Java NIO 框架 Netty 之美：基础篇之一

一、前言

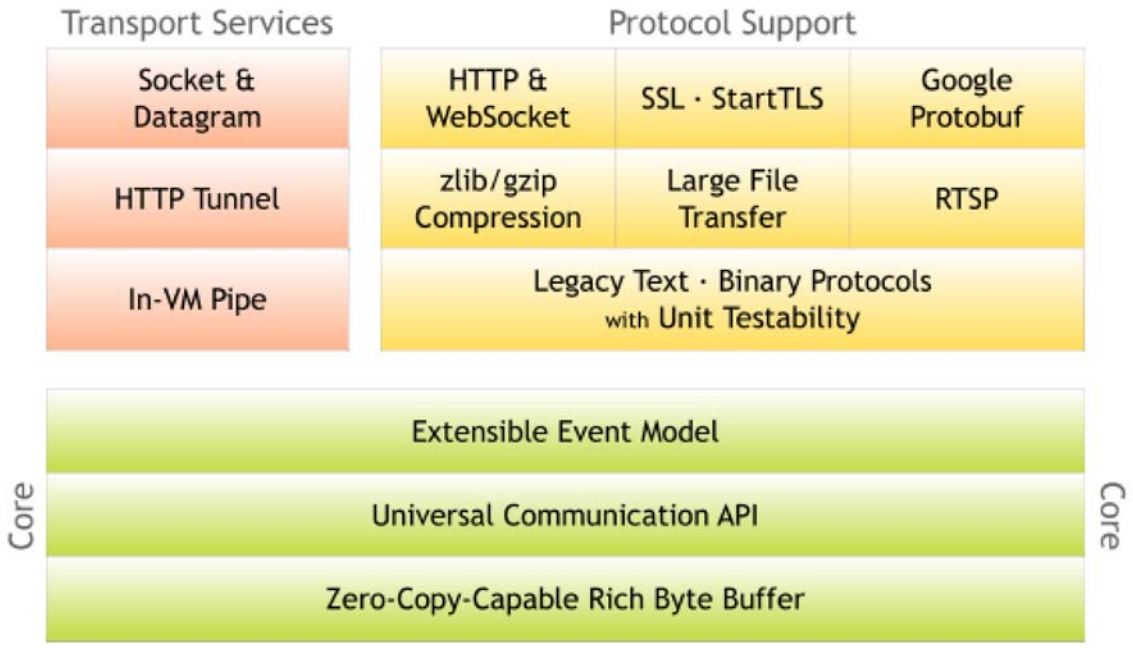
本 Chat 作为 Netty 系列的开篇，主要包含下面内容：

- 初识 Netty；
- 使用 Java NIO 搭建简单的客户端与服务端实现网络通讯；
- 使用 Netty 搭建简单的客户端与服务端实现网络通讯；
- Netty 底层操作与 Java NIO 操作对应关系；
- Netty 中常用术语的概念澄清：Channel 与 Socket 什么关系，Channel 与 EventLoop 什么关系，Channel 与 ChannelPipeline 是什么关系，EventLoop 与 EventLoopGroup 是什么关系等等？

二、初识 Netty

Netty 是一种可以轻松快速的开发协议服务器和客户端网络应用程序的 NIO 框架，它大大简化了 TCP 或者 UDP 服务器的网络编程，但是你仍然可以访问和使用底层的 API，Netty 只是对其进行了高层的抽象。

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。Netty 是被精心设计的，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。



如上图是 Netty 官方的提供的 Netty 的架构图，其中 Transport services 模块是支持的一些传输服务，右侧是支持的传输协议，core 模块是 Netty 的核心特性。下面来看下 Netty 的一些特性：

设计：

- 对不同传输协议类型提供统一的API接口，方便用户使用，比如在使用阻塞和非阻塞套接字时候使用的是同一个 API，只是需要设置的参数不一样。
- 基于一个灵活、可扩展的事件模型来实现。
- 高度可定制的线程模型——单线程、一个或多个线程池。
- 真正的无连接数据报套接字（UDP）的支持

易用性：

- 完善的使用文档和示例代码
- 不需要额外的依赖，JDK 5 (Netty 3.x) 或者 JDK 6 (Netty 4.x) 已经足够。

性能：

- 更好的吞吐量，更低的等待延迟
- 更小的资源消耗
- 最小化不必要的内存拷贝

Netty 作为高性能异步通讯框架，其应用还是比较广泛的，比如阿里巴巴开源的高性能 RPC 框架 Dubbo 的网络通讯默认实现使用的是 Netty， 蚂蚁金服开源的金融级 Sofa-Bolt 框架，底层网络通讯也是基于 Netty 来实现的，还有最近刚开源的 Zuul2.0 使用 Netty 重写了其接受与处理请求的逻辑。

