# 背景

# 概述

组成部分

# 功能

## 3.1服务端启动类ServerBootstrap

### 3.1.1ServerBootstrap构造

图3-1-1-1

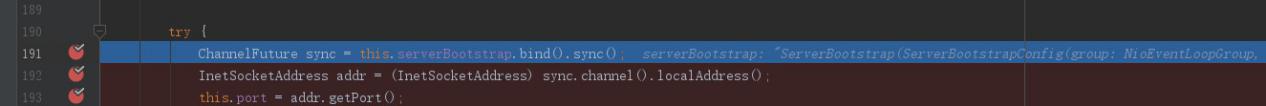


图3-1-1-1 所示，是netty 服务端导向启动类示例（参考rocketmq)

1. 、初始化this.eventLoopGroupBoss对象 参考3.2
2. 、初始化this.eventLoopGroupSelector对象参考3.2
3. 、group方法中，如果当前运行环境是windows，中eventLoopGroupBoss、eventLoopGroupSelector参数是 NioEventLoopGroup类实例；如果是linux环境，eventLoopGroupBoss、eventLoopGroupSelector对象是EpollEventLoopGroup类实例。
4. 、channel方法中，如果当前运行环境是windows，NioServerSocketChannel.class 实例；如果当前环境是linux，EpollServerSocketChannel.class实例，并把Class 实例经过包装赋值给AbstractBootstrap.channelFactory
5. 、option() 方法，把ChannelOption实例 添加到AbstractBootstrap.options Map对象中。
6. 、childOption() 方法，把ChannelOption实例添加到ServerBootstrap.childOptions Map对象中。
7. 、localAddress()方法作用： 绑定服务端ip、端口号。
8. 、childHandler()方法 ，就是把ChannelHandler实例赋值给ServerBootstrap.childHandler成员变量

### 3.1.2 ServerBootstrap 创建通道并绑定

图3-1-2-1



1. 执行AbstractBootstrap.bind()方法

图3-1-2-2

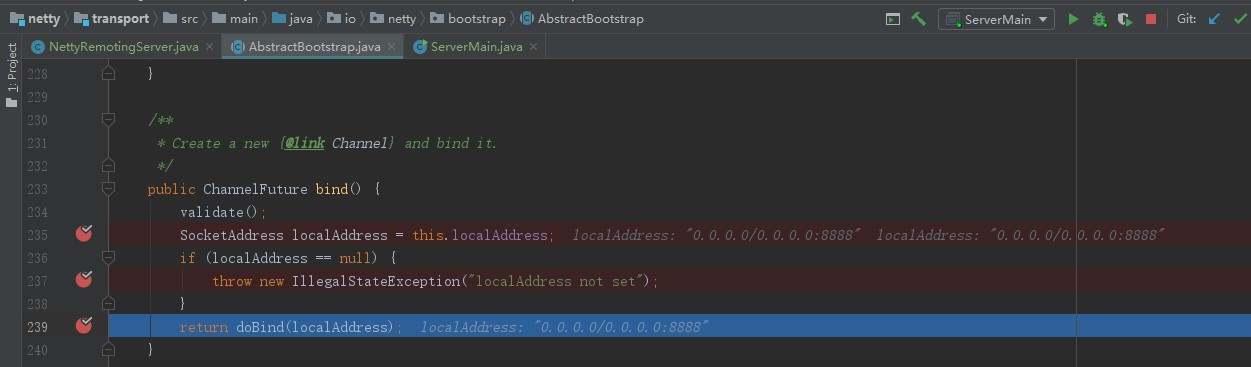
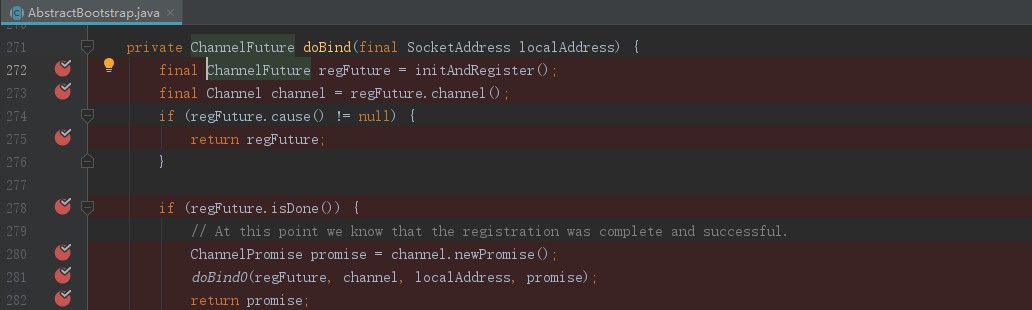


图3-1-2-3



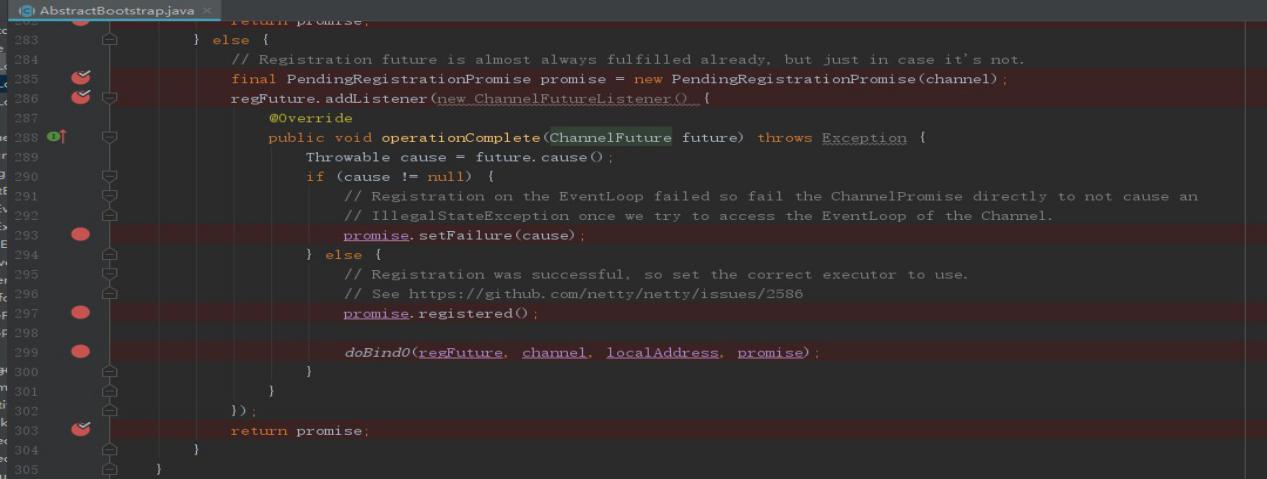
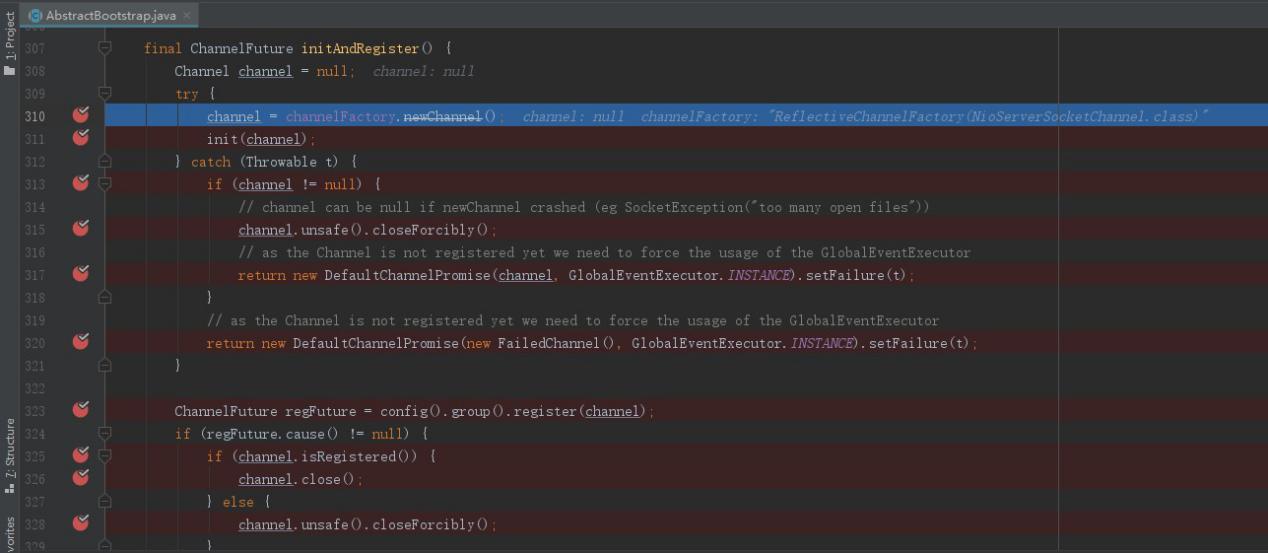


