实战：构建客户端与服务端 pipeline

> 通过上小节 (https://juejin.im/book/5b4bc28bf265da0f60130116/s 的学习，我们已经了解 pipeline 和 channelHandler 的基本概念。本小节，我们使用上一小节的理论知识来重新构建服务端和客户端的 pipeline，把复杂的逻辑从单独的一个 channelHandler 中抽取出来。

Netty 内置了很多开箱即用的 ChannelHandler。下面，我们通过学习 Netty 内置的 ChannelHandler 来逐步构建我们的 pipeline。

ChannelInboundHandlerAdapter 与 ChannelOutboundHandlerAdapter

首先是 ChannelInboundHandlerAdapter ，这个适配器主要用于实现其接口 ChannelInboundHandler 的所有方法，这样我们在编写自己的 handler 的时候就不需要实现 handler 里面的每一种方法，而只需要实现我们所关心的方法，默认情况下，对于 ChannelInboundHandlerAdapter，我们比较关心的是他的如下方法

> ChannelInboundHandlerAdapter.java

```
@Override
public void channelRead(ChannelHandlerContext
ctx, Object msg) throws Exception {
    ctx.fireChannelRead(msg);
}
```

他的作用就是接收上一个 handler 的输出，这里的 msg 就是上一个 handler 的输出。大家也可以看到，默认情况下 adapter 会通过 fireChannelRead() 方法直接把上一个 handler 的输出结果传递到下一个 handler。

与 ChannelInboundHandlerAdapter 类似的类是 ChannelOutboundHandlerAdapter，它的核心方法如下

ChannelOutboundHandlerAdapter.java

```
@Override
public void write(ChannelHandlerContext ctx,
Object msg, ChannelPromise promise) throws
Exception {
    ctx.write(msg, promise);
}
```

默认情况下，这个 adapter 也会把对象传递到下一个 outBound 节点，它的传播顺序与 inboundHandler 相反，这里就不再对这个类展开了。

我们往 pipeline 添加的第一个 handler 中的 channelRead 方法中，msg 对象其实就是 ByteBuf。服务端在接受到数据之后，应该首先要做的第一步逻辑就是把这个 ByteBuf 进行解码，然后把解码后的结果传递到下一个 handler，像这样

```
@Override
public void channelRead(ChannelHandlerContext
ctx, Object msg) throws Exception {
        ByteBuf requestByteBuf = (ByteBuf) msg;
        // 解码
        Packet packet =
PacketCodeC.INSTANCE.decode(requestByteBuf);
        // 解码后的对象传递到下一个 handler 处理
        ctx.fireChannelRead(packet)
}
```

不过在开始解码之前，我们来了解一下另外一个特殊的 handler

ByteToMessageDecoder

通常情况下，无论我们是在客户端还是服务端，当我们收到数据之后，首先要做的事情就是把二进制数据转换到我们的一个 Java 对象，所以 Netty 很贴心地写了一个父类，来专门做这个事情，下面我们来看一下，如何使用这个类来实现服务端的解码

```
public class PacketDecoder extends
ByteToMessageDecoder {

    @Override
    protected void decode(ChannelHandlerContext
ctx, ByteBuf in, List out) {
        out.add(PacketCodeC.INSTANCE.decode(in));
    }
}
```

当我们继承了 ByteToMessageDecoder 这个类之后，我们只需要实现一下 decode() 方法，这里的 in 大家可以看到，传递进来的时候就已经是 ByteBuf 类型，所以我们不再需要强转，第三个参数是

List 类型，我们通过往这个 List 里面添加解码后的结果对象，就可以自动实现结果往下一个 handler 进行传递，这样，我们就实现了解码的逻辑 handler。

