这可能是目前最透彻的 Netty 原理架构解析

作者：panjf2000
原文链接：https://ld246.com/article/1578623410721
来源网站：链滴

本文基于 Netty 4.1 展开介绍相关理论模型，使用场景，基本组件、整体架构，知其然且知其所以然希望给大家在实际开发实践、学习开源项目方面提供参考。

Netty 是一个异步事件驱动的网络应用程序框架，用于快速开发可维护的高性能协议服务器和客户端。

JDK 原生 NIO 程序的问题

JDK 原生也有一套网络应用程序 API，但是存在一系列问题，主要如下：

- **NIO 的类库和 API 繁杂，使用麻烦。** 你需要熟练掌握 Selector、ServerSocketChannel、Socket hannel、ByteBuffer 等。
- **需要具备其他的额外技能做铺垫。** 例如熟悉 Java 多线程编程，因为 NIO 编程涉及到 Reactor 模 ，你必须对多线程和网路编程非常熟悉，才能编写出高质量的 NIO 程序。
- **可靠性能力补齐，开发工作量和难度都非常大。** 例如客户端面临断连重连、网络闪断、半包读写、败缓存、网络拥塞和异常码流的处理等等。

NIO 编程的特点是功能开发相对容易，但是可靠性能力补齐工作量和难度都非常大。

- **JDK NIO 的 Bug。** 例如臭名昭著的 Epoll Bug，它会导致 Selector 空轮询，最终导致 CPU 100 。

官方声称在 JDK 1.6 版本的 update 18 修复了该问题，但是直到 JDK 1.7 版本该问题仍旧存在，只过该 Bug 发生概率降低了一些而已，它并没有被根本解决。

Netty 的特点

Netty 对 JDK 自带的 NIO 的 API 进行封装，解决上述问题，主要特点有：

- **设计优雅，** 适用于各种传输类型的统一 API 阻塞和非阻塞 Socket；基于灵活且可扩展的事件模型可以清晰地分离关注点；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接字支持（自 3.1 起）。
- **使用方便** ，详细记录的 Javadoc，用户指南和示例；没有其他依赖项，JDK 5（Netty 3.x）或 6（ etty 4.x）就足够了。

- **高性能，吞吐量更高，** 延迟更低；减少资源消耗；最小化不必要的内存复制。
- **安全，** 完整的 SSL/TLS 和 StartTLS 支持。
- **社区活跃，不断更新，** 社区活跃，版本迭代周期短，发现的 Bug 可以被及时修复，同时，更多的功能会被加入。

Netty 常见使用场景

Netty 常见的使用场景如下：

互联网行业

在分布式系统中，各个节点之间需要远程服务调用，高性能的 RPC 框架必不可少，Netty 作为异步性能的通信框架，往往作为基础通信组件被这些 RPC 框架使用。

典型的应用有： 阿里分布式服务框架 Dubbo 的 RPC 框架使用 Dubbo 协议进行节点间通信，Dubbo 协议默认使用 Netty 作为基础通信组件，用于实现各进程节点之间的内部通信。

游戏行业

无论是手游服务端还是大型的网络游戏，Java 语言得到了越来越广泛的应用。Netty 作为高性能的基通信组件，它本身提供了 TCP/UDP 和 HTTP 协议栈。

非常方便定制和开发私有协议栈，账号登录服务器，地图服务器之间可以方便的通过 Netty 进行高性的通信。

大数据领域

经典的 Hadoop 的高性能通信和序列化组件 Avro 的 RPC 框架，默认采用 Netty 进行跨界点通信，的 Netty Service 基于 Netty 框架二次封装实现。 有兴趣的读者可以了解一下目前有哪些开源项目用了 Netty：Related Projects。

Netty 高性能设计

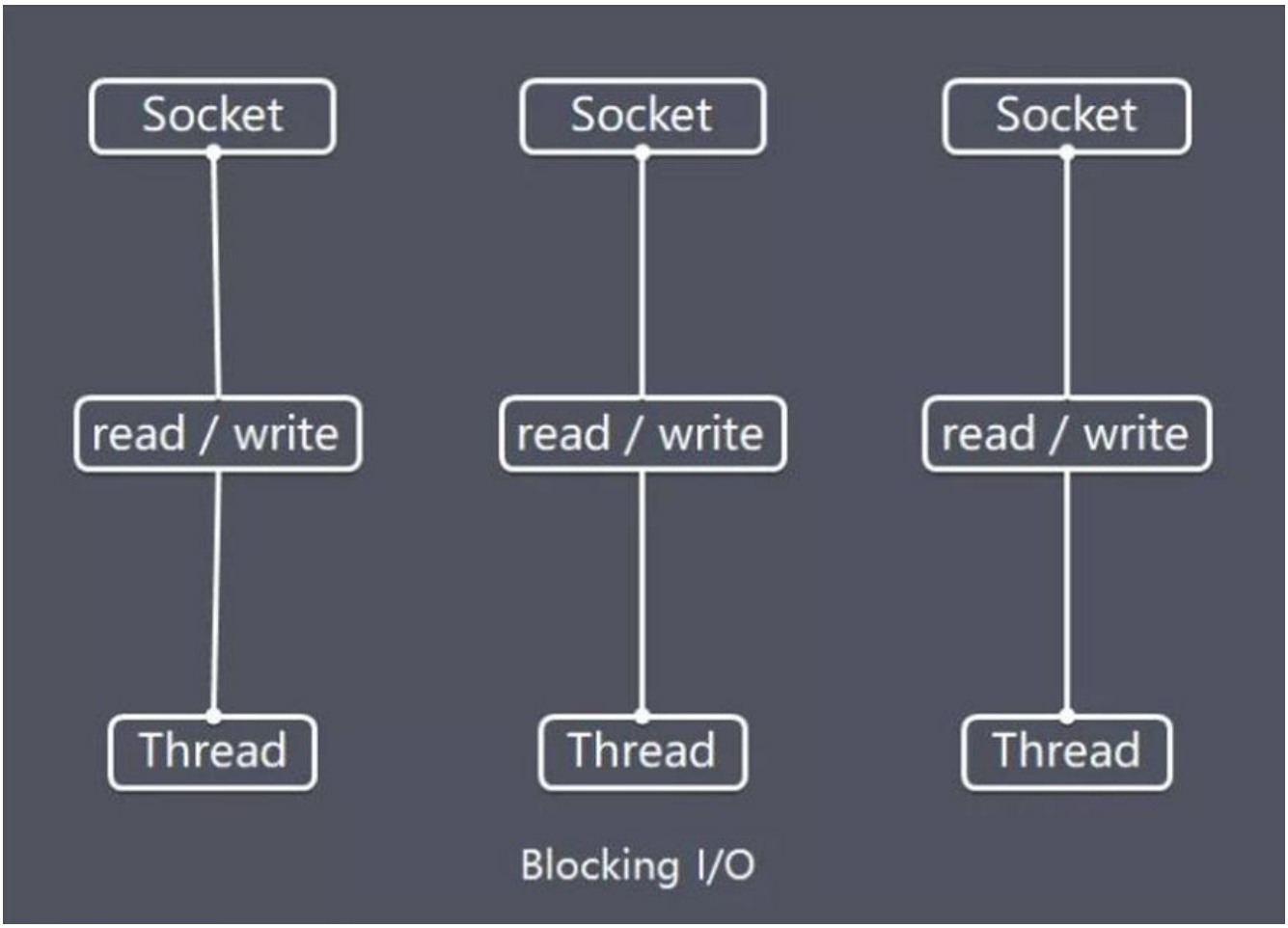
Netty 作为异步事件驱动的网络，高性能之处主要来自于其**I/O 模型**和**线程处理模型**，前者决定如何发数据，后者决定如何处理数据。