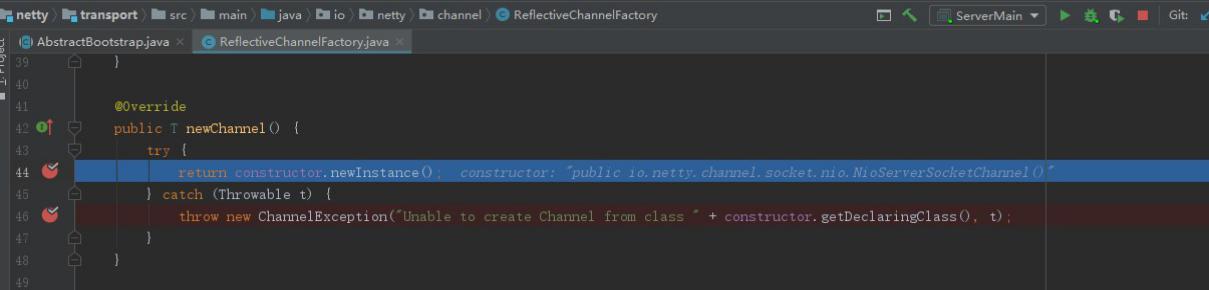
图3-1-2-4



1）图3-1-2-4 310行 通过ReflectiveChannelFactory 创建Channel实例。

具体创建哪EpollServerSocketChannel还是NioServerSocketChannel 要根据当前运行环境，windows是NioServerSocketChannel。

图3-1-2-5

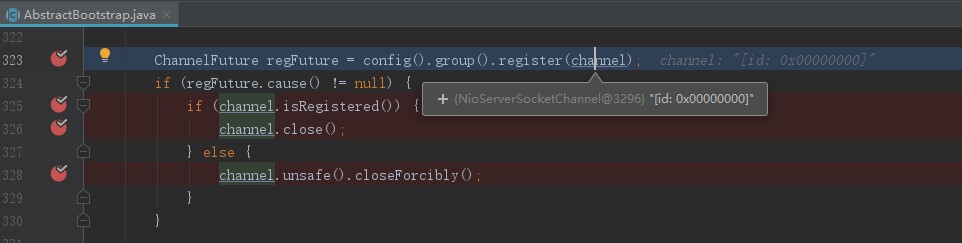
其实就是NioServerSocketChannel 的Constructor构造方法实例

构造NioServerSocketChannel 参考(3.6)

1. 、图3-1-2-4 310行参考3.1.3(ServerBootstrap.init()方法）
2. 、图3-1-2-4 323行 config()获取NioServerSocketChannelConfig实例,config().group()获取

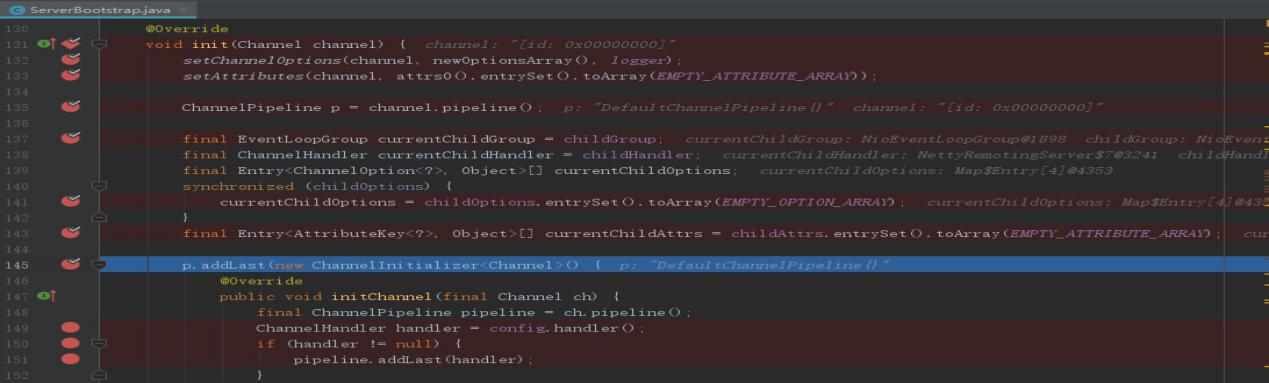
NioEventLoopGroup 实例，config().group().register(channel)即执行NioEventLoopGroup的register()方法，入参为3-1-2-6所示NioServerSocketChannel实例。(参考3.2.3)

