目录

[1.仿微信IM系统简介 1](#_Toc527145435)

[2.Netty是什么? 2](#_Toc527145436)

[3.服务端启动流程 8](#_Toc527145437)

[4.客户端启动流程 11](#_Toc527145438)

[5.实战:客户端与服务端双向通信 11](#_Toc527145439)

[6.数据传输载体ByteBuf介绍 12](#_Toc527145440)

[7.客户端与服务端通信协议编解码 14](#_Toc527145441)

[8.实战:Netty实现客户端登录 15](#_Toc527145442)

[9.实战:实现客户端与服务端收发消息 16](#_Toc527145443)

[10.Pipeline与ChannelHandler 17](#_Toc527145444)

[11.实战:构建客户端与服务端 Pipeline 18](#_Toc527145445)

[12.实战:拆包粘包理论与解决方案 20](#_Toc527145446)

[13.ChannelHandler的生命周期 21](#_Toc527145447)

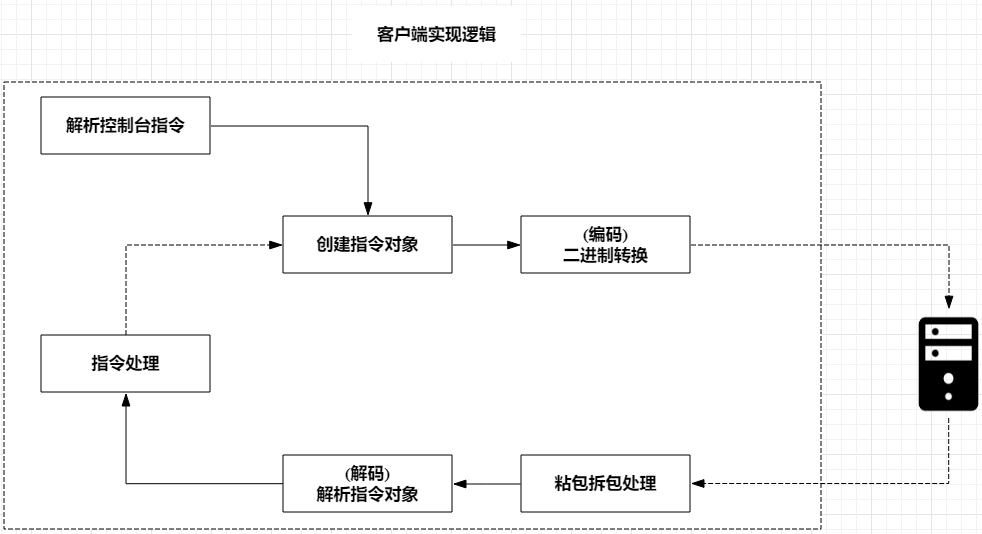
[14.实战:使用ChannelHandler的热插拔实现客户端身份校验 23](#_Toc527145448)

[15.实战:群聊消息的收发及Netty性能优化 23](#_Toc527145449)

[16.心跳与空闲检测 25](#_Toc527145450)

# 1.仿微信IM系统简介

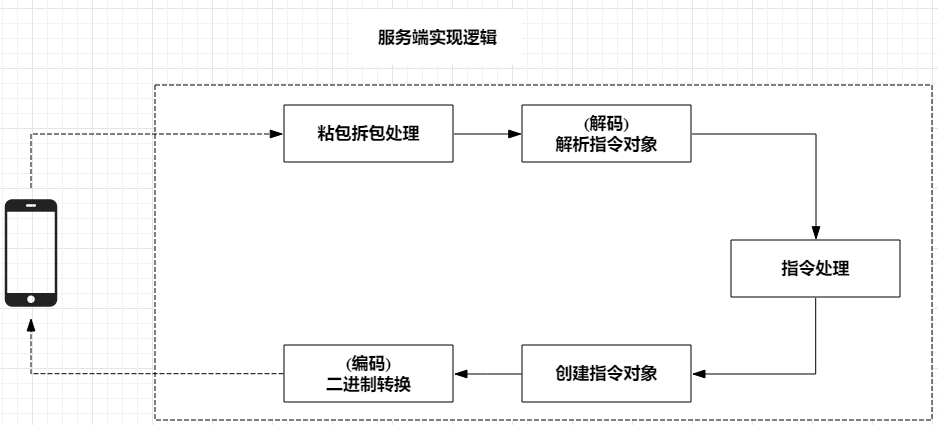
(1)客户端使用Netty程序实现逻辑



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.

