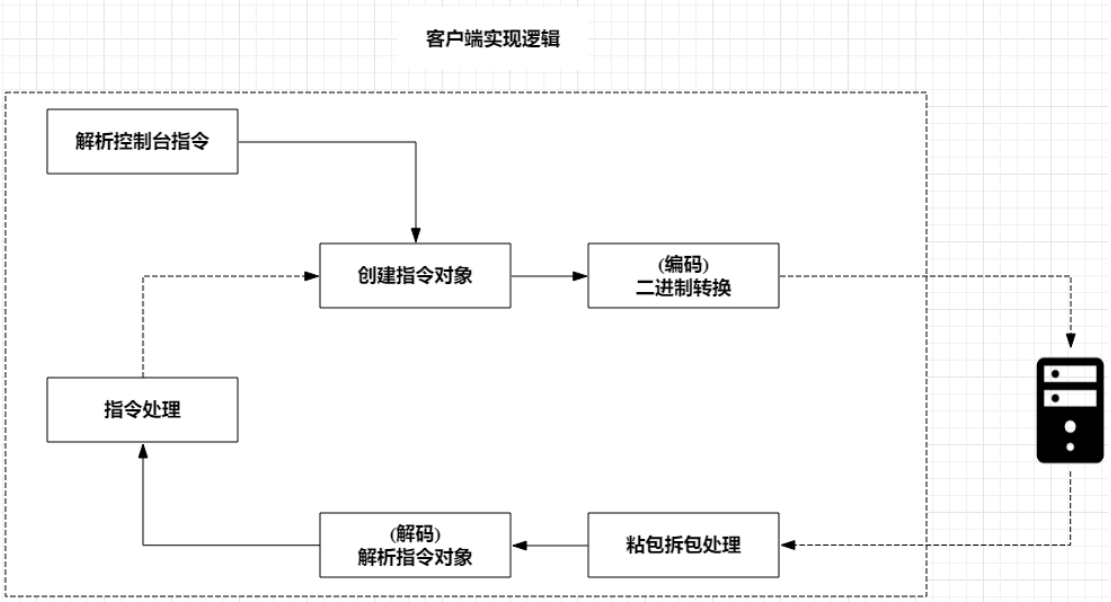
目录

1.仿微信 IM 系统简介

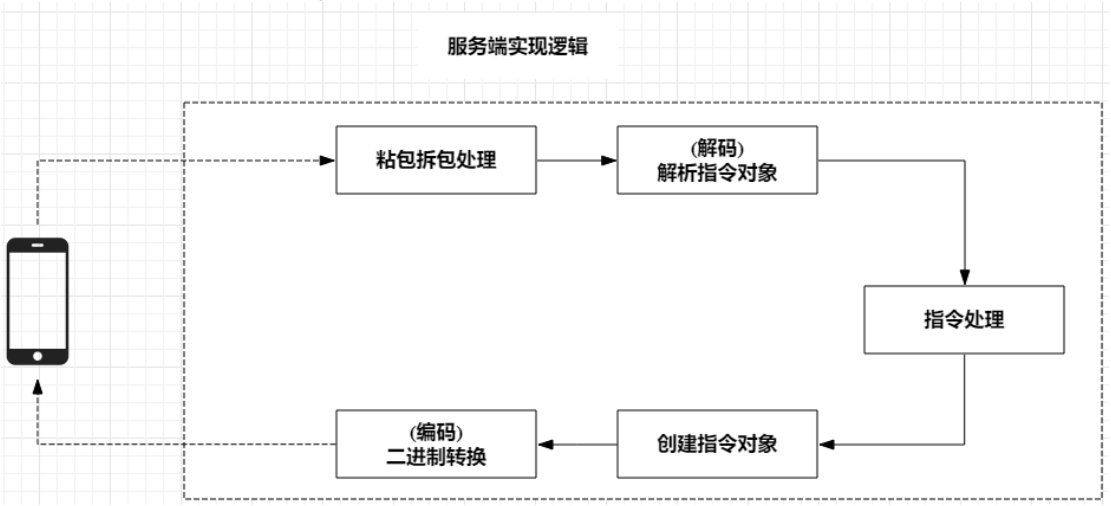
(1)客户端使用 Netty 程序实现逻辑



解析控制台指令(譬如发送消息或者建立群聊等指令)->基于控制台输入创建指令对象->协议的编码(通过自定义二进制协议将指令对象封装成二进制);

接收服务端数据拆包粘包处理(截取一段完整的二进制数据包)-> 协议的解码(将此二进制数据包解析成指令对象)->将指令对象提供给相应逻辑处理器处理.

(2) 服务端使用 Netty 程序实现逻辑



2.Netty 是什么?

