throws Exception;
   protected abstract void decode(ChannelHandlerContext ctx, INBOUND_IN msg,
List<Object> out) throws Exception;
}
```

其他编解码方式

使用编解码器来充当编码器和解码器的组合失去了单独使用编码器或解码器的灵活性，编解码器是要么都有要么都没有。你可能想知道是否有解决这个僵化问题的方式，还可以让编码器和解码器在ChannelPipeline中作为一个逻辑单元。幸运的是，Netty提供了一种解决方案，使用CombinedChannelDuplexHandler。

如何使用CombinedChannelDuplexHandler来结合解码器和编码器呢？下面我们从两个简单的例子看了解。

```
/**
 * 解码器，将byte转成char
 */
public class ByteToCharDecoder extends ByteToMessageDecoder {

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object>
out) throws Exception {
        while(in.readableBytes() >= 2){
            out.add(Character.valueOf(in.readChar()));
        }
    }
```

```
/**
 * 编码器，将char转成byte
 */
public class CharToByteEncoder extends MessageToByteEncoder<Character> {

    @Override
    protected void encode(ChannelHandlerContext ctx, Character msg, ByteBuf
out) throws Exception {
        out.writeChar(msg);
    }
}
```

```
/**
 * 继承CombinedChannelDuplexHandler，用于绑定解码器和编码器
 */
public class CharCodec extends CombinedChannelDuplexHandler<ByteToCharDecoder,