1. 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:

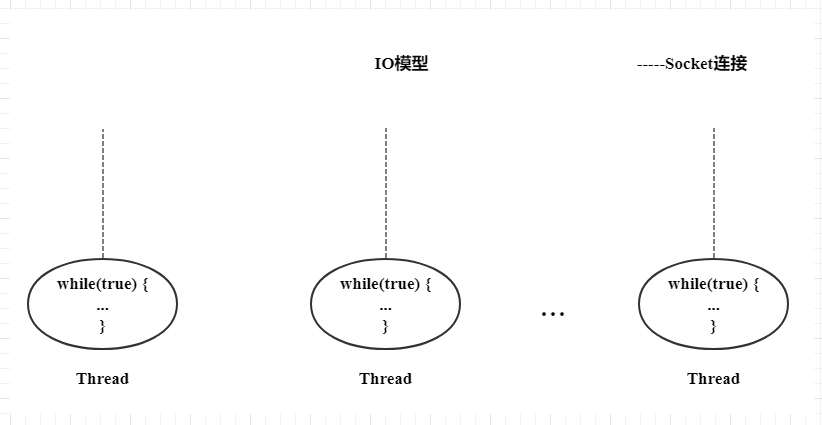
1.线程资源受限:线程是操作系统中非常宝贵的资源,同一时刻有大量的线程处于阻塞状态是非常严重的资源浪费,操作系统耗不起;

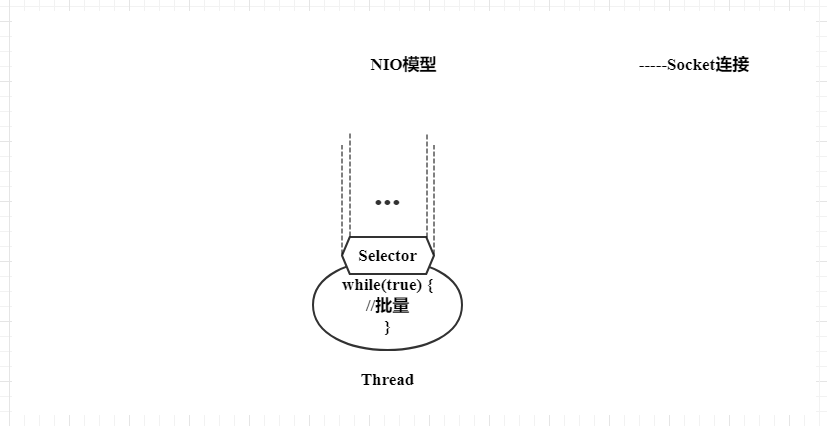
2.线程切换效率低下:单机CPU核数固定,线程爆炸之后操作系统频繁进行线程切换,应用性能急剧下降;

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

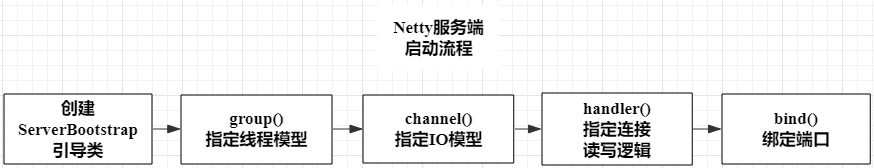
**public class** NettyServer {  
  
 */\*\*  
 \* 1.boss对应,IOServer.java中的接受新连接线程,主要负责创建新连接  
 \* 2.worker对应 IOClient.java中的负责读取数据的线程,主要用于读取数据以及业务逻辑处理  
 \*  
 \** ***@param args*** *\*/* **public static void** main(String[] args) {  
 ServerBootstrap serverBootstrap = **new** ServerBootstrap();  
 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