(1)IO 模型

```
public class IOServer {
```

```
    /**
     * Server 服务端首先创建 ServerSocket 监听 8000 端口,然后创建线程不断调用阻塞方法 serversocket.accept()获取新的连接,当获取到新的连接给每条连接创建新的线程负责从该连接中读取数据,然后读取数据是以字节流的方式
     *
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(8000);

        //接收新连接线程
        new Thread(() -> {
            try {
                //(1)阻塞方法获取新的连接
                Socket socket = serverSocket.accept();

                //(2)每一个新的连接都创建一个线程,负责读取数据
                new Thread(() -> {
                    try {
                        byte[] data = new byte[1024];
                        InputStream inputStream = socket.getInputStream();
                        while (true) {
                            int len;
                            //(3)按照字节流方式读取数据
                            while ((len = inputStream.read(data)) != -1)
                                System.out.println(new String(data, 0, len));
                        }
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }).start();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
public class IOClient {

    /**
     * Client 客户端连接服务端 8000 端口每隔 2 秒向服务端写带有时间戳的 "hello world"
     *
     * @param args
```

```
     */
    public static void main(String[] args) {
        new Thread(() -> {
            try {
                Socket socket = new Socket("127.0.0.1", 8000);
                while (true) {
                    try {
                        socket.getOutputStream().write((new Date() + ": hello
world").getBytes());
                        socket.getOutputStream().flush();
                        Thread.sleep(2000);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

传统的 IO 模型每个连接创建成功都需要一个线程来维护,每个线程包含一个 while 死循环,那么 1w 个连接对应 1w 个线程,继而 1w 个 while 死循环带来如下几个问题:

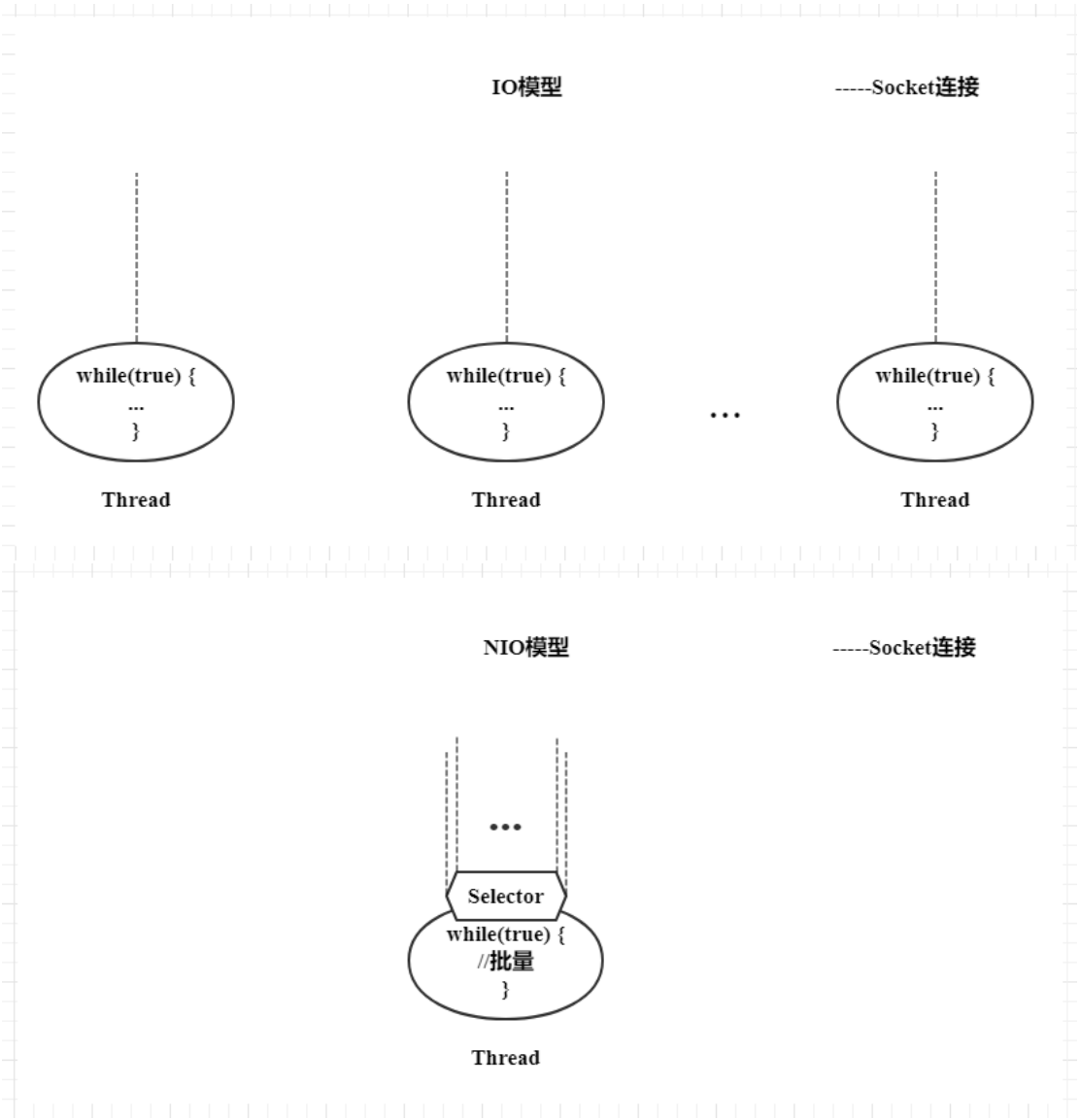
1.线程资源受限:线程是操作系统中非常宝贵的资源,同一时刻有大量的线程处于阻塞状态是非常严重的资源浪费,操作系统耗不起;

2.线程切换效率低下:单机 CPU 核数固定,线程爆炸之后操作系统频繁进行线程切换,应用性能急剧下降;

3.数据读写是以字节流为单位效率不高:每次都是从操作系统底层一个字节一个字节地读取数据.

(2)NIO 模型

1.线程资源受限:NIO 编程模型新来一个连接不再创建一个新的线程,把这条连接直接绑定到某个固定的线程,然后这条连接所有的读写都由该线程来负责.把这么多 while 死循环变成一个死循环,这个死循环由一个线程控制,一条连接来了,不创建一个 while 死循环去监听是否有数据可读,直接把这条连接注册到 Selector 上,然后通过检查 Selector 批量监测出有数据可读的连接进而读取数据.



2.线程切换效率低下:线程数量大大降低,线程切换效率因此也大幅度提高.

3.数据读写是以字节流为单位效率不高:NIO 维护一个缓冲区每次从这个缓冲区里面读取一块的数据,数据读写不再以字节为单位,而是以字节块为单位.

```
public class NIOServer {

    /**
     * serverSelector 负责轮询是否有新的连接,clientSelector 负责轮询连接是否有数据可读.
     * 服务端监测到新的连接不再创建一个新的线程,而是直接将新连接绑定到 clientSelector 上,这样不用 IO 模型中 1w 个 while 循环在死等
     * clientSelector 被一个 while 死循环包裹,如果在某一时刻有多条连接有数据可读通过 clientSelector.select(1)方法轮询出来进而批量处理
     * 数据的读写以内存块为单位
     *
     * @param args
```

```
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        Selector serverSelector = Selector.open();
        Selector clientSelector = Selector.open();

        new Thread(() -> {
            try {
                ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
                serverSocketChannel.socket().bind(new InetSocketAddress(8000));
                serverSocketChannel.configureBlocking(false);
                serverSocketChannel.register(serverSelector, SelectionKey.OP_ACCEPT);

                while (true) {
                    // 轮询监测是否有新的连接
                    if (serverSelector.select(1) > 0) {
                        Set<SelectionKey> selectionKeys = serverSelector.selectedKeys();
                        Iterator<SelectionKey> keyIterator = selectionKeys.iterator();
                        while (keyIterator.hasNext()) {
                            SelectionKey selectionKey = keyIterator.next();
                            if (selectionKey.isAcceptable()) {
                                try {
                                    //(1)每来一个新连接不需要创建一个线程而是直接注册到clientSelector
                                    SocketChannel socketChannel = ((ServerSocketChannel) selectionKey.channel()).accept();
                                    socketChannel.configureBlocking(false);
                                    socketChannel.register(clientSelector, SelectionKey.OP_READ);
                                } finally {
                                    keyIterator.remove();
                                }
                            }
                        }
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }).start();

        new Thread(() -> {
            try {
                while (true) {
```

```
                    // (2)批量轮询是否有哪些连接有数据可读
                    if (clientSelector.select(1) > 0) {
                        Set<SelectionKey> selectionKeys = serverSelector.selectedKeys();
                        Iterator<SelectionKey> keyIterator = selectionKeys.iterator();
                        while (keyIterator.hasNext()) {
                            SelectionKey selectionKey = keyIterator.next();
                            if (selectionKey.isReadable()) {
                                try {
                                    SocketChannel socketChannel = (SocketChannel)
selectionKey.channel();
                                    ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
                                    //(3)读取数据以块为单位批量读取
                                    socketChannel.read(byteBuffer);
                                    byteBuffer.flip();

System.out.println(Charset.defaultCharset().newDecoder().decode(byteBuffer)
                                            .toString());
                                } finally {
                                    keyIterator.remove();
                                    selectionKey.interestOps(SelectionKey.OP_READ);
                                }
                            }
                        }
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

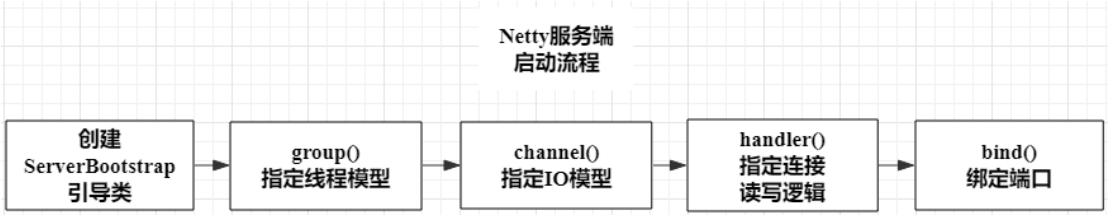
(3)Netty 编程

```
public class NettyServer {

    /**
     * 1.boss 对应 IOServer.java 中的接受新连接线程，主要负责创建新连接
     * 2.worker 对应 IOClient.java 中的负责读取数据的线程，主要用于读取数据以及业务
逻辑处理
     *
     * @param args
     */
    public static void main(String[] args) {
        ServerBootstrap serverBootstrap = new ServerBootstrap();
```

```
            NioEventLoopGroup bossGroup = new NioEventLoopGroup();
            NioEventLoopGroup workerGroup = new NioEventLoopGroup();

            serverBootstrap.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ch.pipeline().addLast(new StringDecoder());
                            ch.pipeline().addLast(new  SimpleChannelInboundHandler<String>()
{
                                @Override
                                protected void channelRead0(ChannelHandlerContext ctx,
String msg) throws Exception {
                                    System.out.println(msg);
                                }
                            });
                        }
                    }).bind(8000);
        }
}
public class NettyClient {

        public static void main(String[] args) throws InterruptedException {
            Bootstrap bootstrap = new Bootstrap();
            NioEventLoopGroup group = new NioEventLoopGroup();

            bootstrap.group(group).channel(NioSocketChannel.class).handler(new
ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new StringEncoder());
                }
            });

            Channel channel = bootstrap.connect("127.0.0.1", 8000).channel();
            while (true) {
                channel.writeAndFlush(new Date() + ": hello world!");
                Thread.sleep(2000);
            }
        }
}
```

3.服务端启动流程

要启动 Netty 服务端,必须要指定三类属性,分别是线程模型、IO 模型、连接读写处理逻辑.
Netty 服务端启动流程:创建引导类->指定线程模型、IO 模型、连接读写处理逻辑->绑定端口.