另外，值得注意的一点，对于 Netty 里面的 ByteBuf，我们使用 4.1.6.Final 版本，默认情况下用的是堆外内存，在 ByteBuf (https://juejin.im/book/5b4bc28bf265da0f60130116/sectior 这一小节中我们提到，堆外内存我们需要自行释放，在我们前面小节的解码的例子中，其实我们已经漏掉了这个操作，这一点是非常致命的，随着程序运行越来越久，内存泄露的问题就慢慢暴露出来了，而这里我们使用 ByteToMessageDecoder，Netty 会自动进行内存的释放，我们不用操心太多的内存管理方面的逻辑，关于如何自动释放内存大家有兴趣可以参考一下 ByteToMessageDecoder的实现原理(8-2) (https://coding.imooc.com/class/chapter/230.html#Anchor)。

当我们通过解码器把二进制数据转换到 Java 对象即指令数据包之后，就可以针对每一种指令数据包编写逻辑了。

SimpleChannelInboundHandler

回顾一下我们前面处理 Java 对象的逻辑

```
if (packet instanceof LoginRequestPacket) {
    // ...
} else if (packet instanceof
MessageRequestPacket) {
    // ...
} else if ...
```

我们通过 if else 逻辑进行逻辑的处理，当我们要处理的指令越来越多的时候，代码会显得越来越臃肿，我们可以通过给 pipeline 添加多个 handler(ChannelInboundHandlerAdapter的子类) 来解决过多的 if else 问题，如下

XXXHandler.java

```
if (packet instanceof XXXPacket) {
    // ...处理
} else {
   ctx.fireChannelRead(packet);
}
```

这样一个好处就是，每次添加一个指令处理器，逻辑处理的框架都是一致的，

但是，大家应该也注意到了，这里我们编写指令处理 handler 的时候，依然编写了一段我们其实可以不用关心的 if else 判断，然后还要手动传递无法处理的对象 (XXXPacket) 至下一个指令处理器，这也是一段重复度极高的代码，因此，Netty 基于这种考虑抽象出了一个 SimpleChannelInboundHandler 对象，类型判断和对象传递的活都自动帮我们实现了，而我们可以专注于处理我们所关心的指令即可。

下面，我们来看一下如何使用 SimpleChannelInboundHandler 来简化我们的指令处理逻辑

LoginRequestHandler.java

```
public class LoginRequestHandler extends
SimpleChannelInboundHandler<LoginRequestPacket> {
    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
LoginRequestPacket loginRequestPacket) {
        // 登录逻辑
    }
}
```

SimpleChannelInboundHandler 从字面意思也可以看到，使用它非常简单，我们在继承这个类的时候，给他传递一个泛型参数，然后在 channelRead0() 方法里面，我们不用再通过 if 逻辑来判断当前对象是否是本 handler 可以处理的对象，也不用强转，不用往下传递本 handler 处理不了的对象，这一切都已经交给父类 SimpleChannelInboundHandler 来实现了，我们只需要专注于我们要处理的业务逻辑即可。

上面的 LoginRequestHandler 是用来处理登录的逻辑，同理，我们可以很轻松地编写一个消息处理逻辑处理器

MessageRequestHandler.java

```
public class MessageRequestHandler extends
SimpleChannelInboundHandler<MessageRequestPacket>
 {
    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
MessageRequestPacket messageRequestPacket) {

    }
}
```

MessageToByteEncoder

在前面几个小节，我们已经实现了登录和消息处理逻辑，处理完请求之后，我们都会给客户端一个响应，在写响应之前，我们需要把响应对象编码成 ByteBuf，结合我们本小节的内容，最后的逻辑框架如下

```
public class LoginRequestHandler extends
SimpleChannelInboundHandler<LoginRequestPacket> {
    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
LoginRequestPacket loginRequestPacket) {
        LoginResponsePacket loginResponsePacket =
login(loginRequestPacket);
        ByteBuf responseByteBuf =
PacketCodeC.INSTANCE.encode(ctx.alloc(),
loginResponsePacket);

ctx.channel().writeAndFlush(responseByteBuf);
    }
}
```

```
public class MessageRequestHandler extends
SimpleChannelInboundHandler<MessageRequestPacket>
 {
    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
MessageRequestPacket messageRequestPacket) {
        MessageResponsePacket
messageResponsePacket =
receiveMessage(messageRequestPacket);
        ByteBuf responseByteBuf =
PacketCodeC.INSTANCE.encode(ctx.alloc(),
messageRequestPacket);

ctx.channel().writeAndFlush(responseByteBuf);
    }
}
```