CharToByteEncoder> {
    public CharCodec(){
        super(new ByteToCharDecoder(), new CharToByteEncoder());
    }
}
```

从上面代码可以看出，使用CombinedChannelDuplexHandler绑定解码器和编码器很容易实现，比使用Codec更灵活。

2 TCP粘包和拆包

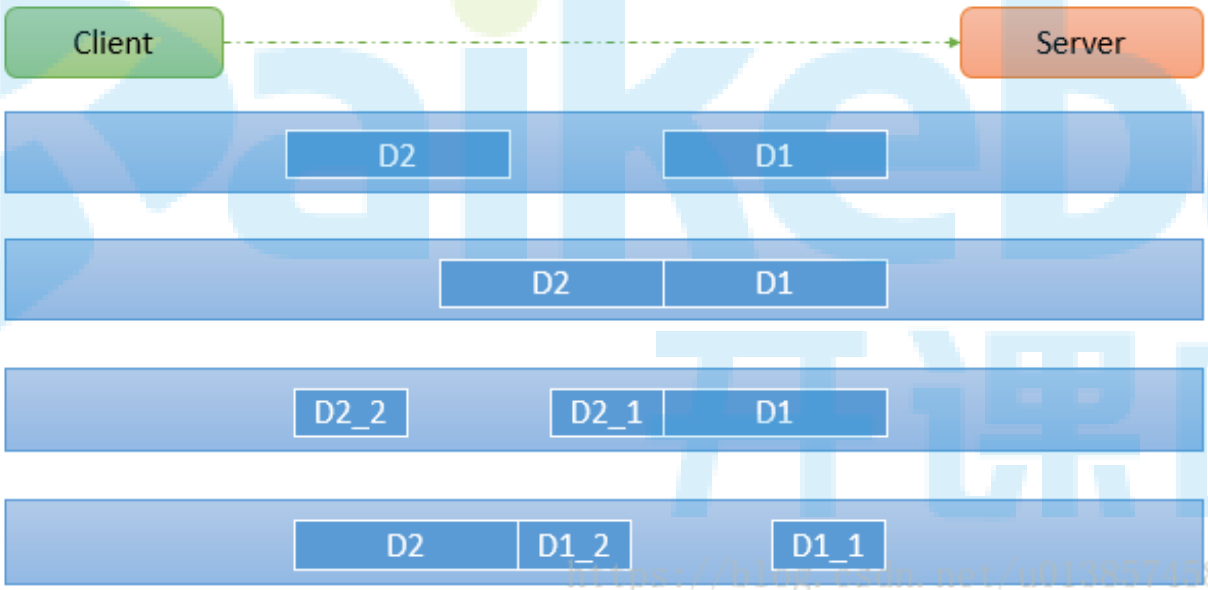
2.1 基本介绍

TCP是个**流**协议，所谓流，就是没有界限的一串数据。TCP底层并不了解上层业务数据的具体含义，它会根据TCP缓冲区的实际情况进行包的划分，所以在业务上认为，一个完整的包可能会被TCP拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的TCP粘包和拆包问题。

> TCP是四层协议，不负责数据逻辑的处理，但是数据在TCP层 “流” 的时候为了保证安全和节约效率会把 “流” 做一些分包处理，比如：
>
> 1. 发送方约定了每次数据传输的最大包大小，超过该值的内容将会被拆分成两个包发送；
> 2. 发送端 和 接收端 约定每次发送数据包长度并随着网络状况动态调整接收窗口大小，这里也会出现拆包的情况；

TCP粘包：把多个小的包封装成一个大的数据包发送，发送方发送的若干数据包到接收方时粘成一个包

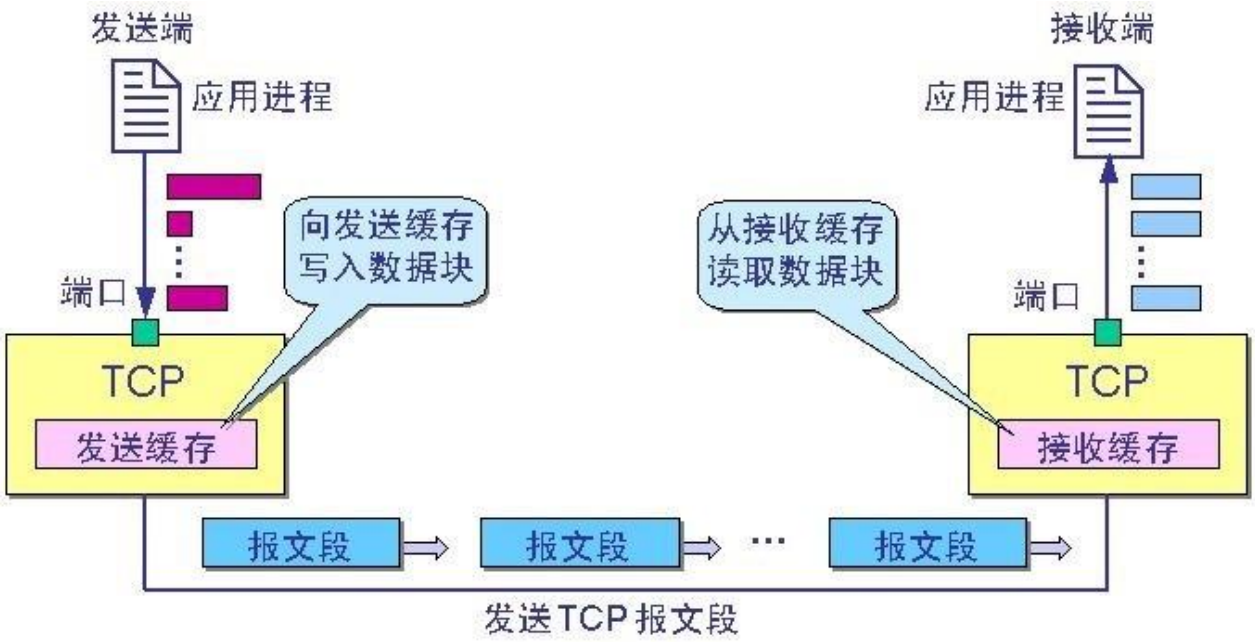
TCP拆包：把一个完整的包拆分为多个小包进行发送，发送方发送一个数据包到接收方时被拆分为若干个小包



假设客户端分别发送了两个数据包 `D1` 和 `D2` 给服务端，由于服务端一次读取到字节数是不确定的，故可能存在以下四种情况：

1. 服务端分两次读取到了两个独立的数据包，分别是 `D1` 和 `D2`，没有粘包和拆包
2. 服务端一次接受到了两个数据包，`D1` 和 `D2` 粘合在一起，称之为 `TCP` 粘包
3. 服务端分两次读取到了数据包，第一次读取到了完整的 `D1` 包和 `D2` 包的部分内容，第二次读取到了 `D2` 包的剩余内容，这称之为 `TCP` 拆包