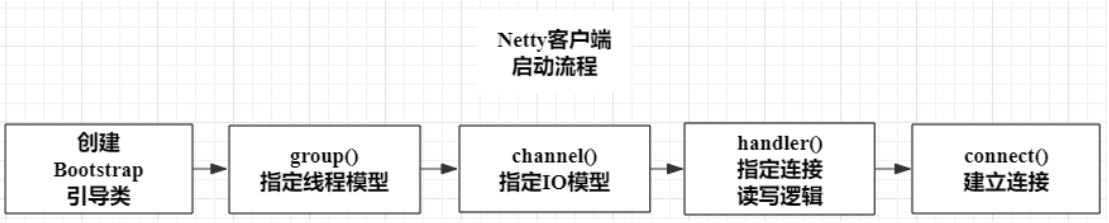

```
/**
 * 01: 服务端启动流程介绍[https://www.jianshu.com/p/ec3ebb396943]
 * 要启动Netty 服务端,必须要指定三类属性,分别是线程模型、IO 模型、连接读写处理逻辑
 * Netty 服务端启动的流程是创建引导类给引导类指定线程模型,IO 模型,连接读写处理逻辑,绑定端口之后服务端就启动起来
 * bind 方法是异步的通过异步机制来实现端口递增绑定
 * Netty 服务端启动额外的参数,主要包括给服务端 channel 或者 channel 设置属性值,设置底层 TCP 参数
 */
public class NettyServer {

    private static final int BEGIN_PORT = 8000;
    private static final AttributeKey<Object> SERVER_NAME_KEY = AttributeKey.newInstance("serverName");
    private static final String SERVER_NAME_VALUE = "nettyServer";
    public static final AttributeKey<Object> CLIENT_KEY = AttributeKey.newInstance("clientKey");
    public static final String CLIENT_VALUE = "clientValue";

    /**
     * 创建两个 NioEventLoopGroup,这两个对象可以看做是传统 IO 编程模型的两大线程组,boosGroup 表示监听端口,创建新连接的线程组,workerGroup 表示处理每一条连接的数据读写的线程组
     * 创建引导类 ServerBootstrap 进行服务端的启动工作,通过.group(boosGroup, workerGroup)给引导类配置两大线程定型引导类的线程模型指定服务端的 IO 模型为 NIO,通过.channel(NioServerSocketChannel.class)来指定 IO 模型
     * 调用 childHandler()方法给引导类创建 ChannelInitializer 定义后续每条连接的数据读写,业务处理逻辑,泛型参数 NioSocketChannel 是 Netty 对 NIO 类型的连接的抽象,而 NioServerSocketChannel 也是对 NIO 类型的连接的抽象
     * serverBootstrap.bind()是异步的方法调用之后是立即返回的,返回值是 ChannelFuture,给 ChannelFuture 添加监听器 GenericFutureListener,在 GenericFutureListener 的 operationComplete 方法里面监听端口是否绑定成功
     * childHandler()用于指定处理新连接数据的读写处理逻辑,handler()用于指定在服务端启动过程中的一些逻辑
```

```
        * attr()方法给服务端的 channel 即 NioServerSocketChannel 指定一些自定义属性,通过 channel.attr()取出该属性,给 NioServerSocketChannel 维护一个 map
        * childAttr()方法给每一条连接指定自定义属性,通过 channel.attr()取出该属性
        * childOption()方法给每条连接设置一些 TCP 底层相关的属性:
        * ChannelOption.SO_KEEPALIVE 表示是否开启 TCP 底层心跳机制,true 为开启
        * ChannelOption.SO_REUSEADDR 表示端口释放后立即就可以被再次使用,因为一般来说,一个端口释放后会等待两分钟之后才能再被使用
        * ChannelOption.TCP_NODELAY 表示是否开始 Nagle 算法,true 表示关闭,false 表示开启,通俗地说,如果要求高实时性,有数据发送时就马上发送,就关闭,如果需要减少发送次数减少网络交互就开启
        * option()方法给服务端 channel 设置一些 TCP 底层相关的属性:
        * ChannelOption.SO_BACKLOG 表示系统用于临时存放已完成三次握手的请求的队列的最大长度,如果连接建立频繁,服务器处理创建新连接较慢,适当调大该参数
        *
        * @param args
        */
    public static void main(String[] args) {
        NioEventLoopGroup bossGroup = new NioEventLoopGroup();
        NioEventLoopGroup workerGroup = new NioEventLoopGroup();

        ServerBootstrap serverBootstrap = new ServerBootstrap();
        serverBootstrap.group(bossGroup, workerGroup)
                .channel(NioServerSocketChannel.class)
                .handler(new ChannelInitializer<ServerSocketChannel>() {
                    @Override
                    protected void initChannel(ServerSocketChannel ch) throws Exception
{
                        System.out.println("服务端启动中");
                        System.out.println(ch.attr(SERVER_NAME_KEY).get());
                    }
                })
                .attr(SERVER_NAME_KEY, SERVER_NAME_VALUE)
                .option(ChannelOption.SO_BACKLOG, 1024)
                .childAttr(CLIENT_KEY, CLIENT_VALUE)
                .childOption(ChannelOption.SO_KEEPALIVE, true)
                .childOption(ChannelOption.SO_REUSEADDR, true)
                .childOption(ChannelOption.TCP_NODELAY, true)
                .childHandler(new ChannelInitializer<NioSocketChannel>() {
                    @Override
                    protected void initChannel(NioSocketChannel ch) throws Exception {
                        System.out.println(ch.attr(CLIENT_KEY).get());
                    }
                });
```

```
            bind(serverBootstrap, BEGIN_PORT);
        }

        private static void bind(ServerBootstrap serverBootstrap, int port) {
            serverBootstrap.bind(BEGIN_PORT).addListener(new  GenericFutureListener<Future<?
super Void>>() {
                    @Override
                    public void operationComplete(Future<? super Void> future) throws Exception {
                        if (future.isSuccess()) {
                            System.out.println("端口[" + port + "]绑定成功!");
                        } else {
                            System.err.println("端口[" + port + "]绑定失败!");
                            bind(serverBootstrap, port + 1);
                        }
                    }
                });
        }
}
```

4.客户端启动流程

要启动 Netty 客户端,必须要指定三类属性,分别是线程模型、IO 模型、连接读写处理逻辑.
Netty 客户端启动流程:创建引导类->指定线程模型、IO 模型、连接读写处理逻辑->建立连接.



