NIO 网络编程框架 Netty
知识点暨面试题总结

课程名称	NIO 网络编程框架 Netty
前 5 次直播 课程内容	——

序号	020501	问题	请简单描述一下 Reactor 模型在 Netty 中的应用。
参考答案	在 Netty-Server 中一般使用的是 Reactor 的多线程池模型，而 Netty-Client 中一般使用的是 Reactor 单线程池模型。具体来说，NioEventLoopGroup 充当着线程池。每一个 NioEventLoopGroup 中都包含了多个 NioEventLoop，而每个 NioEventLoop 又绑定着一个线程。 一个 NioEventLoop 可以处理多个 Channel 中的 IO 操作，而其只有一个线程。所以对于这个线程资源的使用，就存在了竞争。此时为每一个 NioEventLoop 都绑定了一个多跑复用器 Selector，由 Selector 来决定当前 NioEventLoop 的线程为哪些 Channel 服务。 这就是 Reactor 模型在 Netty 中的应用。		

序号	020502	问题	eventLoopGroup 中用于存放 eventLoop 的数据结构是什么？
参考答案	eventLoopGroup 中用于存放 eventLoop 的数据结构是数组。数组元素的个数可以在创建 eventLoopGroup 时指定，默认为当前主机逻辑内核数量的 2 倍。		

序号	020503	问题	NioEventLoop 的本质是什么，其包含了什么重要的变量？
参考答案	NioEventLoop 是一个 EventExecutor，是一个线程 Executor，其还封装着一个 executor，这个 executor 还绑定着一个线程，注册到这个 NioEventLoop 的 selector 的所有 channel 中就绪的 IO 事件及任务，都是由这个线程来完成的。		

序号	020504	问题	NioEventLoopGroup 的本质是什么，其包含了什么重要的变量？
参考答案	NioEventLoopGroup 是一个线程池，是一个线程池 Executor，其还封装着一个总的 executor，这个 executor 绑定着一个 threadFactory，同时		

	NioEventLoopGroup 还封装着一个用于存放 eventLoop 的数组。 这个总的 executor 为其所包含的每一个 eventLoop 创建了一个子 executor，然后这个总的 executor 所绑定的 threadFactory 会为每个子 executor 创建一个线程，用于完成相关任务。

序号	020505	问题	NioEventLoopGroup 默认会创建和包含多少个 NioEventLoop？
参考答案	在不指定创建数量的情况下，NioEventLoopGroup 默认会创建当前主机逻辑内核数量的 2 倍数量的 NioEventLoop。Netty 系统会自动将这些 NioEventLoop 所绑定线程的执行平均分配到各个逻辑内核上。这样做是为了充分利用内核，提高系统性能。 这个类似于 Nginx 中 worker_processes 与 worker_cpu_affinity 两个属性的设置。不同的是，Nginx 是按照物理 CPU 数量进行设置，而 Netty 是按照逻辑内核数据进行设置。		

序号	020506	问题	Server 监听一个 port 仅需一个 channel，而这个 parentChannel 仅会与绑定一个 EventLoop，那么在 parentGroup 中创建多个 EventLoop 有什么用处呢？
参考答案	我们知道，一个 Server 可以监听多个端口，那么就需要多个 parentChannel，那么也就可能会需要绑定 parentGroup 中的多个 EventLoop 了。注意，这里仅仅是可能。因为一个 EventLoop 上可以绑定多个 channel。		

序号	020507	问题	parentGroup 中 EventLoop 的数量与当前 Server 所监听的 port 数量间有关系吗？
参考答案	(通过对上个问题的分析)我们知道，parentGroup 中 EventLoop 的数量与当前 Server 所监听的 port 数量间是没有直接关系的。Server 同时监听 3 个 port，而 parentGroup 中仅有一个 EventLoop 也是可以的。 但 Group 采用的是轮询方式向 channel 分配其 EventLoop 的。所以，一般设置 Group 中 EventLoop 的数量大于等于当前 Server 所监听的 port 的数量。为了提高效率，这个 EventLoop 的数量会取一个 2 的 n 次幂。		

