# 【第13章-服务端创建】【02-服务端源码分析】

## 服务端创建源码分析

### Netty服务端创建时序图

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 可以方便地创建Netty的服务端。时序图如图1 所示。



图-1 Netty服务端创建时序图

步骤l ：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，以降低用户的开发难度。

步骤2 ：设置并绑定Reactor 线程池。Netty 的Reactor 线程池是Eve n tLoopG roup ， 它实际就是EventLoop 的数组。Eve ntLo op 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。EventLoop 的职责不仅仅是处理网络IO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 。从调用层面看，也不存在从EventLoop 线程中再启动其他类型的钱程用于异步执行另外的任务，这样就避免了多线程并发操作和锁竞争，提升了IO 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 。对子用户而言，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。相关代码如下。

|  |
| --- |
| **// io.netty.bootstrap.AbstractBootstrap#channel**  **public** B channel(Class<? **extends** C> channelClass) {  **if** (channelClass == **null**) {  **throw new** NullPointerException(**"channelClass"**);  }  **return** channelFactory(**new** ReflectiveChannelFactory<C>(channelClass)); } |

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。

1. 链路注册
2. 链路激活
3. 链路断开
4. 接收到请求消息
5. 请求消息接收并处理完毕
6. 发送应答消息
7. 链路发生异常
8. 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。

1. 系统编解码框架——ByteTo MessageCodec
2. 通用基于长度的半包解码器——LengthFieldBasedFrameDecoder
3. 码流日志打印Handler——LoggingHandler
4. SSL 安全认证Handler——SslHandler
5. 链路空闲检测Handler——IdleStateHandler
6. 流量整形Handler一一ChannelTrafficShapingHandler
7. Base64 编解码——Base64Decoder 和Base64Encoder。

创建和添加ChannelHandler的代码示例如下。

|  |
| --- |
| .childHandler(**new** ChannelInitializer<SocketChannel>() {  @Override  **public void** initChannel(SocketChannel ch) **throws** IOException {  ch.pipeline().addLast(**new** EchoServerHandler());  } }); |

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，会启动监昕端口，并将ServerSocketChannel 注册到Selector 上监听客户端连接， 相关代码如下。

|  |
| --- |
| **// io.netty.bootstrap.AbstractBootstrap#doBind**  **private** ChannelFuture doBind(**final** SocketAddress localAddress) {  **final** ChannelFuture regFuture = initAndRegister();  **final** Channel channel = regFuture.channel();  **if** (regFuture.cause() != **null**) {  **return** regFuture;  }   **if** (regFuture.isDone()) {ChannelPromise promise = channel.newPromise();  *doBind0*(regFuture, channel, localAddress, promise);  **return** promise;  } **else** {**final** PendingRegistrationPromise promise = **new** PendingRegistrationPromise(channel);  regFuture.addListener(**new** ChannelFutureListener() {  @Override  **public void** operationComplete(ChannelFuture future) **throws** Exception {  Throwable cause = future.cause();  **if** (cause != **null**) {promise.setFailure(cause);  } **else** {promise.**executor** = channel.eventLoop();  }  *doBind0*(regFuture, channel, localAddress, promise);  }  });  **return** promise;  } } |

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。

|  |
| --- |
| **private void** select(**boolean** oldWakenUp) **throws** IOException {  // 此处代码省略．．.  **int** selectedKeys = selector.select(timeoutMillis);  selectCnt ++;  // 此处代码省略．．. } |

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。

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。

|  |
| --- |
| **// io.netty.channel.AbstractChannelHandlerContext#fireChannelRead**  @Override **public** ChannelHandlerContext fireChannelRead(Object msg) {  AbstractChannelHandlerContext next = findContextInbound();  ReferenceCountUtil.*touch*(msg, next);  **invokedNextChannelRead** = **true**;  next.invoker().invokeChannelRead(next, msg);  **return this**; } |

### Netty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。

|  |
| --- |
| EventLoopGroup bossGroup = **new** NioEventLoopGroup(); EventLoopGroup workerGroup = **new** NioEventLoopGroup(); |

NioEventLoopGroup 实际就是Reactor线程池， 负货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。

