实战：客户端与服务端双向通信

在前面两个小节，我们已经学习了服务端启动与客户端启动的流程，熟悉了这两个过程之后，就可以建立服务端与客户端之间的通信了，本小节，我们用一个非常简单的 Demo 来了解一下服务端和客户端是如何来通信的。

> 本小节，我们要实现的功能是：客户端连接成功之后，向服务端写一段数据 ，服务端收到数据之后打印，并向客户端回一段数据，文章里面展示的是核心代码，完整代码请参考 GitHub (https://github.com/lightningMan/flash-netty/tree/%E5%AE%A2%E6%88%B7%E7%AB%AF%E4%B8%8

客户端发数据到服务端

在客户端启动流程 (https://juejin.im/book/5b4bc28bf265da0f60130116/sectior这一小节，我们提到，
客户端相关的数据读写逻辑是通过 Bootstrap 的 handler() 方法指定

```
.handler(new ChannelInitializer<SocketChannel>()
{
    @Override
    public void initChannel(SocketChannel ch) {
        // 指定连接数据读写逻辑
    }
});
```

现在，我们在 initChannel() 方法里面给客户端添加一个逻辑处理器，这个处理器的作用就是负责向服务端写数据

```
.handler(new ChannelInitializer<SocketChannel>()
{
    @Override
    public void initChannel(SocketChannel ch) {
        ch.pipeline().addLast(new
FirstClientHandler());
    }
});
```

1. ch.pipeline() 返回的是和这条连接相关的逻辑处理链，采用了责任链模式，这里不理解没关系，后面会讲到
2. 然后再调用 addLast() 方法 添加一个逻辑处理器，这个逻辑处理器为的就是在客户端建立连接成功之后，向服务端写数据，下面是这个逻辑处理器相关的代码

```
public class FirstClientHandler extends
ChannelInboundHandlerAdapter {
    @Override
    public void
channelActive(ChannelHandlerContext ctx) {
        System.out.println(new Date() + ": 客户端写
出数据");

        // 1. 获取数据
        ByteBuf buffer = getByteBuf(ctx);

        // 2. 写数据
        ctx.channel().writeAndFlush(buffer);
    }

    private ByteBuf
getByteBuf(ChannelHandlerContext ctx) {
        // 1. 获取二进制抽象 ByteBuf
        ByteBuf buffer = ctx.alloc().buffer();

        // 2. 准备数据，指定字符串的字符集为 utf-8
        byte[] bytes = "你好，闪电
侠!".getBytes(Charset.forName("utf-8"));

        // 3. 填充数据到 ByteBuf
        buffer.writeBytes(bytes);

        return buffer;
    }
}
```

1. 这个逻辑处理器继承自 ChannelInboundHandlerAdapter，然后覆盖了

channelActive()方法，这个方法会在客户端连接建立成功之后被调用
2. 客户端连接建立成功之后，调用到 channelActive() 方法，在这个方法里面，我们编写向服务端写数据的逻辑
3. 写数据的逻辑分为两步：首先我们需要获取一个 netty 对二进制数据的抽象 ByteBuf，上面代码中，ctx.alloc() 获取到一个 ByteBuf 的内存管理器，这个
内存管理器的作用就是分配一个 ByteBuf，然后我们把字符串的二进制数据填充到 ByteBuf，这样我们就获取到了 Netty 需要的一个数据格式，最后我们调用 ctx.channel().writeAndFlush() 把数据写到服务端

以上就是客户端启动之后，向服务端写数据的逻辑，我们可以看到，和传统的 socket 编程不同的是，Netty 里面数据是以 ByteBuf 为单位的，
所有需要写出的数据都必须塞到一个 ByteBuf，数据的写出是如此，数据的读取亦是如此，接下来我们就来看一下服务端是如何读取到这段数据的。

服务端读取客户端数据

在服务端端启动流程(https://juejin.im/book/5b4bc28bf265da0f60130116/sectior这一小节，我们提到，
服务端相关的数据处理逻辑是通过 ServerBootstrap 的 childHandler() 方法指定

```
.childHandler(new
ChannelInitializer<NioSocketChannel>() {
   protected void initChannel(NioSocketChannel
ch) {
        // 指定连接数据读写逻辑
   }
});
```

