<https://www.jianshu.com/p/4ad6a08e31a3>

IO模型

<https://my.oschina.net/u/1859679/blog/1839169>

Reator模型

<https://my.oschina.net/u/1859679/blog/1844109>

netty api

<https://netty.io/4.1/api/index.html>

netty exapmle

<https://github.com/sanshengshui/netty-learning-example>

netty book

[https://waylau.gitbooks.io/essential-netty-in-action/content](https://waylau.gitbooks.io/essential-netty-in-action/content/GETTING%20STARTED/Writing%20the%20echo%20server.html)

netty简介

<https://www.cnblogs.com/imstudy/p/9908791.html>

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

特点：

I/O 复用模型

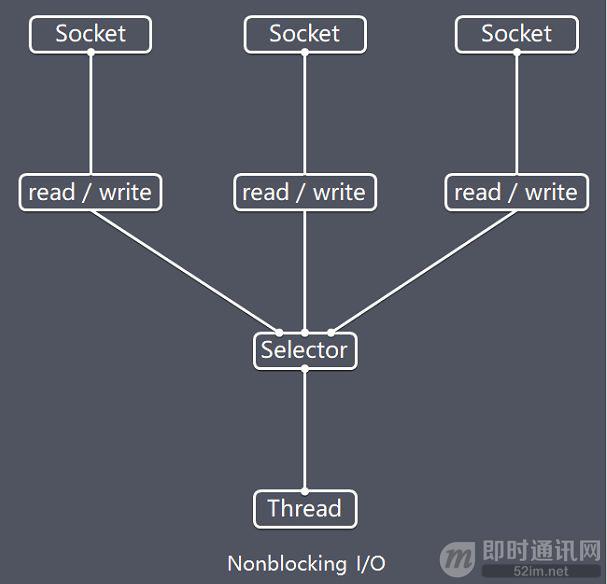
Netty 的 IO 线程 NioEventLoop 由于聚合了多路复用器 Selector，可以同时并发处理成百上千个客户端连接。

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入和输出通道。

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 阻塞导致的线程挂起。

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 一连接一线程模型，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】：

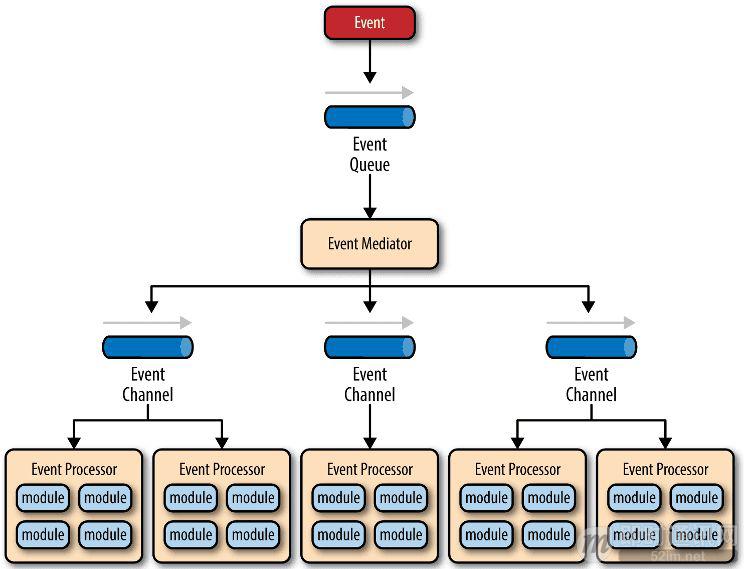
传统的 I/O 是面向字节流或字符流的，以流式的方式顺序地从一个 Stream 中读取一个或多个字节, 因此也就不能随意改变读取指针的位置。