I/O 模型

用什么样的通道将数据发送给对方，BIO、NIO 或者 AIO，I/O 模型在很大程度上决定了框架的性能。

阻塞 I/O

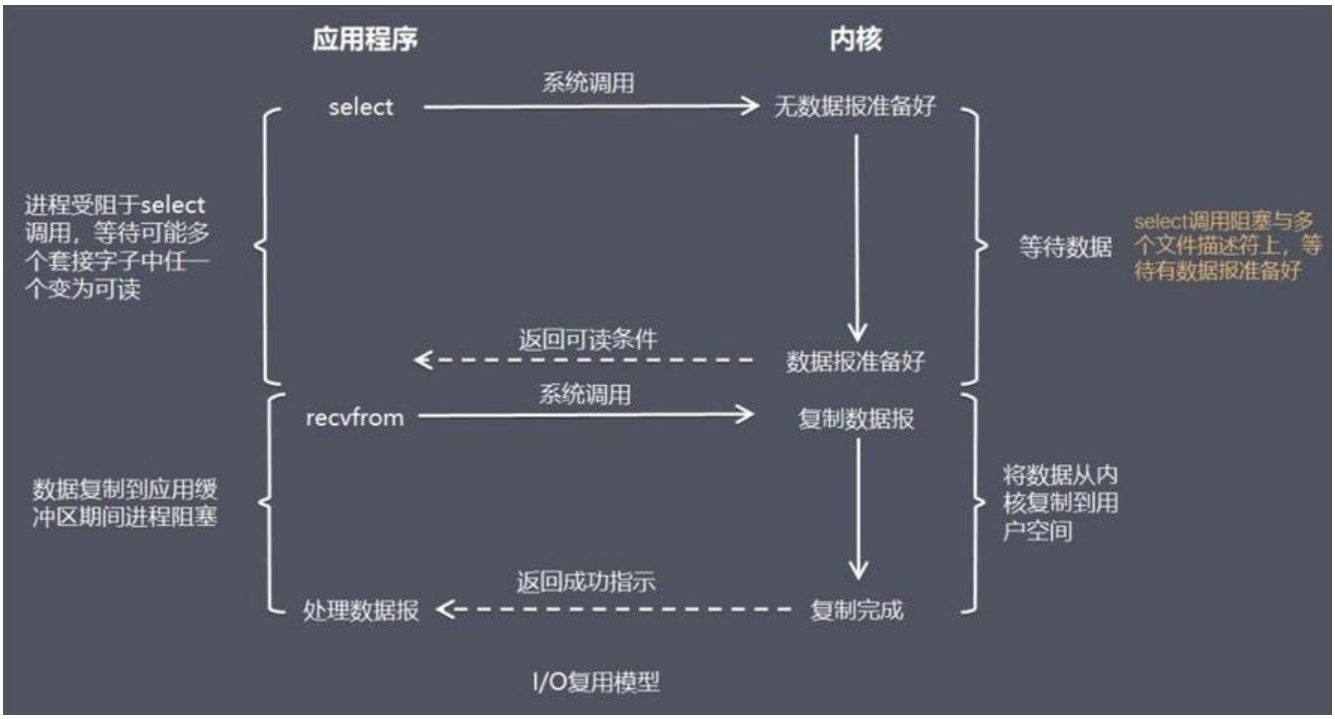
传统阻塞型 I/O(BIO)可以用下图表示：



特点如下：

- 每个请求都需要独立的线程完成数据 Read，业务处理，数据 Write 的完整操作问题。
- 当并发数较大时，需要创建大量线程来处理连接，系统资源占用较大。
- 连接建立后，如果当前线程暂时没有数据可读，则线程就阻塞在 Read 操作上，造成线程资源浪费。