三、使用 Java NIO 搭建简单的客户端与服务端实现网络通讯

本节我们使用 JDK 中原生 NIO API 来创建一个简单的 TCP 客户端与服务器交互的网络程序。

3.1 客户端程序

这个客户端功能是当客户端连接到服务端后，给服务器发送一个 Hello，然后从套接字里面读取服务器端返回的内容并打印，具体代码如下：

```
public class NioClient {

    // (1)创建发送和接受缓冲区
    private static ByteBuffer sendbuffer =
ByteBuffer.allocate(1024);
    private static ByteBuffer receivebuffer =
ByteBuffer.allocate(1024);

    public static void main(String[] args) throws IOException {
        // (2) 获取一个客户端socket通道
        SocketChannel socketChannel = SocketChannel.open();
        // （3）设置socket为非阻塞方式
        socketChannel.configureBlocking(false);
        // （4）获取一个选择器
        Selector selector = Selector.open();
        // （5）注册客户端socket到选择器
        SelectionKey selectionKey =
socketChannel.register(selector, 0);
        // （6）发起连接
        boolean isConnected = socketChannel.connect(new
InetSocketAddress("127.0.0.1", 7001));

        // (7)如果连接没有马上建立成功，则设置对链接完成事件感兴趣
        if (!isConnected) {
            selectionKey.interestOps(SelectionKey.OP_CONNECT);

        }

        int num = 0;
        while (true) {

            // (8) 选择已经就绪的网络IO操作，阻塞方法
            int selectCount = selector.select();
            System.out.println(num + "selectCount:" +
selectCount);
            // （9）返回已经就绪的通道的事件
```

```
            Set<SelectionKey> selectionKeys =
selector.selectedKeys();

            //(10)处理所有就绪事件
            Iterator<SelectionKey> iterator =
selectionKeys.iterator();
            SocketChannel client;
            while (iterator.hasNext()) {
                //(10.1)获取一个事件，并从集合移除
                selectionKey = iterator.next();
                iterator.remove();
                //(10.2)获取事件类型
                int readyOps = selectionKey.readyOps();
                //(10.3)判断是否是OP_CONNECT事件
                if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
                    //（10.3.1）等待客户端socket完成与服务器端的链接
                    client = (SocketChannel)
selectionKey.channel();
                    if (!client.finishConnect()) {
                        throw new Error();

                    }

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

                    //(10.3.2)设置要发送给服务端的数据
                    sendbuffer.clear();
                    sendbuffer.put("hello server,im a
client".getBytes());
                    sendbuffer.flip();
                    //(10.3.3)写入输入。
                    client.write(sendbuffer);
                    //(10.3.4)设置感兴趣事件，读事件

selectionKey.interestOps(SelectionKey.OP_READ);

                //(10.4)判断是否是OP_READ事件
                } else if ((readyOps & SelectionKey.OP_READ) !=
0) {
                    client = (SocketChannel)
selectionKey.channel();
                    //（10.4.1）读取数据并打印
                    receivebuffer.clear();
                    int count = client.read(receivebuffer);
                    if (count > 0) {
                        String temp = new
String(receivebuffer.array(), 0, count);
                        System.out.println(num++ + "receive from
server:" + temp);
                    }
```

```
                }
            }
        }
    }
```

- 代码（1）分别创建了一个发送和接受 buffer，用来发送数据时候 byte 化内容和接受数据。
- 代码（2）获取一个客户端套接字通道。
- 代码（3）设置 socket 通道为非阻塞模式，默认是阻塞模式。
- 代码（4）（5）获取一个选择器，然后注册客户端套接字通道到该选择器，并且设置感兴趣的事情为 0，就是不对任何事件感兴趣。
- 代码（6）（7）调用套接字通道的 connect 方法，连接服务器（服务器套接字地址为 127.0.0.1：7001），由于步骤（3）设置了为非阻塞，所以步骤（6）马上会返回。代码（7）判断连接是否已经完成，如果没有，则设置选择器去监听 OP_CONNECT 事件，也就是指明对该事件感兴趣。
- 然后进入 while 循环进行事件处理，其中代码（8）选择已经就绪的网络IO事件，如果当前没有就绪的则阻塞当前线程。当有就绪事件后，会返回获取的事件个数，会执行代码（9）具体取出来具体事件列表。
- 代码（10）循环处理所有就绪事件，代码（10.1）迭代出一个事件 key，然后从集合中删除，代码（10.2）获取事件 key 感兴趣的标志，代码（10.3）则看兴趣集合里面是否有 OP_CONNECT，如果有则说明有 OP_CONNECT 事件已经就绪了，那么执行步骤（10.3.1）等待客户端与服务端完成三次握手，然后步骤（10.3.2）（10.3.3）写入 hello server,im a client 到服务器端。然后代码（10.3.4）设置对 OP_READ 事件感兴趣。
- 代码（10.4）则看如果当前事件 key 是 OP_READ 事件，说明服务器发来的数据已经在接受 buffer 就绪了，客户端可以去具体拿出来了，然后代码（10.4.1）从客户端套接字里面读取数据并打印。

> 注：设置套接字为非阻塞后，connect 方法会马上返回的，所以需要根据结果判断是否为链接建立 OK 了，如果没有成功，则需要设置对该套接字的 op_connect 事件感兴趣，在这个事件到来的时候还需要调用 finishConnect 方法来具体完成与服务器的链接，在 finishConnect 返回 true 后说明链接已经建立完成了，则这时候可以使用套接字通道发送数据到服务器，并且设置堆该套接字的 op_read 事件感兴趣，从而可以监听到服务端发来的数据，并进行处理。