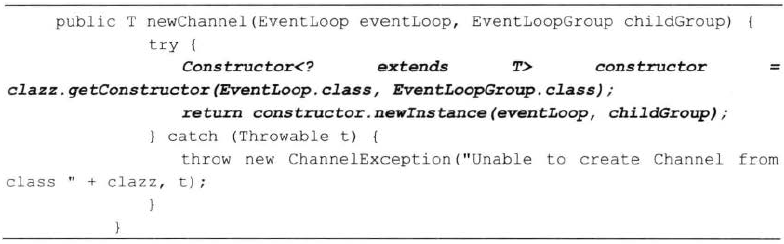
|  |
| --- |
| **// io.netty.bootstrap.ServerBootstrap#group(io.netty.channel.EventLoopGroup, io.netty.channel.EventLoopGroup)**  **public** ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {  **super**.group(parentGroup);  **if** (childGroup == **null**) {  **throw new** NullPointerException(**"childGroup"**);  }  **if** (**this**.**childGroup** != **null**) {  **throw new** IllegalStateException(**"childGroup set already"**);  }  **this**.**childGroup** = childGroup;  **return this**; } |

其中父NioEventLoopGroup 被传入了父类构造函数中， 代码如下。

|  |
| --- |
| **// io.netty.bootstrap.AbstractBootstrap#group(io.netty.channel.EventLoopGroup)**  **public** B group(EventLoopGroup group) {  **if** (group == **null**) {  **throw new** NullPointerException(**"group"**);  }  **if** (**this**.**group** != **null**) {  **throw new** IllegalStateException(**"group set already"**);  }  **this**.**group** = group;  **return** (B) **this**; } |

该方法会被客户端和服务端重用， 用于设置工作IO 线程，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 用于端口监听和客户端链路接入。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。ServerBootstrapChannelFactory 是ServerBootstrap 的内部静态类，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



在源代码中未找到

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。



在哪里可以找到代码？

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：未链接队列和己连接队列，根据TCP 三路握手过程中三个分节来分隔这两个队列。服务器处于listen 状态时， 收到客户端syn 分节（ connect ）时在未完成队列中创建一个新的条目，然后用三路握手的第二个分节即服务器的syn 响应客户端，此条自在第三个分节到达前（客户端对服务器syn 的ack ）一直保留在未完成连接队列中，如果三路握手完成，该条目将从未完成连接队列搬到已完成连接队列尾部。当进程调用accept时，从己完成队列中的头部取出一个条目给进程， 当已完成队列为空时进程将睡眠，直到有条目在已完成连接队列中才唤醒。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l 28\*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，这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Handler 是NioServerSocketChannel 对应的ChannelPipeline 的Handler：父类中的Handler 是客户端新接入的连接SocketChannel 对应的Channel Pipeline的Handler 。两者的区别可以通过图-3 来展示。



图-3 ServerBootstrap的Handler 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。

|  |
| --- |
| **// io.netty.bootstrap.AbstractBootstrap#doBind**  **private** ChannelFuture doBind(**final** SocketAddress localAddress) {  **final** ChannelFuture regFuture = initAndRegister(); // NO.1  **final** Channel channel = regFuture.channel();  **if** (regFuture.cause() != **null**) {  **return** regFuture;  }   **if** (regFuture.isDone()) { // NO.2ChannelPromise promise = channel.newPromise();  *doBind0*(regFuture, channel, localAddress, promise);  **return** promise;  } **else** {**final** PendingRegistrationPromise promise = **new** PendingRegistrationPromise(channel);  regFuture.addListener(**new** ChannelFutureListener() {  @Override // NO.2  **public void** operationComplete(ChannelFuture future) **throws** Exception {  Throwable cause = future.cause();  **if** (cause != **null**) {promise.setFailure(cause);  } **else** {promise.**executor** = channel.eventLoop();  }  *doBind0*(regFuture, channel, localAddress, promise);  }  });  **return** promise;  } } |

先看下NO.1 。首先创建Channel，createChannel 由子类ServerBootstrap 实现，创建新的NioServerSocketChannel 。它有两个参数： 参数1 是从父类的NIO 线程池中顺序获取一个NioEventLoop ， 它就是服务端用于监听和接收客户端连接的Reactor 线程： 参数2 是所谓的workerGroup 线程池，它就是处理IO读写的Reactor 线程组，相关代码如下。

|  |
| --- |
| **// io.netty.bootstrap.AbstractBootstrap#initAndRegister**  **final** ChannelFuture initAndRegister() {  **final** Channel channel = channelFactory().newChannel();  **try** {  init(channel);  } **catch** (Throwable t) {  channel.unsafe().closeForcibly();**return new** DefaultChannelPromise(channel, GlobalEventExecutor.***INSTANCE***).setFailure(t);  }   ChannelFuture regFuture = group().register(channel);  **if** (regFuture.cause() != **null**) {  **if** (channel.isRegistered()) {  channel.close();  } **else** {  channel.unsafe().closeForcibly();  }  }**return** regFuture; } |

