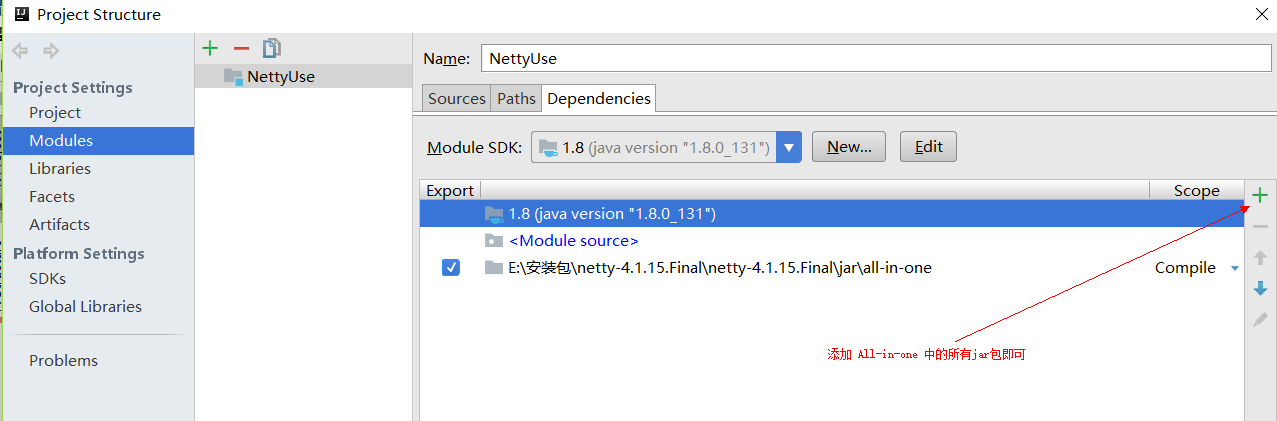
**一、Netty开发环境搭建**

过程：

* 下载Netty类库：<http://netty.io/downloads.html>
* 下载完之后解压，有个文件夹叫做all-in-one，只用导入里面两个jar就可以了
* 开发过程搭建：file->project structure->Modules->dependencies->”+”->jars or directory->找到all-in-one目录即可





Netty 架构图

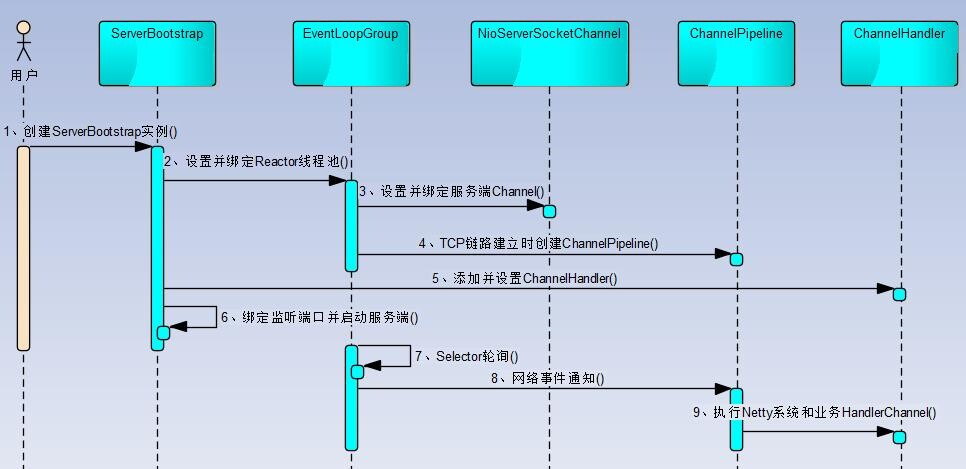
**二、Netty 服务端创建**

当我们直接使用JDK NIO 的类库开发基于NIO 的异步服务端时，需要使用到**多路复用器**Selector、ServerSocketChannel、SocketChannel、ByteBuffer、SelectionKey 等等，相比于传统的BIO 开发，NIO 的开发要复杂很多，开发出稳定、高性能的异步通信框架，一直是个难题。

Netty 为了向使用者**屏蔽NIO 通信的底层细节**，在和用户交互的边界做了封装，目的就是为了**减少用户开发工作量，降低开发难度**。**ServerBootstrap** 是Socket 服务端的启动辅助类，用户通过ServerBootstrap 可以方便的创建Netty的服务端。

**1、创建时序图**

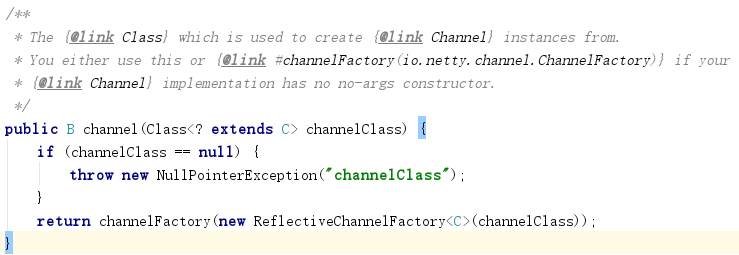
* **步骤1：**创建ServerBootstrap 实例。ServerBootstrap 是Netty 服务端的启动辅助类，它*提供了一系列的方法用于设置服务端启动相关的参数*。底层通过门面模式对各种能力进行抽象和封装，尽量不需要用户跟过多的底层API 打交道，降低用户的开发难度。