3.2 服务端程序

服务端程序代码如下：

```
public class NioServer {

    // （1）缓冲区
    private ByteBuffer sendbuffer = ByteBuffer.allocate(1024);
    private ByteBuffer receivebuffer = ByteBuffer.allocate(1024);
```

```
    private Selector selector;

    public NioServer(int port) throws IOException {
        // (2)获取一个服务器套接字通道
        ServerSocketChannel serverSocketChannel =
ServerSocketChannel.open();
        // （3）socket为非阻塞
        serverSocketChannel.configureBlocking(false);
        // （4）获取与该通道关联的服务端套接字
        ServerSocket serverSocket = serverSocketChannel.socket();
        // （5）绑定服务端地址
        serverSocket.bind(new InetSocketAddress(port));
        // （6）获取一个选择器
        selector = Selector.open();
        // （7）注册通道到选择器，选择对OP_ACCEPT事件感兴趣
        serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);
        System.out.println("----Server Started----");

        // (8)处理事件
        int num = 0;
        while (true) {
            // (8.1)获取就绪的事件集合
            int selectKeyCount = selector.select();
            System.out.println(num++ + "selectCount:" +
selectKeyCount);

            Set<SelectionKey> selectionKeys =
selector.selectedKeys();
            // (8.2)处理就绪事件
            Iterator<SelectionKey> iterator =
selectionKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey selectionKey = iterator.next();
                iterator.remove();
                processSelectedKey(selectionKey);
            }
        }
    }

    private void processSelectedKey(SelectionKey selectionKey)
throws IOException {

        SocketChannel client = null;
        // (8.2.1)客户端完成与服务器三次握手
        if (selectionKey.isAcceptable()) {
            // (8.2.1.1)获取完成三次握手的链接套接字
            ServerSocketChannel server = (ServerSocketChannel)
selectionKey.channel();
            client = server.accept();
            if (null == client) {
                return;
```

```
                }
                System.out.println("--- accepted client---");

                //  （8.2.1.2）该套接字为非阻塞模式
                client.configureBlocking(false);
                //  （8.2.1.3）注册该套接字到选择器，对OP_READ事件感兴趣
                client.register(selector, SelectionKey.OP_READ);

                // (8.2.2)为读取事件
            } else if (selectionKey.isReadable()) {
                // (8.2.2.1) 读取数据
                client = (SocketChannel) selectionKey.channel();
                receivebuffer.clear();
                int count = client.read(receivebuffer);
                if (count > 0) {
                    String receiveContext = new
String(receivebuffer.array(), 0, count);
                    System.out.println("receive client info:" +
receiveContext);
                }
                // (8.2.2.2)发送数据到client
                sendbuffer.clear();
                client = (SocketChannel) selectionKey.channel();
                String sendContent = "hello client ,im server";
                sendbuffer.put(sendContent.getBytes());
                sendbuffer.flip();
                client.write(sendbuffer);
                System.out.println("send info to client:" +
sendContent);

            }

        }


        public static void main(String[] args) throws IOException {
            int port = 7001;
            NioServer server = new NioServer(port);
        }
}
```


- 代码（1）分别创建了一个发送和接受 buffer，用来发送数据时候 byte 化内容，和接受数据。
- 代码（2）获取一个服务端监听套接字通道。
- 代码（3）设置 socket 通道为非阻塞模式，默认是阻塞模式。
- 代码（4）获取与该通道关联的服务端套接字
- 代码（5）绑定服务端套接字监听端口为 7001
- 代码（6）（7）获取一个选择器，并注册通道到选择器，选择对 OP_ACCEPT 事件感兴趣，到这里服务端已经开始监听客户端链接了。

- 代码（8） 具体处理事件，8.1 选择当前就绪的事件，8.2 遍历所有就绪事件，顺序调用 processSelectedKey 进行处理。
- 代码（8.2.1） 当前事件key对应的 `OP_ACCEPT` 事件，则执行代码 8.2.1.1 获取已经完成三次握手的链接套接字，并通过代码 8.2.1.2 设置该链接套接字为非阻塞模式，通过代码 8.2.1.3 注册该链接套接字到选择器，并设置对对 `OP_READ` 事件感兴趣。
- 代码（8.2.2） 判断如果当前事件 key 为 `OP_READ` 则通过代码 8.2.2.1 链接套接字里面获取客户端发来的数据，通过代码 8.2.2.2 发送数据到客户端。

> 注：在这个例子里面监听套接字 serverSocket 和 serverSocket 接受到的所有链接套接字都注册到了同一个选择器上，其中 processSelectedKey 里面 8.2.1 是用来处理 serverSocket 接受的新链接的，8.2.2 是用来处理链接套接字的读写的。

到这里服务端和客户端就搭建好了，首先启动服务器，然后运行客户端，会输入如下：

```
0selectCount:1
--- client already connected----
1selectCount:1
2receive from server:hello client ,im server
```

这时候服务器的输出结果为：


```
----Server Started----
0selectCount:1
--- accepted client---
1selectCount:1
receive client info:hello server,im a client
send info to client:hello client ,im server
```

简单分析下结果:

- 服务器端启动后，会先输出 `----Server Started----`
- 客户端启动后去链接服务器端，三次握手完毕后，服务器会获取 `op_accept` 事件，会通过 accept 获取链接套接字，所以输出了：

```
0selectCount:1
--- accepted client---
```

- 然后客户端接受到三次握手信息后，获取到了 `op_connect` 事件，所以输出：

```
0selectCount:1
--- client already connected----
```

然后发送数据到服务器端。

- 服务端收到数据后，选择器会选择出 op_read 事件，读取客户端发来的内容，并发送回执到客户端：

```
1selectCount:1
receive client info:hello server,im a client
send info to client:hello client ,im server
```

- 客户端收到服务器端回执后，选择器会选择出 op_read 事件，所以客户端会读取服务器端发来的内容，所以输出：

```
1selectCount:1
2receive from server:hello client ,im server
```

四、使用 Netty 搭建简单的客户端与服务端实现网络通讯

本节我们使用 Netty 来实现上节使用 Java 原生 NIO 实现的功能，首先需要引入 Netty 的依赖，只需要添加下面的 Maven 配置：

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.13.Final</version>
</dependency>
```

4.1 客户端程序

首先看下主函数代码：