序号	020508	问题	一个 Server 同时监听了多个 port 的意义是什么呢？
参考答案	一个 Server 监听多个 port 可以创建多个 parentChannel，而多个 parentChannel 一般会绑定多个 EventLoop，从而使 EventLoop 的负载均衡了。 不过，一般情况下，并不会使 Client 自行指定要连接的 Server 的 port，因为这样仍无法真正实现对 parentGroup 的 EventLoop 的的负载均衡。一般会使 Client 访问一个路由服务器，例如 Nginx，让路由服务器根据负载均衡策略，路由到相同 Server 的不同 Port 上。这样，就实现了对 parentGroup 中 EventLoop 的负载均衡。		

序号	020509	问题	与 eventLoop 所绑定的线程是由谁创建的，是何时创建的？
参考答案	与 eventLoop 所绑定的线程是由 eventLoopGroup 所封装的 executor 创建的，确切地说，是由与这个 executor 所绑定的 ThreadFactory 创建的。是在完成 channel 注册到 eventLoop 的 selector 时创建的。		

序号	020510	问题	由 EventLoop 绑定线程来完成的任务有几种类型？分别存放在哪里？
参考答案	由 EventLoop 绑定线程来完成的任务有三种类型： ● 定时任务：该类型任务会存放到定时任务队列 scheduledTaskQueue 中。 ● 普通任务：该类型任务会存放到任务队列 taskQueue 中。我们在当前解析的 Netty 源码中见到的任务，都是这类任务。当然，定时任务最终也会从定时任务队列中逐个取出，然后放入到 taskQueue 中来执行。 ● 收尾任务：该类型任务会存放到任务队列 tailTasks 中。其定义方式与普通任务定义方式相同，只不过由于其主要用于完成一些收尾工作，所以被添加到了 tailTasks 队列中了。		

序号	020511	问题	请简述 Channel 实例在创建过程中都完成了哪些重要任务。
参考答案	在创建 Channel 过程中完成了以下几个重要任务： ● 生成了 Channel 的 id ● 创建了真正的数据传输对象 Unsafe ● 创建了与 Channel 相绑定的 ChannelPipeline ● 为 Channel 创建了其配置类 ChannelConfig		

序号	020512	问题	请简述一下 ChannelPromise。
参考答案	ChannelPromise 是一种特殊的 ChannelFuture，其是可写的。该接口中具有很多的 set 方法，即是可以修改的，也就是可写的。其作用是，根据异步操作的结果：成功或失败，来修改这个 Promise 的相关属性，以使 Promise 的获得者可以获知异步执行结果。		

序号	020513	问题	对于 parentChannel，其在初始化过程中主要完成了哪些工作？
参考答案	该初始化 channel 过程，其实主要完成了两项工作： 获取 ServerBootStrap 中的非 child 开头的属性，并初始化到 channel 中		

	获取 ServerBootStrap 中的 child 开头的属性，并初始化到一个连接处理器中，然后再将这个连接处理器添加到 channel 的 pipeline 中

序号	020514	问题	连接处理器 ServerBootstrapAcceptor 是何时实例化的，作用是什么？
参考答案	连接处理器 ServerBootstrapAcceptor 主要是用于处理 Client 的连接请求。该处理器在 Server 启动时实例化到 parentChannel 的 pipeline 中。也就是说，一个 parentChannel 就会绑定一个连接处理器实例。而该连接处理器用于处理“由这个 parentChannel 接收的 Client 的连接请求”。 当这个 parentChannel 接收到连接请求后，在其 channelRead()方法中会将其接收到的消息强转为一个 childChannel，然后使用在 ServerBootstrap 中配置的 child 开头的属性初始化这个 childChannel，并将 ServerBootstrap 中通过 childHandler 配置的 ChannelInitializer 中指定的处理器添加到 childChannel 的 pipeline 中。			