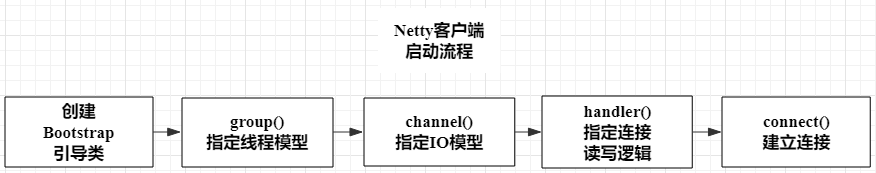


*/\*\*  
 \* 01: 服务端启动流程介绍[https://www.jianshu.com/p/ec3ebb396943]  
 \* 要启动Netty服务端,必须要指定三类属性,分别是线程模型、IO模型、连接读写处理逻辑  
 \* Netty服务端启动的流程是创建引导类给引导类指定线程模型,IO模型,连接读写处理逻辑,绑定端口之后服务端就启动起来  
 \* bind方法是异步的通过异步机制来实现端口递增绑定  
 \* Netty服务端启动额外的参数,主要包括给服务端channel或者channel设置属性值,设置底层TCP参数  
 \*/***public class** NettyServer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

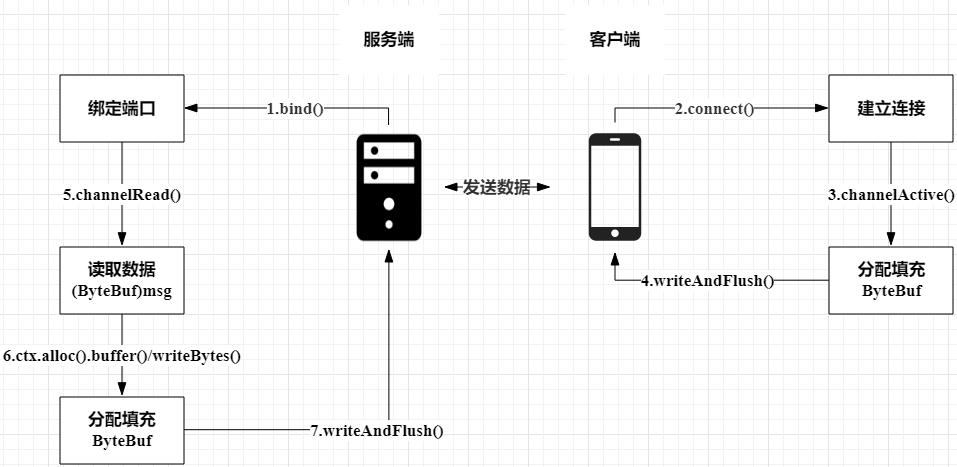
bootstrap.config().group().schedule(() -> *connect*(bootstrap, host, port, retry - 1), delay, TimeUnit.***SECONDS***);

# 5.实战:客户端与服务端双向通信

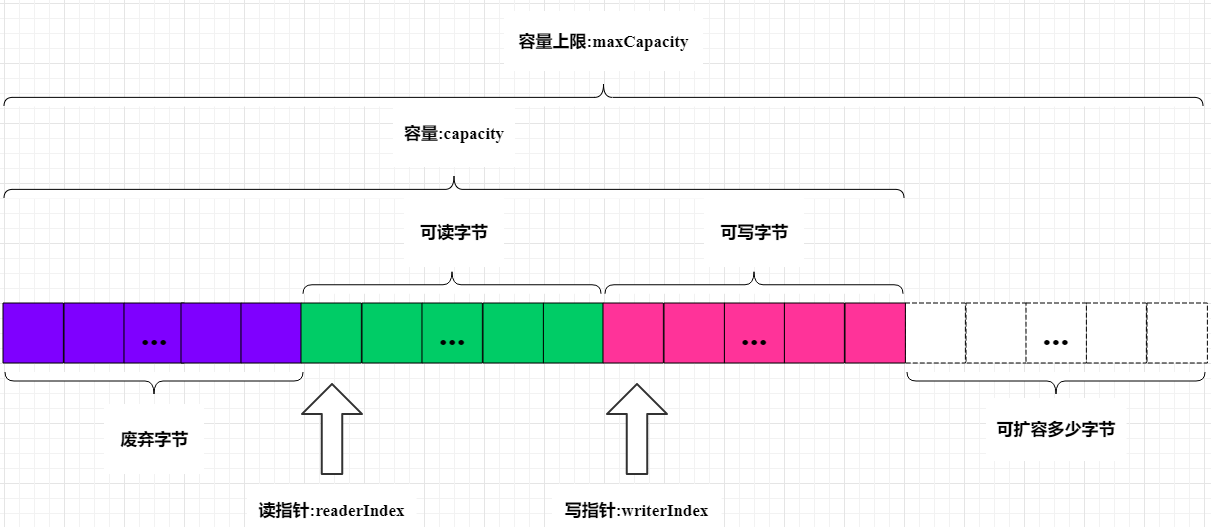
客户端/服务端连接读写逻辑处理均是启动阶段通过给逻辑处理链 Pipeline 添加逻辑处理器实现连接数据的读写逻辑.