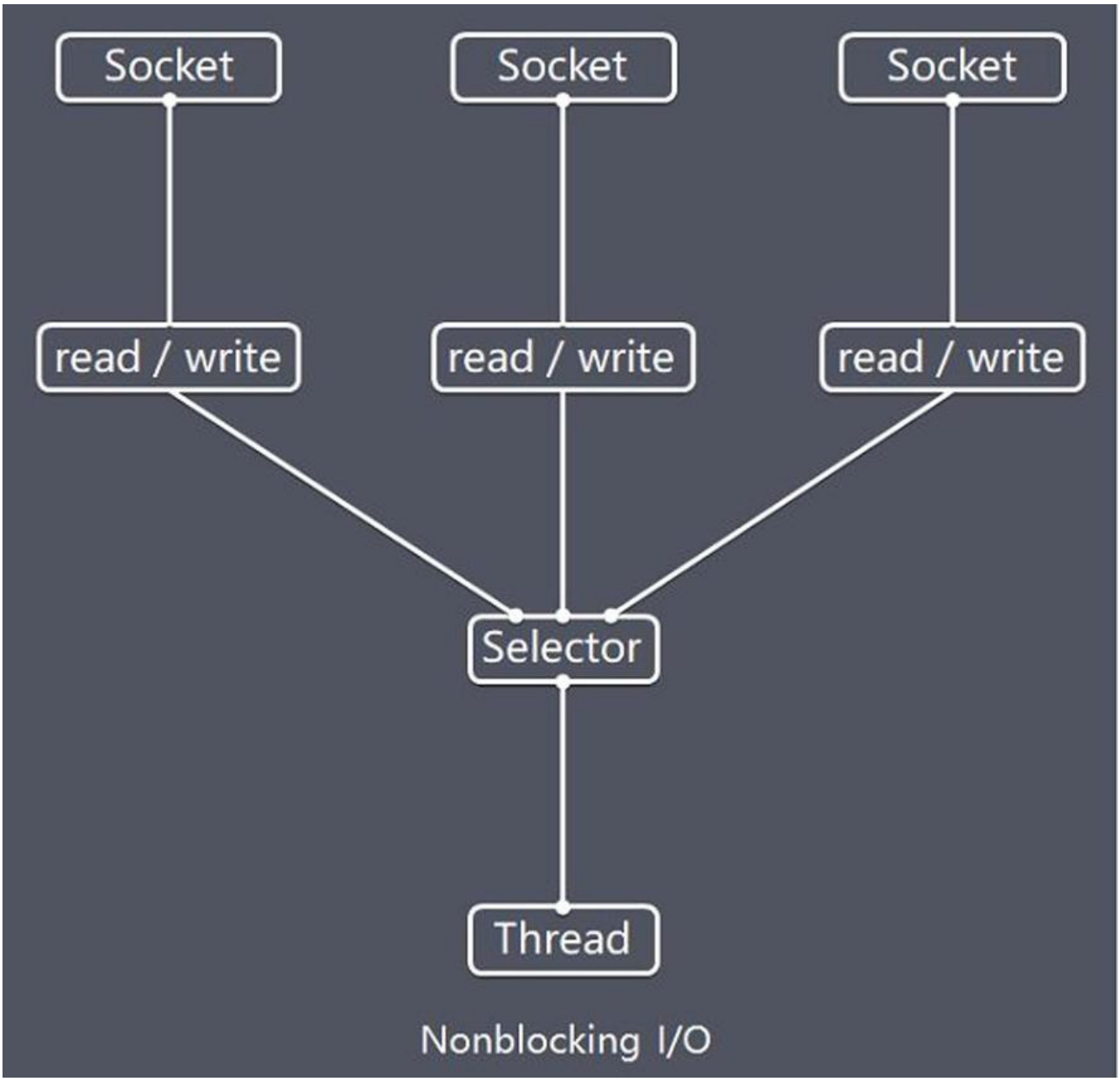
I/O 复用模型



在 I/O 复用模型中，会用到 Select，这个函数也会使进程阻塞，但是和阻塞 I/O 所不同的是这两个数可以同时阻塞多个 I/O 操作。

而且可以同时对多个读操作，多个写操作的 I/O 函数进行检测，直到有数据可读或可写时，才真正调 I/O 操作函数。

Netty 的非阻塞 I/O 的实现关键是基于 I/O 复用模型，这里用 Selector 对象表示：



Netty 的 IO 线程 NioEventLoop 由于聚合了多路复用器 Selector，可以同时并发处理成百上千个客端连接。

当线程从某客户端 Socket 通道进行读写数据时，若没有数据可用时，该线程可以进行其他任务。

线程通常将非阻塞 IO 的空闲时间用于在其他通道上执行 IO 操作，所以单独的线程可以管理多个输和输出通道。

由于读写操作都是非阻塞的，这就可以充分提升 IO 线程的运行效率，避免由于频繁 I/O 阻塞导致的程挂起。

一个 I/O 线程可以并发处理 N 个客户端连接和读写操作，这从根本上解决了传统同步阻塞 I/O 一连一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

基于 Buffer

传统的 I/O 是面向字节流或字符流的，以流式的方式顺序地从一个 Stream 中读取一个或多个字节，因此也就不能随意改变读取指针的位置。

在 NIO 中，抛弃了传统的 I/O 流，而是引入了 Channel 和 Buffer 的概念。在 NIO 中，只能从 Channel 中读取数据到 Buffer 中或将数据从 Buffer 中写入到 Channel。