图3-1-2-6



### 3.1.3 ServerBootstrap.init()方法

图3-1-3-1





1）、132行初始化相关option选项。如图3-1-3-2通过channel.config()获取

NioServerSocketChannelConfig实例，再通过channel.config.setOption方法

实际设置参数图如3-1-3-3、图3-1-3-4、图3-1-3-5所示，当然不同的option

具体设置可能不同，例如3-1-3-6、3-1-3-7。

图3-1-3-2

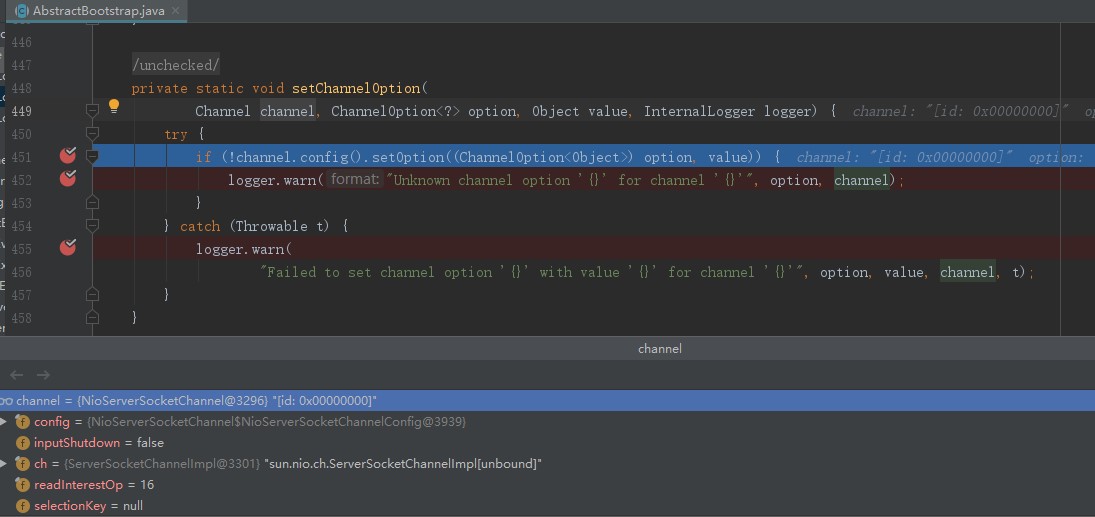


图3-1-3-3

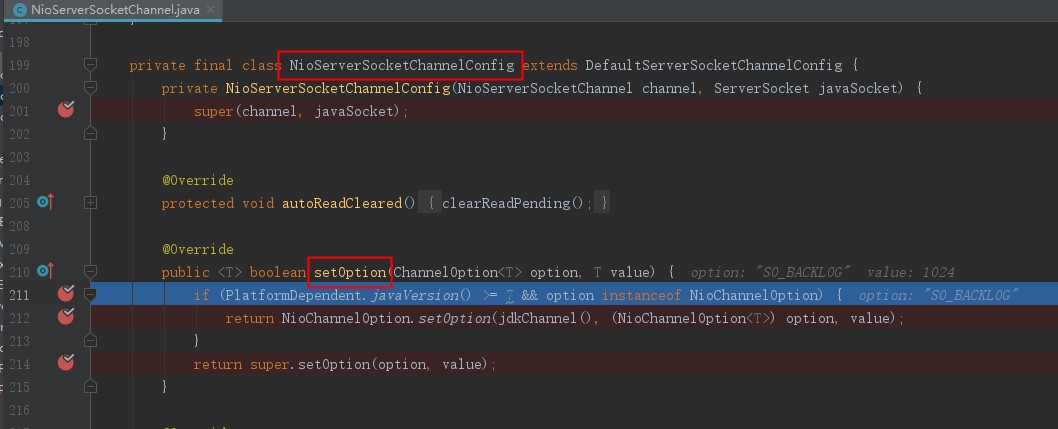


图3-1-3-4

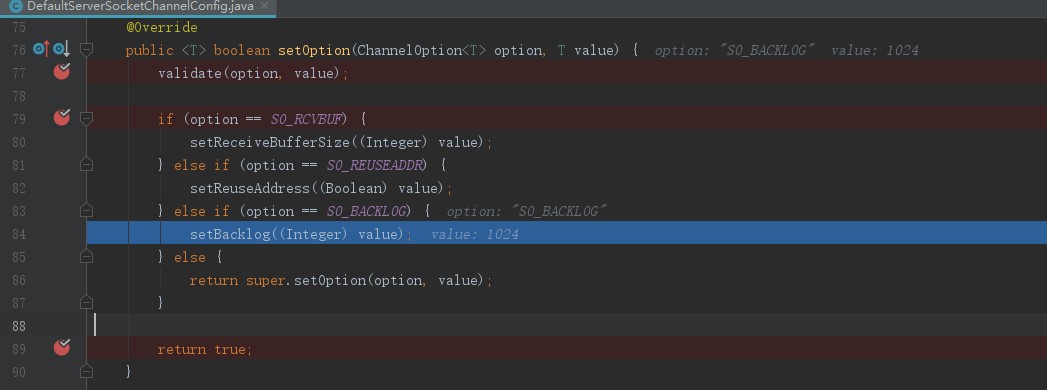


图3-1-3-5

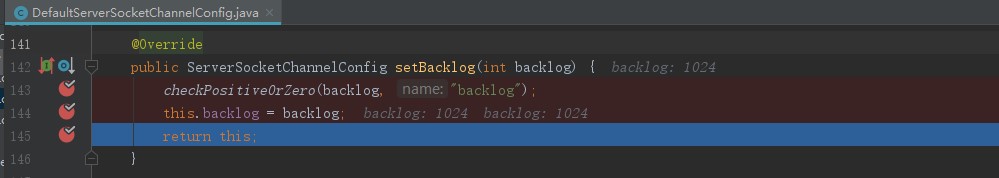


图 3-1-3-6

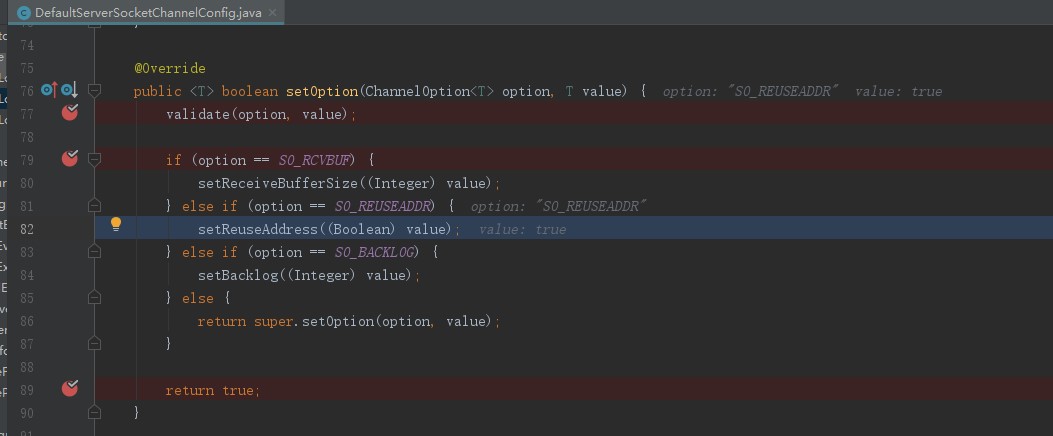
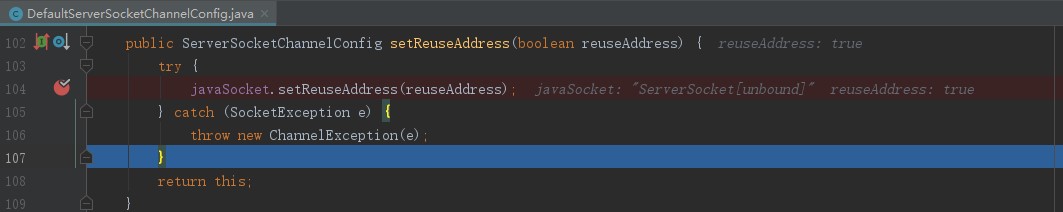