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.

****

# 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;

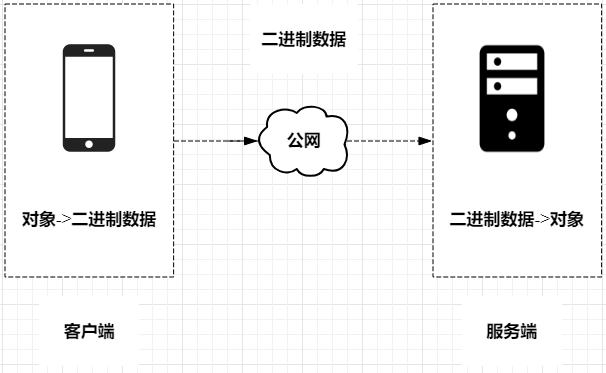
7.slice()/duplicate()/copy()方法均维护着自身的读写指针,与原始的 ByteBuf的读写指针无关,相互之间不受影响.

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.

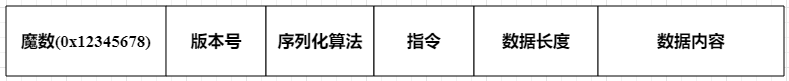
# 客户端与服务端通信协议编解码

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.

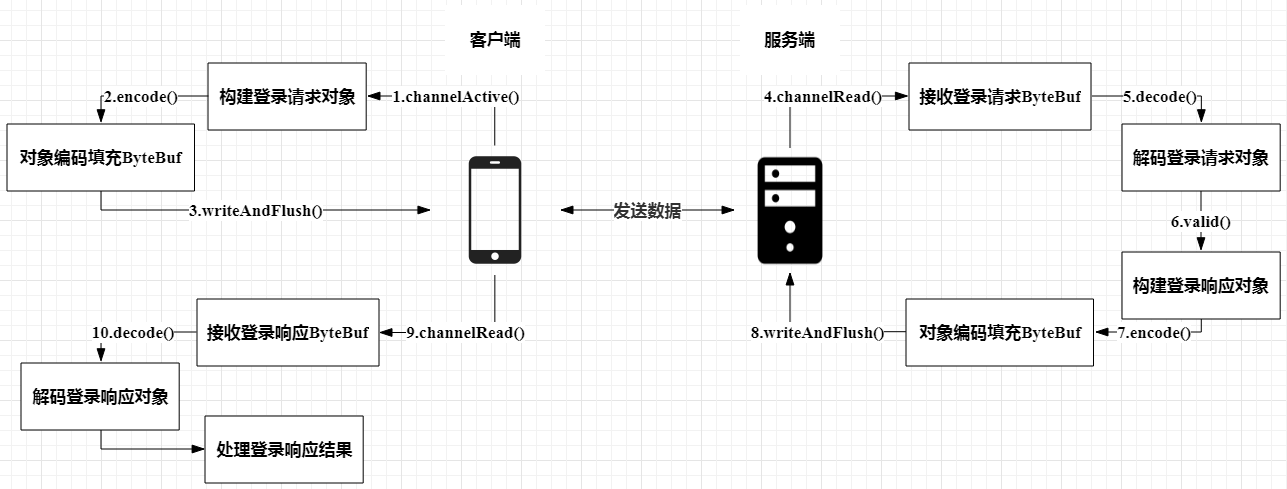
编码(封装成二进制过程)流程:1.调用ByteBuf分配器ByteBufAllocator创建ByteBuf对象,ioBuffer()方法返回适配io读写相关的内存,尽可能创建直接内存,写到 IO 缓冲区的效果更高;2.把Java对象序列化成二进制数据包;3.按照通信协议逐个往ByteBuf对象写入字段即实现编码.