序号	020515	问题	NioEventLoop 中有一个成员变量 wakenUp，其是一个原子布尔类型。这个变量的值对于 selector 选择源码的阅读很重要。它的值代表什么意义？
参考答案	NioEventLoop 中有一个成员变量 wakenUp，其是一个原子布尔类型。其取值意义为： ● true：表示当前 eventLoop 所绑定的线程处于非阻塞状态，即唤醒状态 ● false：表示当前 eventLoop 所绑定的线程即将被阻塞			

序号	020516	问题	在 selector 中有一个方法 wakeup()，其意义对于 selector 选择源码的阅读很重要。但这个方法的意义仅从其方法名上来理解，很容易产生误解。那么这个方法表示什么意思呢？
参考答案	该方法会使选择操作立即结束，保存选择结果到 selector。而选择操作的结束，会使其调用者线程被唤醒。			

课程名称	NIO 网络编程框架 Netty
第 6 次直播课程内容	1) client 端启动源码解析（重点） 2) Pipeline 源码解析（重点）

序号	020601	问题	Client 端 Channel 的创建过程与 Server 端 parentChannel 的创建过程相同。但其初始化过程略有不同。不同主要体现在哪里？
参考答案	parentChannel 在初始化过程中需要将连接处理器注册到 channel 的 pipeline 中。这个连接处理器用于处理 Client 端的连接操作，为 Client 端		

	在 Server 处生成其对应的 childChannel，并注册到相应的 selector。但 Client 端的 channel 无需注册连接处理器，因为它是连接的发出者，而非接收者。

序号	020602	问题	通过对 ChannelFuture 的 isDone()方法的判断来处理异步完成后的情况，与为 ChannelFuture 添加的监听器来处理异步完成的情况，有什么不同。
参考答案	ChannelFuture 的 isDone()方法是对异步操作是否完成进行立即判断，而 ChannelFuture 的监听器回调方法则是在异步操作真正完成后才会触发。		

序号	020603	问题	ChannelPipeline 是在什么时候创建的？
参考答案	ChannelPipeline 是在创建 Channel 是创建的，其是 Channel 一个很重要的成员。		

序号	020604	问题	请简述 channelPipeline。
参考答案	ChannelPipeline 本质上是一个双向链表，默认具有头、尾两个节点。除了这两个节点外，其还可以通过 channelPipeline 的 addLast()方法向其中添加处理器节点。每一个处理器最终都会被封装为一下 channelPipeline 上的节点。		

序号	020605	问题	请简述一般处理器从 ChannelPipeline 中的删除过程。
参考答案	处理器从 ChannelPipeline 中的删除过程主要做了如下几项工作： ● 从 ChannelPipeline 中查找是否存在该处理器对应的节点。若存在，则进行删除。 ● 由于其删除的是节点，所以，会首先从 ChannelPipeline 中找到该处理器节点，然后从 ChannelPipeline 的双向链表中删除该节点。 ● 最后触发该处理器 handlerRemoved()方法的执行。		

序号	020606	问题	在添加处理器到 ChannelPipeline 时可以为该处理器指定名称，若没有指定系统会为其自动生成一个名称。这个自动生成的名称格式是怎样的？
参考答案	在将处理器添加到 ChannelPipeline 中时若没有指定名称，系统会自动为其生成一个名称，该名称为该处理器类的简单类名后跟一个#，然后是一个数字。从 0 开始尝试。若该名称在 ChannelPipeline 中存在，则数字加一，直到找到不重复的数字为止。		

课程名称	NIO 网络编程框架 Netty
第 7 次直播 课程内容	1) Pipeline 源码解析 2（重点） 2) Inbound/Outbound 处理器源码解析（重点）

序号	020701	问题	在 ChannelInitializer 类上为什么需要添加@Sharable?
参考答案	@Sharable 注解添加到一个处理器类上表示该处理器是共享的，可以被多次添加到同一个 ChannelPipeline 中，也可以被添加到多个 ChannelPipeline 中。 服务端启动类中定义的 ChannelInitializer 实例是在 Server 启动时创建的，然后每过来一个 Client 连接，就会将其添加到一个 childChannel 的 pipeline 中。即一个 ChannelInitializer 处理器实例被添加到了多个不同的 pipeline 中。这也就是为什么需要在 ChannelInitializer 类上添加@Sharable 注解的原因。		

