1. 什么是Netty
   1. Netty简介

netty是一个高性能异步事件驱动的网络开发框架，它基于java NIO提供的API实现。传统的NIO类库和api是非常繁杂的，需要熟练掌握Selector、ServerSocketChannel、ByteBuffer等，还需要处理断线重连、网络闪断、心跳处理、半包读写、网络拥塞等一场情况，开发难度较大，而Netty对JDK自带的NIO进行了良好封装，简化了开发难度，且Netty拥有高性能、高吞吐、低延迟、低资源消耗等优点

Netty常见的有3.x、4.x和5.x这几个版本，但是截至2022年，广泛使用的仍是Netty4

* 1. Netty使用场景

互联网行业：在微服务系统中，各节点之间的通讯需要高性能RPC框架，而Netty作为高性能通讯框架，往往被用于开发RPC框架，比如Dubbo、RocketMq等

游戏行业：无论是手游还是大型端游，往往都离不开Netty的支持

大数据领域：大数据领域也大量使用Netty作为高性能网络框架，如Spark、Flink、Hadoop等

* 1. Netty快速开始

使用Netty，首先我们需要引入依赖包：

<dependency>  
 <groupId>io.netty</groupId>  
 <artifactId>netty-all</artifactId>  
 <version>4.1.65.Final</version>  
</dependency>

我们通过一个小例子快速开始，这个例子由4个类组成，NettyClient是客户端，NettyClientHandler是客户端处理器，NettyServer是服务端，NettyServerHandler是服务端处理器



public class NettyServer {  
 public static void main(String[] args) throws Exception {  
 //创建两个线程组bossGroup和workerGroup,含有的子线程NioEventLoop的个数默认为cpu核数的两倍,可以简单理解为两个线程池  
 //bossGroup只是处理连接请求，真正的和客户端业务处理，会交给workerGroup完成

//bossGroup相当于主从reactor模型中的mainReactor，workerGroup相当于subReacotr  
 EventLoopGroup bossGroup = new NioEventLoopGroup(1);  
 EventLoopGroup workerGroup = new NioEventLoopGroup(8);  
 try {  
 //创建服务器端的启动对象  
 ServerBootstrap bootstrap = new ServerBootstrap();  
 //使用链式编程来配置参数  
 bootstrap.group(bossGroup, workerGroup) //设置两个线程组  
 //使用NioServerSocketChannel作为服务器的通道实现  
 .channel(NioServerSocketChannel.class)  
 //初始化服务器连接队列大小，服务端处理客户端连接请求是顺序处理的，所以同一时间只能处理一个客户端连接  
 //多个客户端同时来的时候，服务端将不能处理的客户端连接请求放在队列中等待处理  
 .option(ChannelOption.*SO\_BACKLOG*, 1024)  
 .childHandler(new ChannelInitializer<SocketChannel>() {  
 @Override  
 protected void initChannel(SocketChannel socketChannel) throws Exception {  
 //对workerGroup 的Socket Channel 设置处理器  
 socketChannel.pipeline().addLast(new NettyServerHandler());  
 }  
 });  
 System.*out*.println("netty server start 。");  
 //绑定一个端口并且同步，生成了一个ChaimelFuture异步对象，通过isDone ()等方法可以判断异步事件的执行情况  
 //启动服务器(并绑定端口)，bind是异步操作，sync方法是等待异步操作执行完毕  
 ChannelFuture cf = bootstrap.bind(9000).sync();  
 //注册监听器  
 cf.addListener(new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture channelFuture) throws Exception {  
 if (cf.isSuccess()) {  
 System.*out*.println("监听9000端口成功");  
 } else {  
 System.*out*.println("监听9000端口失败");  
 }  
 }  
 });  
 //等待服务端监听端口关闭，closeFuture是异步操作  
 //通过sync同步方法同步等待通道关闭处理完毕，这里会阻塞直到通道关闭完成，内部调用的是Object的wait()  
 cf.channel().closeFuture().sync();  
 } finally {  
 bossGroup.shutdownGracefully();  
 workerGroup.shutdownGracefully();  
 }  
 }  
}

public class NettyServerHandler extends ChannelInboundHandlerAdapter {  
 */\*\*  
 \* 读取客户端发送的消息  
 \** ***@param*** *ctx  
 \** ***@param*** *msg  
 \** ***@throws*** *Exception  
 \*/* @Override  
 public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
 System.*out*.println("服务端读取线程 = " + Thread.*currentThread*().getName());  
 ByteBuf buf = (ByteBuf) msg;  
 System.*out*.println("服务端收到来自客户端的消息 = " + buf.toString(CharsetUtil.*UTF\_8*));  
 }  
  