```
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，用来处理io请求，默认线程个数为内核cpu*2
```

```
        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();
        }
    }
}
```

- 如上代码（1.1） 创建 Reactor 线程池，用来处理 IO 请求，默认线程个数为内核 CPU*2。
- 代码（1.2）创建启动类 Bootstrap 实例，用来设置客户端相关参数，其中（1.2.1）设置创建的线程池用来处理 IO 请求；（1.2.2）指定用于创建客户端 NIO 通道的 Class 对象，这里为 NioSocketChannel；（1.2.3）设置客户端套接字参数，这里是打开 TCP_NODELAY 选项；（1.2.4）设置用户自定义 handler，这里在管线里面添加了用户自定义的 NettyClientHandler，其代码后面会讲解；
- 代码（1.3）异步启动与服务端的链接，并且等待链接完成。
- 代码（1.4）同步等待客户端与服务端链接断开。
- 代码（1.5）优雅关闭创建的 Reactor 线程池。

其中用户自定义的 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();
    }
    //(2.1)
    @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);
        }
    }
    //(2.2)
    @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();
    }
}
```


其中代码（2.1）channelActive 函数是当客户端与服务器端链接建立完毕后被回调的函数，这里函数里面把 `hello server,im a client` 二进制化后发送到了服务器，并且刷新缓存让数据立刻发送到服务器端。

代码（2.2）channelRead 函数是当客户端接受 buffer 里面数据就绪后被回调的函数，这里函数里面从 buffer 里面读取服务端发来的数据并打印。

4.2 服务器端程序

首先看下服务器主函数代码：

```
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 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();
        }
```

```
    }
}
```

- 如上代码（1.1） 创建主从 Reactor 线程池，其中 bossGroup 线程池线程个数为 1，用来接收客户端发来的 TCP 请求；workerGroup 线程池线程个数默认为内核 CPU 个数 * 2，用来具体处理 IO 相关的操作。
- 代码（1.2）创建启动类 ServerBootstrap 实例，用来设置客户端相关参数，其中（1.2.1）设置创建的线程池；（1.2.2）指定用于创建客户端 NIO 通道的 Class 对象，这里为 NioServerSocketChannel；（1.2.3）设置客户端套接字参数，这里是设置 SO_BACKLOG 大小为 100，监听套接字在接受客户端请求时候会维护两个队列，一个是存放已经完成 TCP 三次握手的套接字的队列，一个是存放还没有完成三次握手的套接字的队列，这个 backlog 就是两个队列大小之和；（1.2.4）设置日志 handler；（1.2.5）用户自定义 handler，这里在管线里面添加了用户自定义的 NettyServerHandler，其代码后面会讲解；
- 代码（1.3）绑定监听端口，并且等待完成。
- 代码（1.4）同步等待服务端套接字关闭。
- 代码（1.5）优雅关闭创建的 Reactor 线程池。

然后看下服务器主函数代码：

```
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(response));
        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();
        }
    }
```

其中（2.1）channelRead 函数是当服务端收到客户端发来的数据时候被回调，这里函数内部是首先读取客户端发来的数据并打印，然后向客户端写入一些数据。

其中（2.2）channelActive 函数是当服务端监听到客户端连接，并且完成三次握手后回调。

到这里使用 Netty 搭建客户端与服务器网络通讯程序就结束了，首先启动服务器，然后在启动客户端，会有下面结果：

其中客户端结果：

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

其中服务器结果：

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

五、Netty 底层操作与 Java NIO 操作对应关系

5.1 Netty 客户端底层与 Java NIO 对应关系

在讲解 Netty 客户端程序时候我们提到指定 NioSocketChannel 用于创建客户端 NIO 套接字通道的实例，下面我们来看 NioSocketChannel 是如何创建一个 Java NIO 里面的 SocketChannel 的。

首先我们来看 NioSocketChannel 的构造函数：

```
    public NioSocketChannel() {
        this(DEFAULT_SELECTOR_PROVIDER);
    }
```

其中 DEFAULT_SELECTOR_PROVIDER 定义如下：

```
    private static final SelectorProvider
DEFAULT_SELECTOR_PROVIDER = SelectorProvider.provider();
```

然后继续看

```
//这里的provider为DEFAULT_SELECTOR_PROVIDER
    public NioSocketChannel(SelectorProvider provider) {
        this(newSocket(provider));
    }
```

其中 newSocket 代码如下：


```
    private static SocketChannel newSocket(SelectorProvider
provider) {
        try {
            return provider.openSocketChannel();
        } catch (IOException e) {
            throw new ChannelException("Failed to open a
socket.", e);
        }
    }
```

所以 NioSocketChannel 内部是管理一个客户端的 SocketChannel 的，这个 SocketChannel 就是讲 Java NIO 时候的 SocketChannel，也就是创建 NioSocketChannel 实例对象时候相当于执行了 Java NIO 中：

```
SocketChannel socketChannel = SocketChannel.open();
```

另外在 NioSocketChannel 的父类 AbstractNioChannel 的构造函数里面默认会记录队 op_read 事件感兴趣，这个后面当链接完成后会使用到：

```
    protected AbstractNioByteChannel(Channel parent,
SelectableChannel ch) {
```

```
            super(parent, ch, SelectionKey.OP_READ);
        }
```

另外在 NioSocketChannel 的父类 AbstractNioChannel 的构造函数里面设置了该套接字为非阻塞的：

```
    protected AbstractNioChannel(Channel parent, SelectableChannel
    ch, int readInterestOp) {
            super(parent);
            this.ch = ch;
            this.readInterestOp = readInterestOp;
            try {
                ch.configureBlocking(false);
            } catch (IOException e) {
             ...
            }
        }
```

