# 【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**; } |