 */\*\*  
 \* 数据读取完毕后调用该方法  
 \** ***@param*** *ctx  
 \** ***@throws*** *Exception  
 \*/* @Override  
 public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {  
 ByteBuf buf = Unpooled.*copiedBuffer*("hello，我收到了你的消息".getBytes(CharsetUtil.*UTF\_8*));  
 ctx.writeAndFlush(buf);  
 }  
}

public class NettyClient {  
 public static void main(String[] args) throws Exception {  
 //客户端需要一个事件循环组  
 EventLoopGroup group = new NioEventLoopGroup();  
 try {  
 //创建客户端启动对象  
 //注意客户端使用的不是ServerBoot strap而是Boot strap  
 Bootstrap bootstrap = new Bootstrap();  
 //设置相关参数  
 bootstrap.group(group) //设置线程组  
 .channel(NioSocketChannel.class) // 使用\ioSocketChannel作为客户端的通道实现  
 .handler((new ChannelInitializer<SocketChannel>() { //加入处理器  
 @Override  
 protected void initChannel(SocketChannel socketChannel) throws Exception {  
 //对workerGroup 的Socket Channel 设置处理器  
 socketChannel.pipeline().addLast(new NettyClientHandler());  
 }  
 }));  
 System.*out*.println("netty client start。。。");  
 //启动客户端去连接服务器端  
 ChannelFuture cf = bootstrap.connect("127.0.0.1", 9000).sync();  
 //对通道关闭进行监听  
 cf.channel().closeFuture().sync();  
 } finally {  
 group.shutdownGracefully();  
 }  
 }  
}

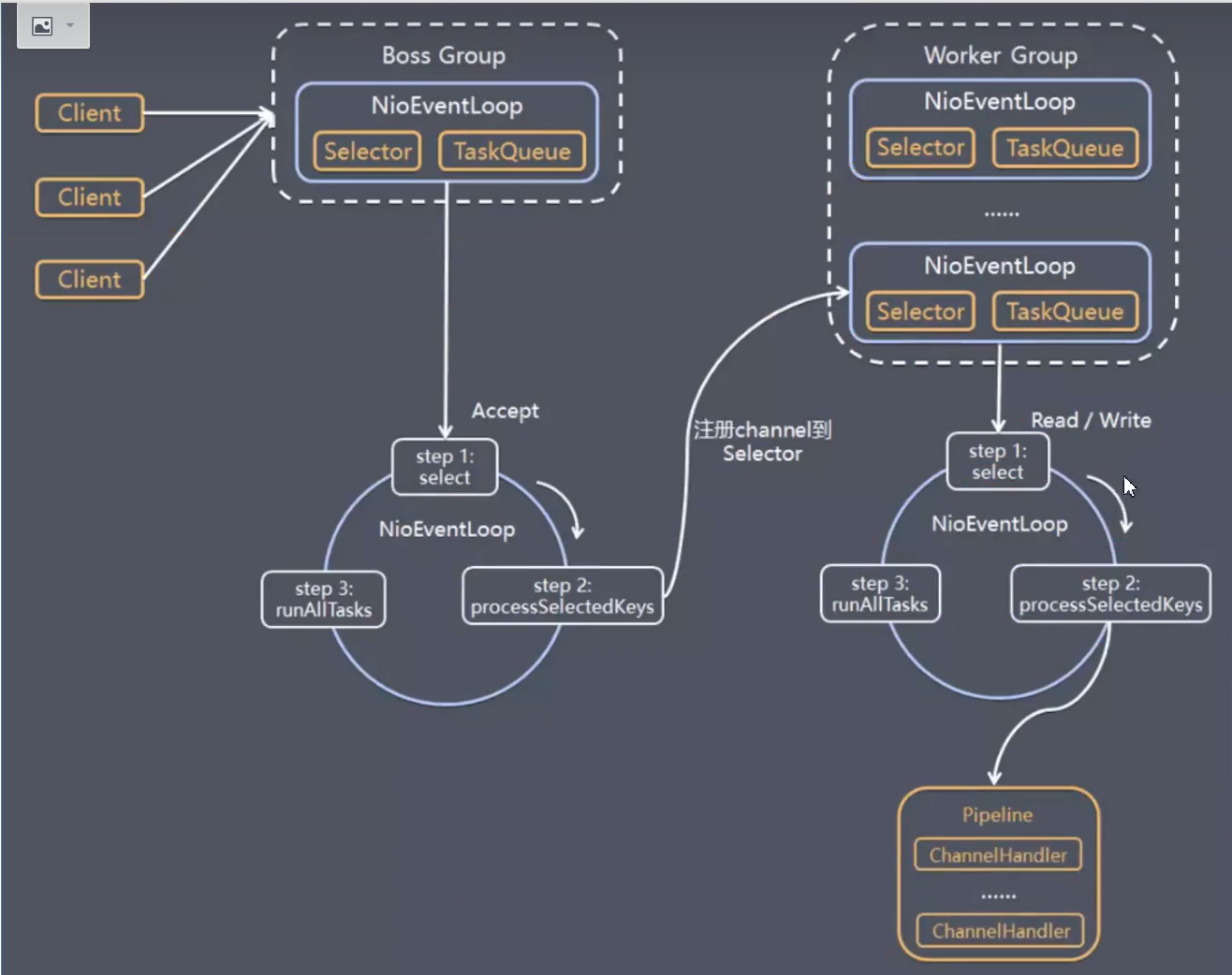
public class NettyClientHandler extends ChannelInboundHandlerAdapter {  
 */\*\*  
 \* 客户端连接服务器完成后触发此方法  
 \** ***@param*** *ctx  
 \** ***@throws*** *Exception  
 \*/* @Override  
 public void channelActive(ChannelHandlerContext ctx) throws Exception {  
 System.*out*.println("客户端连接服务器成功。。。" );  
 ByteBuf buf = Unpooled.*copiedBuffer*("hello，客户端".getBytes(CharsetUtil.*UTF\_8*));  
 ctx.writeAndFlush(buf);  
 }  
  