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。

事件驱动模型

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者模式的思路。



主要包括 4 个基本组件：

1）事件队列（event queue）：接收事件的入口，存储待处理事件；

2）分发器（event mediator）：将不同的事件分发到不同的业务逻辑单元；

3）事件通道（event channel）：分发器与处理器之间的联系渠道；

4）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。

Reactor

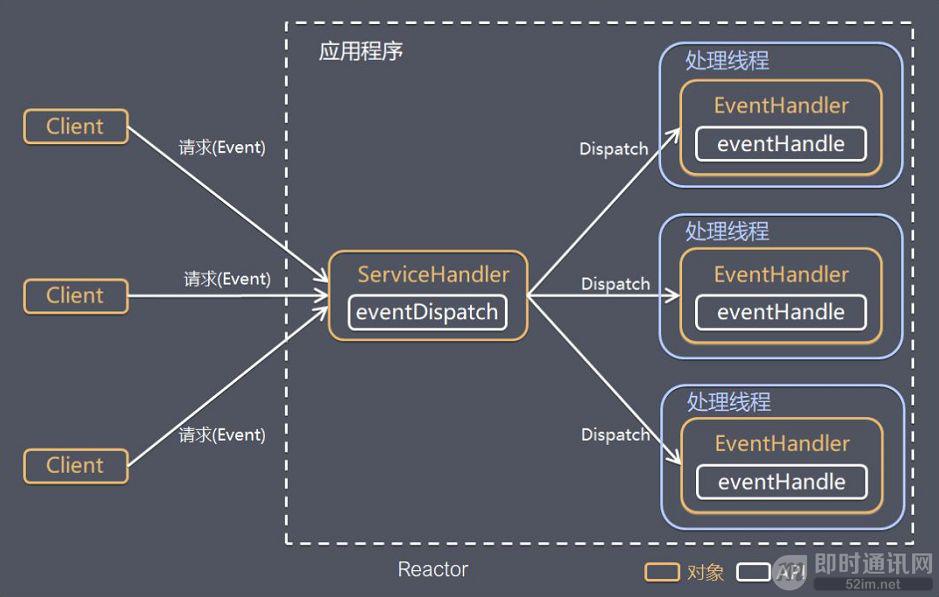
基于NIO中实现多路复用的一种模式

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的事件驱动处理模式

Reactor 模型中有 2 个关键组成：

1）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 IO 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；

2）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官员。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。



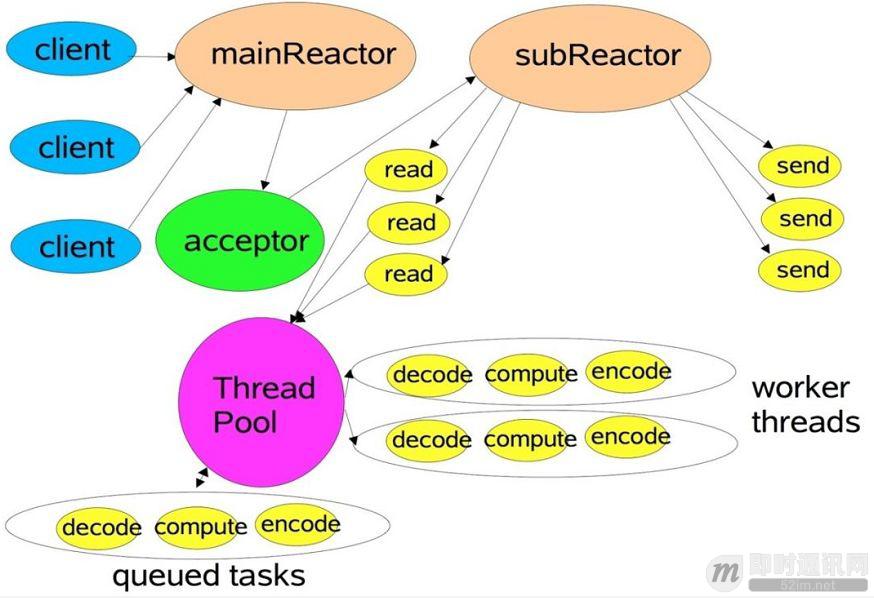
取决于 Reactor 的数量和 Hanndler 线程数量的不同，Reactor 模型有 3 个变种：

1）单 Reactor 单线程；

2）单 Reactor 多线程；

3）主从 Reactor 多线程。

Netty 主要基于主从 Reactors 多线程模型



EventLoopGroup bossGroup = newNioEventLoopGroup();

EventLoopGroup workerGroup = newNioEventLoopGroup();

ServerBootstrap server = newServerBootstrap();

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 group 均是线程池：

1）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，专门处理端口的 Accept 事件，每个端口对应一个 Boss 线程；