NioServerSocketChannel 创建成功后，对它进行初始化， 初始化工作主要有以下三点。

1. 设置Socket 参数和N ioServerSocketChannel 的附加属性。
2. 将AbstractBootstrap 的Handler 添加到NioServerSocketChannel 的ChannelPipeline中。
3. 将用于服务端注册的HandlerServerBootstrapAcceptor 添加到ChannelPipeline 中。

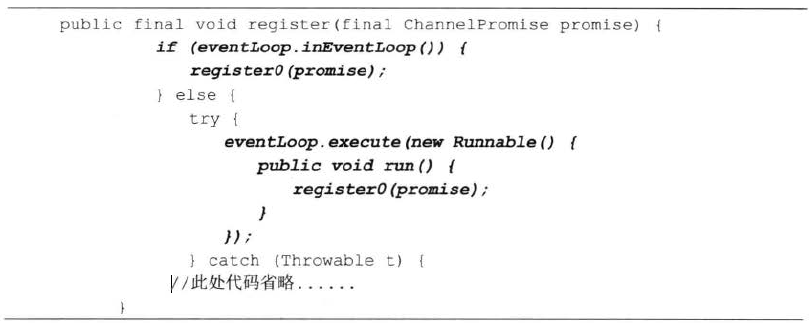
|  |
| --- |
| // io.netty.bootstrap.ServerBootstrap#init  @Override **void** init(Channel channel) **throws** Exception {  // （1）  **final** Map<ChannelOption<?>, Object> options = options();  **synchronized** (options) {  channel.config().setOptions(options);  }   **final** Map<AttributeKey<?>, Object> attrs = attrs();  **synchronized** (attrs) {  **for** (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {  @SuppressWarnings(**"unchecked"**)  AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();  channel.attr(key).set(e.getValue());  }  }  // （2）  ChannelPipeline p = channel.pipeline();  **if** (handler() != **null**) {  p.addLast(handler());  }   **final** EventLoopGroup currentChildGroup = **childGroup**;  **final** ChannelHandler currentChildHandler = **childHandler**;  **final** Entry<ChannelOption<?>, Object>[] currentChildOptions;  **final** Entry<AttributeKey<?>, Object>[] currentChildAttrs;  **synchronized** (**childOptions**) {  currentChildOptions = **childOptions**.entrySet().toArray(*newOptionArray*(**childOptions**.size()));  }  **synchronized** (**childAttrs**) {  currentChildAttrs = **childAttrs**.entrySet().toArray(*newAttrArray*(**childAttrs**.size()));  }  // （3）  p.addLast(**new** ChannelInitializer<Channel>() {  @Override  **public void** initChannel(Channel ch) **throws** Exception {  ch.pipeline().addLast(**new** ServerBootstrapAcceptor(  currentChildGroup, currentChildHandler, currentChildOptions, currentChildAttrs));  }  }); } |

到此， Netty 服务端监听的相关资源已经初始化完毕， 就剩下最后一步一一注册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，我们先通过图-4 看看目前NioServerSocketChannel 的ChannelPipeline的组成。



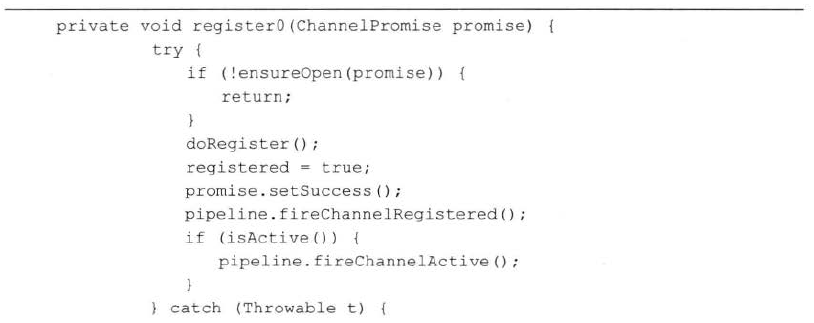
图-4 NioServerSocketChannel 的ChannelPipeline

最后，我们看下NioServerSocketChannel 的注册。当NioServerSocketCbannel 初始化完成之后，需要将它注册到Reactor 线程的多路复用器上监听新客户端的接入，代码如下。



源码中没有找到

首先判断是否是NioEventLoop 自身发起的操作。如果是，则不存在并发操作， 直接执行Channel 注册： 如果由其他线程发起，则封装成一个Task 放入消息队列中异步执行。此处，由于是由ServerBootstrap 所在线程执行的注册操作， 所以会将其封装成Task 投递到NioEventLoop 中执行，代码如下。



源码中没有找到

将NioServerSocketChannel 注册到NioEventLoop 的Selector 上，代码如下：