 */\*\*  
 \* 读取服务端发送的消息  
 \** ***@param*** *ctx  
 \** ***@param*** *msg  
 \** ***@throws*** *Exception  
 \*/* @Override  
 public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {  
 ByteBuf buf = (ByteBuf) msg;  
 System.*out*.println("客户端收到来自服务端的消息 = " + buf.toString(CharsetUtil.*UTF\_8*));  
 }  
  
}

* 1. Reactor模型

参考《java se学习笔记》

1. Netty模块组件
   1. Netty架构

Netty整体是基于主从Reactor模型来构建的，其具体结构如下：



﻿

1. Netty抽象出两组线程池BossGroup和WorkGroup，他们都是NioEventLoopGroup类型，其中BossGroup专门负责接收客户端的新连接，WorkerGroup专门负责处理读写事件
2. NioEventLoopGroup相当于一个事件循环线程组，这个组中含有多个事件循环线程，每一个单独的事件循环线程都是NioEventLoop
3. 每个NioEventLoop都有一个selector，用于监听注册在其上的socketChannel的网络通讯
4. 每个Boss NioEventLoop线程内部循环执行的步骤有3步
5. 处理accept事件，与client建立连接，并生成NioSocketChannel
6. 将NioSocketChannel注册到某个worker NIOEventLoop上的selector
7. 处理任务队列的任务，即runAIITasks
8. 每个worker NIOEventLoop线程循环执行的步骤：
9. 轮询注册到自己selector上的所有NioSocketChannel的read/ write事件
10. 处理I/O事件，即read /write事件，在对应NioSocketChannel处理业务
11. runAIITasks处理任务队列TaskQueue的任务，一些耗时的业务处理一般可以放入TaskQueue中慢慢处理，这样不影响数据在pipeline中的流动处理
12. 每个worker NIOEventLoop处理NioSocketChannel业务时，会使用pipeline (管道)，管道中维护了很多handler处理器用来处理channel中的数据
    1. Netty组件
       1. ﻿Bootstrap和serverBootStrap

Bootstrap意思是引导，一个Netty应用通常由一个Bootstrap开始，主要作用是配置整个Netty程序，串联各个组件，Netty中Bootstrap类是客户端程序的启动引导类， ServerBootstrap是服务端启动引导类。