Netty 服务端创建时序图

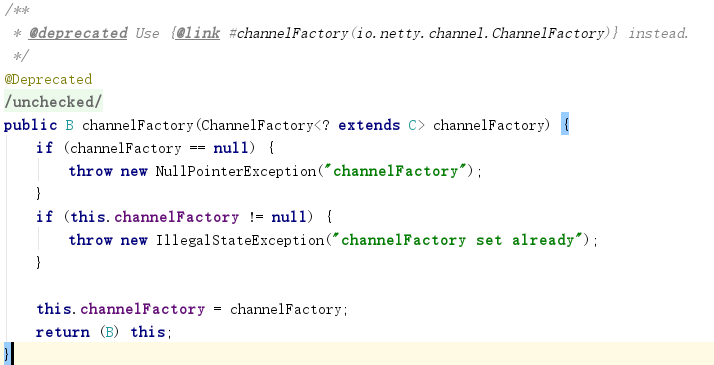
我们在创建 ServerBootstrap 实例时，会惊讶的发现 **ServerBootstrap 只有一个无参的构造函数**，作为启动辅助类这让人不可思议，因为它需要与多个其它组件或者类交互。ServerBootstrap 构造函数没有参数的根本原因是因为它的参数太多了，而且未来也可能会发生变化，为了解决这个问题，就需要引入**Builder 模式**。

* **步 骤 2：** 设 置 并 绑 定 Reactor 线 程 池。**Netty 的 Reactor 线 程 池 是EventLoopGroup**，它实际就是 EventLoop 的数组。EventLoop 的职责是处理所有注册到本线程多路复用器 Selector 上的 Channel，Selector 的轮询操作由绑定的 EventLoop 线程 run 方法驱动，在一个循环体内循环执行。值得说明的是，EventLoop 的职责不仅仅是处理网络 I/O 事件，用户**自定义的 Task 和定时任务Task** 也统一由 EventLoop 负责处理，这样线程模型就实现了统一。从调度层面看也不存在在 EventLoop 线程中再**启动其它类型的线程**用于异步执行其它的任务，这样就避免了多线程并发操作和锁竞争，提升了 I/O 线程的处理和调度性能。
* **步 骤 3：** 设 置 并 绑 定 服 务 端 Channel。 作 为 NIO 服 务 端， 需 要 创 建**ServerSocketChannel**，Netty 对 原 生 的 NIO 类 库 进 行 了 封 装， 对 应 实 现 是**NioServerSocketChannel**。对于用户而言，不需要关心服务端 Channel 的底层实现细节和工作原理，只需要指定具体使用哪种服务端 Channel 即可。因此，Netty 的 ServerBootstrap 方法*提供了 channel 方法用于指定服务端 Channel 的类型*。Netty 通过**工厂类**，利用**反射创建** NioServerSocketChannel 对象。由于服务端监听端口往往只需要在系统启动时才会调用，因此反射对性能的影响并不大。



通过反射机制获取ChanelFactory的实例：



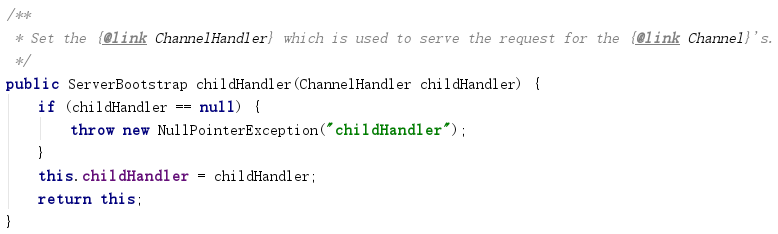


* **步骤 4：**链路建立的时候创建并初始化 **ChannelPipeline**。ChannelPipeline并不是 NIO 服务端必需的，它本质就是一个**负责处理网络事件的职责链，负责管理和执行ChannelHandler**。网络事件以事件流的形式在ChannelPipeline中流转，由 ChannelPipeline 根据 ChannelHandler 的**执行策略调度** ChannelHandler 的执行。典型的网络事件如下：

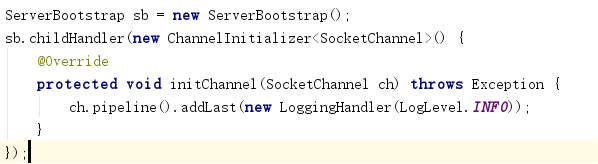
链路注册；链路激活；链路断开；接收到请求消息；请求消息接收并处理完毕；发送应答消息；链路发生异常；发生用户自定义事件。

* **步骤 5：**初始化 ChannelPipeline 完成之后，添加并设置 ChannelHandler。ChannelHandler 是 Netty 提 供 给 用 户 定 制 和 扩 展 的 关 键 接 口。 利 用ChannelHandler 用户可以完成大多数的**功能定制**，例如消息编解码、心跳、安全认证、TSL/SSL 认证、流量控制和流量整形等。Netty 同时也提供了大量的**系统** ChannelHandler 供用户使用，比较实用的系统 ChannelHandler 总结如下：
* 系统编解码框架 - ByteToMessageCodec；
* 通用基于长度的半包解码器 - LengthFieldBasedFrameDecoder;
* 码流**日志打印**Handler - LoggingHandler；
* SSL安全认证Handler - SslHandler；
* 链路空闲检测Handler - IdleStateHandler；
* 流量整形Handler - ChannelTrafficShapingHandler;
* Base64编解码 - Base64Decoder和Base64Encoder。