2）workerGroup 线程池会被各个 SubReactor 和 Worker 线程充分利用。

功能特性

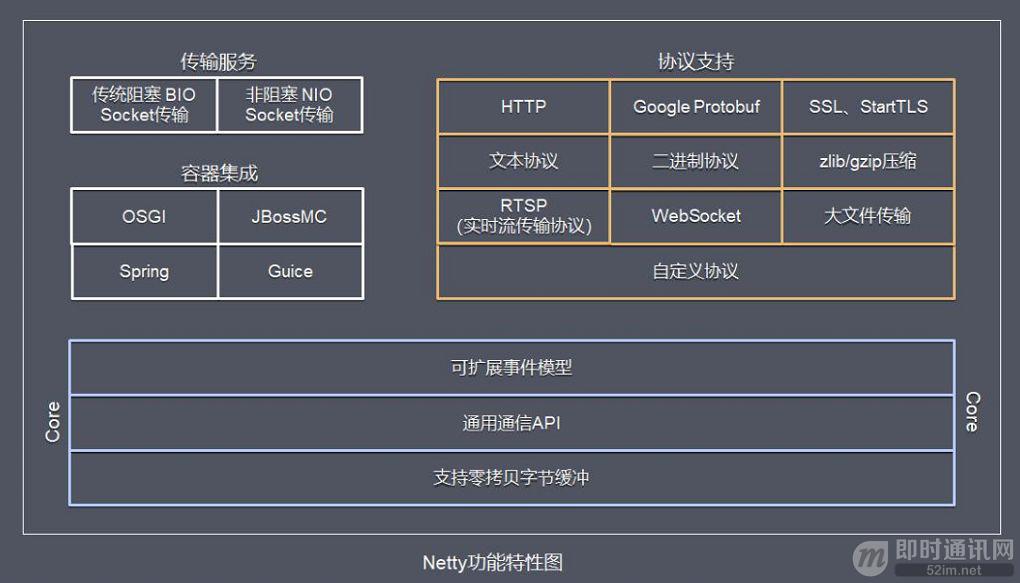
Netty 功能特性如下：

1）传输服务：支持 BIO 和 NIO；

2）容器集成：支持 OSGI、JBossMC、Spring、Guice 容器；

3）协议支持：HTTP、Protobuf、二进制、文本、WebSocket 等一系列常见协议都支持。还支持通过实行编码解码逻辑来实现自定义协议；

4）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 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。

【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 ChannelFutures，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：

1）当前网络连接的通道的状态（例如是否打开？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)，异步调用意味着任何 I/O 调用都将立即返回，并且不保证在调用结束时所请求的 I/O 操作已完成。

4）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。

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 类型与之对应。

下面是一些常用的 Channel 类型：

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 以及文件 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 。

【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 runAllTasks 方法触发。

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 任务的执行时间相等。

【NioEventLoopGroup】：

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

【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(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，方便使用期间，可以继承它的子类：

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 ChannelHandler 通常如何处理 I/O 事件。

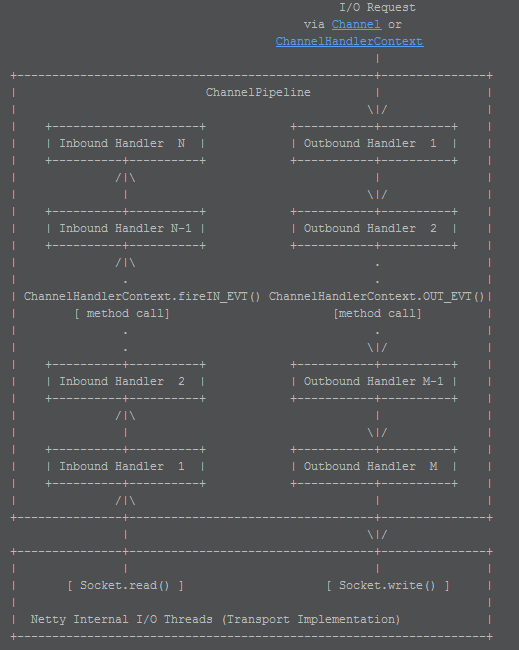
I/O 事件由 ChannelInboundHandler 或 ChannelOutboundHandler 处理，并通过调用 ChannelHandlerContext 中定义的事件传播方法。