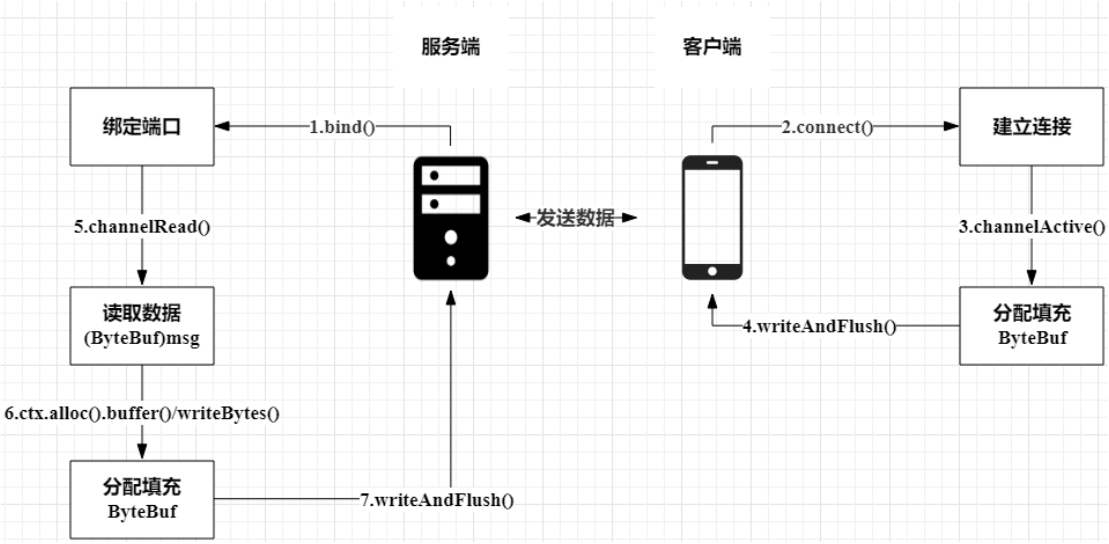
失败重连通过 connect()异步回调机制实现指数退避重连逻辑:

```
// 第几次重连
int order = (MAX_RETRY - retry) + 1;
// 本次重连的间隔
int delay = 1 << order;
bootstrap.config().group().schedule(() -> connect(bootstrap, host, port, retry - 1), delay,
TimeUnit.SECONDS);
```

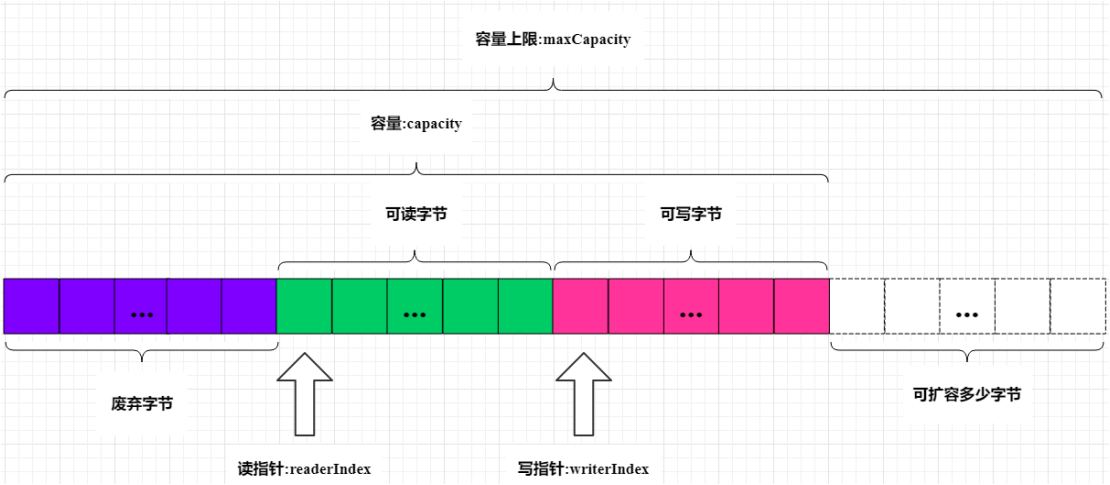
5.实战:客户端与服务端双向通信

客户端/服务端连接读写逻辑处理均是启动阶段通过给逻辑处理

链 Pipeline 添加逻辑处理器实现连接数据的读写逻辑.
客户端连接成功回调逻辑处理器 channelActive()方法,客户端/服务端接收连接数据调用 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传输载体为 ByteBuf,ByteBuf 通过连接的内存管理器创建即 ctx.alloc().buffer(),通过 writeBytes()方法将字节数据填充到 ByteBuf 写到对端.



6. 数据传输载体 ByteBuf 介绍



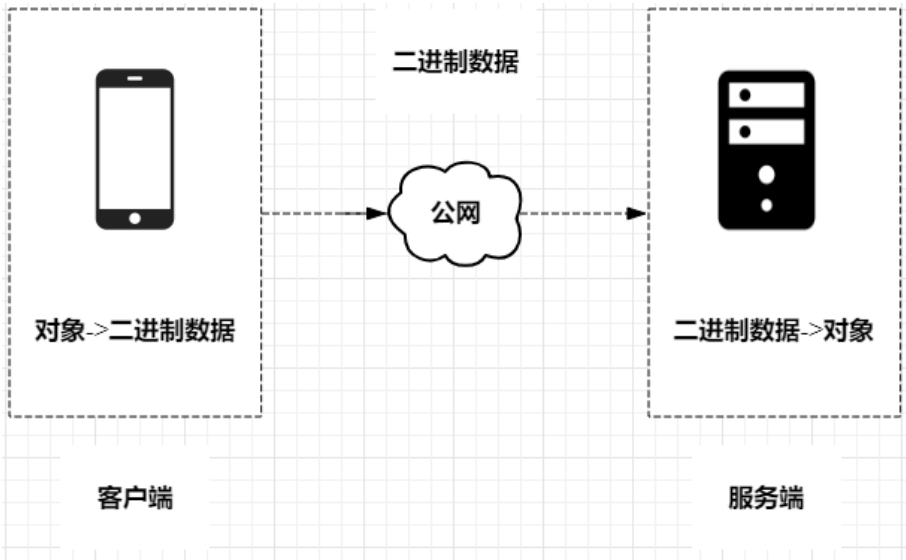
二进制数据抽象 ByteBuf 是字节容器,容器里面的的数据分为三个部分,第一个部分是已经丢弃的字节,这部分数据是无效的;第二部分是可读字节,这部分数据是 ByteBuf 的主体数据,从 ByteBuf 里面读取的数据都来自这一部分;最后一部分的数据是可写字节,所有写到 ByteBuf 的数据都会写到这一段.最后一部分虚线表示的是该 ByteBuf 最多还能扩容多少容量.这三部分内容是被两个指针给划分出来的,从