* + 1. Future和ChannelFuture

正如前面介绍，在Netty中所有的IO操作都是异步的，不能立刻得知消息是否被正确处理，但是可以过一会等它执行完成或者直接通过Future和Channel Futures注册一个监听，当IO成功或失败时监听会自动触发注册的监听事件。

* + 1. Channel

Netty网络通信的组件，能够用于执行网络I/O操作。Channel为用户提供以下功能：

1. 当前网络连接的通道的状态，例如是通道否打开，通道是否已连接
2. 网络连接的配置参数，例如接收缓冲区大小
3. 提供异步的网络I/O操作，如建立连接，读写，绑定端口等，异步调用意味着任何I/O调用方法都将立即返回，并且不保证在调用结束时所请求的IO操作已完成。
4. IO调用将返回一个ChannelFuture实例，通过注册监听器到ChannelFuture上，可以指定在I/O操作成功、失败或取消时回调通知调用方。
5. 支持关联I/O操作与对应的处理程序。

不同协议、不同的阻塞类型的连接都有不同的Channel对象类型与之对应：

1. ***NioSocketChannel：***异步的客户端 TCP Socket 连接。
2. ***NioServerSocketChannel：***异步的服务器端 TCP Socket 连接。
3. ***NioDatagramChannel*：**异步的 UDP 连接。
4. ***NioSctpChannel：***异步的客户端Sctp连接。
5. ***NioSctpServerChannel：***异步的Sctp服务器端连接。

这些通道涵盖了 UDP和TCP等网络IO

﻿

* + 1. Selector

Netty基于Selector对象实现I/O多路复用，通过Selector 一个线程可以监听多个连接的Channel事件。

当向一个Selector中注册Channel后，Netty的Selector内部机制就可以自动不断地查询(Select)这些注册的Channel是否有已就绪的I/O事件(例如可读，可写，网络连接完成等)， 这样程序就可以很简单地使用一个线程高效地管理多个Channel。

* + 1. NioEventLoop

NioEventLoop中维护了一个线程和任务队列，支持异步提交执行任务，线程启动时会调用NioEventLoop的run方法，执行I/O任务和非I/O任务：

I/O 任务，即 selectionKey 中 ready 的事件，如 accept、connect、read、write 等，由 processSelectedKeys 方法触发。

非IO任务，添加到taskQueue中的任务，如register0. bind0等任务，由runAIITasks方法触发。

* + 1. NioEventLoopGroup

NioEventLoopGroup，主要管理NioEventLoop的生命周期，可以理解为一个线程池，内部维护了一组NioEventLoop线程，每个NioEventLoop负责处理多个Channel上的事件，而一个NioEventLoop则应于一个线程。

* + 1. ChannelHandler

ChannelHandler是一个接口，处理I/O事件或拦截I/O操作，处理之后将其转发到ChannelPipeline(业务处理链)中的下一个处理程序。

ChannelHandler本身并没有提供很多方法，因为这个接口有许多的方法需要实现，为了方便使用，可以继承它的子类：

1. ChannelInboundHandler用于处理入站I/O事件。
2. ChannelOutboundHandler用于处理出站I/O操作。

或者使用以下适配器类：

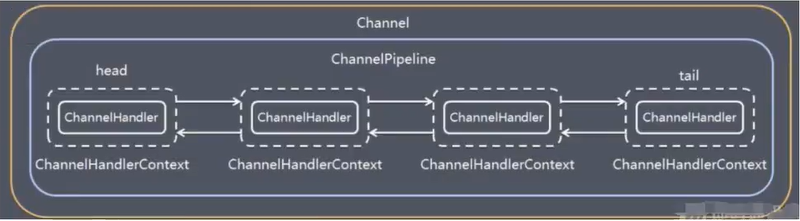
1. ChannelInboundHandlerAdapter 用于处理入站 I/O 事件。
2. ChannelOutboundHandlerAdapter 用于处理出站I/O操作。
   * 1. ChannelHandlerContext

保存Channel相关的所有上下文信息，同时关联一个ChannelHandler对象。

* + 1. ChannelPipline

保存ChannelHandler的List，用于处理或拦截Channel的入站事件和出站操作。

ChannelPipeline实现了一种高级形式的拦截过滤器模式，使用户可以完全控制事件的处理方式，以及Channel中各个的ChannelHandler如何相互交互。 在Netty中每个Channel都有且仅有一个ChannelPipeline与之对应，它们的组成关系如下：