图3-1-3-7

、

2）、attrs设置

3）、145行，p即DefaltChannelPipeline实例，执行addLast方法，参考3-1-3-8、3-1-3-所示，3-1-3-9 executor为null。

图3-1-3-8

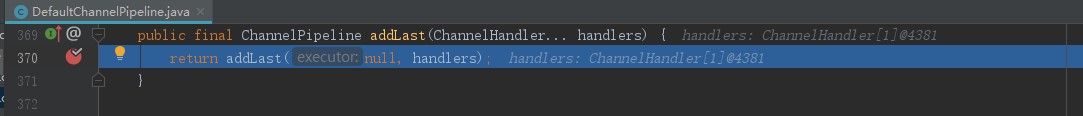


图3-1-3-9



图3-1-3-10

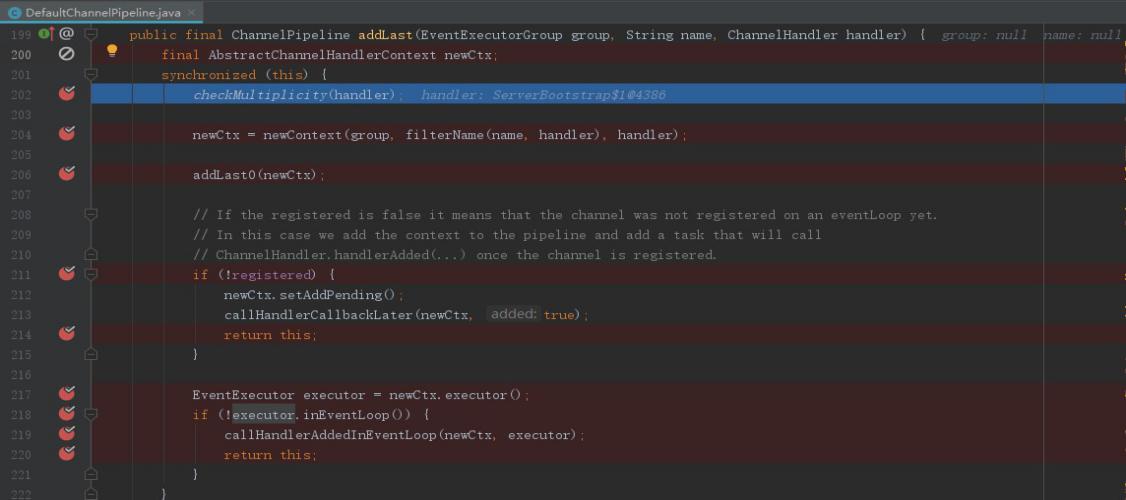




图3-1-3-10 202行校验是否重复添加或没有@Sharable注解 如果直接抛出异常，否则该设置ChannelHandler.added=true。

图3-1-3-10 204行 创建newCtx实例，参考3-1-3-11

图3-1-3-10 206行，把DefaultChannelHandlerContext newCtx实例添加到双向链表尾部，参考图3-1-3-14

图3-1-3-11

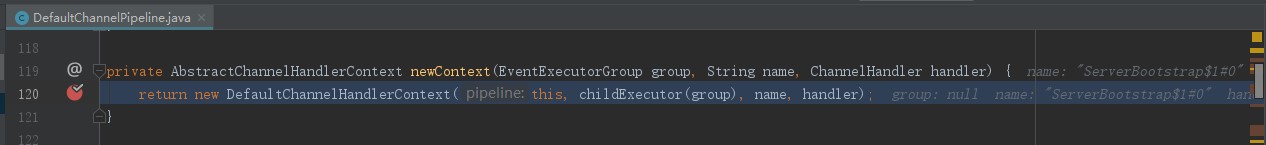
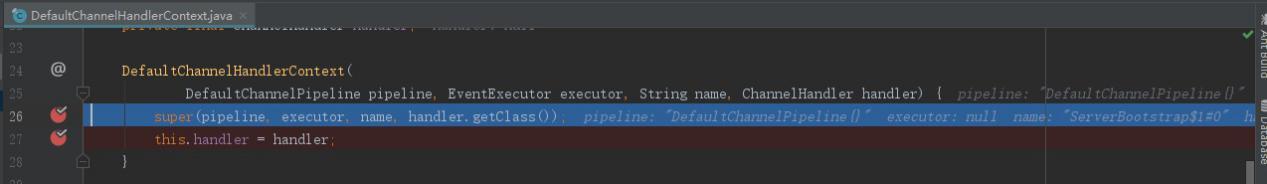


图3-1-3-12



piepeline为DefaultChannelPipeline实例，

executor为null

略

图3-1-3-13

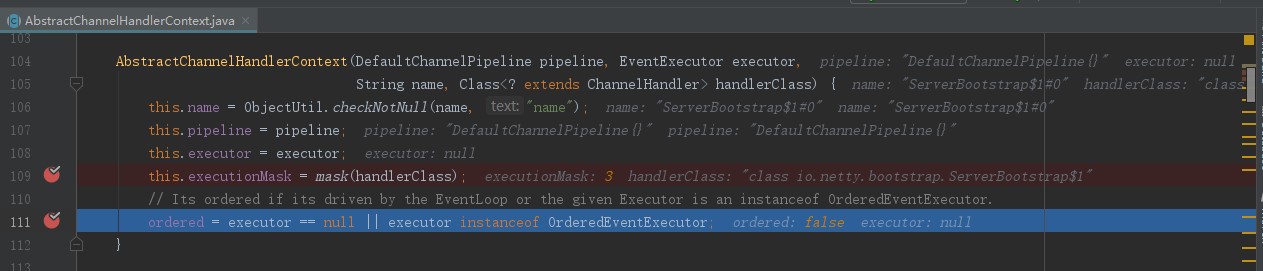
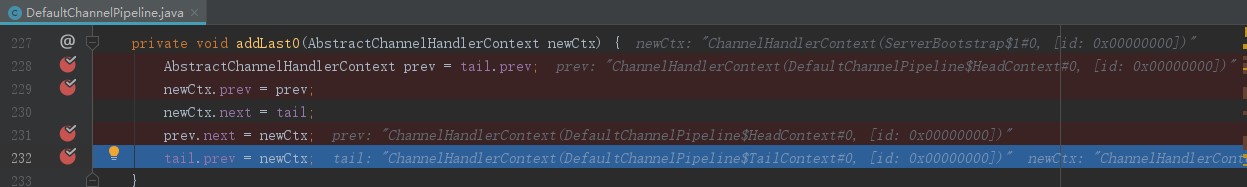