左到右依次是读指针(readerIndex)、写指针(writerIndex),然后还有容量 capacity 表示 ByteBuf 底层内存的总容量.
读数据是从 ByteBuf 里每读取一个字节,readerIndex 自增 1,ByteBuf 里面共有 writerIndex-readerIndex 个字节可读, 由此推论当 readerIndex 与 writerIndex 相等的时候,ByteBuf 不可读.
写数据是从 writerIndex 指向的部分开始写,每写一个字节,writerIndex 自增 1,直到增加至容量 capacity,此时表示 ByteBuf 不可写.
ByteBuf 里容量上限 maxCapacity 表示当向 ByteBuf 写数据时容量 capacity 不足进行扩容,直至扩容到最大容量 maxCapacity,超过 maxCapacity 报错.
推荐 API:markReaderIndex()&resetReaderIndex()/markWriterIndex()& resetWriterIndex():前者表示把当前的读/写指针保存起来,后者表示把当前的读/写指针恢复到之前保存的值,常见使用场景为解析自定义协议的数据包.
解析数据注意:get/set***()方法不会改变读写指针,read/write***()方法会改变读写指针.
Netty 使用堆外内存,堆外内存是不被 JVM 直接管理的,申请到的内存无法被垃圾回收器直接回收需要手动回收,即申请到的内存必须手工释放,否则造成内存泄漏.
ByteBuf 通过引用计数方式管理,如果 ByteBuf 没有地方被引用到,需要回收底层内存.默认情况下,当创建完 ByteBuf 其引用为 1,然后每次调用 retain()方法引用加 1,release()方法原理是将引用计数减 1,减完发现引用计数为 0 回收 ByteBuf 底层分配内存.
slice()/duplicate()/copy():
1.slice()方法从原始 ByteBuf 截取一段,这段数据是从 readerIndex 到 writeIndex,返回的新的 ByteBuf 的最大容量 maxCapacity 为原始 ByteBuf 的 readableBytes();
2.duplicate()方法把整个 ByteBuf 都截取出来包括所有的数据、指针信息;
3.slice()方法与 duplicate()方法的相同点是:底层内存以及引用计数与原始的 ByteBuf 共享即经过 slice()或者 duplicate()返回的 ByteBuf 调用 write 系列方法都会影响到原始的 ByteBuf,但是它们都维持着与原始 ByteBuf 相同的内存引用计数和不同的读写指针;
4.slice()方法与 duplicate()方法的不同点是:slice()只截取从 readerIndex 到 writerIndex 之间的数据,返回的 ByteBuf 最大容量被限制到原始 ByteBuf 的 readableBytes(), duplicate()是把整个 ByteBuf 都与原始的 ByteBuf 共享;
5.slice()方法与 duplicate()方法不会拷贝数据,它们只是通过改变读写

指针来改变读写的行为,copy()直接从原始的 ByteBuf 拷贝所有的信息包括读写指针以及底层对应的数据,因此往 copy()返回的 ByteBuf 写数据不会影响到原始的 ByteBuf;
6.slice()方法与 duplicate()方法不会改变 ByteBuf 的引用计数,所以原始的 ByteBuf 调用 release()之后发现引用计数为零开始释放内存,调用这两个方法返回的 ByteBuf 也会被释放,此时如果再对它们进行读写报错.因此通过调用一次 retain()方法增加引用,表示它们对应的底层的内存多一次引用,引用计数为 2,在释放内存的时候需要调用两次 release()方法将引用计数降到零才会释放内存;
7.slice()/duplicate()/copy()方法均维护着自身的读写指针,与原始的 ByteBuf 的读写指针无关,相互之间不受影响.
函数体里面只要增加引用计数包括 ByteBuf 的创建和手动调用 retain()方法必须调用 release()方法.多个 ByteBuf 引用同一段内存通过引用计数来控制内存的释放,遵循谁 retain()谁 release()的原则.

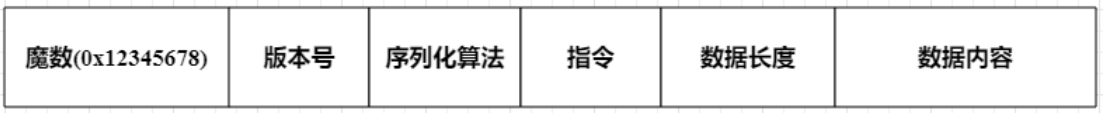
7. 客户端与服务端通信协议编解码

客户端与服务端的通信协议是客户端与服务端事先商量好的,每一个二进制数据包每一段字节分别代表什么含义的规则.
客户端与服务端的通信过程:首先客户端把 Java 对象按照通信协议转换成二进制数据包;然后通过网络把这段二进制数据包发送到服务端,数据的传输过程由 TCP/IP 协议负责数据的传输;服务端接收到数据按照协议截取二进制数据包的相应字段包装成 Java 对象交给应用逻辑处理;服务端处理完毕如果需要返回响应给客户端按照相同的流程进行.



通信协议的设计:第一个字段是魔数,通常情况下为固定的几个字节;

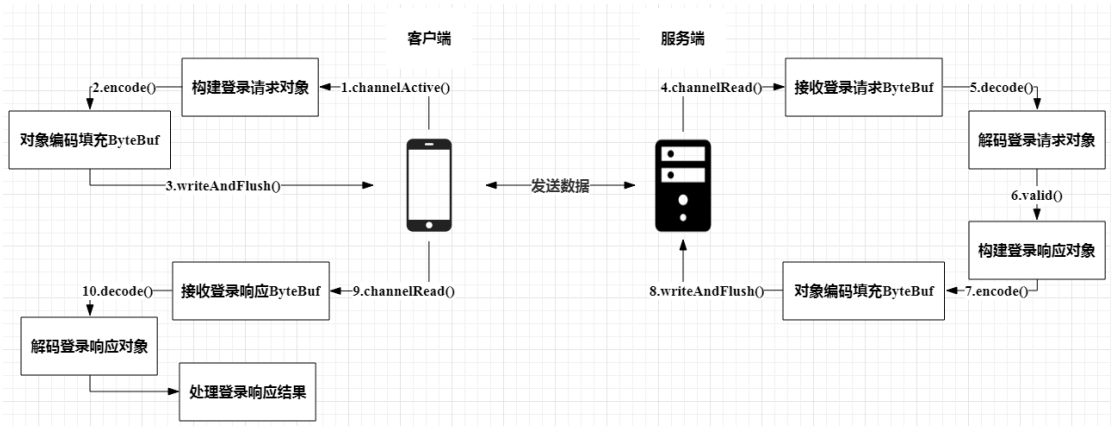
接下来一个字节为版本号,通常情况下是预留字段用于协议升级;第三部分序列化算法表示如何把 Java 对象转换二进制数据以及二进制数据如何转换回 Java 对象;第四部分的字段表示指令,服务端或者客户端每收到一种指令都有相应的处理逻辑;接下来的字段为数据部分的长度;最后一个部分为数据内容.



通信协议的实现:把 Java 对象根据协议封装成二进制数据包的过程称为编码;把从二进制数据包中解析出 Java 对象的过程称为解码.
编码(封装成二进制过程)流程:1.调用 ByteBuf 分配器 ByteBufAllocator 创建 ByteBuf 对象,ioBuffer()方法返回适配 io 读写相关的内存,尽可能创建直接内存,写到 IO 缓冲区的效果更高;2.把 Java 对象序列化成二进制数据包;3.按照通信协议逐个往 ByteBuf 对象写入字段即实现编码.
解码(解析 Java 对象过程)流程:1.调用 skipBytes()方法跳过魔数 0x12345678 字节;2.调用 ByteBuf 的 API 获取序列化算法标识、指令、数据包长度;3.根据获取的数据包长度取出数据,通过指令拿获取数据包对应的 Java 对象类型,根据序列化算法标识获取序列化对象,将字节数组转换为 Java 对象即实现解码.