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

Channel、ChannelPipeline、ChannelHandler、ChannelHandlerContext 之间的关系

<https://www.cnblogs.com/nanaheidebk/p/11025362.html>

Netty 中有两个方向的数据流，图3.4 显示的入站(ChannelInboundHandler)和出站(ChannelOutboundHandler)之间有一个明显的区别：若数据是从用户应用程序到远程主机则是“出站(outbound)”，相反若数据时从远程主机到用户应用程序则是“入站(inbound)”。



ChannelHandlerContext 的主要功能是管理它所关联的 ChannelHandler 和在同一个 ChannelPipeline 中的其他 ChannelHandler 之间的交互。事件从一个 ChannelHandler 到下一个 ChannelHandler 的移动是由 ChannelHandlerContext 上的调用完成的。

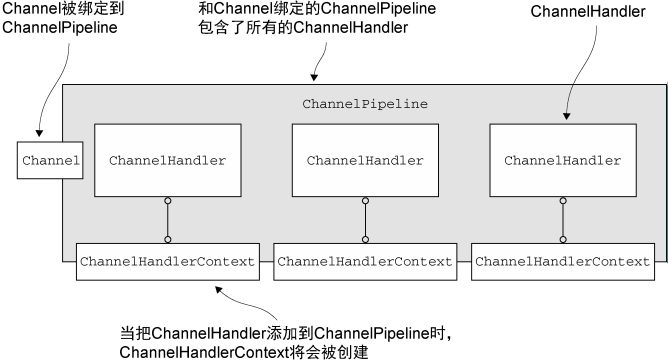
NioEventLoopGroup 实际上就是个线程池，一个 EventLoopGroup 包含一个或者多个 EventLoop；

一个 EventLoop 在它的生命周期内只和一个 Thread 绑定；

所有有 EnventLoop 处理的 I/O 事件都将在它专有的 Thread 上被处理；

一个 Channel 在它的生命周期内只注册于一个 EventLoop；

每一个 EventLoop 负责处理一个或多个 Channel；



NioEventLoopGroup 实际上就是个线程池，一个 EventLoopGroup 包含一个或者多个 EventLoop；

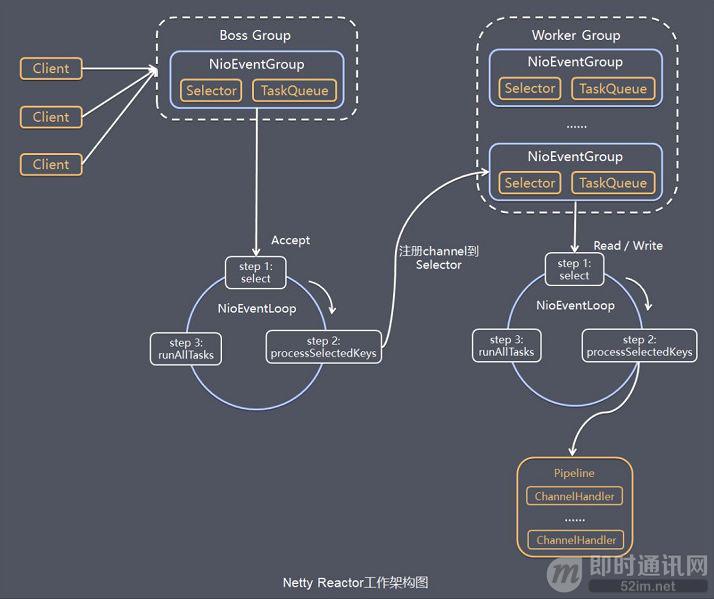
一个 EventLoop 在它的生命周期内只和一个 Thread 绑定；

所有有 EnventLoop 处理的 I/O 事件都将在它专有的 Thread 上被处理；

一个 Channel 在它的生命周期内只注册于一个 EventLoop；

每一个 EventLoop 负责处理一个或多个 Channel；

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：

1）轮询 Accept 事件；

2）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；

3）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：

1）轮询 Read、Write 事件；

2）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；

3）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(newRunnable() {

@Override

publicvoidrun() {

//...

}

});

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 类方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(newRunnable() {

@Override

publicvoidrun() {

}

}, 60, TimeUnit.SECONDS);