我们注意到，我们处理每一种指令完成之后的逻辑是类似的，都需要进行编码，然后调用 writeAndFlush() 将数据写到客户端，这个编码的过程其实也是重复的逻辑，而且在编码的过程中，我们还需要手动去创建一个 ByteBuf，如下过程

PacketCodeC.java

```
public ByteBuf encode(ByteBufAllocator
byteBufAllocator, Packet packet) {
    // 1. 创建 ByteBuf 对象
    ByteBuf byteBuf =
byteBufAllocator.ioBuffer();
    // 2. 序列化 java 对象

    // 3. 实际编码过程

    return byteBuf;
}
```

而Netty 提供了一个特殊的 channelHandler 来专门处理编码逻辑，我们不需要每一次将响应写到对端的时候调用一次编码逻辑进行编码，也不需要自行创建 ByteBuf，这个类叫做 MessageToByteEncoder，从字面意思也可以看出，它的功能就是将对象转换到二进制数据。

下面，我们来看一下，我们如何来实现编码逻辑

```
public class PacketEncoder extends
MessageToByteEncoder<Packet> {

    @Override
    protected void encode(ChannelHandlerContext
ctx, Packet packet, ByteBuf out) {
        PacketCodeC.INSTANCE.encode(out, packet);
    }
}
```

PacketEncoder 继承自 MessageToByteEncoder，泛型参数 Packet 表示这个类的作用是实现 Packet 类型对象到二进制的转换。

这里我们只需要实现 encode() 方法，我们注意到，在这个方法里面，第二个参数是 Java 对象，而第三个参数是 ByteBuf 对象，我们在这个方法里面要做的事情就是把 Java 对象里面的字段写到 ByteBuf，我们不再需要自行去分配 ByteBuf，因此，大家注意到，PacketCodeC 的 encode() 方法的定义也改了，下面是更改前后的对比

PacketCodeC.java

```
// 更改前的定义
public ByteBuf encode(ByteBufAllocator
byteBufAllocator, Packet packet) {
    // 1. 创建 ByteBuf 对象
    ByteBuf byteBuf =
byteBufAllocator.ioBuffer();
    // 2. 序列化 java 对象

    // 3. 实际编码过程

    return byteBuf;
}
// 更改后的定义
public void encode(ByteBuf byteBuf, Packet
packet) {
    // 1. 序列化 java 对象

    // 2. 实际编码过程
}
```

我们可以看到，PacketCodeC 不再需要手动创建对象，不再需要再把创建完的 ByteBuf 进行返回。当我们向 pipeline 中添加了这个编码器之后，我们在指令处理完毕之后就只需要 writeAndFlush java