基于 Buffer 操作不像传统 IO 的顺序操作，NIO 中可以随意地读取任意位置的数据。

线程模型

数据报如何读取？读取之后的编解码在哪个线程进行，编解码后的消息如何派发，线程模型的不同，性能的影响也非常大。

事件驱动模型

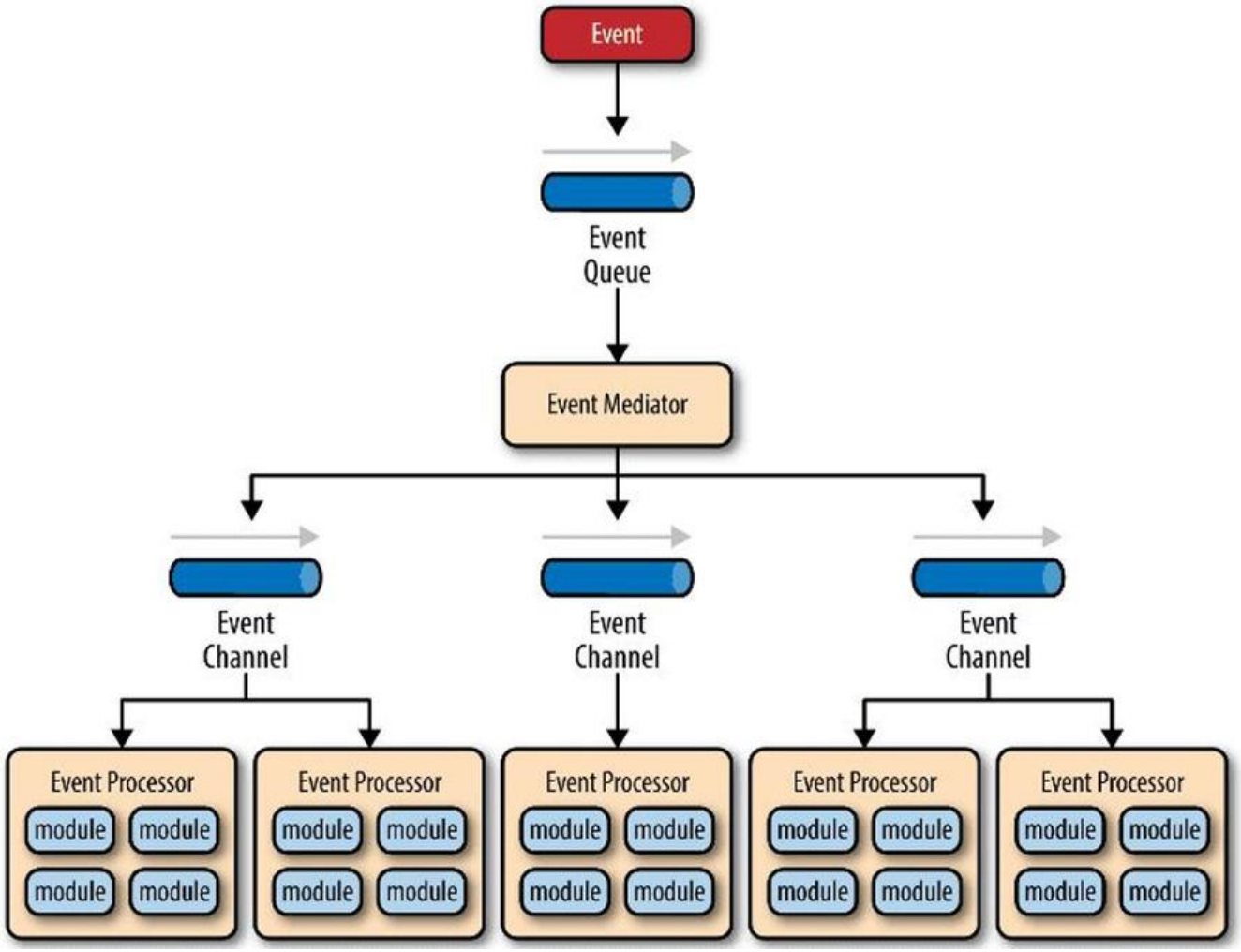
通常，我们设计一个事件处理模型的程序有两种思路：

- **轮询方式，** 线程不断轮询访问相关事件发生源有没有发生事件，有发生事件就调用事件处理逻辑。
- **事件驱动方式，** 发生事件，主线程把事件放入事件队列，在另外线程不断循环消费事件列表中的事件，调用事件对应的处理逻辑处理事件。事件驱动方式也被称为消息通知方式，其实是设计模式中观察模式的思路。

以 GUI 的逻辑处理为例，说明两种逻辑的不同：

- **轮询方式，** 线程不断轮询是否发生按钮点击事件，如果发生，调用处理逻辑。
- **事件驱动方式，** 发生点击事件把事件放入事件队列，在另外线程消费的事件列表中的事件，根据事类型调用相关事件处理逻辑。

这里借用 O'Reilly 大神关于事件驱动模型解释图：



主要包括 4 个基本组件：

- **事件队列（event queue）：** 接收事件的入口，存储待处理事件。
- **分发器（event mediator）：** 将不同的事件分发到不同的业务逻辑单元。
- **事件通道（event channel）：** 分发器与处理器之间的联系渠道。
- **事件处理器（event processor）：** 实现业务逻辑，处理完成后会发出事件，触发下一步操作。

可以看出，相对传统轮询模式，事件驱动有如下优点：

- **可扩展性好，** 分布式的异步架构，事件处理器之间高度解耦，可以方便扩展事件处理逻辑。
- **高性能，** 基于队列暂存事件，能方便并行异步处理事件。

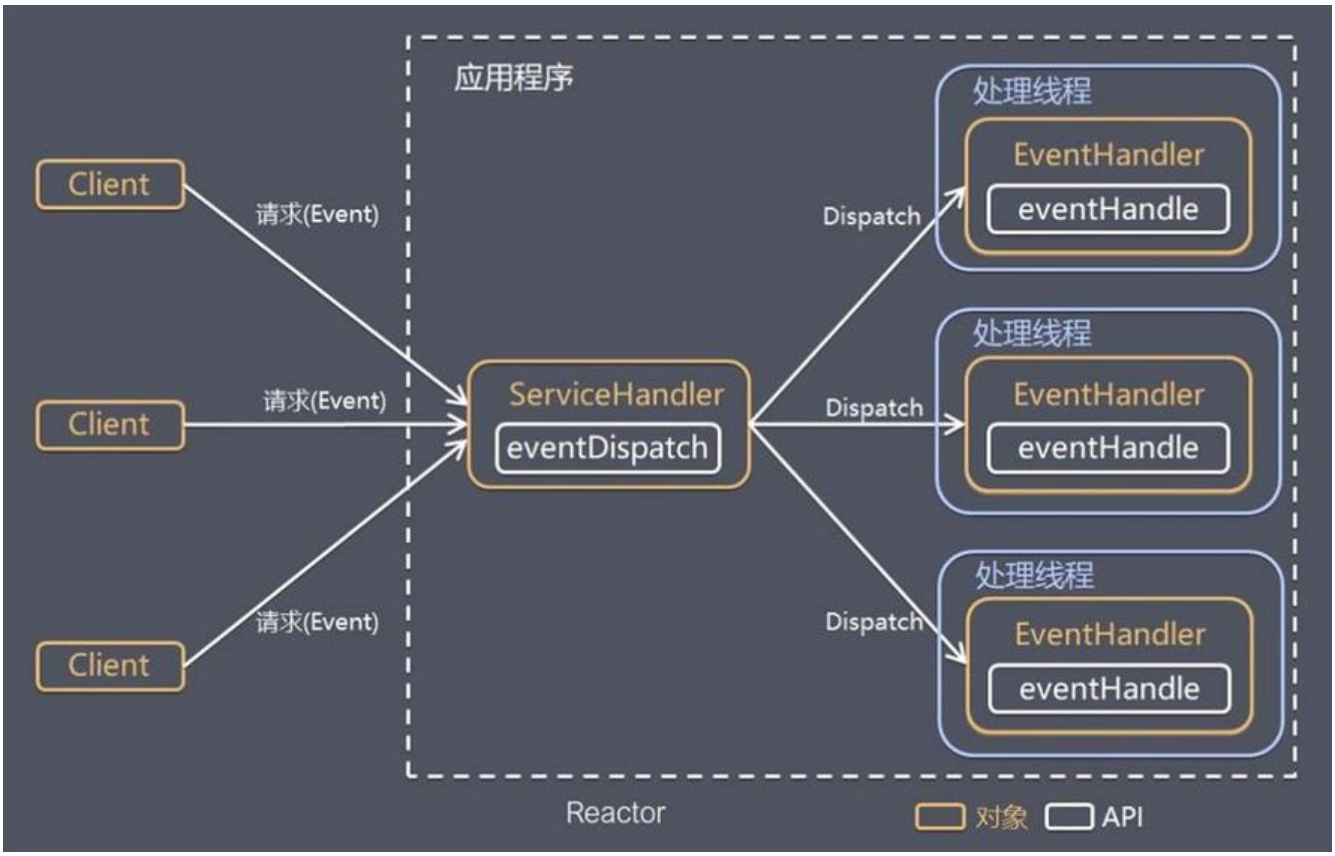
Reactor 线程模型

Reactor 是反应堆的意思，Reactor 模型是指通过一个或多个输入同时传递给服务处理器的服务请求事件驱动处理模式。

服务端程序处理传入多路请求，并将它们同步分派给请求对应的处理线程，Reactor 模式也叫 Dispatcher 模式，即 I/O 多了复用统一监听事件，收到事件后分发(Dispatch 给某进程)，是编写高性能网络务器的必备技术之一。