下面我们看 Netty 里面是哪里创建的 NioSocketChannel 实例，哪里注册到选择器的。

下面我们看下 Bootstrap 的 connect 操作代码：


```
        public ChannelFuture connect(InetAddress inetHost, int
    inetPort) {
            return connect(new InetSocketAddress(inetHost,
    inetPort));
        }
```

类似 Java NIO 传递了一个 InetSocketAddress 对象用来记录服务端 IP 和端口：

```
        public ChannelFuture connect(SocketAddress remoteAddress) {
           ...
            return doResolveAndConnect(remoteAddress,
    config.localAddress());
        }
```

下面我们看下 doResolveAndConnect 的代码：

```
    private ChannelFuture doResolveAndConnect(final SocketAddress
    remoteAddress, final SocketAddress localAddress) {
            //(1)
            final ChannelFuture regFuture = initAndRegister();
            final Channel channel = regFuture.channel();

            if (regFuture.isDone()) {
```

```
                if (!regFuture.isSuccess()) {
                    return regFuture;
                }
                 //(2)
                return doResolveAndConnect0(channel, remoteAddress,
    localAddress, channel.newPromise());
            }
                ...
           }
    }
```

首先我们来看代码（1）initAndRegister：

```
        final ChannelFuture initAndRegister() {
            Channel channel = null;
            try {
                //(1.1)
                channel = channelFactory.newChannel();
                //(1.2)
                init(channel);
            } catch (Throwable t) {
                ...
            }
            //(1.3)
            ChannelFuture regFuture =
    config().group().register(channel);
            if (regFuture.cause() != null) {
                if (channel.isRegistered()) {
                    channel.close();
                } else {
                    channel.unsafe().closeForcibly();
                }
            }
    }
```

其中（1.1）作用就是创建一个 NioSocketChannel 的实例，代码（1.2）是具体设置内部套接字的选项的。

代码（1.3)则是具体注册客户端套接字到选择器的，其首先会调用 NioEventLoop 的 register 方法，最后调用 NioSocketChannelUnsafe 的 register 方法：

```
    public final void register(EventLoop eventLoop, final
    ChannelPromise promise) {
                ...
                AbstractChannel.this.eventLoop = eventLoop;

                if (eventLoop.inEventLoop()) {
                    register0(promise);
```

```
            } else {
                try {
                    eventLoop.execute(new Runnable() {
                        @Override
                        public void run() {
                            register0(promise);
                        }
                    });
                } catch (Throwable t) {
                   ...
                }
            }
    }
```

其中 register0 内部调用 doRegister，其代码如下：

```
    protected void doRegister() throws Exception {
            boolean selected = false;
            for (;;) {
                try {
                    //注册客户端socket到当前eventloop的selector上
                    selectionKey =
javaChannel().register(eventLoop().unwrappedSelector(), 0, this);
                    return;
                } catch (CancelledKeyException e) {
                    ...
                }
            }
    }
```

到这里代码（1）initAndRegister 的流程讲解完毕了，下面我们来看代码（2）的：

```
      public final void connect(
                    final SocketAddress remoteAddress, final
SocketAddress localAddress, final ChannelPromise promise) {
              ...
                try {
                    ...

                    boolean wasActive = isActive();
                    if (doConnect(remoteAddress, localAddress)) {
                        fulfillConnectPromise(promise, wasActive);
                    } else {
                      。。。
                    }
                } catch (Throwable t) {
                   ...
```

```
            }
    }
```

其中 doConnect 代码如下：

```
    protected boolean doConnect(SocketAddress remoteAddress,
SocketAddress localAddress) throws Exception {
        ...
         boolean success = false;
         try {
             //2.1
             boolean connected =
SocketUtils.connect(javaChannel(), remoteAddress);
             //2.2
             if (!connected) {

selectionKey().interestOps(SelectionKey.OP_CONNECT);
             }
             success = true;
             return connected;
         } finally {
             if (!success) {
                 doClose();
             }
         }
    }
```


- 其中 2.1 具体调用客户端套接字的 connect 方法，等价于 Java NIO 里面的。
- 代码 2.2 由于 connect 方法是异步的，所以类似 JavaNIO 调用 connect 方法进行判断，如果当前没有完成链接则设置对 op_connect 感兴趣。

最后一个点就是何处进行的从选择器获取就绪的事件的，具体是在该客户端套接关联的 NioEventLoop 里面的做的，每个 NioEventLoop 里面有一个线程用来循环从选择器里面获取就绪的事件，然后进行处理：

```
    protected void run() {
        for (;;) {
            try {
                ...
               select(wakenUp.getAndSet(false));
                ...
                processSelectedKeys();
                ...
            } catch (Throwable t) {
                handleLoopException(t);
            }
            ...
```

```
            }
        }
```

其中 select 代码如下：

```
    private void select(boolean oldWakenUp) throws IOException {
            Selector selector = this.selector;
            try {
                ...
                for (;;) {
                    ...
                    int selectedKeys =
    selector.select(timeoutMillis);
                    selectCnt ++;

                  ...
            } catch (CancelledKeyException e) {
                ...
            }
    }
```

可知会从选择器选取就绪的事件，其中 processSelectedKeys 代码如下：


```
    private void processSelectedKeys() {
            ...
         processSelectedKeysPlain(selector.selectedKeys());
            ...
        }
```