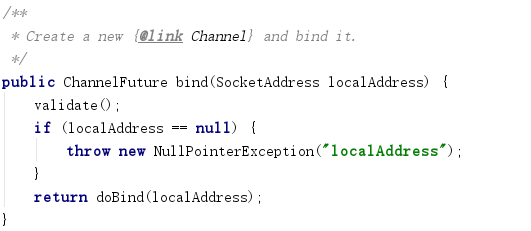
**创建和添加 ChannelHandler** 的代码示例如下：



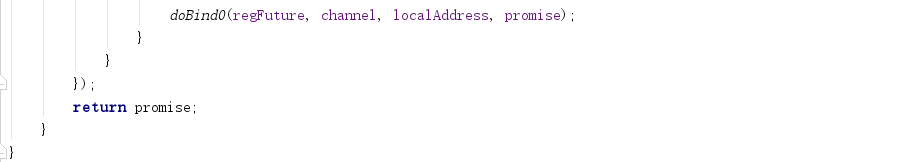
主函数：



* **步骤 6：**绑定并启动监听端口。在绑定监听端口之前系统会做一系列的**初始化和检测工作**，完成之后，会启动监听端口，并将 **ServerSocketChannel** 注册到Selector 上监听客户端连接，相关代码如下

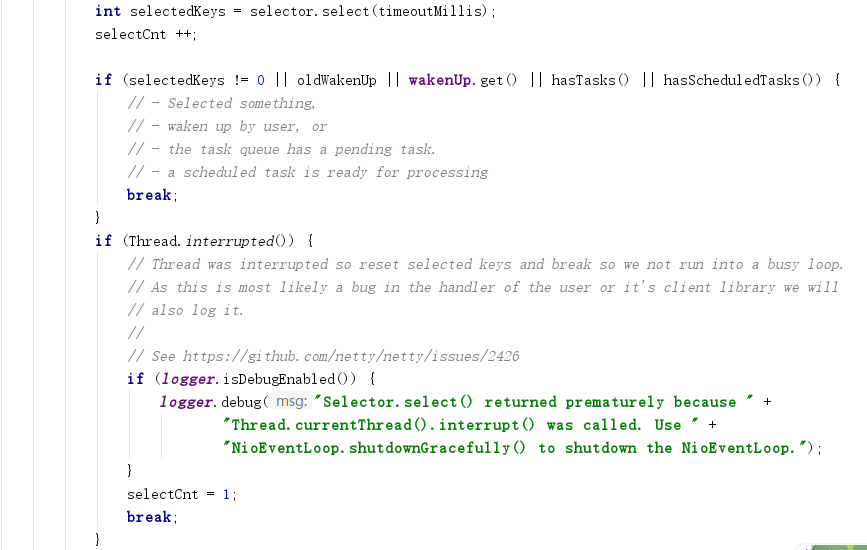


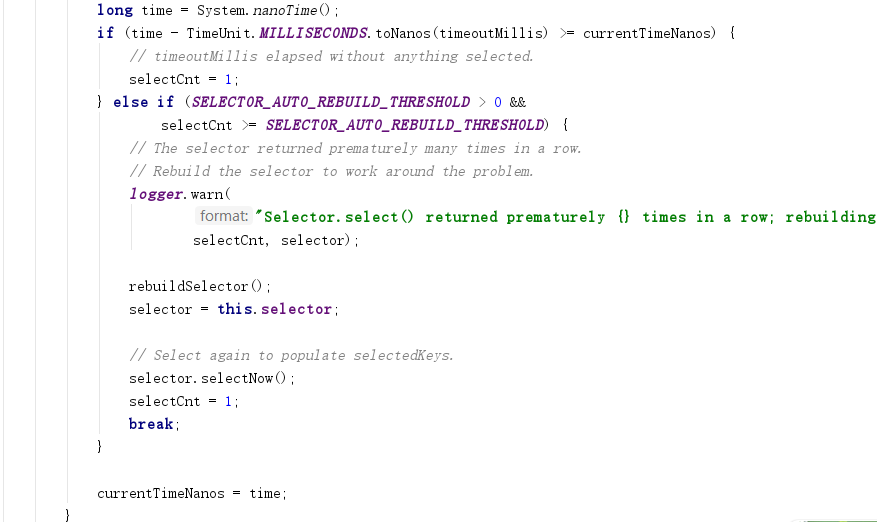


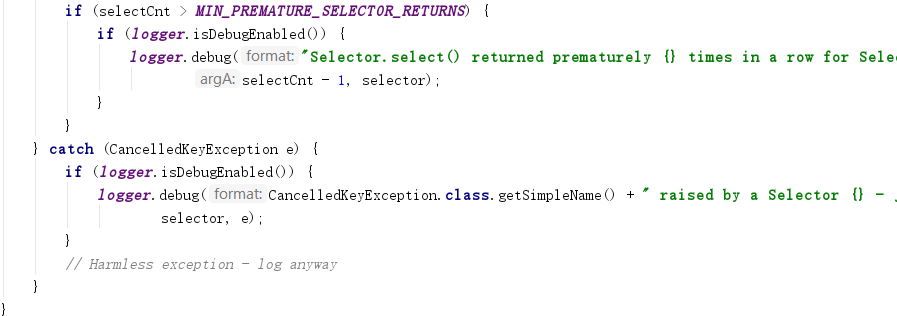


* **步骤 7：**Selector 轮询。由 Reactor 线程 **NioEventLoop** 负责**调度和执行Selector 轮询**操作，选择准备就绪的 Channel 集合，相关代码如下：