Reactor 模型中有 2 个关键组成：

- **Reactor，** Reactor 在一个单独的线程中运行，负责监听和分发事件，分发给适当的处理程序来对 O 事件做出反应。它就像公司的电话接线员，它接听来自客户的电话并将线路转移到适当的联系人。
- **Handlers，** 处理程序执行 I/O 事件要完成的实际事件，类似于客户想要与之交谈的公司中的实际员。Reactor 通过调度适当的处理程序来响应 I/O 事件，处理程序执行非阻塞操作。



取决于 Reactor 的数量和 Hanndler 线程数量的不同，Reactor 模型有 3 个变种：

- **单 Reactor 单线程。**
- **单 Reactor 多线程。**
- **主从 Reactor 多线程。**

可以这样理解，Reactor 就是一个执行 while (true) { selector.select(); ...} 循环的线程，会源源不断产生新的事件，称作反应堆很贴切。

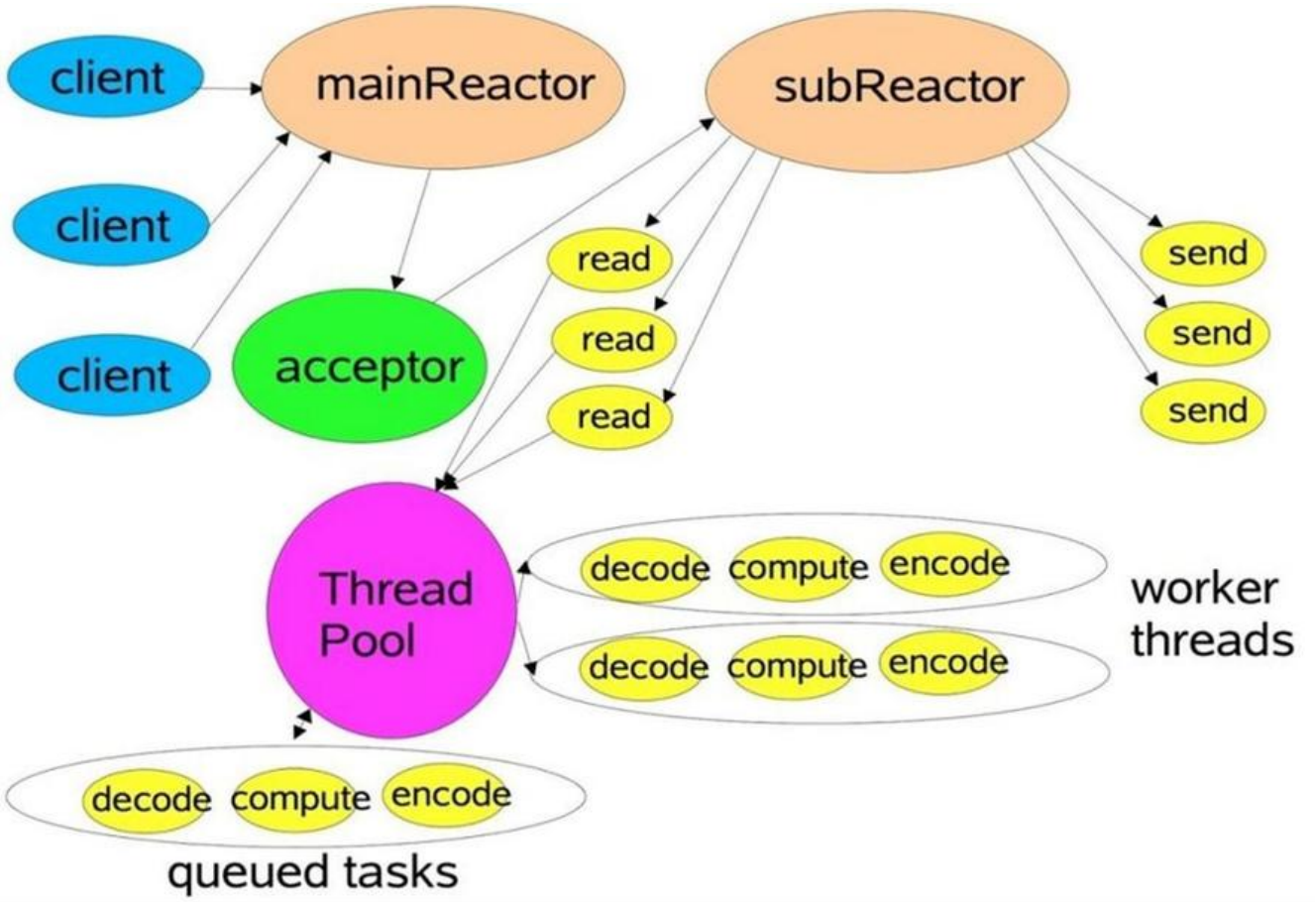
篇幅关系，这里不再具体展开 Reactor 特性、优缺点比较，有兴趣的读者可以参考我之前另外一篇文：《理解高性能网络模型》。

Netty 线程模型

Netty 主要基于主从 Reactors 多线程模型（如下图）做了一定的修改，其中主从 Reactor 多线程模有多个 Reactor：

- MainReactor 负责客户端的连接请求，并将请求转交给 SubReactor。
- SubReactor 负责相应通道的 IO 读写请求。
- 非 IO 请求（具体逻辑处理）的任务则会直接写入队列，等待 worker threads 进行处理。

这里引用 Doug Lee 大神的 Reactor 介绍：Scalable IO in Java 里面关于主从 Reactor 多线程模型图：



特别说明的是： 虽然 Netty 的线程模型基于主从 Reactor 多线程，借用了 MainReactor 和 SubReactor 的结构。但是实际实现上 SubReactor 和 Worker 线程在同一个线程池中：

```
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();
ServerBootstrap server = new ServerBootstrap();
server.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
```

上面代码中的 bossGroup 和 workerGroup 是 Bootstrap 构造方法中传入的两个对象，这两个 grou 均是线程池：

- bossGroup 线程池则只是在 Bind 某个端口后，获得其中一个线程作为 MainReactor，专门处理口的 Accept 事件，每个端口对应一个 Boss 线程。
- workerGroup 线程池会被各个 SubReactor 和 Worker 线程充分利用。

异步处理

异步的概念和同步相对。当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用部件在完成后，通过状态、通知和回调来通知调用者。

Netty 中的 I/O 操作是异步的，包括 Bind、Write、Connect 等操作会简单的返回一个 ChannelFutu e。

调用者并不能立刻获得结果，而是通过 Future-Listener 机制，用户可以方便的主动获取或者通过通

机制获得 IO 操作结果。

当 Future 对象刚刚创建时，处于非完成状态，调用者可以通过返回的 ChannelFuture 来获取操作执的状态，注册监听函数来执行完成后的操作。

常见有如下操作：

- 通过 isDone 方法来判断当前操作是否完成。
- 通过 isSuccess 方法来判断已完成的当前操作是否成功。
- 通过 getCause 方法来获取已完成的当前操作失败的原因。
- 通过 isCancelled 方法来判断已完成的当前操作是否被取消。
- 通过 addListener 方法来注册监听器，当操作已完成(isDone 方法返回完成)，将会通知指定的监器；如果 Future 对象已完成，则理解通知指定的监听器。