一个Channel中包含了一个ChannelPipeline，而ChannelPipeline又维护了一个由ChannelHandlerContext上下文组成的双向链表，并且每个ChannelHandlerContext中又关联了一个ChannelHandler

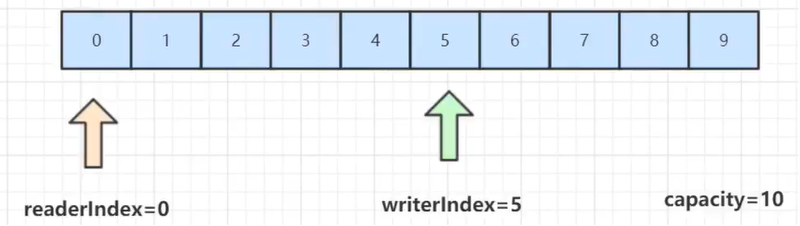
read事件(入站事件)和write事件(出站事件)在同一个双向链表中传递，但入站事件会从链表的head传递到最后一个入站的handler，而出站事件会从链表的tail传递到最前一个出站的handler，二者互不打扰

* + 1. ByteBuf

﻿ 从结构上来说，ByteBuf由一串字节数组构成。ByteBuf提供了两个索引，一个用于读取数据，一个用于写入数据。这两个索引通过在字节数组中移动，来定位需要读或者写信息的位置。

当从ByteBuf读取时，它的readerlndex读索引 将会根据读取的字节数递增。

当写ByteBuf时，它的writerlndex也会根据写入的字节数进行递增。



读取不会超过写入，如果readerlndex超过writerlndex，将会抛出IndexOutOfBoundException

eg：

public static void main(String[] args) {  
 //创建byteBuf对象，该对象内部包含一个字节数组byte［10］  
 //通过readerIndex和writerIndex以及capacity,将buffer分成三个区域  
 //己经读取的区域：［0, readerIndex)  
 //可读取的区域：［readerIndex，writerIndex)  
 //可写的区域：［writerIndex, capacity)  
 ByteBuf byteBuf = Unpooled.*buffer*(10);  
 System.*out*.println("byteBuf=" + byteBuf);  
 //写入8个数字，此时已读区域为[0,0)，可读区域为[0,8)，可写区域为[8,10)  
 for (int i = 0; i < 8; i++) {  
 byteBuf.writeByte(i);  
 }  
 System.*out*.println("byteBuf=" + byteBuf);  
 //getByte()不会移动readerIndex和writerIndex索引，此时已读区域为[0,0)，可读区域为[0,8)，可写区域为[8,10)  
 for (int i = 0; i < 5; i++) {  
 System.*out*.println(byteBuf.getByte(i));  
 }  
 System.*out*.println("byteBuf=" + byteBuf);  
 //读取5个数字，此时已读区域为[0,5)，可读区域为[0,8)，可写区域为[8,10)  
 for (int i = 0; i < 5; i++) {  
 System.*out*.println(byteBuf.readByte());  
 }  
}

1. Netty原理
   1. Netty粘包拆包

Udp是面向报文的协议，当用户数据通过udp协议传输时，操作系统不会对消息进行拆分，也就是说对于UDP来说，每个UDP报文就是一个用户消息的边界，因此不存在粘包拆包的问题

而tcp是面向字节流的的协议，TCP作为传输层并不了解上层的业务数据的具体涵义。因此当用户消息通过tcp传输时，一个完整的上层用户消息可能被操作系统拆分为多个tcp报文进行传输，而消息的接收者必须通过某种机制确认消息的边界，否则就会出现粘包的问题。

其实粘包也是为了提高网络传输效率，试想一下，假如有两个用户消息，大小都远小于MSS，那还不如合并起来发送一次，效率肯定要高于分两次发送。对于tcp协议，如果包长度达到MSS，立即发送，否则等到下一个包过来进行拆分发送；如果含有Fin包或者等待超时，即使没有达到MSS，也会立即发送

解决方案：

* + - 1. **加入特殊标志位作为边界：**类似的如c语言的字符串，但是对于消息内容中带有特殊标志位的需要进行转义处理，如FTP协议
      2. **固定包长度**：规定固定长度的包，空位补空格。极少使用，很不灵活
      3. **自定义消息结构**：给每个用户消息自定义消息结构，消息结构必须包含两部分，一部分说明数据长度，另一部分则是实际数据，这样就可以通过数据长度一直读取包数据，直到整个消息读取完成