可知会获取已经就绪的事件集合，然后交给 processSelectedKeysPlain 处理，后者循环调用 processSelectedKey 具体处理每个事件，代码如下：

```
    private void processSelectedKey(SelectionKey k,
    AbstractNioChannel ch) {
            ...
            try {
                //(3)如果是op_connect事件
                int readyOps = k.readyOps();
                if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
                    int ops = k.interestOps();
                    ops &= ~SelectionKey.OP_CONNECT;
                    k.interestOps(ops);
                    //3.1
                    unsafe.finishConnect();
                }
                //4
                if ((readyOps & SelectionKey.OP_WRITE) != 0) {
```

```
                ch.unsafe().forceFlush();
            }
            //5
            if ((readyOps & (SelectionKey.OP_READ |
SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
                unsafe.read();
            }
        } catch (CancelledKeyException ignored) {
            unsafe.close(unsafe.voidPromise());
        }
    }
```

代码（3）如果当前事件 key 为 op_connect 则去掉 op_connect，然后调用 NioSocketChannel 的 doFinishConnect：

```
    protected void doFinishConnect() throws Exception {
        if (!javaChannel().finishConnect()) {
            throw new Error();
        }
    }
```

可知是调用了客户端套接字的 finishConnect 方法，最后会调用 NioSocketChannel 的 doBeginRead 方法设置对 op_read 事件感兴趣：


```
    protected void doBeginRead() throws Exception {
       ...
        final int interestOps = selectionKey.interestOps();
        if ((interestOps & readInterestOp) == 0) {
            selectionKey.interestOps(interestOps |
readInterestOp);
        }
    }
```

这里 interestOps 为 op_read，上面在讲解 NioSocketChannel 的构造函数时候提到过。

代码（5）如果当前是 op_accept 事件说明是服务器监听套接字获取到了一个链接套接字，如果是 op_read，则说明可以读取客户端发来的数据了，如果是后者则会激活管线里面的所有 handler 的 channelRead 方法，这里会激活我们自定义的 NettyClientHandler 的 channelRead 读取客户端发来的数据，然后在向客户端写入数据。

注：本节讲解了 Netty 客户端底层如何使用 Java NIO 进行实现的，可见与我们前面讲解的 Java NIO 设计的客户端代码步骤是一致的，只是 Netty 对其进行了封装，方便了我们使用，了解了这些对深入研究 Netty 源码提供了一个骨架指导。

5.2 Netty 服务端底层与 Java NIO 对应关系

在讲解 Netty 服务端程序时候我们提到指定 NioServerSocketChannel 用于创建服务器端 NIO 监听套接字通道的实例，下面我们来看 NioServerSocketChannel 是如何创建一个 Java NIO 里面的 ServerSocketChannel 的。

NioServerSocketChannel 的构造函数如下：

```
    public NioServerSocketChannel() {
        this(newSocket(DEFAULT_SELECTOR_PROVIDER));
    }
```

类似客户端代码，这里 DEFAULT_SELECTOR_PROVIDER 的定义如下：

```
    private static final SelectorProvider
DEFAULT_SELECTOR_PROVIDER = SelectorProvider.provider();
```

其中 newSocket 代码如下：

```
    private static ServerSocketChannel newSocket(SelectorProvider
provider) {
        try {

            return provider.openServerSocketChannel();
        } catch (IOException e) {
            throw new ChannelException(
                    "Failed to open a server socket.", e);
        }
    }
```


可知创建了一个 ServerSocketChannel 通道，这个和 Java NIO 里面创建的 ServerSocketChannel 是一个东西，只是 NioServerSocketChannel 封装了创建步骤。

然后我们继续看 NioServerSocketChannel 的构造函数：

```
    public NioServerSocketChannel(ServerSocketChannel channel) {
        super(null, channel, SelectionKey.OP_ACCEPT);
        config = new NioServerSocketChannelConfig(this,
javaChannel().socket());
    }
```

可知这里设置了 readInterestOp 为 OP_ACCEPT。

最后 NioServerSocketChannel 的父类 AbstractNioChannel 额构造函数设置 ServerSocketChannel 为非阻塞：

```
protected AbstractNioChannel(Channel parent, SelectableChannel
ch, int readInterestOp) {
        super(parent);
        this.ch = ch;
        this.readInterestOp = readInterestOp;
        try {
            ch.configureBlocking(false);
        } catch (IOException e) {
           ...
        }
    }
```

下面我们看哪里注册 ServerSocketChannel 到选择器，哪里进行的 bind 操作，在讲解 Netty 服务端程序时候最后会调用 ServerBootstrap 的 bind 启动服务，下面我们具体看看 bind 内部做了啥：

```
    public ChannelFuture bind(int inetPort) {
        return bind(new InetSocketAddress(inetPort));
    }
public ChannelFuture bind(SocketAddress localAddress) {
       ...
        return doBind(localAddress);
    }
```


可知类似 Java NIO 首先创建了一个监听地址，下面重点看 doBind：

```
    private ChannelFuture doBind(final SocketAddress
localAddress) {
        //(1)
        final ChannelFuture regFuture = initAndRegister();
        final Channel channel = regFuture.channel();
        if (regFuture.cause() != null) {
            return regFuture;
        }

        if (regFuture.isDone()) {
            //(2)
            doBind0(regFuture, channel, localAddress, promise);
            return promise;
        } else {
           ...
        }
    }
```

其中代码（1）initAndRegister 的内容在上节讲解过了，不同在于这里首先创建的是 NioServerSocketChannel 的一个实例，然后设置其内部管理的 ServerSocketChannel 的套接字选项，最后注册监听套接字到对应的 NioEventLoop 管理的选择器。