例如下面的代码中绑定端口是异步操作，当绑定操作处理完，将会调用相应的监听器处理逻辑。

```
serverBootstrap.bind(port).addListener(future -> {
            if (future.isSuccess()) {
                System.out.println(new Date() + ": 端口[" + port + "]绑定成功!");
            } else {
                System.err.println("端口[" + port + "]绑定失败!");
            }
        });
```

相比传统阻塞 I/O，执行 I/O 操作后线程会被阻塞住， 直到操作完成；异步处理的好处是不会造成线阻塞，线程在 I/O 操作期间可以执行别的程序，在高并发情形下会更稳定和更高的吞吐量。

Netty 架构设计

前面介绍完 Netty 相关一些理论，下面从功能特性、模块组件、运作过程来介绍 Netty 的架构设计。

功能特性

原文链接：这可能是目前最透彻的 Netty 原理架构解析

Netty 功能特性如下：

- **传输服务，** 支持 BIO 和 NIO。
- **容器集成，** 支持 OSGI、JBossMC、Spring、Guice 容器。
- **协议支持，** HTTP、Protobuf、二进制、文本、WebSocket 等一系列常见协议都支持。还支持通实行编码解码逻辑来实现自定义协议。
- **Core 核心，** 可扩展事件模型、通用通信 API、支持零拷贝的 ByteBuf 缓冲对象。

模块组件

Bootstrap、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 序，串联各个组件，Netty 中 Bootstrap 类是客户端程序的启动引导类，ServerBootstrap 是服务端动引导类。

Future、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注册一个监听，具体的实现就是通过 Future 和 ChannelFutur s，他们可以注册一个监听，当操作执行成功或失败时监听会自动触发注册的监听事件。

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：

- 当前网络连接的通道的状态（例如是否打开？是否已连接？）
- 网络连接的配置参数 （例如接收缓冲区大小）
- 提供异步的网络 I/O 操作(如建立连接，读写，绑定端口)，异步调用意味着任何 I/O 调用都将立即回，并且不保证在调用结束时所请求的 I/O 操作已完成。

调用立即返回一个 ChannelFuture 实例，通过注册监听器到 ChannelFuture 上，可以 I/O 操作成功失败或取消时回调通知调用方。

- 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 类型与之对应。下面是一些常用的 Channel 型：

- NioSocketChannel，异步的客户端 TCP Socket 连接。
- NioServerSocketChannel，异步的服务器端 TCP Socket 连接。
- NioDatagramChannel，异步的 UDP 连接。
- NioSctpChannel，异步的客户端 Sctp 连接。
- NioSctpServerChannel，异步的 Sctp 服务器端连接，这些通道涵盖了 UDP 和 TCP 网络 IO 以及件 IO。

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，Selector 内部的机制就可以自动不断地查询(Select) 这些注的 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地使用一个线程高效地管理多个 Channel 。

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 NioEvent oop 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 proces SelectedKeys 方法触发。
- **非 IO 任务，** 添加到 taskQueue 中的任务，如 register0、bind0 等任务，由 runAllTasks 方法触 。

两种任务的执行时间比由变量 ioRatio 控制，默认为 50，则表示允许非 IO 任务执行的时间与 IO 任 的执行时间相等。

NioEventLoopGroup

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

ChannelHandler

ChannelHandler 是一个接口，处理 I/O 事件或拦截 I/O 操作，并将其转发到其 ChannelPipeline(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，方便使用期间，以继承它的子类：

- ChannelInboundHandler 用于处理入站 I/O 事件。
- ChannelOutboundHandler 用于处理出站 I/O 操作。

或者使用以下适配器类：

- ChannelInboundHandlerAdapter 用于处理入站 I/O 事件。
- ChannelOutboundHandlerAdapter 用于处理出站 I/O 操作。
- ChannelDuplexHandler 用于处理入站和出站事件。

ChannelHandlerContext

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

ChannelPipline

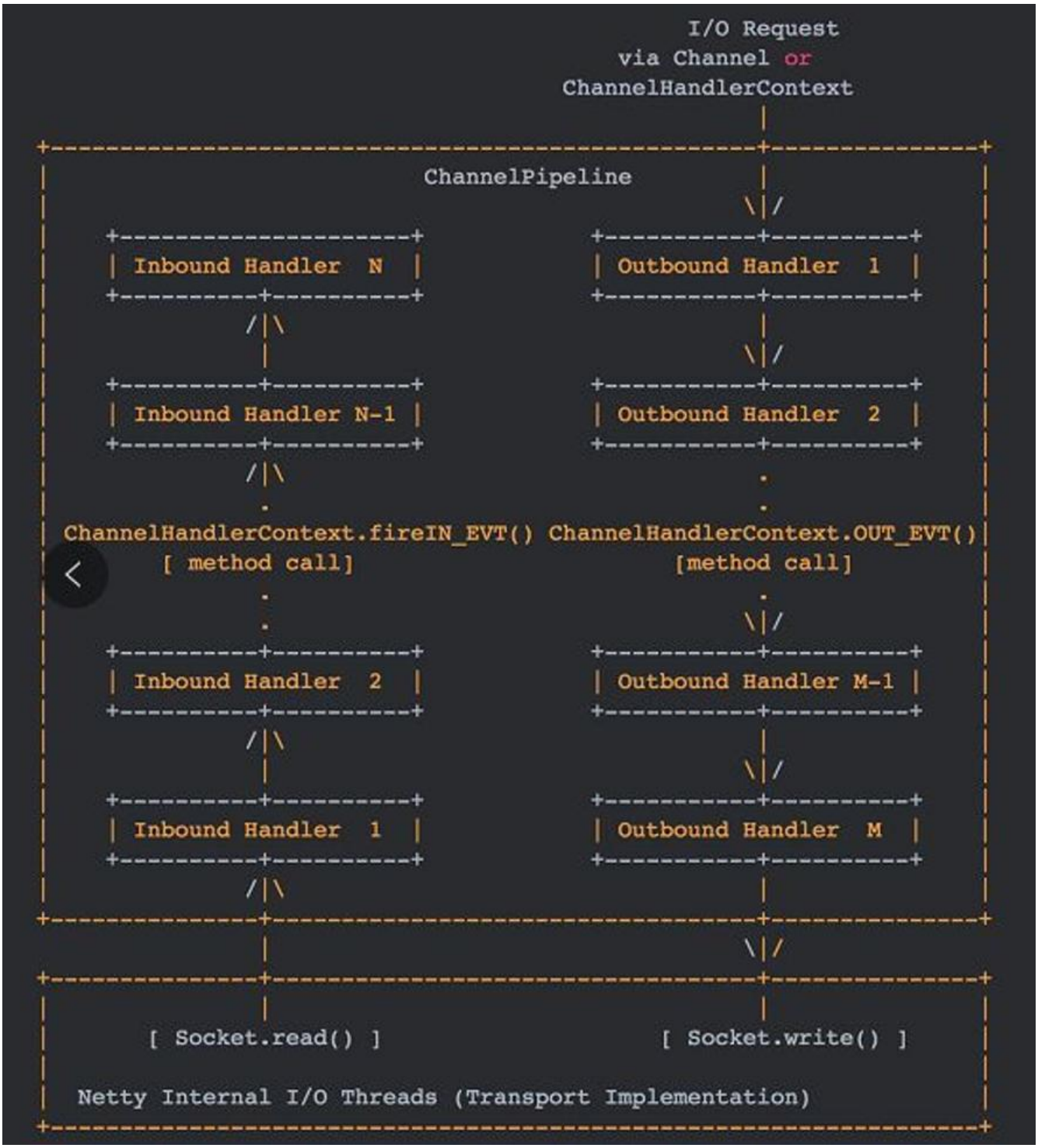
保存 ChannelHandler 的 List，用于处理或拦截 Channel 的入站事件和出站操作。