4. 服务端分两次读取到了数据包，第一次读取到了 `D1` 包的部分内容 `D1_1`，第二次读取到了 `D1` 包的剩余部分内容 `D1_2` 和完整的 `D2` 包。

2.2 产生原因



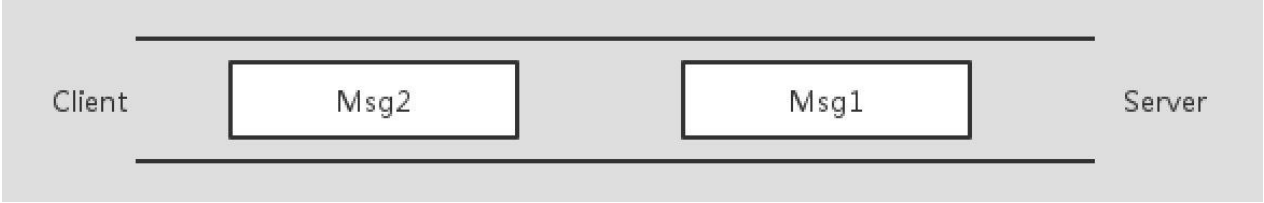
发生TCP粘包、拆包主要是由于下面一些原因：

- 应用程序写入的数据大于缓冲区大小，这将会发生拆包。
- 应用程序写入数据小于字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包。
- 进行MSS（最大报文长度）大小的TCP分段，当TCP报文长度-TCP头部长度>MSS的时候将发生拆包。
- 接收方法不及时读取套接字缓冲区数据，这将发生粘包。

> **MSS:** 是Maximum Segement Size缩写，表示TCP报文中data部分的最大长度，是TCP协议在OSI五层网络模型中传输层对一次可以发送的最大数据的限制。
>
> **MTU:** 最大传输单元是Maxitum Transmission Unit的简写，是OSI五层网络模型中链路层(datalink layer)对一次可以发送的最大数据的限制。
>
> 当需要传输的数据大于MSS或者MTU时，数据会被拆分成多个包进行传输。由于MSS是根据MTU计算出来的，**因此当发送的数据满足MSS时，必然满足MTU。**

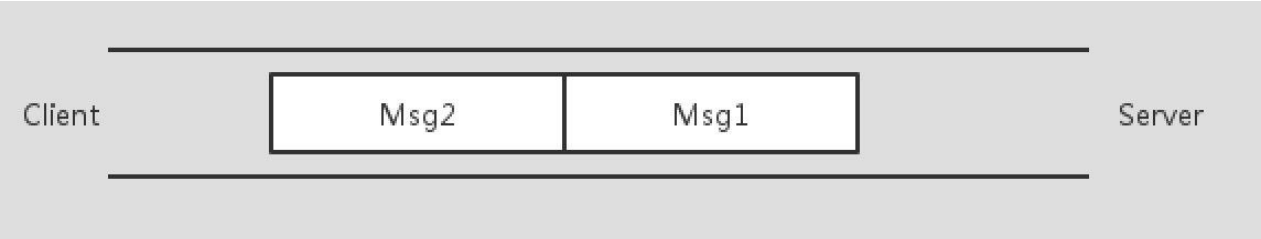
发送端的字节流都会先传入缓冲区，再通过网络传入到接收端的缓冲区中，最终由接收端获取。

当我们发送两个完整包到接收端的时候：

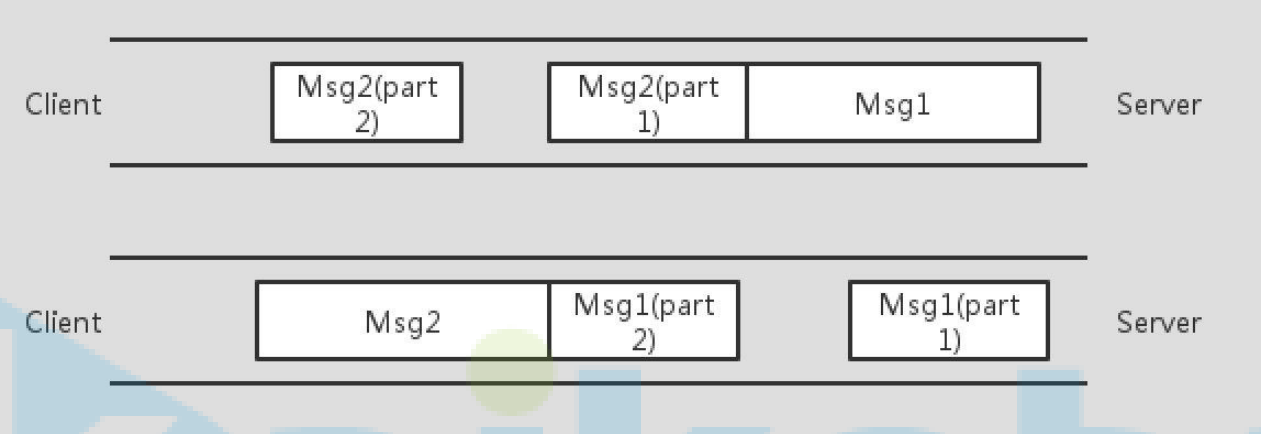


正常情况会接收到两个完整的报文。

但也有以下的情况：



接收到的是一个报文，它是由发送的两个报文组成的，这样对于应用程序来说就很难处理了（这样称为粘包）。



还有可能出现上面这样的虽然收到了两个包，但是里面的内容却是互相包含，对于应用来说依然无法解析（拆包）。

2.3 解决方案

拆包解决思路：

基本思路就是不断的从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

- 若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