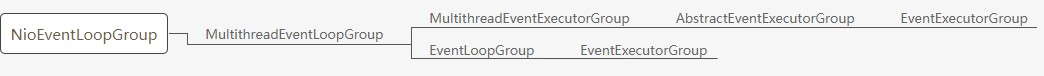
图3-1-3-14



## 3.2 NioEventLoopGroup类

### 3.2.1继承关系

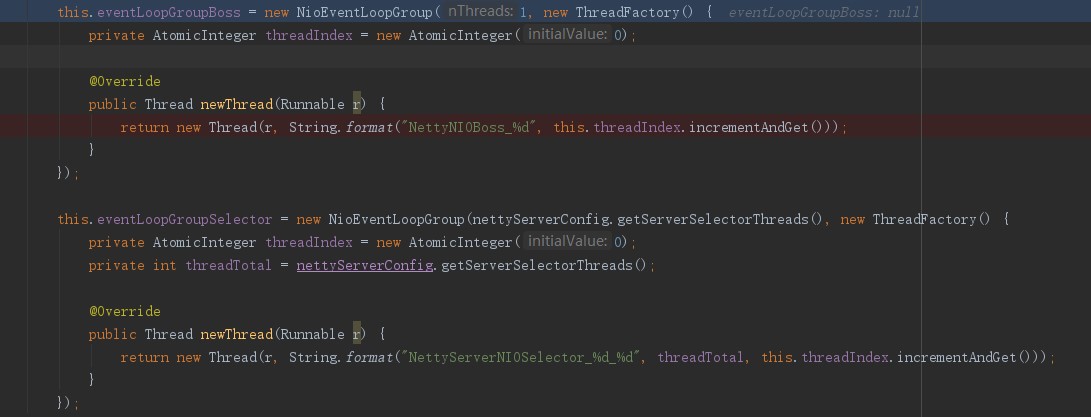
图3-2-1-1



### 3.2.2 构造方法

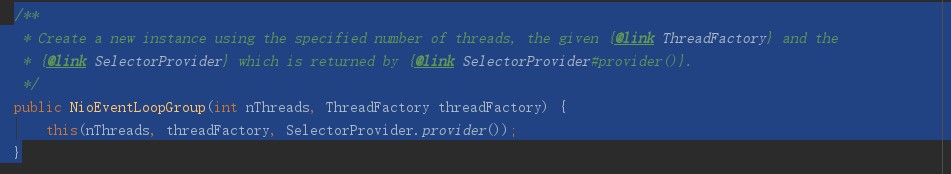
#### 3.2.2.1 构造方法1

图3-2-2-1



#### 3.2.2.2 构造方法2

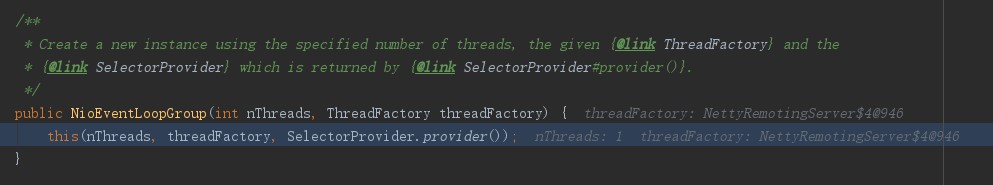
图3-2-2-2



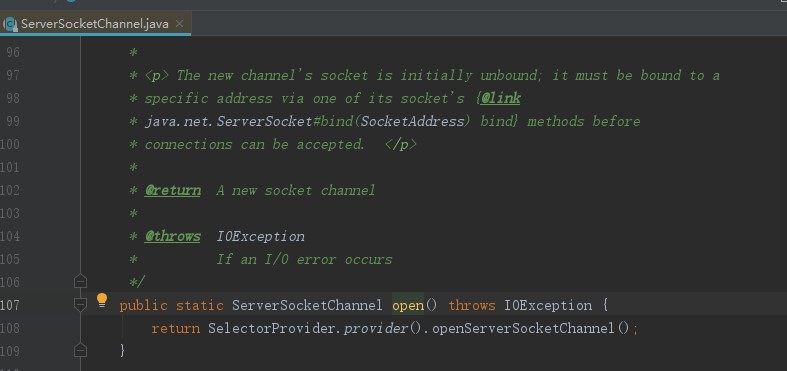
1. nThreads线程数
2. threadFactory用于创建线程的工厂实例（具体代码参考，图3-2-2-1）

#### 3.2.2.3 构造方法3

图3-2-2-3

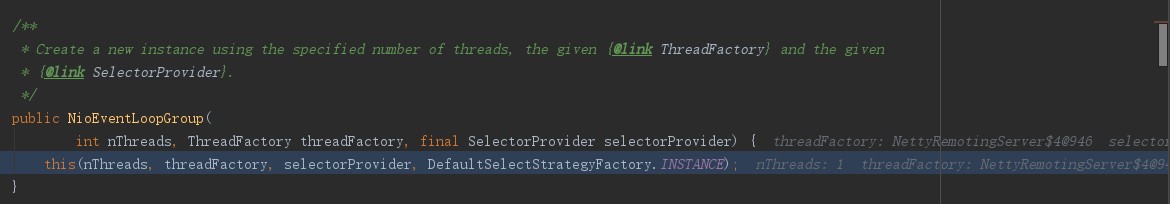


1. SelectorProvider.provider() 返回一个SelectorProvider 的实现类实例。Java.nio.channels.ServerSocketChannel.open()方法中就用到了该对象。如图3-2-2-4



#### 3.2.2.4 构造方法4

图3-2-2-5



1. DefaultSelectStrategyFactory.INSTANCE实例

图3-2-2-6

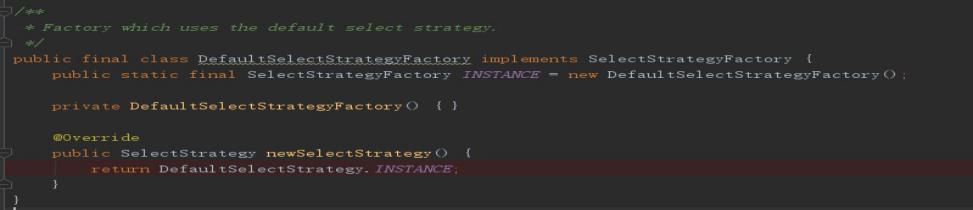
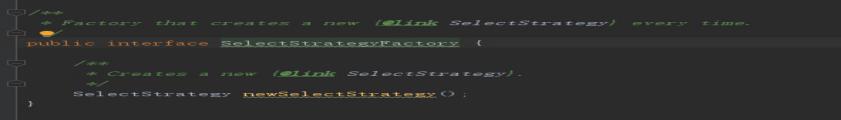


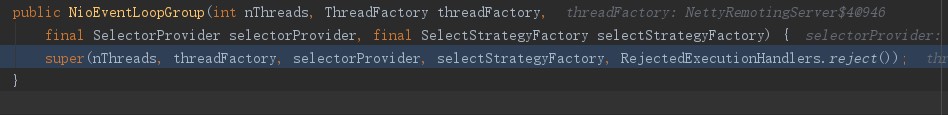
图3-2-2-7



感觉没有啥用

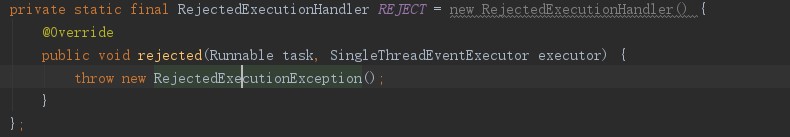
#### 3.2.2.5 构造方法5

图3-2-2-8



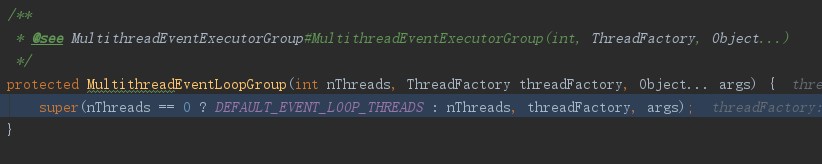
1. RejectedExecutionHandlers.reject()方法返回RejectedExecutionHandler实现类。

