11 Netty是如何支持常见的编解码方式的

更新时间：2020-07-30 11:53:33

古之立大事者，不唯有超世之才，亦必有坚韧不拔之志。——苏轼

前言

你好，我是彤哥。

上一节，我们一起学习了粘包 / 半包的相关知识以及解决方案，在上一节的最后，我们说粘包 / 半包的处理在 Netty 中是一次编解码，那么，二次编解码是什么呢？

所以，本节，我们就来谈谈 Netty 中常见的二次编解码。

好了，让我们进入今天的学习吧。

一次编解码和二次编解码

关于一次编解码和二次编解码，我想通过三个问题来叙述：

- 为什么需要一次编解码和二次编解码呢？
- 一次编解码和二次编解码可以合并吗？
- Netty 中如何快速地区分一次编解码和二次编解码呢？

首先，让我们先来看第一个问题：**为什么需要一次编解码和二次编解码呢？**

> 为了便于描述，我这里统一使用解码过程来描述，也就是收到请求处理的过程，编码的过程正好是反过来的。

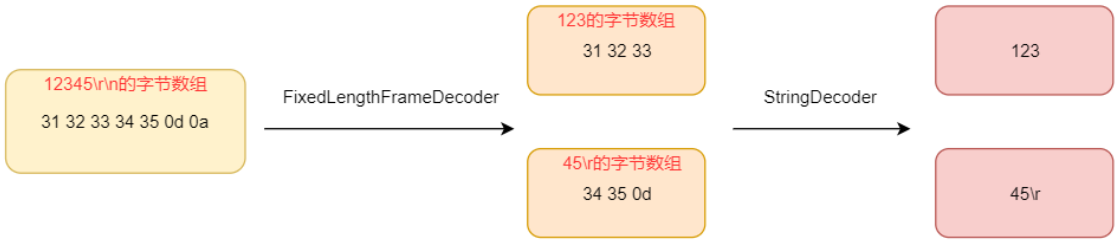
上一节，我们说了，一次解码主要用于解决粘包 / 半包的问题，将缓冲区中的字节数组按照协议本身的格式进行分割，其实，分割后的数据还是字节数组。

那么，分割后的字节数组如何转换成 Java 里面我们可以直接使用的对象呢？

这就需要二次解码了，通过二次解码，可以将字节数组转换成 Java 对象，然后传入我们自定义的 Handler 里面进行业务逻辑的处理。

比如，上一节中，固定长度为 3 的一次编解码器的那个例子，如果我们需要在控制台打印出来输入的内容，那么就要经历以下几个过程：

1. 运用一次解码将 “12345\r\n” 的字节数组拆分成 “123” 和 “45\r” 的字节数组；
2. 运用二次解码将 “123” 和 “45\r” 转换成 Java 的 String 类型的对象；
3. 打印上面的 String 对象；



既然一次解码的时候都已经解出了对应的字节数组，何不顺势而为将其序列化成 Java 对象呢？

所以，**一次编解码和二次编解码可以合并吗？**

可以，但是不建议，这里主要运用了分层的思想，举个简单的例子，比如一次编解码我们采用的是 “长度 + 内容法”，二次编解码一开始使用的是 XML，后面换成了 JSON，其实一次编解码我们不需要修改，只需要修改二次编解码就可以了。但是，如果二者合为一体了，那我们在后面实现 JSON 编解码的时候又要重新实现一下 “长度 + 内容” 的一次编解码的过程。

分层的思想很重要，在 Java 中随处可见，比如，著名的 MVC 分层思想。

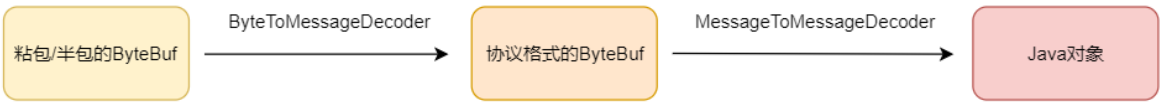
> 凡事都有特例，Netty 中也有一些编解码没有严格地按照分层的思想来实现，比如 MarshallingEncoder，但是，还是那句话，不建议合并，分层很重要。

最后，**Netty 中如何快速区分一次编解码和二次编解码呢？**

其实，贴心的 Netty 也想到了这个问题，所以她定义了下面两组类来分别表示一次编解码和二次编解码：

- 一次编解码：MessageToByteEncoder/ByteToMessageDecoder
- 二次编解码：MessageToMessageEncoder/MessageToMessageDecoder

正常来说，继承自 MessageToByteEncoder 或者 ByteToMessageDecoder 类的就是一次编解码，继承自 MessageToMessageEncoder 或者 MessageToMessageDecoder 类的就是二次编解码，其实，也很好理解，服务端接收请求的过程也是先拿到字节数组（在 Netty 中可以理解为 ByteBuf），然后通过 ByteToMessageDecoder 转换成协议格式的字节数组，再把协议格式的字节数组通过 MessageToMessageDecoder 转换成 Java 对象。



> 正如前文所说，凡事都有特例，比如 MarshallingEncoder，它继承自 MessageToByteEncoder，但是它把二次编码的工作也给干了。从 ByteToMessageDecoder 的名称也可以知道，字节数组直接转成 Java 对象也没有毛病，而且，MessageToMessageDecoder 也可以表示 Java 对象 A 转换成 Java 对象 B。不过，对于我们自己来写编解码，最好还是遵循分层的思想来实现。