解码(解析Java对象过程)流程:1.调用skipBytes()方法跳过魔数 0x123

45678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.

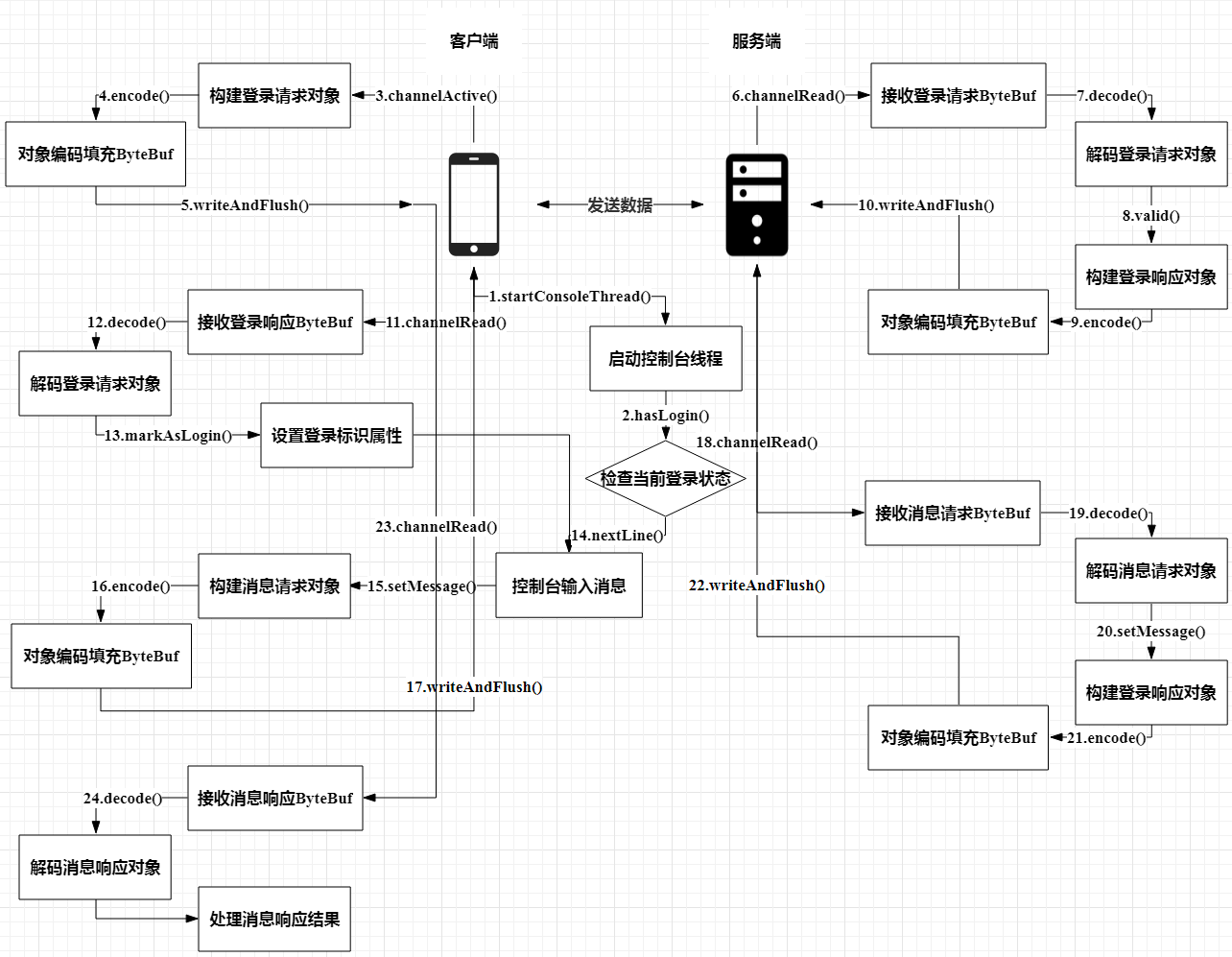
# 实战:Netty实现客户端登录

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.