- 若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

关键点是如何判断是一个完整的数据包

解决策略

1、设置消息边界(分隔符，对应Netty提供的LineBasedFrameDecoder、DelimiterBasedFrameDecoder解码器)

2、设置定长消息(对应Netty提供的FixedLengthFrameDecoder解码器)

3、使用带消息头的协议，消息头存储消息开始标识及消息的长度信息Header+Body(对应Netty提供的LengthFieldBasedFrameDecoder解码器)

4、发送消息长度，自定义消息解码器

2.4 LineBasedFrameDecoder

LineBasedFrameDecoder是回车换行解码器，如果用户发送的消息以回车换行符作为消息结束的标识，则可以直接使用Netty的LineBasedFrameDecoder对消息进行解码，只需要在初始化Netty服务端或者客户端时将LineBasedFrameDecoder正确的添加到ChannelPipeline中即可，不需要自己重新实现一套换行解码器。

LineBasedFrameDecoder的工作原理是它依次遍历ByteBuf中的可读字节，判断是否有**“\n”或“\r\n”**，如果有，就以此位置为结束位置，从可读索引到结束位置区间的字节就组成了一行。它是以换行符为结束标志的解码器，支持携带结束符或不携带结束符两种解码方式，同时支持配置单行的最大长度。如果连接读取到最大长度后仍然没有发现换行符，就会抛出异常，同时忽略掉之前读到的异常码流。防止由于数据报没有携带换行符导致接收到 ByteBuf 无限制积压，引起系统内存溢出。

通常LineBasedFrameDecoder会和StringDecoder搭配使用。StringDecoder的功能非常简单，就是将接收到的对象转换成字符串，然后继续调用后面的Handler。LineBasedFrameDecoder+StringDecoder组合就是按行切换的文本解码器，用来支持TCP的粘包和拆包。

对于文本类协议的解析，文本换行解码器非常实用，例如对 HTTP 消息头的解析、FTP 协议消息的解析等。

使用起来十分简单，只需要在ChannelPipeline 中添加即可，如下所示