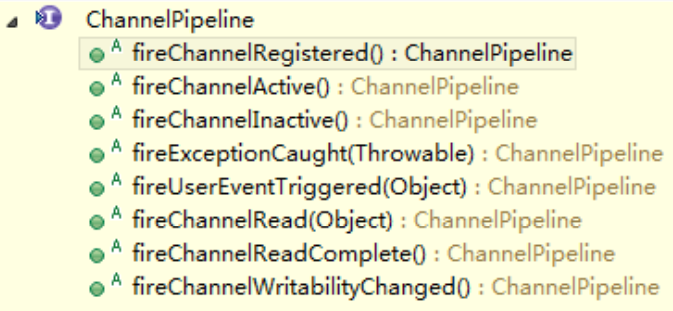




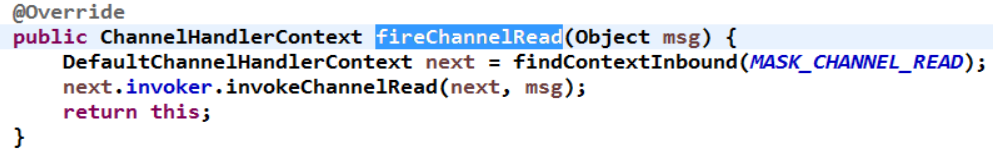




* **步 骤 8：** 当 轮 询 到 准 备 就 绪 的 Channel 之 后， 就 由 Reactor 线程 NioEventLoop 执 行 ChannelPipeline 的 相 应 方 法， 最 终 调 度 并 执 行ChannelHandler，代码如下：

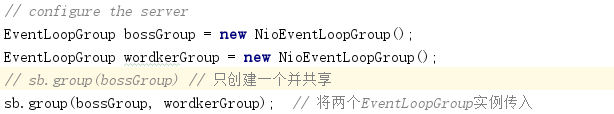


* **步 骤 9：** 执 行 Netty 系 统 ChannelHandler 和 用 户 添 加 定 制 的ChannelHandler。ChannelPipeline 根 据 网 络 事 件 的 类 型， 调 度 并 执 行ChannelHandler，相关代码如下所示：

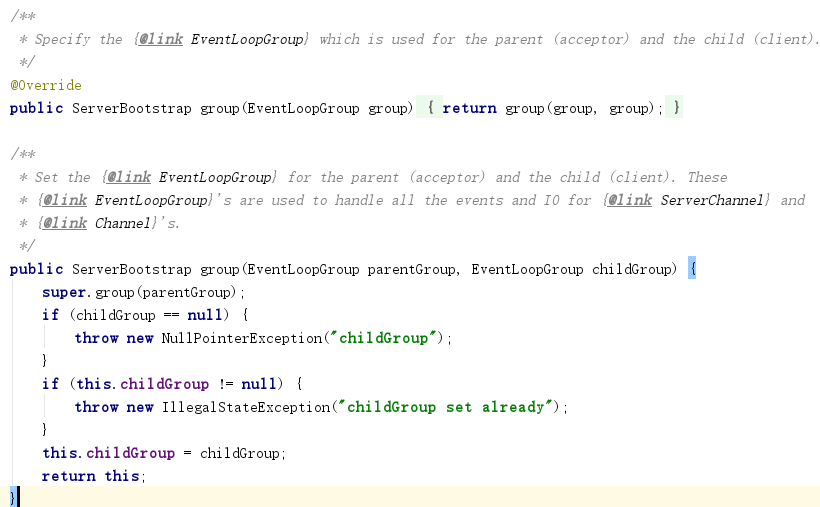


**2、服务端创建源码分析**

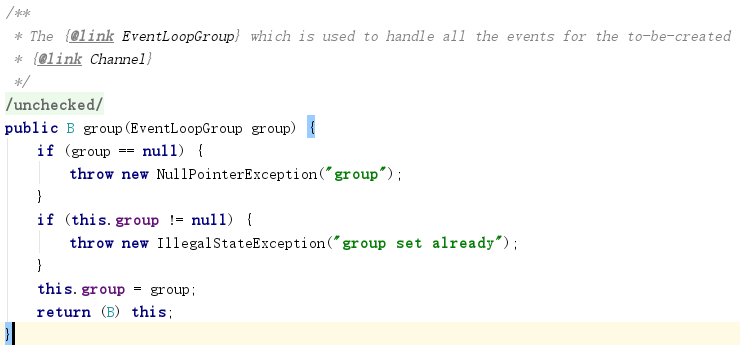
首先通过构造函数**创建 ServerBootstrap 实例**，随后，通常会**创建两个**EventLoopGroup（并不是必须要创建两个不同的 EventLoopGroup，也可以**只创建一个并共享**），代码如下所示：



NioEventLoopGroup 实际就是 Reactor 线程池，负责*调度和执行客户端的接入、网络读写事件的处理、用户自定义任务和定时任务的执行*。通过ServerBootstrap 的 group 方法将两个 EventLoopGroup **实例传入**，代码如下：