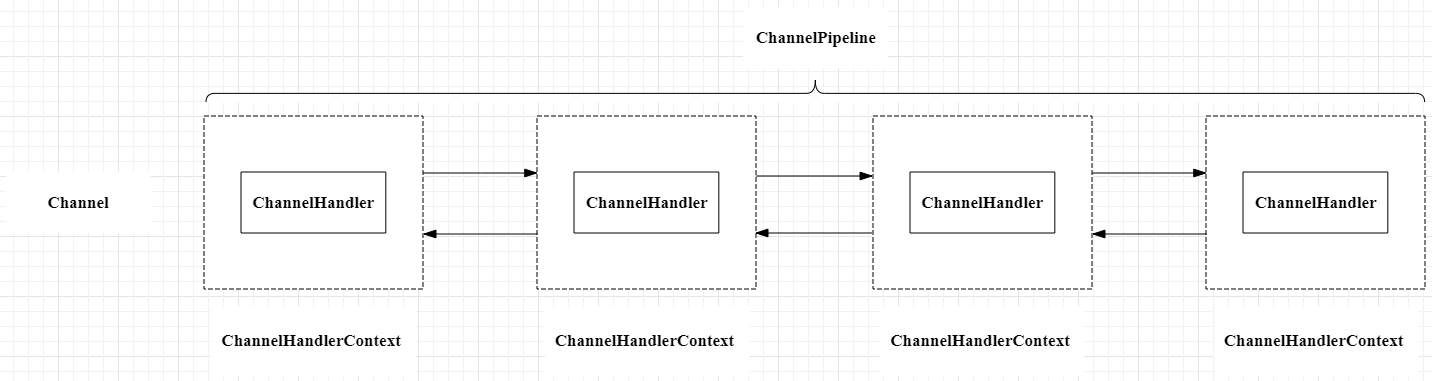
# 实战:实现客户端与服务端收发消息

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.



# 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.



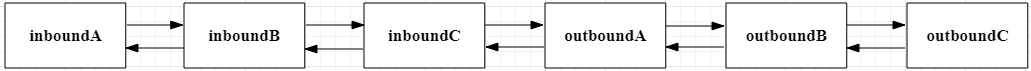
ChannelHandler的分类:(1)ChannelInboundHandler接口处理读数据逻辑,定义组装响应前期处理逻辑,重要方法为channelRead();(2)Channel

OutBoundHandler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/ChannelOutBo

undHandler接口默认实现为ChannelInboundHandlerAdapter/ChannelO

utBoundHandlerAdapter,默认情况下把读写事件传播到下一个Handler.

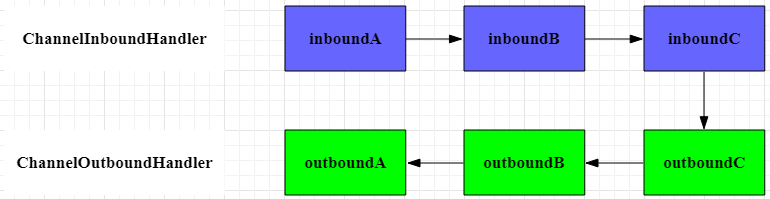
ChannelInboundHandler事件传播顺序与通过addLast()方法添加顺序相同. ChannelOutboundHandler事件传播顺序与通过addLast()方法添加顺序相反. ChannelHandler以双向链表方式连接,实际链表的节点是ChannelHandlerContext.



ChannelInboundHandler事件通常只会传播到下一个ChannelInboundH

andler,ChannelOutboundHandler事件通常只会传播到下一个ChannelO

utboundHandler,两者相互不受干扰.