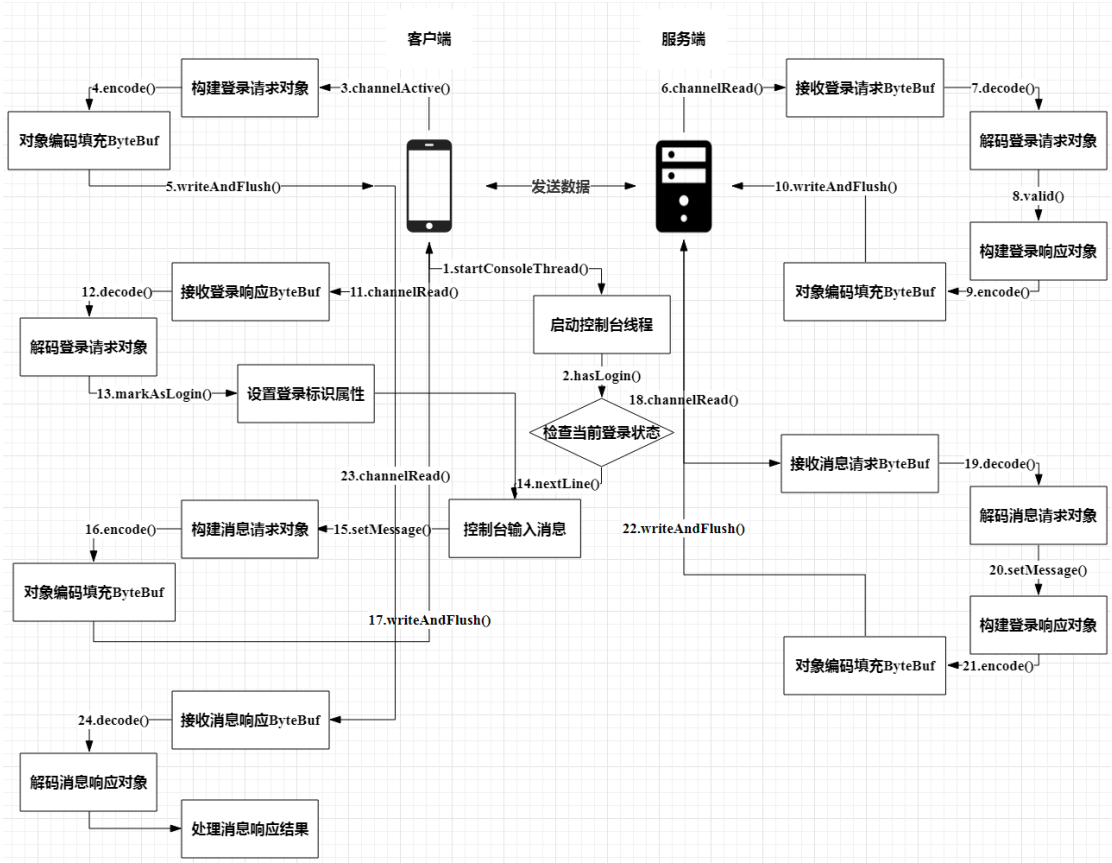
8. 实战:Netty 实现客户端登录

登录流程:客户端 channelActive()构建登录请求对象,通过 ctx.alloc()获取 ByteBuf 分配器,将登录请求对象编码填充到 ByteBuf,调用 ctx.channel()获取当前连接,writeAndFlush()把二进制数据写到服务端;服务端 channelRead()接收登录请求 ByteBuf 解码登录请求对象进行登录校验;服务端登录校验通过构造登录响应对象,登录响应对象编码 ByteBuf 写到客户端;客户端接收登录响应 ByteBuf 解码登录响应对象,判断是否登录成功处理登录响应即实现登录.



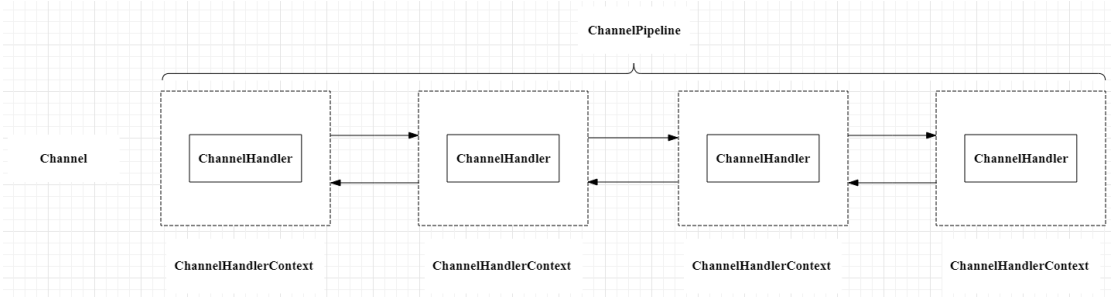
9. 实战:实现客户端与服务端收发消息

收发消息流程:客户端连接服务端启动控制台线程检查当前登录状态,登录成功通过 channel.attr().set()绑定 Channel 属性设置登录标识, 构建消息请求对象,通过 ctx.alloc()获取 ByteBuf 分配器,将消息请求对象编码填充到 ByteBuf,调用 ctx.channel()获取当前连接,writeAndFlush()把二进制数据写到服务端;服务端 channelRead()接收消息请求 ByteBuf 解码消息请求对象进行消息处理;服务端处理消息完毕通过构造消息响应对象, 消息响应对象编码 ByteBuf 写到客户端;客户端接收消息响应ByteBuf解码消息响应对象处理消息响应即实现收发消息.



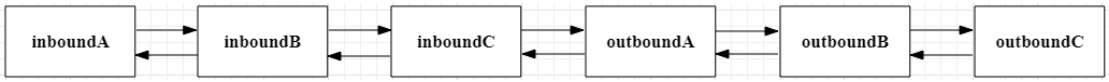
10. Pipeline 与 ChannelHandler

Pipeline 与 ChannelHandler 的构成:一条连接对应一个 Channel,Channel 处理逻辑在 ChannelPipeline 对象里面,ChannelPipeline 是双向链表结构,与 Channel 之间是一对一的关系.ChannelPipeline 里面每个节点是 ChannelHandlerContext 对象, ChannelHandlerContext 获取与 Channel 相关的上下文信息,包含逻辑处理器 ChannelHandler.

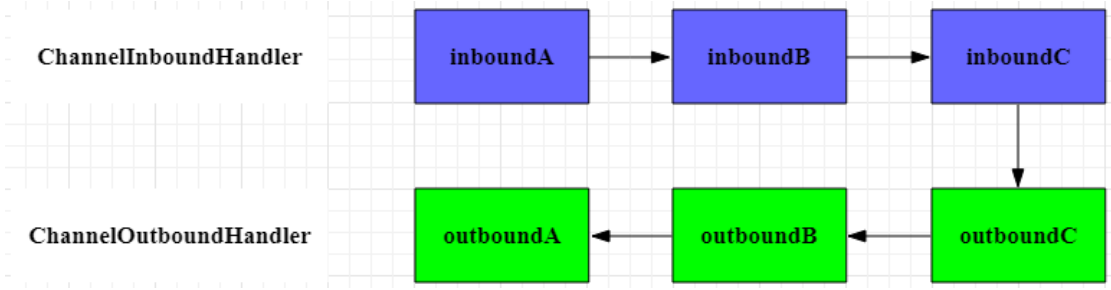


ChannelHandler 的分类:(1)ChannelInboundHandler 接口处理读数据逻

辑,定义组装响应前期处理逻辑,重要方法为 channelRead();(2)ChannelOutBoundHandler 接口处理写数据逻辑,定义组装响应后期把数据写到对端逻辑,核心方法为 write().ChannelInboundHandler/ChannelOutBoundHandler 接口默认实现为 ChannelInboundHandlerAdapter/ChannelOutBoundHandlerAdapter,默认情况下把读写事件传播到下一个 Handler. ChannelInboundHandler 事件传播顺序与通过 addLast()方法添加顺序相同. ChannelOutboundHandler 事件传播顺序与通过 addLast()方法添加顺序相反. ChannelHandler 以双向链表方式连接,实际链表的节点是 ChannelHandlerContext.



ChannelInboundHandler 事件通常只会传播到下一个 ChannelInboundHandler,ChannelOutboundHandler 事件通常只会传播到下一个 ChannelOutboundHandler,两者相互不受干扰.