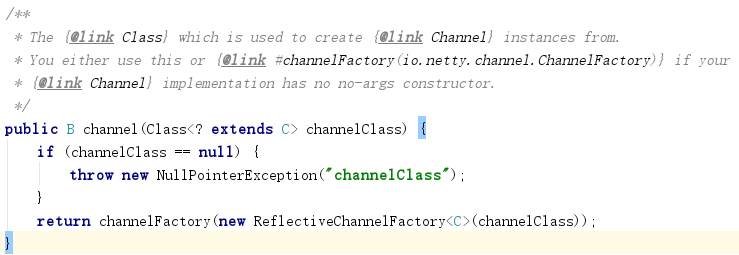


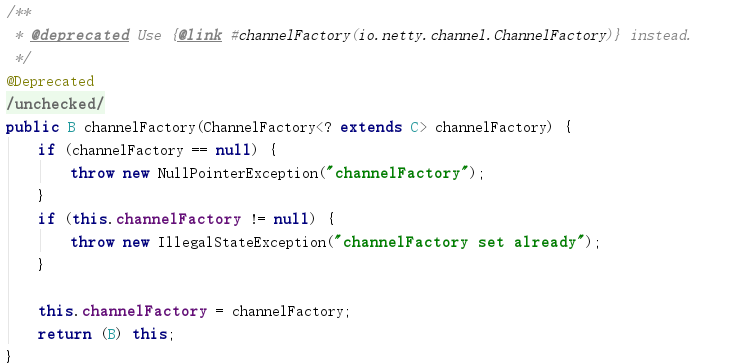
父类Group函数：

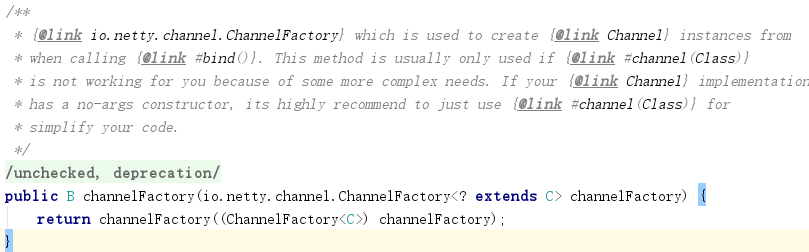


父 NioEventLoopGroup 被传入了**父类**构造函数中，该方法会被客户端和服务端重用，用于**执行和调度网络事件的读写。**

线 程 组 和 线 程 类 型 设 置 完 成 后， 需 要 *设 置 服 务 端 Channel*，Netty **通过 Channel 工 厂 类 来 创 建 不 同 类 型 的 Channel**， 对 于 服 务 端， 需 要 创 建NioServerSocketChannel。所以，通过指定 Channel 类型的方式创建 Channel 工厂。ReflectiveChannelFactory 职责是根据 Channel 的类型通过反射创建 Channel 的实例，服务端需要创建的是NioServerSocketChannel 实例，代码如下





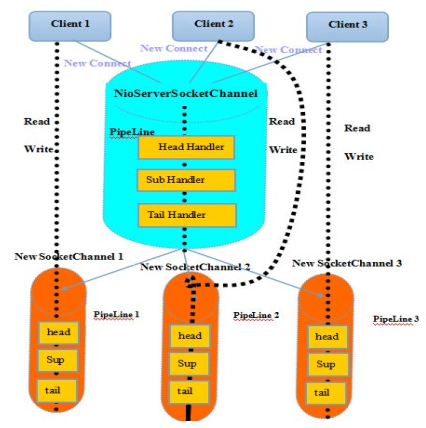


指定 NioServerSocketChannel 后，需要设置 TCP 的一些参数，作为服务端，主要是要设置 TCP 的 backlog 参数，底层 C 的对应接口定义如下：

int listen(int fd, int backlog);

backlog 指定了内核为**此套接口排队的最大连接个数，对于给定的监听套接口，内核要维护两个队列，未链接队列和已连接队列**，根据 TCP 三路握手过程中三个分节来分隔这两个队列。backlog 被规定为两个队列总和的最大值，大多数实现默认值为 5，但在高并发 web 服务器中此值显然不够，lighttpd 中此值达到 128\*8 。需要设置此值更大一些的原因是未完成连接队列的长度可能因为客户端 SYN 的到达及等待三路握手第三个分节的到达延时而增大。Netty 默认的backlog 为 100，当然，用户可以修改默认值，用户需要根据实际场景和网络状况进行灵活设置。

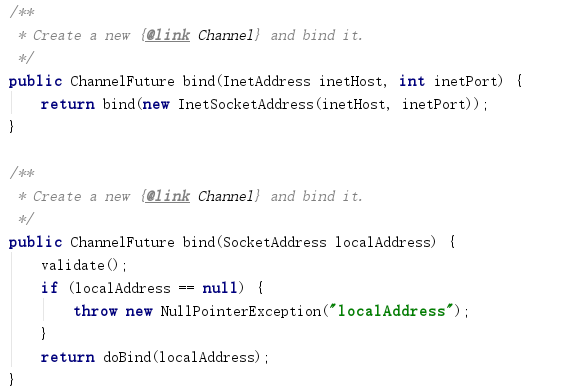
TCP 参数设置完成后，用户可以为**启动辅助类和其父类**分别指定 Handler，两 类 Handler 的 用 途 不 同， 子 类 中 的 Hanlder 是 NioServerSocketChannel 对应的 ChannelPipeline 的 Handler，父类中的 Hanlder 是*客户端新接入的连接SocketChannel 对应的 ChannelPipeline 的 Handler*。两者的区别可以通过下图来展示：