```
ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
            .channel(NioServerSocketChannel.class)
            .option(ChannelOption.SO_BACKLOG, 1024)
            .childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws Exception {

                    ChannelPipeline p = ch.pipeline();
                    p.addLast(new LineBasedFrameDecoder(1024));
                    p.addLast(new StringDecoder());
                    p.addLast(new StringEncoder());

                    p.addLast(new LineServerHandler());
                }
            });
```

2.5 DelimiterBasedFrameDecoder

DelimiterBasedFrameDecoder是分隔符解码器，用户可以指定消息结束的分隔符，它可以自动完成以分隔符作为码流结束标识的消息的解码。回车换行解码器实际上是一种特殊的DelimiterBasedFrameDecoder解码器。

首先将分隔符转换成 ByteBuf 对象，作为参数构造 DelimiterBasedFrameDecoder，将其添加到 ChannelPipeline 中，然后依次添加字符串解码器（通常用于文本解码）和用户 Handler。

DelimiterBasedFrameDecoder 原理分析：解码时，判断当前已经读取的 ByteBuf 中是否包含分隔符 ByteBuf，如果包含，则截取对应的 ByteBuf 返回。

```
protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws
Exception {
        if (lineBasedDecoder != null) {
            return lineBasedDecoder.decode(ctx, buffer);
        }
        // Try all delimiters and choose the delimiter which yields the
shortest frame.
        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;
            }
        }
```

示例

```
//todo 客户端发送数据
public class ClientHandler extends ChannelInboundHandlerAdapter {
    /**
     * todo 当客户端连接服务器完成就会触发该方法
     * @param ctx
     * @throws Exception
     */
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        String message =
"aaaaaaaaaaaaaaaa&_bbbbbbbbbbbbbbbbbb&_ccccccccccc&_";
        ByteBuf byteBuf = Unpooled.buffer(message.getBytes().length);
        byteBuf.writeBytes(message.getBytes());
        ctx.writeAndFlush(byteBuf);
    }
}
```

```
//todo 服务端添加分隔符解码器
public class EchoServer {
    public static void main(String[] args) {

//todo 设置两个线程组
```

```
serverBootstrap.group(bossGroup,workerGroup)
        .channel(NioServerSocketChannel.class)
        .option(ChannelOption.SO_BACKLOG,1024)
        .childHandler(new ChannelInitializer<SocketChannel>() {
            @Override
            protected void initChannel(SocketChannel ch) throws Exception {
                //todo 向pipeline加入分隔符解码器
                ByteBuf delimiter = Unpooled.copiedBuffer("&_".getBytes());
                ch.pipeline().addLast(new
DelimiterBasedFrameDecoder(1024,true,delimiter));
                ch.pipeline().addLast(new StringDecoder());
                ch.pipeline().addLast(new ServerHandler());
            }
        });
    }
}
```

2.6 FixedLengthFrameDecoder

FixedLengthFrameDecoder是固定长度解码器，它能够按照指定的长度对消息进行自动解码，开发者不需要考虑TCP的粘包/拆包等问题，非常实用。

对于定长消息，如果消息实际长度小于定长，则往往会进行补位操作，

它在一定程度上导致了空间和资源的浪费。但是它的优点也是非常明显的，编解码比较简单。

```
ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .option(ChannelOption.SO_BACKLOG, 100)
                    .handler(new LoggingHandler(LogLevel.INFO))//配置日志输出
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch)
                                throws Exception {
                            ch.pipeline().addLast(new
FixedLengthFrameDecoder(10));
                            ch.pipeline().addLast(new StringDecoder());
                            ch.pipeline().addLast(new StringEncoder());

                            ch.pipeline().addLast(new ServerHandler());
                        }
                    });
```

利用 FixedLengthFrameDecoder 解码器，无论一次接收到多少数据报，它都会按照构造函数中设置的固定长度进行解码，如果是半包消息，FixedLengthFrameDecoder 会缓存半包消息并等待下个包到达后进行拼包，直到读取到一个完整的包。

2.7 LengthFieldBasedFrameDecoder

大多数的协议（私有或者公有），协议头中会携带长度字段，用于标识消息体或者整包消息的长度，例如SMPP、HTTP协议等。由于基于长度解码需求的通用性，以及为了降低用户的协议开发难度，Netty提供了LengthFieldBasedFrameDecoder，自动屏蔽TCP底层的拆包和粘包问题，只需要传入正确的参数，即可轻松解决“读半包”问题。