现在，我们在 initChannel() 方法里面给服务端添加一个逻辑处理器，这个处理器的作用就是负责读取客户端来的数据

```
.childHandler(new
ChannelInitializer<NioSocketChannel>() {
    protected void initChannel(NioSocketChannel
ch) {
        ch.pipeline().addLast(new
FirstServerHandler());
    }
});
```

这个方法里面的逻辑和客户端侧类似，获取服务端侧关于这条连接的逻辑处理链 pipeline，然后添加一个逻辑处理器，负责读取客户端发来的数据

```
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")));
    }
}
```

服务端侧的逻辑处理器同样继承自 ChannelInboundHandlerAdapter，与客户端不同的是，这里覆盖的方法是 channelRead()，这个方法在接收到客户端发来的数据之后被回调。

这里的 msg 参数指的就是 Netty 里面数据读写的载体，为什么这里不直接是 ByteBuf，而需要我们强转一下，我们后面会分析到。这里我们强转之后，然后调用 byteBuf.toString() 就能够拿到我们客户端发过来的字符串数据。

我们先运行服务端，再运行客户端，下面分别是服务端控制台和客户端控制台的输出

服务端

```
端口[8000]绑定成功!
Sat Aug 04 09:27:40 CST 2018: 服务端读到数据 -> 你好，闪电侠!
```

客户端

```
连接成功!
Sat Aug 04 09:19:23 CST 2018: 客户端写出数据
```

到目前为止，我们已经实现了客户端发数据服务端打印，离我们本小节开始的目标还差一半，接下来的部分我们来实现另外一半：服务端收到数据之后向客户端回复数据

服务端回数据给客户端

服务端向客户端写数据逻辑与客户端侧的写数据逻辑一样，先创建一个 ByteBuf，然后填充二进制数据，最后调用 writeAndFlush() 方法写出去，下面是服务端回数据的代码

```
public class FirstServerHandler extends
ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext
ctx, Object msg) {
        // ... 收数据逻辑省略

        // 回复数据到客户端
        System.out.println(new Date() + ": 服务端写
出数据");
        ByteBuf out = getByteBuf(ctx);
        ctx.channel().writeAndFlush(out);
    }

    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;
    }
}
```

现在，轮到客户端了。客户端的读取数据的逻辑和服务端读取数据的逻辑一样，同样是覆盖 ChannelRead() 方法

```
public class FirstClientHandler extends
ChannelInboundHandlerAdapter {

    // 写数据相关的逻辑省略

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

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

将这段逻辑添加到客户端之后逻辑处理器 FirstClientHandler 之后，客户端就能收到服务端发来的数据，完整的代码参考 GitHub (https://github.com/lightningMan/flash-netty/tree/%E5%AE%A2%E6%88%B7%E7%AB%AF%E4%B8%8E%E6%

客户端与服务端的读写数据的逻辑完成之后，我们先运行服务端，再运行客户端，控制台输出如下

服务端

```
端口[8000]绑定成功!
Sat Aug 04 09:49:12 CST 2018: 服务端读到数据 -> 你好，闪电侠!
Sat Aug 04 09:49:12 CST 2018: 服务端写出数据
```

客户端

```
连接成功!
Sat Aug 04 09:49:12 CST 2018: 客户端写出数据
Sat Aug 04 09:49:12 CST 2018: 客户端读到数据 -> 你好，欢迎关注我的微信公众号，《闪电侠的博客》！
```

到这里，我们本小节要实现的客户端与服务端双向通信的功能实现完毕，最后，我们对本小节做一个总结。

总结

- 本文中，我们了解到客户端和服务端的逻辑处理是均是在启动的时候，通过给逻辑处理链 pipeline 添加逻辑处理器，来编写数据的读写逻辑，pipeline 的逻辑我们在后面会分析。
- 接下来，我们学到，在客户端连接成功之后会回调到逻辑处理器的 channelActive() 方法，而不管是服务端还是客户端，收到数据之后都会调用到 channelRead 方法。
- 写数据调用writeAndFlush方法，客户端与服务端交互的二进制数据载体为 ByteBuf，ByteBuf 通过连接的内存管理器创建，字节数据填充到 ByteBuf 之后才能写到对端，接下来一小节，我们就来重点分析 ByteBuf。

思考题

如何实现新连接接入的时候，服务端主动向客户端推送消息，客户端回复服务端消息？欢迎留言讨论。