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。

下图引用 Netty 的 Javadoc 4.1 中 ChannelPipeline 的说明，描述了 ChannelPipeline 中 Channel andler 通常如何处理 I/O 事件。

I/O 事件由 ChannelInboundHandler 或 ChannelOutboundHandler 处理，并通过调用 ChannelH ndlerContext 中定义的事件传播方法。

例如 ChannelHandlerContext.fireChannelRead（Object）和 ChannelOutboundInvoker.write（ bject）转发到其最近的处理程序。



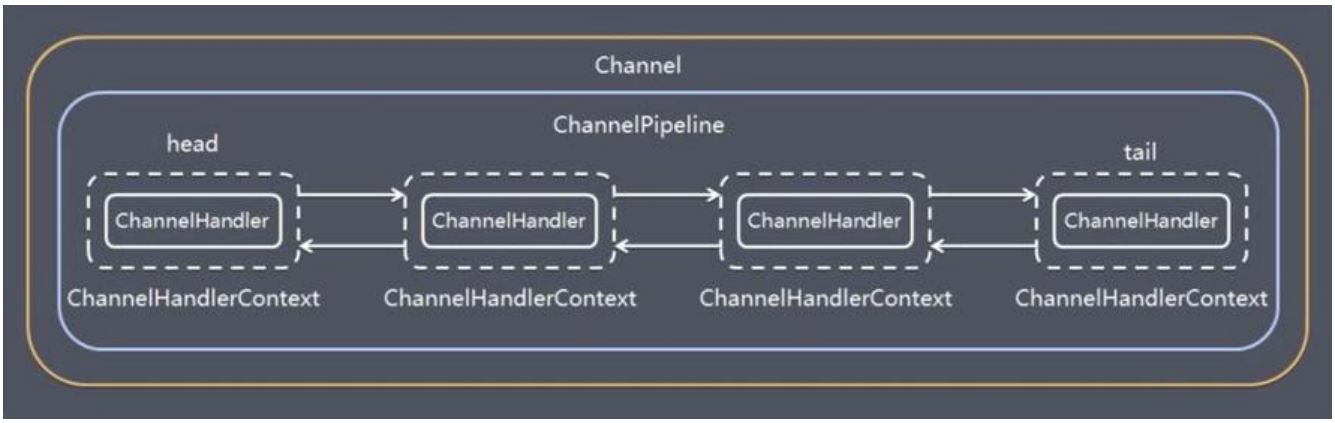
入站事件由自下而上方向的入站处理程序处理，如图左侧所示。入站 Handler 处理程序通常处理由底部的 I/O 线程生成的入站数据。

通常通过实际输入操作（例如 SocketChannel.read（ByteBuffer））从远程读取入站数据。

出站事件由上下方向处理，如图右侧所示。出站 Handler 处理程序通常会生成或转换出站传输，例如 write 请求。

I/O 线程通常执行实际的输出操作，例如 SocketChannel.write（ByteBuffer）。

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 包含了一个 ChannelPipeline，而 ChannelPipeline 中又维护了一个由 ChannelHandlerContext 组成的双向链表，并且每个 ChannelHandlerContext 中又关联着一个 ChannelHandler。

入站事件和出站事件在一个双向链表中，入站事件会从链表 head 往后传递到最后一个入站的 handle，出站事件会从链表 tail 往前传递到最前一个出站的 handler，两种类型的 handler 互不干扰。

Netty 工作原理架构

初始化并启动 Netty 服务端过程如下：

```
public static void main(String[] args) {
        // 创建mainReactor
        NioEventLoopGroup boosGroup = new NioEventLoopGroup();
        // 创建工作线程组
        NioEventLoopGroup workerGroup = new NioEventLoopGroup();

        final ServerBootstrap serverBootstrap = new ServerBootstrap();
        serverBootstrap
                // 组装NioEventLoopGroup
                .group(boosGroup, workerGroup)
                // 设置channel类型为NIO类型
                .channel(NioServerSocketChannel.class)
                // 设置连接配置参数
                .option(ChannelOption.SO_BACKLOG, 1024)
                .childOption(ChannelOption.SO_KEEPALIVE, true)
                .childOption(ChannelOption.TCP_NODELAY, true)
                // 配置入站、出站事件handler
                .childHandler(new ChannelInitializer<NioSocketChannel>() {
                    @Override
                    protected void initChannel(NioSocketChannel ch) {
                        // 配置入站、出站事件channel
                        ch.pipeline().addLast(...);
                        ch.pipeline().addLast(...);
                    }
                });

        // 绑定端口
        int port = 8080;
        serverBootstrap.bind(port).addListener(future -> {
            if (future.isSuccess()) {
                System.out.println(new Date() + ": 端口[" + port + "]绑定成功!");
```