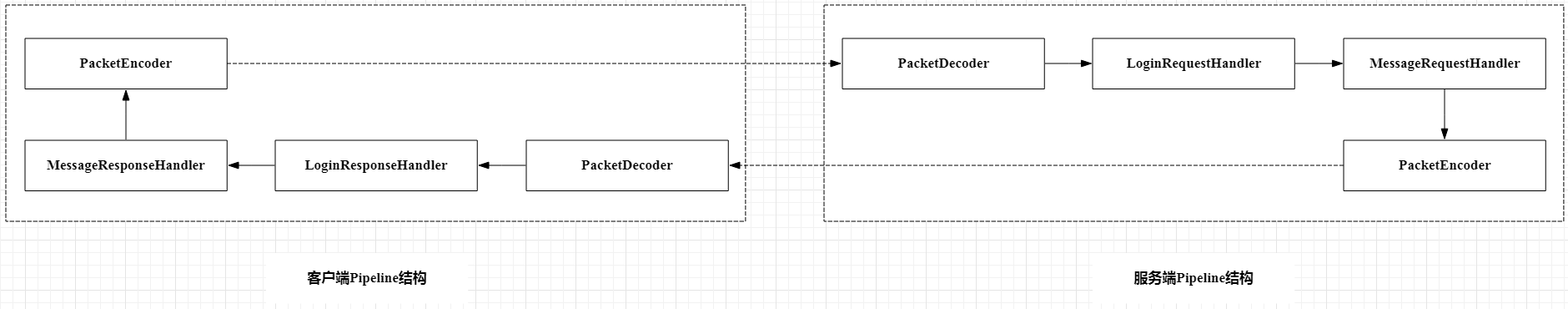
# 实战:构建客户端与服务端 Pipeline

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.



# 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:

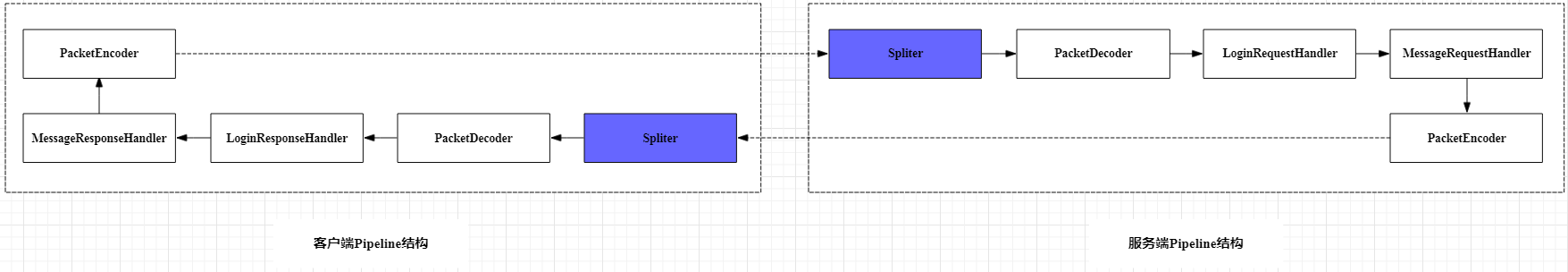
1.固定长度的拆包器FixedLengthFrameDecoder:如果应用层协议非常简单,每个数据包的长度都是固定的,需要把拆包器加到Pipeline,Netty把指定长度的数据包(ByteBuf)传递到下一个ChannelHandler.

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.



# 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NioEventL

oop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.

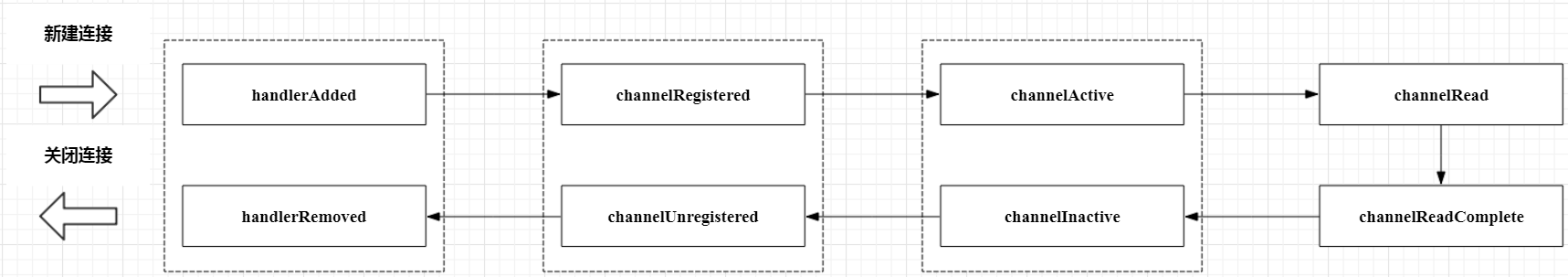
关闭连接ChannelHandler回调方法执行顺序:

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.