图3-2-2-9



#### 3.2.2.6 父构造方法6

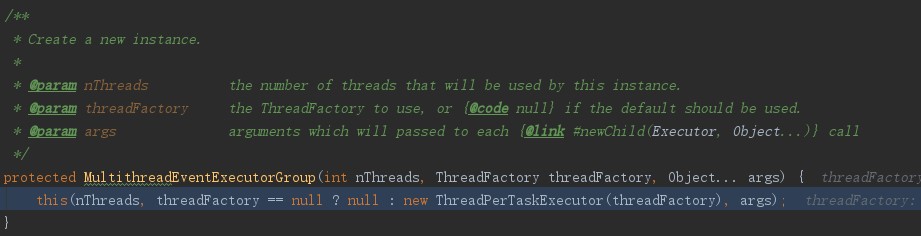
图3-2-2-10



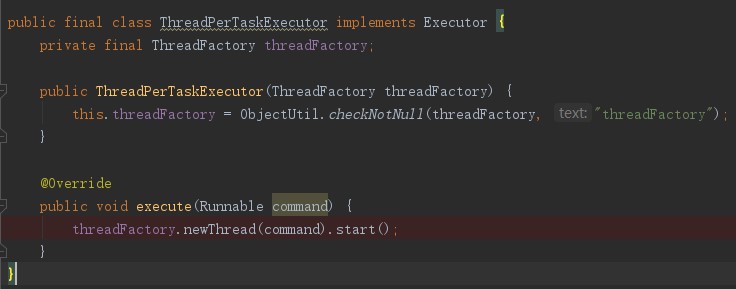
1. 如果nThreads为0，取默认值，默认值最小为1，最大可用核数的2倍。

#### 3.2.2.7 父构造方法7

图3-2-2-11

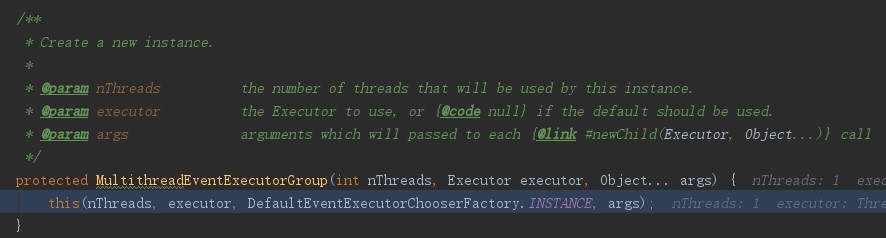


如果传的threadFactory对象为空，创建一个新对象ThreadPerTaskExecutor，入参为threadFactory对象。图3-2-2-11

所示，ThreadPerTaskExecutor继承了java.util.concurrent.Executor，其execute方法就是使用线程工厂类创建一个线程并执行start()方法，让线程运行起来。

#### 3.2.2.8 父构造方法8

图3-2-2-12

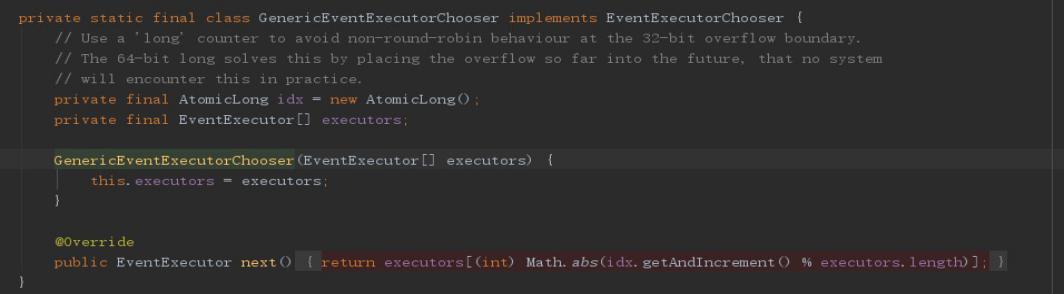


1. 、executor就是ThreadPerTaskExecutor实例，只是经过一层包装。
2. 、DefaultEventExecutorChooserFactory.INSTANCE对是DefaultEventExecutorChooserFactory实例。其中DefaultEventExecutorChooserFactory#newChooser()方法是返回一个EventExecutorChooser实现类实例。

3）EventExecutorChooser有PowerOfTwoEventExecutorChooser、GenericEventExecutorChooser两个实现类，图3-2-2-13



图3-2-2-14



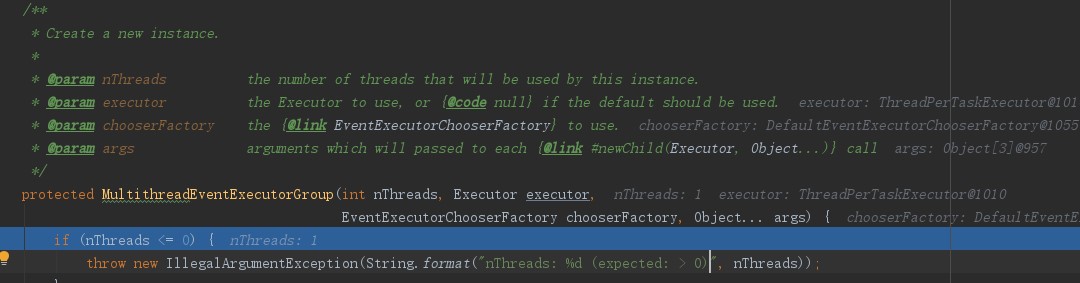
这两个实现类都实现next()方法，只是获取executors数组中的索引方式不一样。

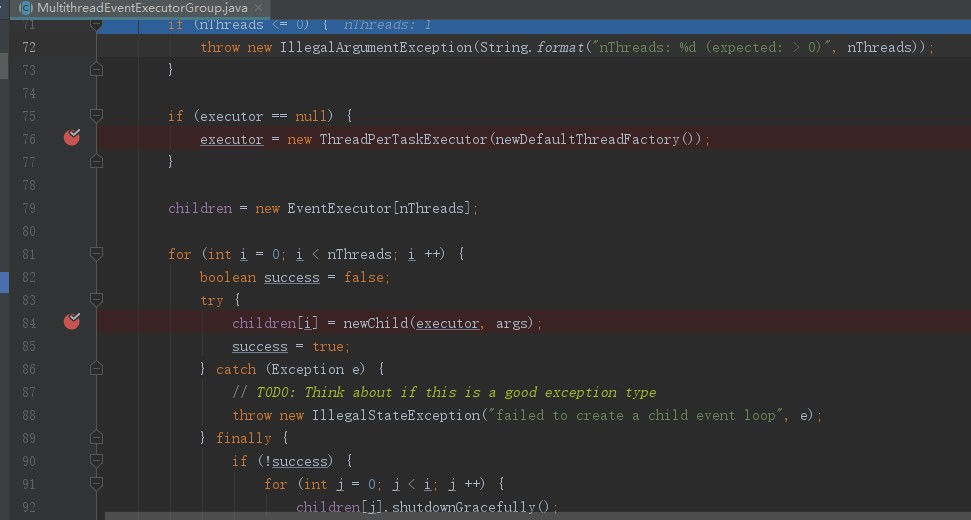
PowerOfTwoEventExecutorChooser#next()方法，计算数组索引方式：增量数字&数据长度-1。GenericEventExecutorChooser#next方法，计算数组索引方式：增量取余数据量长度，返回绝对值。