```
* <pre>
* lengthFieldOffset   = 0
* lengthFieldLength   = 2
* lengthAdjustment    = 0
* <b>initialBytesToStrip</b> = <b>2</b> (= the length of the Length field)
*
* BEFORE DECODE (14 bytes)         AFTER DECODE (12 bytes)
* +--------+----------------+      +----------------+
* | Length | Actual Content |----->| Actual Content |
* | 0x000C | "HELLO, WORLD" |      | "HELLO, WORLD" |
* +--------+----------------+      +----------------+
* </pre>
```

lengthFieldOffset = 0，长度字段偏移位置为0表示从包的第一个字节开始读取；

lengthFieldLength = 2，长度字段长为2，从包的开始位置往后2个字节的长度为长度字段；

lengthAdjustment = 0 ，解析的时候无需跳过任何长度；

initialBytesToStrip = 2，去掉当前数据包的开头2字节，去掉 header。

0x000C 转为 int = 12。

```
public LengthFieldBasedFrameDecoder(
            int maxFrameLength,
            int lengthFieldOffset, int lengthFieldLength,
            int lengthAdjustment, int initialBytesToStrip) {
        this(
                maxFrameLength,
                lengthFieldOffset, lengthFieldLength, lengthAdjustment,
                initialBytesToStrip, true);
    }
```

示例

服务端

```
package com.kkb.demo.netty.example.packageEvent;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;

public class PeServer {

}
```

客户端

```
package com.kkb.demo.netty.example.packageEvent;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;

public class PeClient {
    private  int port;
    private  String address;

    public PeClient(int port, String address) {
        this.port = port;
        this.address = address;
    }

    public void start(){
        EventLoopGroup group = new NioEventLoopGroup();

        Bootstrap bootstrap = new Bootstrap();
        bootstrap.group(group)
                .channel(NioSocketChannel.class)
                .option(ChannelOption.TCP_NODELAY, true)
                .handler(new ClientChannelInitializer());
        try {
            ChannelFuture future = bootstrap.connect(address,port).sync();
            future.channel().writeAndFlush("Hello world, i'm online");
```

```
            future.channel().closeFuture().sync();
        } catch (Exception e) {
            e.printStackTrace();
        }finally {
            group.shutdownGracefully();
        }

    }

    public static void main(String[] args) {
        PeClient client = new PeClient(7788,"127.0.0.1");
        client.start();
    }
}
```

2.8 自定义消息解码器

```
package com.kkb.demo.netty.example.protocoltcp;

//协议包
public class MessageProtocol {
    private int len; //关键

    private byte[] content;

    public int getLen() {
        return len;
    }

    public void setLen(int len) {
        this.len = len;
    }

    public byte[] getContent() {
        return content;
    }

    public void setContent(byte[] content) {
        this.content = content;
    }
}
```

服务端

```
package com.kkb.demo.netty.example.protocoltcp;
```

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;

public class MyProtocolServer {
    private int port;

    public MyProtocolServer(int port) {
        this.port = port;
    }

    public void start(){
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workGroup = new NioEventLoopGroup();

        ServerBootstrap server = new
ServerBootstrap().group(bossGroup,workGroup)
                .channel(NioServerSocketChannel.class)
                .childHandler(new ServerChannelInitializer());

        try {
            ChannelFuture future = server.bind(port).sync();
            future.channel().closeFuture().sync();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally {
            bossGroup.shutdownGracefully();
            workGroup.shutdownGracefully();
        }
    }

    public static void main(String[] args) {
        MyProtocolServer server = new MyProtocolServer(7788);
        server.start();
    }
}
```

客户端

```
package com.kkb.demo.netty.example.protocoltcp;


import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
```

```java
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioSocketChannel;

public class MyProtocolClient {
    private  int port;
    private  String address;

    public MyProtocolClient(int port, String address) {
        this.port = port;
        this.address = address;
    }

    public void start(){
        EventLoopGroup group = new NioEventLoopGroup();

        Bootstrap bootstrap = new Bootstrap();
        bootstrap.group(group)
                .channel(NioSocketChannel.class)
                .option(ChannelOption.TCP_NODELAY, true)
                .handler(new ClientChannelInitializer());
        try {
            ChannelFuture future = bootstrap.connect(address,port).sync();
            future.channel().writeAndFlush("Hello world, i'm online");
            future.channel().closeFuture().sync();
        } catch (Exception e) {
            e.printStackTrace();
        }finally {
            group.shutdownGracefully();
        }

    }

    public static void main(String[] args) {
        MyProtocolClient client = new MyProtocolClient(7788,"127.0.0.1");
        client.start();
    }
}
```