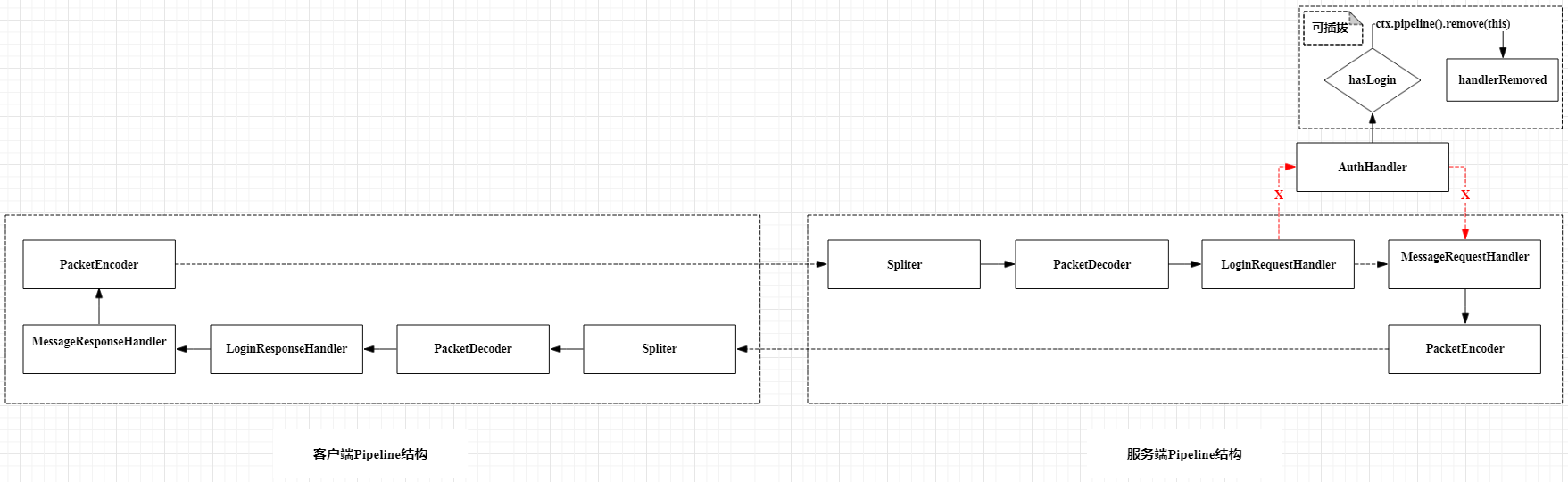
channelActive()/channelInActive()用于TCP连接的建立与释放统计单机的连接数,过滤客户端连接IP黑白名单.

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.

# 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.



# 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.

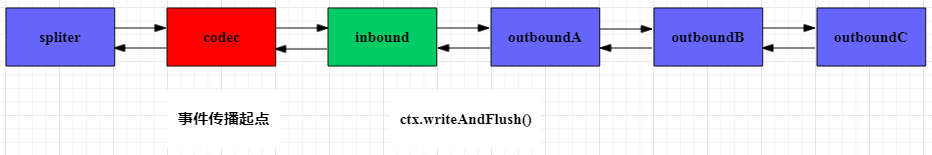
缩短事件传播路径:

1.压缩Handler-合并平行ChannelHandler:平行指令处理器ChannelHan

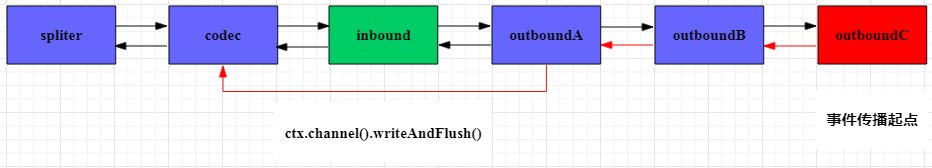
dler压缩到一个指令处理器ChannelHandler, 此指令处理器维护指令到各个指令处理器的映射Map,每次回调此指令处理器的channelRead0()方法通过指令寻找指令处理器ChannelHandler调用其channelRead()方法,最终调用每个指令处理器ChannelHandler的channelRead0()方法.

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.

# 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.