下面我们重点看代码（2）doBind0，其内部最终调用 NioMessageUnsafe 的 bind 方法：

```
  public final void bind(final SocketAddress localAddress, final
ChannelPromise promise) {
            ...
            boolean wasActive = isActive();
            try {//(3)
                doBind(localAddress);
            } catch (Throwable t) {
                ...
            }

            if (!wasActive && isActive()) {
               //(4)
                pipeline.fireChannelActive();

            }
             ...
 }
```

其中代码（3）调用了 NioServerSocketChannel 的 doBind 方法：

```
    protected void doBind(SocketAddress localAddress) throws
Exception {
        if (PlatformDependent.javaVersion() >= 7) {
            javaChannel().bind(localAddress,
config.getBacklog());
        } else {
            javaChannel().socket().bind(localAddress,
config.getBacklog());
        }
    }
```

可知具体调用了 NioServerSocketChannel 内部维护的 ServerSocket 的 bind 方法，这个类似 Java NIO 里面的。

然后代码（4）最终会调用 NioServerSocketChannel 的 doBeginRead 方法设置该套接字对 op_accept 事件感兴趣。

同理由于监听套接字注册到了 NioEventLoop 的选择器上，所以 NioEventLoop 内部的线程会从选择器获取就绪的 op_accept 事件进行处理，NioEventLoop 的 run 方法上节我们讲过了：

```
            if ((readyOps & (SelectionKey.OP_READ |
SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
                unsafe.read();
            }
```

不同在于这里 unsafe.read 调用的是 NioMessageUnsafe 的 read 方法，调用监听套接字的 accept 方法获取完成三次握手的链接套接字：

```
    protected int doReadMessages(List<Object> buf) throws
Exception {
        SocketChannel ch = SocketUtils.accept(javaChannel());

        try {
            if (ch != null) {
                buf.add(new NioSocketChannel(this, ch));
                return 1;
            }
        } catch (Throwable t) {
         ...
        }

        return 0;
    }
```