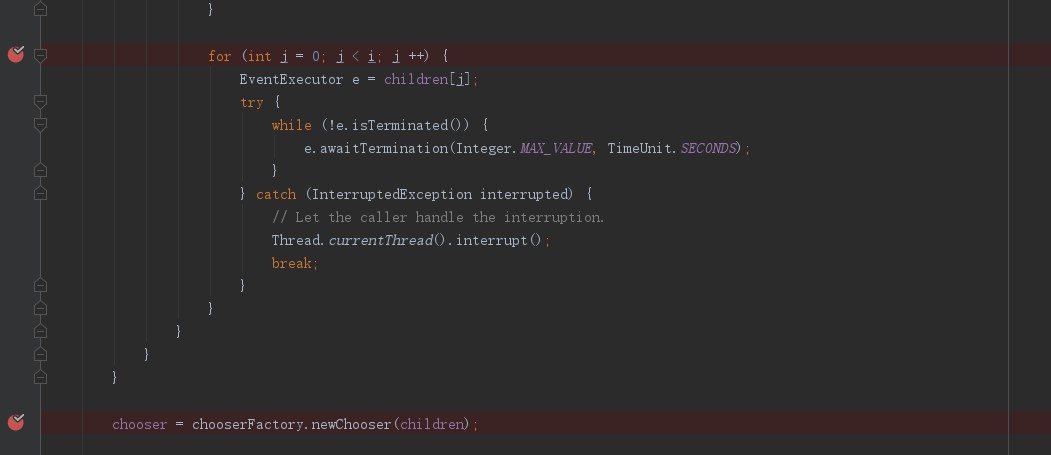
4）EventExecutor对象 其实就是NioEventLoopGroup父类MultithreadEventExecutorGroup中的children，就是NioEventLoop实例。

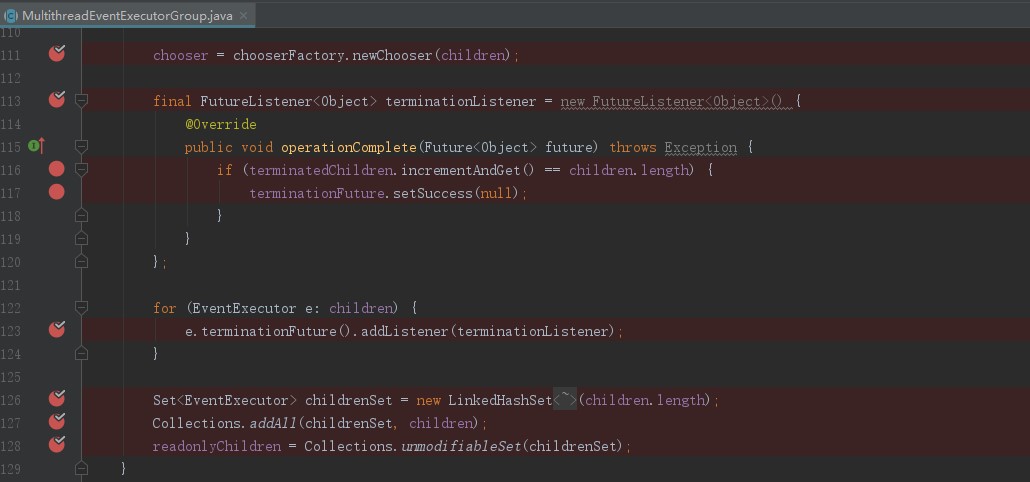
#### 3.2.2.9 父构造方法9

图3-2-2-15(多张整合成一个代表一个方法）









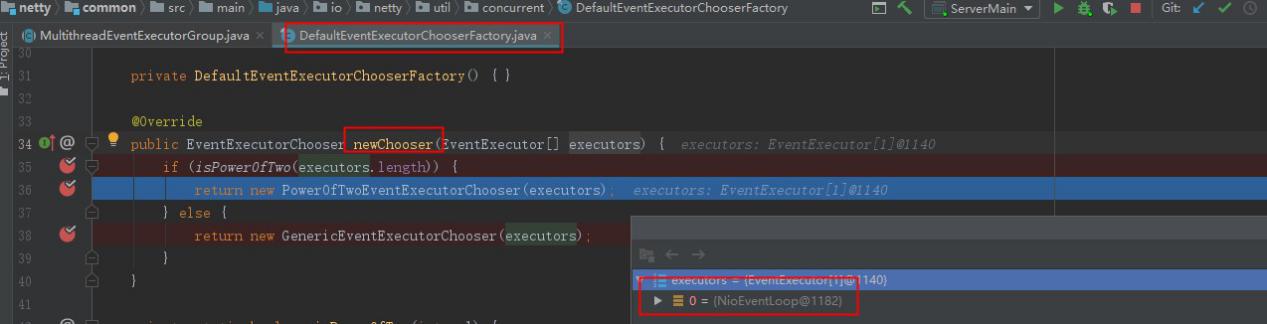
1）nThreads是初始化EventExecutor子类的个数。windows是指NioEventLoop

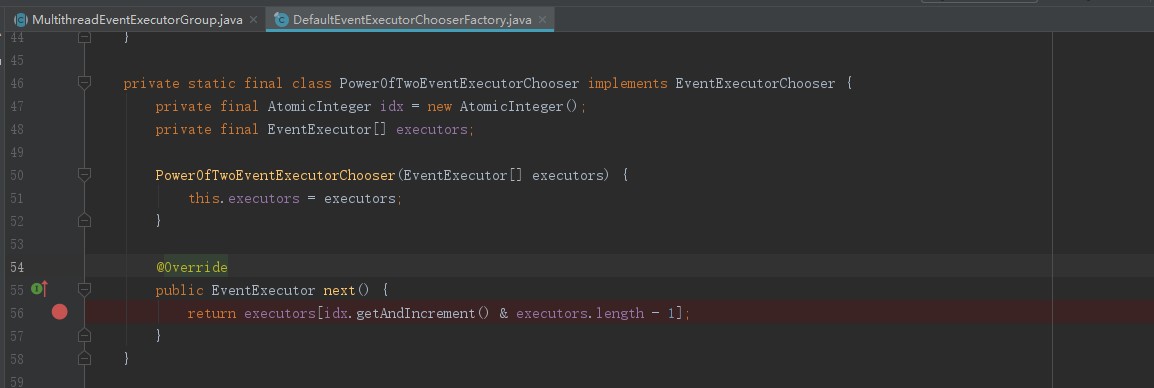
2）如果executor为空，重新创建一个ThreadPerTaskExecutor实例。

3）MultithreadEventExecutorGroup.children在for循环中，对过newChild方法初始化。

newChild方法是MultithreadEventExecutorGroup的子类中实现的，本例中在NioEventLoopGroup方法中实现（参考3.3）