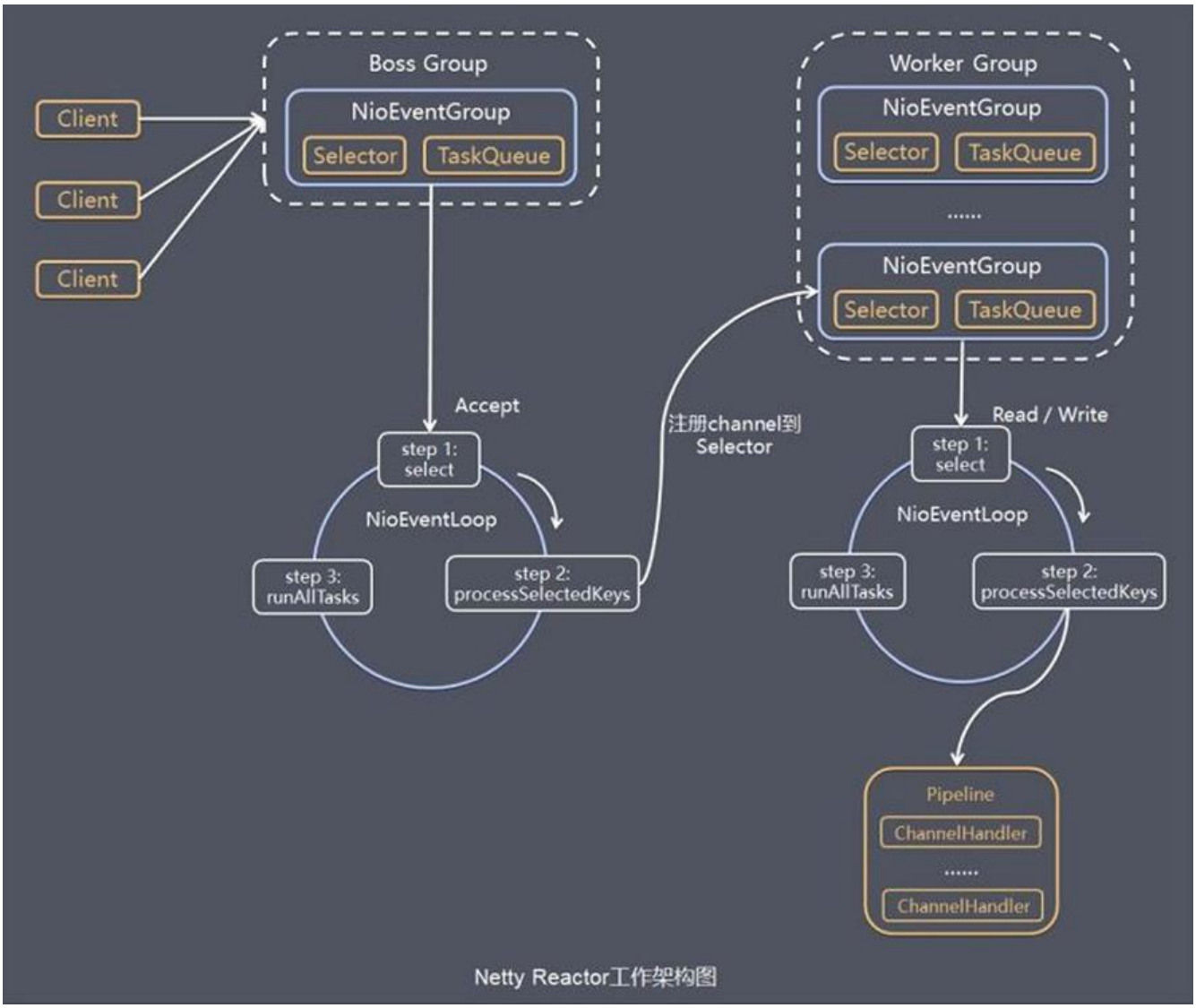
原文链接：这可能是目前最透彻的 Netty 原理架构解析

```
                } else {
                    System.err.println("端口[" + port + "]绑定失败!");
                }
            });
    }
```

基本过程如下：

- 初始化创建 2 个 NioEventLoopGroup，其中 boosGroup 用于 Accetpt 连接建立事件并分发请，workerGroup 用于处理 I/O 读写事件和业务逻辑。
- 基于 ServerBootstrap(服务端启动引导类)，配置 EventLoopGroup、Channel 类型，连接参数、置入站、出站事件 handler。
- 绑定端口，开始工作。

结合上面介绍的 Netty Reactor 模型，介绍服务端 Netty 的工作架构图：



Server 端包含 1 个 Boss NioEventLoopGroup 和 1 个 Worker NioEventLoopGroup。

NioEventLoopGroup 相当于 1 个事件循环组，这个组里包含多个事件循环 NioEventLoop，每个 NioEventLoop 包含 1 个 Selector 和 1 个事件循环线程。

每个 Boss NioEventLoop 循环执行的任务包含 3 步：

- **轮询 Accept 事件。**
- **处理 Accept I/O 事件，** 与 Client 建立连接，生成 NioSocketChannel，并将 NioSocketChannel 注册到某个 Worker NioEventLoop 的 Selector 上。
- **处理任务队列 中的任务，runAllTasks。** 任务队列中的任务包括用户调用 eventloop.execute 或 schedule 执行的任务，或者其他线程提交到该 eventloop 的任务。

每个 Worker NioEventLoop 循环执行的任务包含 3 步：

- 轮询 Read、Write 事件。
- 处理 I/O 事件，即 Read、Write 事件，在 NioSocketChannel 可读、可写事件发生时进行处理。
- 处理任务队列中的任务，runAllTasks。

其中任务队列中的 Task 有 3 种典型使用场景。

① 用户程序自定义的普通任务

```
ctx.channel().eventLoop().execute(new Runnable() {
    @Override
    public void run() {
        //...
    }
});
```

② 非当前 Reactor 线程调用 Channel 的各种方法

例如在推送系统的业务线程里面，根据用户的标识，找到对应的 Channel 引用，然后调用 Write 类法向该用户推送消息，就会进入到这种场景。最终的 Write 会提交到任务队列中后被异步消费。

③ 用户自定义定时任务

```
ctx.channel().eventLoop().schedule(new Runnable() {
    @Override
    public void run() {

    }
}, 60, TimeUnit.SECONDS);
```

总结

现在稳定推荐使用的主流版本还是 Netty4，Netty5 中使用了 ForkJoinPool，增加了代码的复杂度但是对性能的改善却不明显，所以这个版本不推荐使用，官网也没有提供下载链接。

Netty 入门门槛相对较高，是因为这方面的资料较少，并不是因为它有多难，大家其实都可以像搞透 pring 一样搞透 Netty。

在学习之前，建议先理解透整个框架原理结构，运行过程，可以少走很多弯路。

参考资料：

- Netty 入门与实战：仿写微信 IM 即时通讯系统

- Netty 官网
- Netty 4.x 学习笔记 - 线程模型
- Netty 入门与实战
- 理解高性能网络模型
- Netty 基本原理介绍
- software-architecture-patterns.pdf
- Netty 高性能之道 —— 李林锋
- 《Netty In Action》
- 《Netty 权威指南》

标题：这可能是目前最透彻的 Netty 原理架构解析
作者：Hollis
原文：再有人问你Netty是什么，就把这篇文章发给他