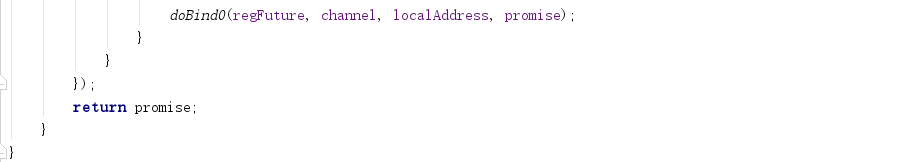
ServerBootstrap的Handle模型

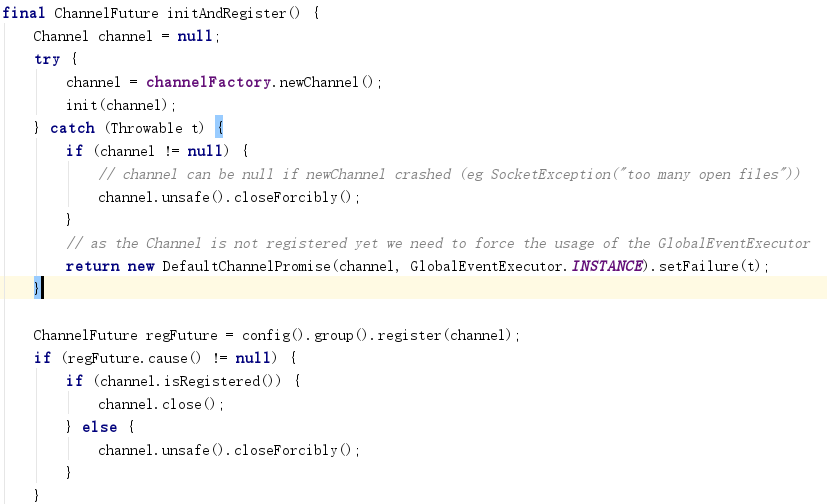
**本质区别就是：**ServerBootstrap 中的 Handler 是 NioServerSocketChannel使用的，所有连接该监听端口的客户端都会执行它，父类 AbstractBootstrap 中的 Handler 是个工厂类，它为每个新接入的客户端都创建一个新的 Handler。

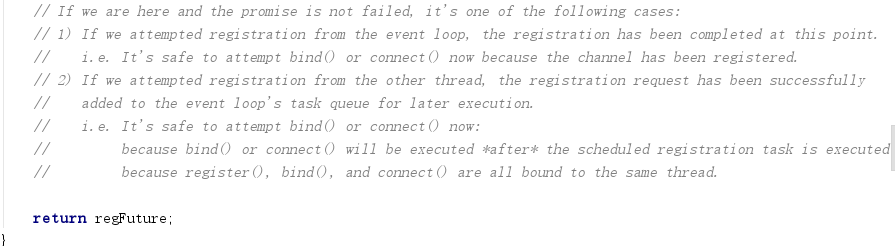
服务端启动的最后一步，就是**绑定本地端口，启动服务**，下面我们来分析下这部分代码：

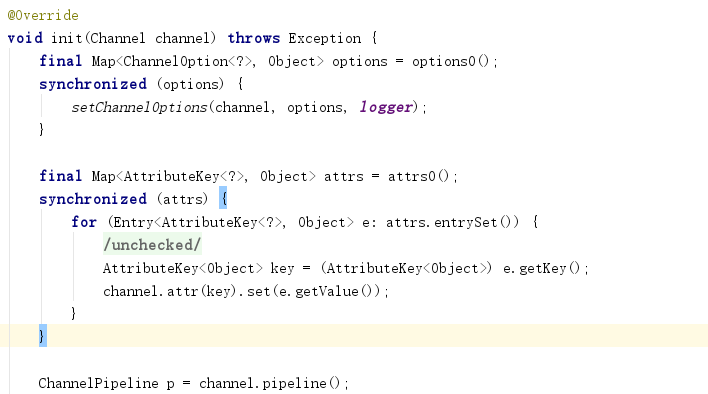






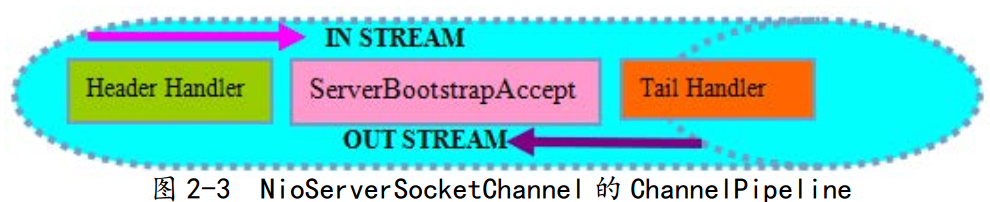








到此处，Netty 服务端监听的相关资源已经初始化完毕，就剩下最后一 步 - **注 册 NioServerSocketChannel 到 Reactor 线 程 的 多 路 复 用 器** 上， 然后**轮询客户端连接事件**。在分析注册代码之前，我们先通过下图看看目前NioServerSocketChannel 的 ChannelPipeline 的组成：