11. 实战:构建客户端与服务端 Pipeline

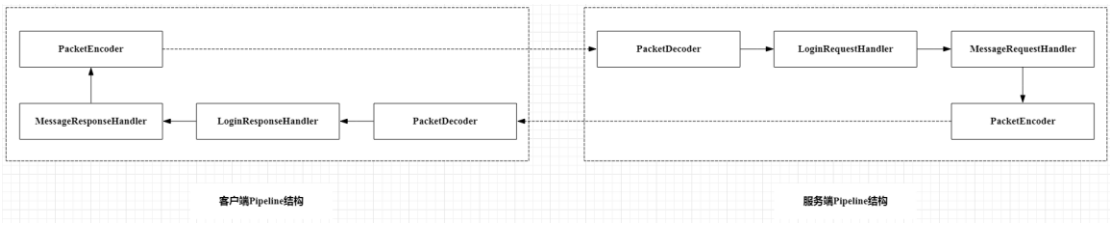
ChannelInboundHandlerAdapter/ChannelOutboundHandlerAdapter 适配器用于实现 ChannelInboundHandler/ChannelOutboundHandler 接口方法. 默认情况下 ChannelInboundHandlerAdapter 的 channelRead()方法通过 fireChannelRead()方法把上一个 Handler 的输出结果传递到下一

个 Handler, ChannelOutboundHandlerAdapter 的 write()方法把对象传递到下一个 OutBound 节点,传播顺序与 ChannelInboundHandler 相反. Pipeline 添加的第一个 Handler 的 channelRead()方法参数 msg 对象是 ByteBuf,服务端接收数据首先要做的第一步逻辑是把 ByteBuf 进行解码,然后把解码结果传递到下一个 Handler.

ByteToMessageDecoder 的 decode()方法参数 in 是 ByteBuf 类型,参数 out 是 List 类型,通过参数 out 添加解码结果对象实现结果往下一个 Handler 进行传递,不用关心 ByteBuf 强制转换和解码结果传递,自动进行内存释放.

继承 SimpleChannelInboundHandler 类传递泛型参数,channelRead0()方法不用通过 if 逻辑判断当前对象是否为本 Handler 能够处理的对象,不用强制转换当前对象,不用往下传递本 Handler 无法处理的对象,交给父类 SimpleChannelInboundHandler 实现,只需要专注于业务处理逻辑即可.

MessageToByteEncoder 将对象转换到二进制数据,不需要创建 ByteBuf,只需要实现 encode()方法自定义编码,encode()方法第 2 个参数是 Java 对象,第 3 个参数是 ByteBuf 对象,要做的事情是把 Java 对象字段写到 ByteBuf,不再需要手动分配 ByteBuf,不用关心 ByteBuf 创建,不用每次向对端写 Java 对象都进行一次编码.



12. 实战:拆包粘包理论与解决方案

拆包基本原理是不断从 TC 缓冲区读取数据,每次读取完毕需要判断是否为一个完整的数据包.(1)如果当前读取的数据不足以拼接成一个完整的业务数据包,保留此数据继续从 TCP 缓冲区读取,直到得到一个完整的数据.(2)如果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,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.

Netty 自带的拆包器:

1.固定长度的拆包器 FixedLengthFrameDecoder:如果应用层协议非常简单,每个数据包的长度都是固定的,需要把拆包器加到 Pipeline,Netty 把指定长度的数据包(ByteBuf)传递到下一个 ChannelHandler.

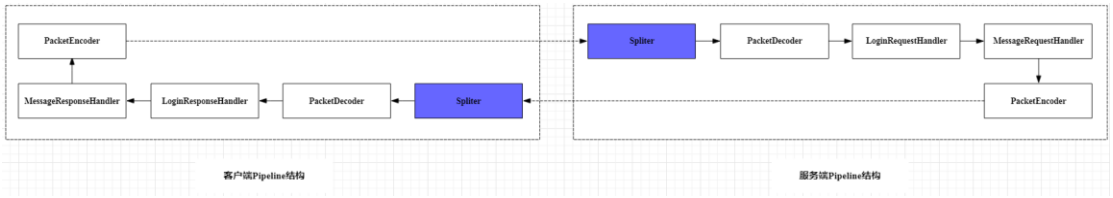
2.行拆包器 LineBasedFrameDecoder:发送端发送数据包的时候,每个数据包之间以换行符作为分隔,接收端通过 LineBasedFrameDecoder 将粘包的 ByteBuf 拆分成一个个完整的应用层数据包.

3.分隔符拆包器 DelimiterBasedFrameDecoder:行拆包器的通用版本,自定义分隔符.

4.基于长度域拆包器 LengthFieldBasedFrameDecoder:最通用的一种拆包器,只要自定义协议包含长度域字段均使用此拆包器实现应用层拆包.使用 LengthFieldBasedFrameDecoder 需要长度域相对整个数据包的偏移量和长度域的长度构造拆包器,Pipeline 最前面添加此拆包器.

拒绝非本协议连接:数据包的开头加上魔数尽早屏蔽非本协议的客户

端,通常放在第一个 Handler 处理此逻辑,每个客户端发过来的数据包都做一次快速判断,判断当前发来的数据包是否是满足自定义协议,需要继承 LengthFieldBasedFrameDecoder 覆盖 decode()方法,调用 decode()方法之前判断前四个字节是否是等于魔数 0x12345678. 基于 Netty 自带的拆包器在拆包之前判断当前连上来的客户端是否是支持自定义协议的客户端,如果不支持尽早关闭连接节省资源.



13. ChannelHandler 的生命周期

新建连接 ChannelHandler 回调方法执行顺序:

handlerAdded()->channelRegistered()->channelActive()->channelRead()->channelReadComplete().

handlerAdded():当检测到新连接后调用 ch.pipeline().addLast()方法回调,表示当前 Channel 成功添加 Handler 处理器.

channelRegistered():表示当前 Channel 逻辑处理和 Nio 线程 NioEventLoop 建立绑定关系,从线程池里面获取线程绑定在 Channel 上面.

channelActive():表示当前 Channel 业务逻辑处理链准备就绪即当前 Channel 的 Pipeline 添加所有 Handler 处理器完毕以及绑定 Nio 线程后,该连接真正激活回调到此方法.

channelRead():客户端向服务端发来数据,每次都会回调此方法,表示有数据可读.

channelReadComplete():服务端每次读完一次完整的数据后回调此方法,表示数据读取完毕.

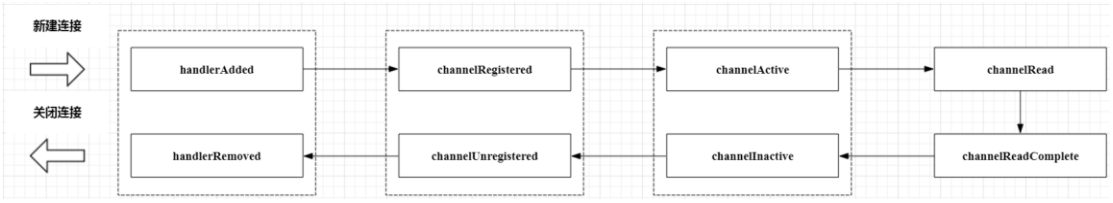
关闭连接 ChannelHandler 回调方法执行顺序:

channelInactive()->channelUnregistered()->handlerRemoved()

channelInactive():表示连接已经被关闭,此连接在 TCP 层面不再是 ESTABLISH 状态.

channelUnregistered():表示与此连接对应的 Nio 线程移除对此连接的处理.

handlerRemoved():表示把此连接添加的所有业务逻辑 Handler 处理器移除.



ChannelInitializer 实现原理: ChannelInitializer 定义 initChannel()抽象方法,服务端启动流程实现逻辑是往 Pipeline 里面添加 Handler 处理器链.handlerAdded()/channelRegistered()方法尝试调用 initChannel()方法,initChannel()使用 putIfAbsent()防止 initChannel()调用多次.

handlerAdded()/handlerRemoved()通常用于资源的申请和释放.