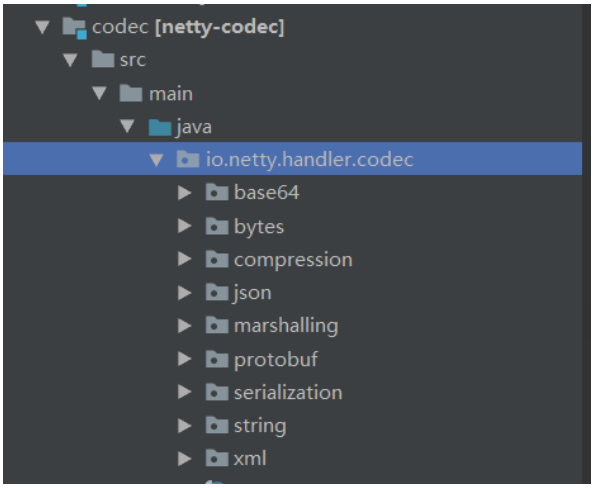
常见的二次编解码方式

常见的二次编解码方式有很多，比如 XML、JSON、Java 序列化等，这些大家都比较熟悉，也比较常用，特别是 JSON，现在随着 RESTful 的流行，基本上基于 Web 开发都使用 JSON 来传输数据。还有一种序列化方式比较流行 ——Google 的 Protobuf，它主要运用在客户端与服务端需要长连接的场景，比如游戏行业，另外，Go 语言中也喜欢用 Protobuf，非常方便，而且高效。

> 二次编解码略等同于序列化方式，如果非要说区别，二次编解码的范围略大于序列化，序列化仅指把 Java 对象转换成字节数组的过程，而二次编解码实际上还包括 Java 对象之间的互相转换，也就是 Message to Message，比如 String 转 Integer，当然，一般不会为这么小的需求还写一个编解码器。

那么，Netty 中支持哪些二次编解码方式呢？

让我们打开 Netty 工程，找到 netty-codec 这个工程，展开目录：



> 不要打开了 netty-codec-xxx 工程了，那些是对各种协议的支持，编解码的范围比较广，Netty 也是因为有了这么多协议、序列化方式的支持才变得这么好用。

可以看到，这个目录下有 base64、bytes、json、protobuf 等等，让我们一个一个来看一下：

- base64，大家都比较熟，BASE64 的支持，常用来把一个字符串转换成另一个字符串，简单加密
- bytes，ByteBuf 与 Java 本身的字节数组 byte[] 之间的互相转换
- compression，各种压缩协议的支持，比如 BZip、Snappy、Zlib 等
- json，通过 JSON 的形式来分割协议，不过，这里只有一个 JSON 一次解码器，因为 JSON 比较简单，只需要 toString () 就能拿到 JSON 文本了，所以，没有相应的二次编解码器，JSON 的优点很多，跨语言，结构清晰，易读
- marshalling，JBoss 的 Marshalling 的支持，也是比较有名的，不过这里的实现没有很好地分层，通过源码可以看到 MarshallingEncoder 继承自 MessageToByteEncoder，而 MarshallingDecoder 继承自 LengthFieldBasedFrameDecoder，缺少一种对称美
- protobuf，Google 的 Protobuf，因体积小，多语言支持而出名，而且不用写多少代码，只需要简单地定义好协议，使用工具一键生成 Java 对象，而且非常方便客户端与服务端不同语言的开发场景
- serialization，基于 Java 序列化做了一些优化，减小了序列化之后字节数组的大小，缺点很明显，只能 Java 中使用
- string，将 ByteBuf 转换成 Java 中的 String 对象，查看源码，其实很简单，只是调用 msg.toString (charset) 就完事了
- xml，XML 的支持，现在很少系统使用 XML 来传输数据了，缺点很明显，报文太大了

好了，我们这里拿三个比较常用的做下简单地对比：

序列化方式	优点	缺点
serialization（优化过的 Java 序列化）	Java 原生，使用方便	报文太大，不便于阅读，只能 Java 使用
json	结构清晰，便于阅读，效率较高，跨语言	报文较大
protobuf	使用方便，效率很高，报文很小，跨语言	不便于阅读

其实，对于性能要求不是特别高的系统，我是非常推荐使用 JSON 这种方式的，毕竟写起来简单，看起来也简单。如果对于性能要求比较高，强烈推荐使用 Protobuf，性能非常高，而且也不用写多少代码，还能很好地定义客户端与服务端之间的协议，比如客户端使用 Javascript，服务端使用 Java，只要双方定好协议，各自使用工具生成对应的代码就可以直接使用了，再也不会为了协议的事儿扯皮了。

后记

本节，我们一起学习了 Netty 中常见的二次编解码，可以看到，Netty 对于大部分的编解码方式都是支持的，即使有少部分不支持，参考现有的代码，相信你也能很快地实现出来。

通过这两节关于 Netty 中编解码的学习，你会发现，其实，使用 Netty 编写服务端程序，只需要写一点 Handler 来处理自己的业务逻辑即可，其它事基本上 Netty 都为我们考虑到了，是不是很爽？

思维导图

}

10 如何解决粘包半包问题

12 Netty服务启动的时候都做了什么