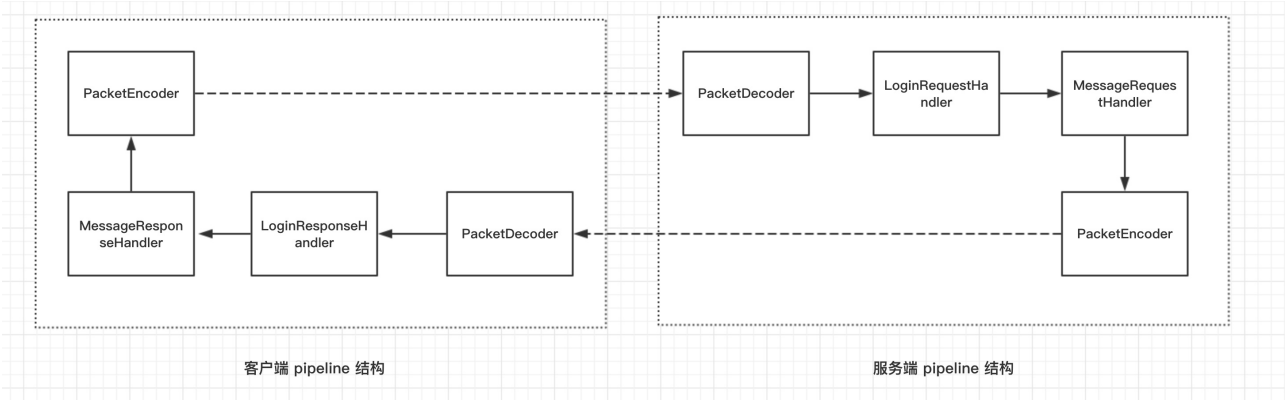
对象即可，像这样

```
public class LoginRequestHandler extends
SimpleChannelInboundHandler<LoginRequestPacket> {
    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
LoginRequestPacket loginRequestPacket) {

ctx.channel().writeAndFlush(login(loginRequestPac
ket));
    }
}

public class MessageRequestHandler extends
SimpleChannelInboundHandler<MessageResponsePacket
> {
    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
MessageResponsePacket messageRequestPacket) {

ctx.channel().writeAndFlush(receiveMessage(messag
eRequestPacket));
    }
}
```

通过我们前面的分析，可以看到，Netty 为了让我们逻辑更为清晰简洁，帮我们做了很多工作，能直接用 Netty 自带的 handler 来解决的问题，不要重复造轮子。在接下里的小节，我们会继续探讨 Netty 还有哪些开箱即用的 handler。

分析完服务端的 pipeline 的 handler 组成结构，相信读者也不难自行分析出客户端的 handler 结构，最后，我们来看一下服务端和客户端完整的 pipeline 的 handler 结构

构建客户端与服务端 pipeline



对应我们的代码

服务端

```
serverBootstrap
                .childHandler(new
ChannelInitializer<NioSocketChannel>() {
                    protected void
initChannel(NioSocketChannel ch) {
                        ch.pipeline().addLast(new
PacketDecoder());
                        ch.pipeline().addLast(new
LoginRequestHandler());
                        ch.pipeline().addLast(new
MessageRequestHandler());
                        ch.pipeline().addLast(new
PacketEncoder());
                    }
            });
```

客户端

```
bootstrap
        .handler(new
ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel
ch) {
                ch.pipeline().addLast(new
PacketDecoder());
                ch.pipeline().addLast(new
LoginResponseHandler());
                ch.pipeline().addLast(new
MessageResponseHandler());
                ch.pipeline().addLast(new
PacketEncoder());
            }
        });
```

> 完整代码在[github (https://github.com/lightningMan/flash-netty/tree/%E6%9E%84%E5%BB%BA%E5%AE%A2%E6%88%B](https://github.com/lightningMan/flash-netty/tree/%E6%9E%84%E5%BB%BA%E5%AE%A2%E6%88%B)
> 对应的本小节分支，大家在本地可以行切换对应分支进行学习。

总结

本小节，我们通过学习 netty 内置的 channelHandler 来逐步构建我们的服务端 pipeline，通过内置的 channelHandler 可以减少很多重复逻辑。

1. 基于 ByteToMessageDecoder，我们可以实现自定义解码，而不用关心 ByteBuf 的强转和 解码结果的传递。
2. 基于 SimpleChannelInboundHandler，我们可以实现每一种

指令的处理，不再需要强转，不再有冗长乏味的 if else 逻辑，不需要手动传递对象。

3. 基于 MessageToByteEncoder，我们可以实现自定义编码，而不用关心 ByteBuf 的创建，不用每次向对端写 Java 对象都进行一次编码。

思考

在 LoginRequestHandler 以及 MessageRequestHandler 的 channelRead0() 方法中，第二个参数对象（XXXRequestPacket）是从哪里传递过来的?
欢迎留言讨论。