序号	020702	问题	对于 ChannelInitializer 处理器实例的创建、删除，都与一个 initMap 有成员变量相关。请简述这个 initMap 在 ChannelInitializer 处理器中的作用。
参考答案	ChannelInitializer 处理器是一个共享处理器，为了减少内存的使用，在其中定义了一个成员变量 initMap。它是一个 Set 集合，其中仅存放着一个元素：当前 ChannelInitializer 处理器构成的节点 ctx。 虽然 ChannelInitializer 处理器实例是共享的，但处理器构成的节点实例却是多例的。所以，对于 ChannelInitializer 处理器的删除，其仅仅就是从 pipeline 中删除了该节点 ctx，然后从这个共享的 initMap 中删除了其存放的节点 ctx。但 ChannelInitializer 处理器实例并没有被删除。 一个 childChannel 会封装一个 ChannelPipeline，而一个 pipeline 中就会有一个“由 ChannelInitializer 处理器构成的” ctx 实例注册。但这个 ChannelInitializer 处理器实例却是共享的。		

序号	020703	问题	简述在 Server 端 bootstrap 中定义的 ChannelInitializer 处理器的创建、添加时机，及添加到哪个 channel 的 pipeline 中了？
参考答案	在 Server 端 bootstrap 中定义的 ChannelInitializer 处理器的创建、添加时机，及添加位置如下： ● 创建时机：在 Server 启动时被创建 ● 添加位置：最终会被添加到 childChannel 的 pipeline 中。因为其是通过 bootstrap 中的 childHandler()完成的初始化 ● 添加时机：每过来一个 Client 连接，就会将该处理器添加到一个 childChannel 的 pipeline 中，但添加的这个处理器实例，都是在 Server		

	启动时创建的那一个

序号	020704	问题	简述在 Client 端 bootstrap 中定义的 ChannelInitializer 处理器的创建、添加时机，及添加到哪个 channel 的 pipeline 中了？
参考答案	在 Client 端 bootstrap 中定义的 ChannelInitializer 处理器的创建、添加时机，及添加位置如下： ● 创建时机：在 Client 启动时被创建 ● 添加位置：被添加到 Channel 的 pipeline 中，Client 端没有 parentChannel 与 childChannel 的区分 ● 添加时机：在 Client 启动时被添加		

序号	020705	问题	ChannelInboundHandler 中都包含了哪一类的方法？
参考答案	ChannelInboundHandler 中包含了像 channelRead()、channelRegistered()、channelActive()等回调方法，即由其它事件所触发的发法。		

序号	020706	问题	ChannelOutboundHandler 中都包含了哪一类的方法？
参考答案	ChannelOutboundHandler 中包含了像 bind()、connet()、close()等方法，这些方法一般都是由 Outbound 处理器实例主动调用执行的，而最终是由 channel 的 unsafe 完成的。		

序号	020707	问题	简述一下 ChanneHandler 接口。
参考答案	ChanneHandler 接口是 ChannelInboundHandler 与 ChannelOutboundHandler 接口的父接口，其包含两个方法 handlerAdded()与 handlerRemoved()。也就是说，这两个方法是所有处理器都具有的方法。		

序号	020708	问题	ChanneHandlerContext 接口对于 ChannelPipeline 的理解很重要，请简述一下 ChanneHandlerContext 接口。
参考答案	ChanneHandlerContext 实例就是一个 ChannelPipeline 节点，是一个双向链表节点，其可以调用 InboundHandler 的方法，也可以调用 OutboundHandler 的方法，以引来触发下一个节点相应方法的执行。同时也可以获取到设置到 channel 中的 attr 属性。		

序号	020709	问题	Channe 中的 attr 属性是在哪里设置的？
参考答案	无论是 Server 还是 Client，它们在启动时会创建并初始化 bootstrap，		