……

**3、客户端接入源码分析**

负 责 处 理 网 络 读 写、 连 接 和 客 户 端 请 求 接 入 的 Reactor 线 程 就 是NioEventLoop， 下 面 我 们 分 析 下 **NioEventLoop 是 如 何 处 理 新 的 客 户 端 连接 接 入 的**。 当 多 路 复 用 器 **检 测 到 新 的 准 备 就 绪 的 Channel** 时， 默 认 执 行**processSelectedKeysOptimized** 方法，代码如下：

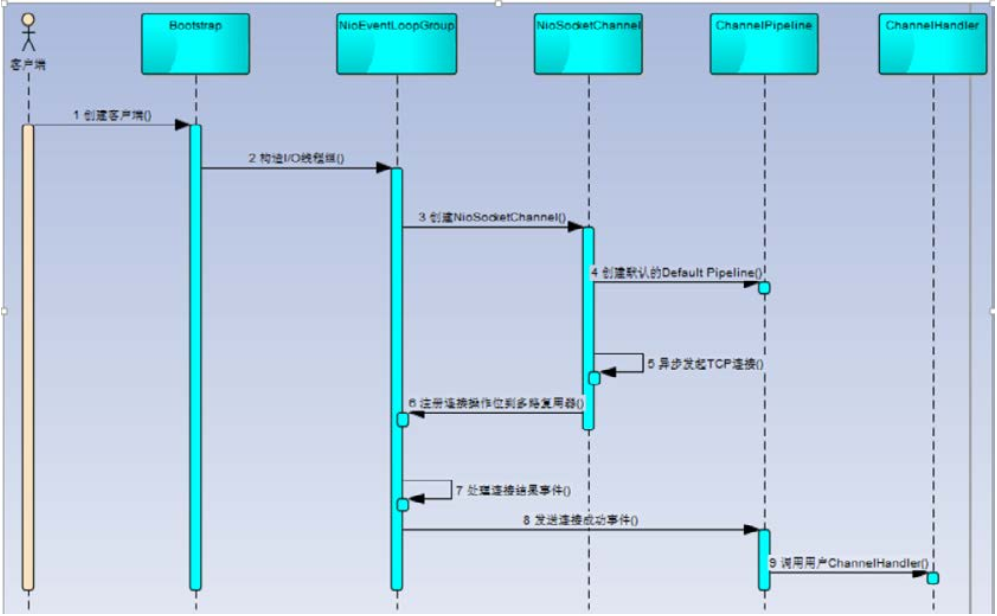
…..

**三、Netty 客户端创建**

Netty 为了向使用者屏蔽 NIO 通信的底层细节，在和用户交互的边界做了封装，目的就是为了减少用户开发工作量，降低开发难度。*Bootstrap 是 Socket客户端创建工具类，用户通过 Bootstrap 可以方便的创建 Netty 的客户端并发起异步 TCP 连接操作*。

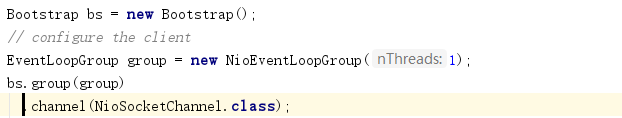
**1、Netty客户端创建时序图**

* **步骤 1：**用户线程创建 Bootstrap 实例，通过 API 设置创建客户端相关的参数，异步发起客户端连接。
* **步 骤 2：** 创 建 处 理 客 户 端 连 接、I/O 读 写 的 Reactor 线 程 组NioEventLoopGroup, 可以*通过构造函数指定 I/O 线程的个数*，默认为 CPU 内核数的 2 倍；
* **步骤 3**：通过 Bootstrap 的 **ChannelFactory** 和用户**指定的 Channel 类型**创建用于客户端连接的 **NioSocketChannel**，它的功能类似于 JDK NIO 类库提供的SocketChannel;
* **步骤 4：**创建默认的 Channel Handler Pipeline，用于调度和执行网络事件；
* **步骤 5：**异步发起 TCP 连接，判断连接是否成功，如果成功，则直接将NioSocketChannel 注册到**多路复用器**上，监听读操作位，用于数据报读取和消息发送；如果没有立即连接成功，则注册连接监听位到多路复用器，等待连接结果；
* **步骤 6：**注册对应的网络监听状态位到多路复用器；
* **步骤 7：**由多路复用器在 I/O 现场中轮询各 Channel，处理连接结果；
* **步骤 8：**如果连接成功，设置 Future 结果，发送连接成功事件，触发ChannelPipeline 执行；
* **步骤 9：**由 ChannelPipeline 调度执行系统和用户的 ChannelHandler，执行业务逻辑



**2、Netty客户端创建源码分析**

首 先， 创 建 Bootstrap 的 实 例， 类 似 ServerBootstrap, 客 户 端 也 使 用Builder 模式来构造。对于客户端，由于它不需要监听和处理来自客户端的连接，所以，只需要一个 Reactor 线程组即可，且完成连接辅助类和 Reactor 线程组的初始化操作后，继续设置发起连接的Channel 为 **NioSocketChannel**，代码如下：