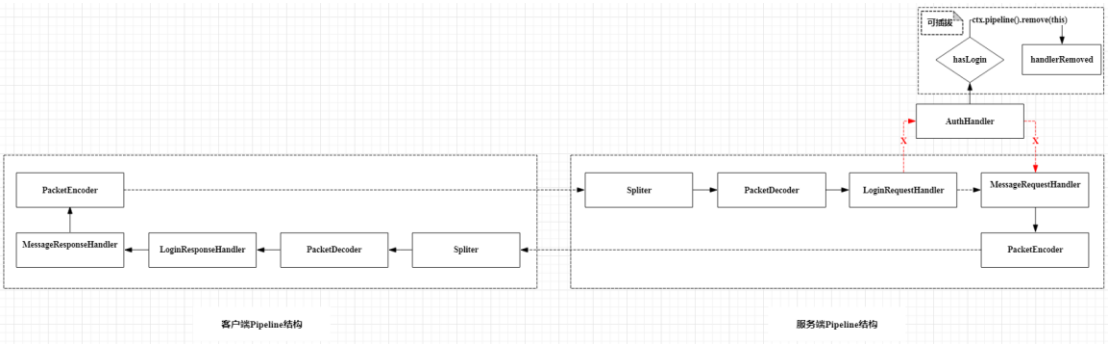
channelActive()/channelInActive()用于 TCP 连接的建立与释放统计单机的连接数,过滤客户端连接 IP 黑白名单.

channelRead()用于服务端根据自定义协议来进行拆包等.

channelReadComplete()用于调用 ctx.channel().flush()方法进行批量刷新.

14. 实战:使用 ChannelHandler 的热插拔实现客户端身份校验

身份校验处理器 AuthHandler 继承 ChannelInboundHandlerAdapter,覆盖 channelRead()方法决定是否把读到的数据传递到后续指令处理器之前,首先判断是否登录成功,如果未登录直接强制关闭连接,否则把读到的数据向下传递给后续指令处理器. AuthHandler 判断如果经过权限认证,调用 Pipeline 的 remove()方法删除自身,此客户端连接的逻辑链不再有该处理逻辑实现热插拔机制动态删除逻辑.



15. 实战:群聊消息的收发及 Netty 性能优化

共享 Handler:每次有新连接到来调用 ChannelInitializer 的 initChannel()方法, 每个指令处理器 ChannelHandler 内部没有成员变量即无状态时调用 pipeline().addLast()方法使用单例模式添加 ChannelHandler,不需要每次都 new 创建指令处理器对象,提高效率避免创建很多小的对象. 指令处理器 ChannelHandler 要被多个 Channel 进行共享必须要加上 @ChannelHandler.Sharable 注解显式告诉 Netty 此 ChannelHandler 支持多个 Channel 共享否则报错.

压缩 Handler-合并编解码器:继承 MessageToMessageCodec 覆盖

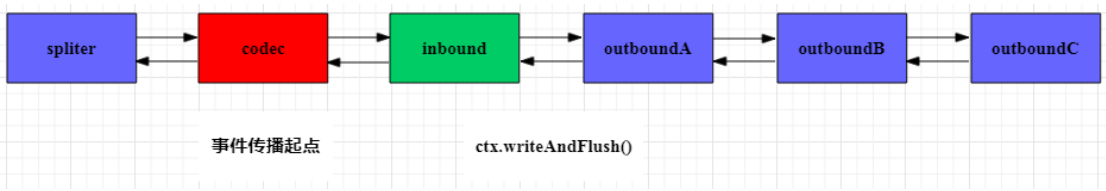
decode()/encode()方法实现编/解码操作.

缩短事件传播路径:

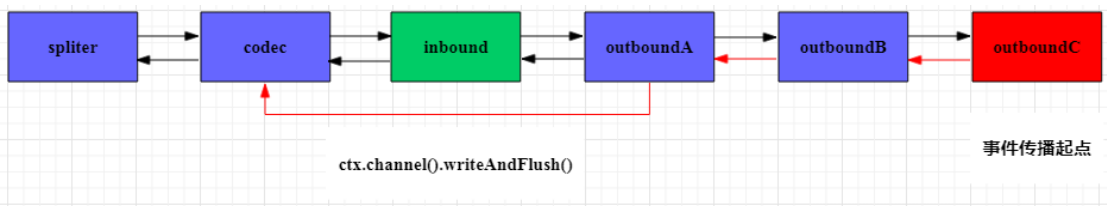
1.压缩 Handler-合并平行 ChannelHandler:平行指令处理器 ChannelHandler 压缩到一个指令处理器 ChannelHandler, 此指令处理器维护指令到各个指令处理器的映射 Map, 每次回调此指令处理器的 channelRead0()方法通过指令寻找指令处理器 ChannelHandler 调用其 channelRead()方法,最终调用每个指令处理器 ChannelHandler 的 channelRead0()方法.

2.更改事件传播源: 如果 OutBound 类型的 ChannelHandler 较多,写数据使用 ctx.writeAndFlush()减短事件传播路径.

ctx.writeAndFlush()是从 Pipeline 链的当前节点开始往前找到第一个 OutBound 类型的 ChannelHandler 把对象往前进行传播,如果此对象确认不需要经过其他 OutBound 类型的 ChannelHandler 处理则使用此方法.



ctx.channel().writeAndFlush()是从 Pipeline 链的最后一个 OutBound 类型的 ChannelHandler 开始把对象往前进行传播,如果确认当前创建的对象需要经过后面 OutBound 类型的 ChannelHandler 则调用此方法.



减少阻塞主线程操作:需要把耗时操作丢到自定义业务线程池处理,Nio 线程会有很多 Channel 共享不能阻塞.

如何准确统计处理时长:writeAndFlush()方法返回 ChannelFuture 对象添加监听器 Listener,回调方法监听 writeAndFlush()执行结果进而执行其他逻辑最后统计耗时.Netty 里面很多方法都是异步操作,业务线程统计操作耗时需要使用监听器回调方式统计,Nio 线程调用直接统计操作耗时即可.

16. 心跳与空闲检测

连接假死现象:在某一端(服务端或者客户端)看来底层 TCP 连接已经断开,但是应用程序并没有捕获到,因此认为这条连接仍然是存在的,从 TCP 层面来说,只有收到四次握手数据包或者一个 RST 数据包,连接的状态表示已断开.

连接假死问题:对于服务端来说,因为每条连接都耗费 CPU 和内存资源,大量假死的连接逐渐耗光服务器的资源,最终导致性能逐渐下降,程序奔溃;对于客户端来说,连接假死造成发送数据超时,影响用户体验.

连接假死原因:1.应用程序出现线程堵塞,无法进行数据读写;2.客户端或者服务端网络相关设备出现故障,比如网卡,机房故障;3.公网丢包,公网环境相对内网而言容易出现丢包、网络抖动等现象,如果在一段时间内用户接入网络连续出现丢包现象,则对客户端来说数据一直发送不出去,服务端接收不到客户端的数据,连接一直耗着.

服务端空闲检测:服务端对于连接假死的应对策略是空闲检测.空闲检测是指每隔一段时间检测这段时间内是否有数据读写,通过IdleStateHandler 回调 channelIdle()方法关闭连接实现空闲检测.

客户端定时发心跳:客户端通过 executor() 获取当前 Channel 绑定的 Nio 线程调用 scheduleAtFixedRate()方法每隔一段时间定期发送心跳数据包到服务端. 通常空闲检测时间要比发送心跳时间的两倍要长一些,排除偶发的公网抖动防止误判.

服务端回复心跳与客户端空闲检测: 客户端向服务端定期发送心跳,服务端接收心跳回复客户端,给客户端发送心跳响应包即可.如果在一段时间之内客户端没有接收服务端发来的数据判定连接为假死状态.