然后在 ServerBootstrapAcceptor 的 channelRead 方法注册链接套接字到对应的选择器：


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

    try {
        childGroup.register(child).addListener(new
ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future)
throws Exception {
                if (!future.isSuccess()) {
                    forceClose(child, future.cause());
                }
            }
        });
    } catch (Throwable t) {
        forceClose(child, t);
    }
}
```

注：本节讲解了 Netty 服务端底层如何使用 Java NIO 进行实现的，可见与我们前面讲解的 Java NIO 设计的服务端代码步骤是一致的，只是 Netty 对其进行了封装，方便了我们使用，了解了这些对深入研究 Netty 源码提供了一个骨架指导。

六、Netty 中常用术语的概念澄清

Channel 也就是通道，这个概念是在 JDK NIO 类库里面提供的一个概念，JDK 中其实现类有客户端套接字通道 java.nio.channels.SocketChannel 和服务端监听套接字通道 java.nio.channels.ServerSocketChannel，Channel 的出现是为了支持异步 IO 操作，JDK 里面的通道是 java.nio.channels.Channel。

io.netty.channel.Channel 是 Netty 框架自己定义的一个通道接口，Netty 实现的客户端 NIO 套接字通道是 NioSocketChannel，提供的服务器端 NIO 套接字通道是 NioServerSocketChannel。

- NioSocketChannel

客户端套接字通道，内部管理了一个 Java NIO 中的 java.nio.channels.SocketChannel 实例，用来创建 SocketChannel 实例和设置该实例的属性，并调用 Connect 方法向服务端发起 TCP 链接等。

- NioServerSocketChannel

服务器端监听套接字通道，内部管理了一个 Java NIO 中的 java.nio.channels.ServerSocketChannel 实例，用来创建 ServerSocketChannel 实例和设置该实例属性，并调用该实例的 bind 方法在指定端口监听客户端的链接。

- Channel 与 socket 的关系

在 Netty 中 Channel 有两种，对应客户端套接字通道 NioSocketChannel，内部管理 java.nio.channels.SocketChannel 套接字，对应服务器端监听套接字通道 NioServerSocketChannel，其内部管理自己的 java.nio.channels.ServerSocketChannel 套接字。也就是 Channel 是对 socket 的装饰或者门面，其封装了对 socket 的原子操作。

- EventLoopGroup

Netty 之所以能提供高性能网络通讯，其中一个原因是因为它使用 Reactor 线程模型。在 netty 中每个 EventLoopGroup 本身是一个线程池，其中包含了自定义个数的 NioEventLoop，每个 NioEventLoop 是一个线程，并且每个 NioEventLoop 里面持有自己的 selector 选择器。

在 Netty 中客户端持有一个 EventLoopGroup 用来处理网络 IO 操作，在服务器端持有两个 EventLoopGroup，其中 boss 组是专门用来接收客户端发来的 TCP 链接请求的，worker 组是专门用来具体处理完成三次握手的链接套接字的网络 IO 请求的。

- Channel 与 EventLoop 的关系

Netty 中 NioEventLoop 是 EventLoop 的一个实现，每个 NioEventLoop 中会管理自己的一个 selector 选择器和监控选择器就绪事件的线程；每个 Channel 只会关联一个 NioEventLoop；

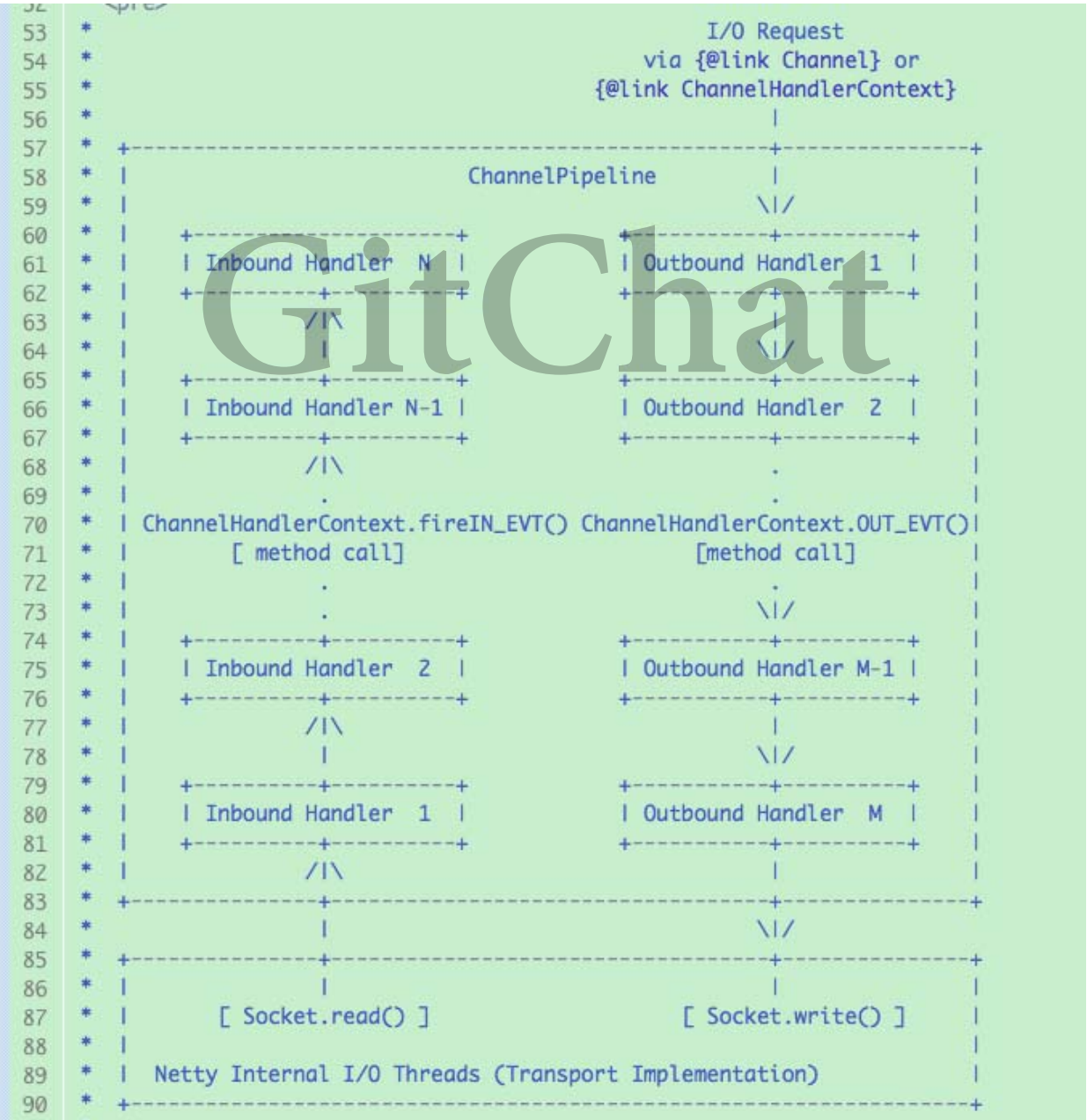
当 Channel 是客户端通道 NioSocketChannel 时候，会注册 NioSocketChannel 管理的 SocketChannel 实例到自己关联的 NioEventLoop 的 selector 选择器上，然后 NioEventLoop 对应的线程会通过 select 命令监控感兴趣的网络读写事件；

当 Channel 是服务端通道 NioServerSocketChannel 时候，NioServerSocketChannel 本身会被注册到 boss EventLoopGroup 里面的某一个 NioEventLoop 管理的 selector 选择器上，而完成三次握手的链接套接字是被注册到了 worker EventLoopGroup 里面的某一个 NioEventLoop 管理的 selector 选择器上；

需要注意是多个 Channel 可以注册到同一个 NioEventLoop 管理的 selector 选择器上，这时候 NioEventLoop 对应的单个线程就可以处理多个 Channel 的就绪事件；但是每个 Channel 只能注册到一个固定的 NioEventLoop 管理的 selector 选择器上。

- ChannelPipeline

Netty 中的 ChannelPipeline 类似于 Tomcat 容器中的 Filter 链，属于设计模式中的责任链模式，其中链上的每个节点就是一个 ChannelHandler。在 netty 中每个 Channel 有属于自己的 ChannelPipeline，对从 Channel 中读取或者要写入 Channel 中的数据进行依次处理，如下图是 netty 源码里面的一个图：



需要注意一点是虽然每个 Channel（更底层说是每个 socket）有自己的 ChannelPipeline，但是每个 ChannelPipeline 里面可以复用一个 ChannelHandler。

七、总结

本文首先讲解了如何使用 Java NIO 来搭建一个简单的 TCP 网络通讯程序，然后讲解了如何使用 Netty 来搭建类似的网络程序，并讲解了 Netty 底层操作是如何与 Java NIO API 对应起来的，最后讲解了 Netty 中一些常用概念，希望读者能够在阅读完本文后，能够下去根据本文提供的关键类和方法进行断点 debug，以便加深理解。