**四、Netty 示例**

<http://netty.io/wiki/user-guide-for-4.x.html#wiki-h3-6>

<http://blog.csdn.net/albertfly/article/details/51526322>

**1、DISCARD服务**

丢弃服务，指的是会忽略所有接收的数据的一种协议

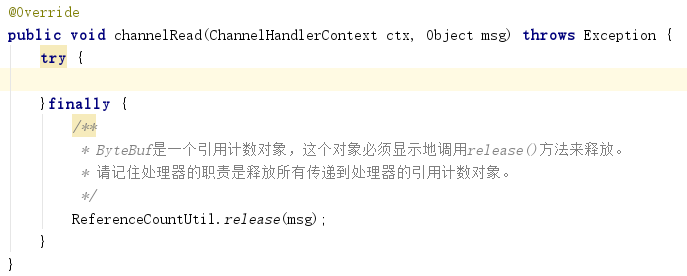






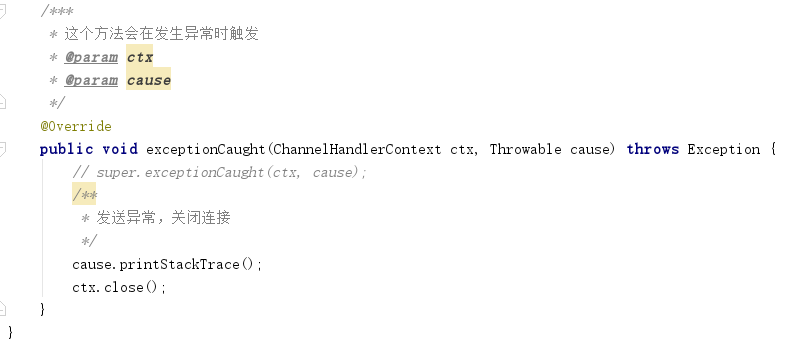


* **DiscardServerHandler** extends **ChannelInboundHandlerAdapter**, which is an implementation of **ChannelInboundHandler**. ChannelInboundHandler provides various event handler methods that you can override. For now, it is just enough to extend ChannelInboundHandlerAdapter rather than to implement the handler interface by yourself.
* We override the **channelRead()** event handler method here. This method is called with the received message, whenever new data is received from a client. In this example, the type of the received message is **ByteBuf**.
* To implement the DISCARD protocol, the handler has to ignore the received message. **ByteBuf is a reference-counted object which has to be released explicitly via the release() method**. Please keep in mind that it is the handler's responsibility to release any reference-counted object passed to the handler. Usually, channelRead() handler method is implemented like the following:



完整代码：



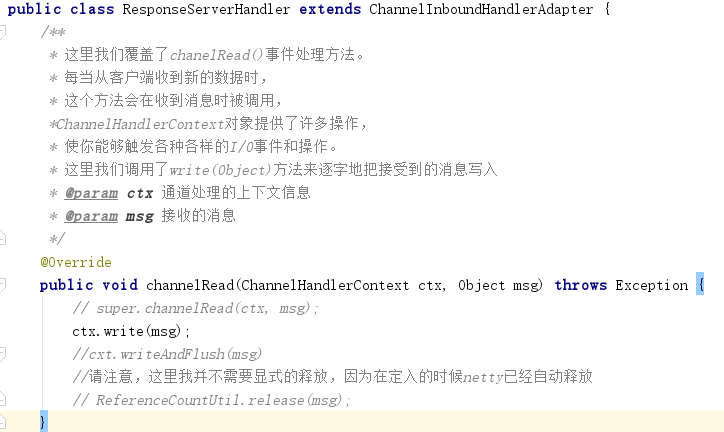


以上是一个丢弃服务的处理方式，你可以运行后通过telnet来发送消息，来查看是否正常运行，注意console里会打印你的输入内容。

**2、ECHO服务（响应式协议）**

到目前为止，我们虽然接收到了数据，但没有做任何的响应。然而**一个服务端通常会对一个请求作出响应**。让我们学习怎样在ECHO协议的实现下编写一个响应消息给客户端，这个协议**针对任何接收的数据都会返回一个响应**。

和discard server唯一不同的是**把在此之前我们实现的channelRead()方法，返回所有的数据替代打印接收数据到控制台上的逻辑**。说明NettyServer 还是用上面已经提供的类，只是把这段里的注销部分修改成如下。





**输出结果：**命令行console中打印出所输入的字符

**3、TIME服务(时间协议的服务)**

在这个部分被实现的协议是TIME协议。和之前的例子不同的是在不接受任何请求时他会发送一个含32位的整数的消息，并且一旦消息发送就会立即关闭连接。在这个例子中，你会学习到如何构建和发送一个消息，然后在完成时主动关闭连接。

因为我们将会**忽略任何接收到的数据，而只是在连接被创建发送一个消息**，所以这次我们不能使用channelRead()方法了，代替他的是，我们需要覆盖**channelActive()**方法，下面的就是实现的内容：说明NettyServer 还是用上面已经提供的类，只是把这段里的注销部分修改成如下。

不像DISCARD和ECHO的服务端，对于TIME协议我们需要一个客户端因为人们不能把一个32位的二进制数据翻译成一个日期或者日历。在这一部分，我们将会讨论如何确保服务端是正常工作的，并且学习怎样用Netty编写一个客户端。在Netty中,**编写服务端和客户端最大的并且唯一不同的使用了不同的BootStrap和Channel的实现。**