	此时可以调用其 attr()或 childAttr()方法，将指定的 attr 属性初始化到 bootstrap 中。然后在 channel 创建后进行初始化时会将 bootstrap 中配置的设置信息初始化到 channel 中。这些信息中就包含 attr 属性。

序号	020710	问题	简述一下 ChannePipeline 接口及其重要实现类 DefaultChannelPipeline。
参考答案			ChannePipeline 是一个 ChannelHandlers 列表。该接口是继承了 ChannelInboundInvoker、ChannelOutboundInvoker 接口，说明其可以触发 Inbound 处理器方法，可以调用 Outbound 处理器方法。同时，其也继承了 Iterator 操，说明其是可迭代的。 该接口有一个重要实现类 DefaultChannelPipeline。 DefaultChannelPipeline 类实现了 ChannelPipline 接口中有关 ChannelInboundInvoker 中的方法，这些方法基本都是调用了抽象节点类 AbstractChannelHandlerContext 的相关静态方法，去调用 head 节点的相应方法。 DefaultChannelPipeline 类还实现了 ChannelPipline 接口中有关 ChannelOutboundInvoker 中的方法，这些方法基本都是调用 tail 结点的相关方法，完成底层的真正执行。 另外，DefaultChannelPipeline 类还实现了 ChannelPipline 接口中 Iterable 接口的方法 iterator()。其迭代的是一个 map 的 entrySet，这个 map 的 key 为节点名称，而 value 为节点所封装的处理器实例。

序号	020711	问题	简述一下 ChannelInboundHandlerAdapter 与 SimpleChannelInboundHandler 处理器的区别及应用场景。
参考答案			若我们使用 ChannelInboundHandlerAdapter，则需要我们自己释放 msg，而使用 SimpleChannelInboundHandler，则系统会自动释放。所以，使用哪个类作为处理器的父类，关键要看是否有需要释放的消息。 一般情况下，若 channelRead()中从对端接收到的 msg(或其封装实例)需要通过 writeAndFlush()等方法发送给对端，则该 msg 不能释放，所以需要使用 ChannelInboundHandlerAdapter 由我们自行控制 msg 的释放。当然，若根本就不需要从对端读取数据，则直接使用 ChannelInboundHandlerAdapter。若使用 SimpleChannelInboundHandler 还需要重写 channelRead0()方法。

序号	020712	问题	ChannelPipline 与 ChannelHandlerContext 都具有 fireChannelRead()方法，请简述一下它们的区别。
参考答案			ChannelPipline 中的 fireChannelRead()方法会从 head 节点的 channelRead()方法开始触发 pipeline 中节点的 channelRead()方法；而 ChannelHandlerContext 中的 fireChannelRead()方法则是触发当前节点的后面节点的 channelRead()方法。

序号	020713	问题	简述在 pipeline 中的多个处理器中都定义了 channelActive()与 handlerAdded()两个方法，请简述它们执行的区别。
参考答案	在 pipeline 中的多个处理器中的多个 channelActive()方法，只有第一个该方法会执行，因为 channel 只会被激活一次。而 handlerAdded()方法则不同，所以处理器中的该方法都会在当前处理器被添加到 pipeline 时被触发执行。		
序号	020714	问题	简述消息在 inboundHandler、outboundHandler 中的传递顺序，及发生异常后，异常信息在 inboundHandler、outboundHandler 中的传递顺序。
参考答案	消息在 inboundHandler 中 channelRead()方法中的传递顺序为，从 head 节点开始逐个向后传递，直到传递给 tail 节点将该消息释放。 消息在 outboundHandler 中 write()方法中的传递顺序为，从 tail 节点开始逐个向前传递，直到传递到 head 节点，然后调用 unsafe 的 write()方法完成底层写操作。 若发生异常，异常信息会从当前发生异常的节点开始调用 exceptionCaught()方法，并向后面节点传递，无论后面节点是 inboundHandler 还是 outboundHandler，最后传递到 tail 节点的 exceptionCaught()方法，将异常消息释放。 当然，前述的向后传递或向前传递的前提是，必须要在节点方法中调用传递到下一个节点的方法，否则是无法传递的。		