1. 如果不成功，就把children对象中的资源释放掉
2. 初始化MultithreadEventExecutorGroup.chooser实例，chooser实现可能是PowerOfTwoEventExecutorChooser或GenericEventExecutorChooser实现类。



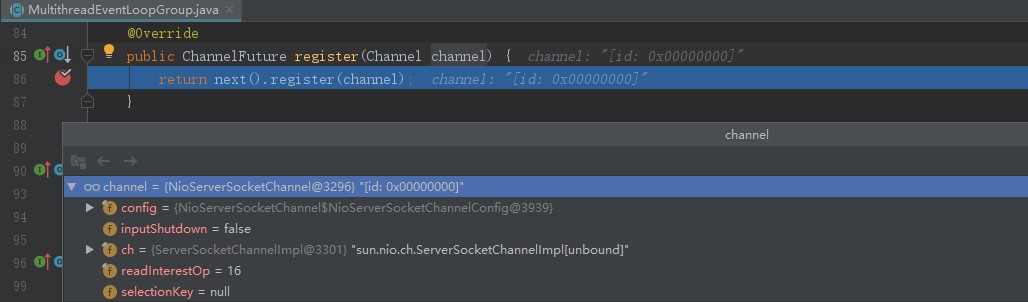


其实就是把NioEventLoop数组 赋值给PowerOfTwoEventExecutorChooser.executors，从而使用NioEventLoopGroup、NioEventLoop产生联系。

1. 实例化FutureListener terminationListener对象，并实现operationComplete方法。
2. 遍历MultithreadEventExecutorGroup.children（NioEventLoop数组）数组，每个NioEventLoop对象terminationFuture成员变量都加把6)中的terminationListener 变量加到terminationFuture中，从而使用NioEventLoop跟FutrueListener 产生关系。具体是什么关系？不知

### 3.2.3 register方法

图3-2-3-1



1. channel为NioServerSocketChannel 实例
2. 、next()如3-2-3-2、3-2-3-3、3-2-3-4所示，最终返回NioEventLoop实例，具体可以参考

NioEventLoopGroup的构造方法3.2.2讲解的设计。

3-2-3-2

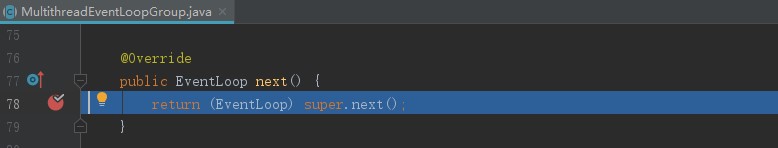


图3-2-3-3

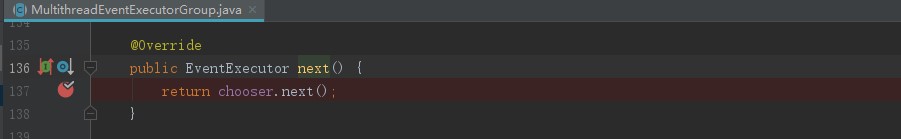
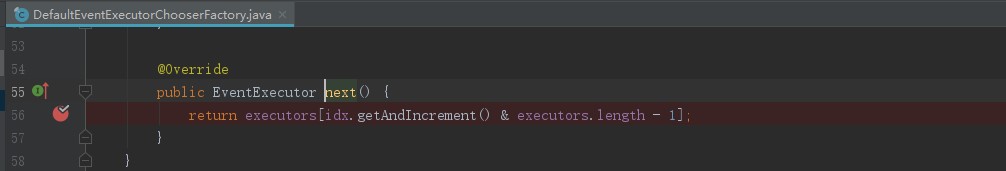
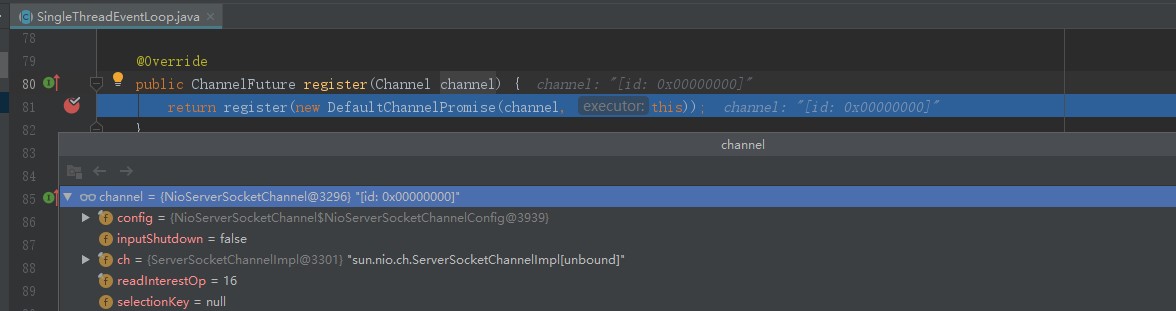


图3-2-3-4



1. 、register()方法最终调用的是NioEventLoop实例的。

图3-2-3-5



参考3.3.3

## 3.3 NioEventLoop

### 3.3.1继承关系

图3-3-1-1



### 3.3.2构造方法

#### 3.3.2.1 构造方法1

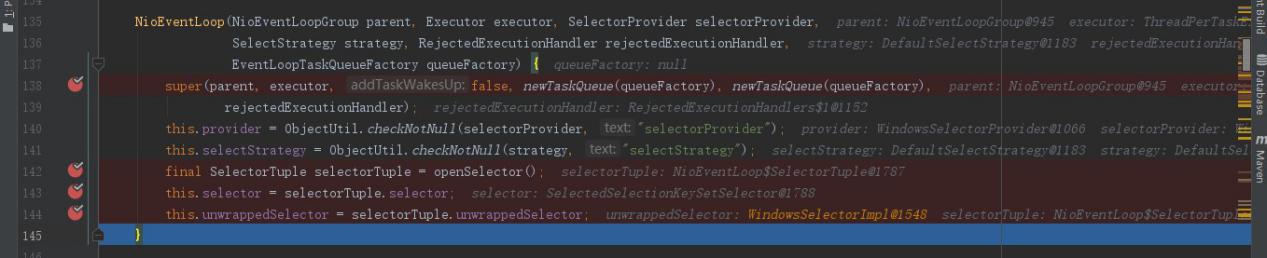
图3-3-2-1



1. this代表NioEventLoopGroup当前对象
2. executor 代表ThreadPerTaskExecutor实例
3. args[0 ] 代表SelectorProvider 子实现类实例
4. args[1]代表DefaultSelectStrategyFactory类实现，args[1].newSelectStrategy()方法返回DefaultSelectStrategy实例
5. queueFactory是null

#### 3.3.2.2 构造方法/初始化2

图3-3-2-2-1



1）执行父构造方法（参考3.3.2.3）

2）入参parent 是NioEventLoopGroup对象

3）入参executor是ThreadPerTaskExecutor对象

4）selectorProvider是SelectorProvider.provider()返回对象参考NioEventLoopGroup

5）strategy是DefaultSelectStrategy实例。其中有一个对外的calculateStrategy方法

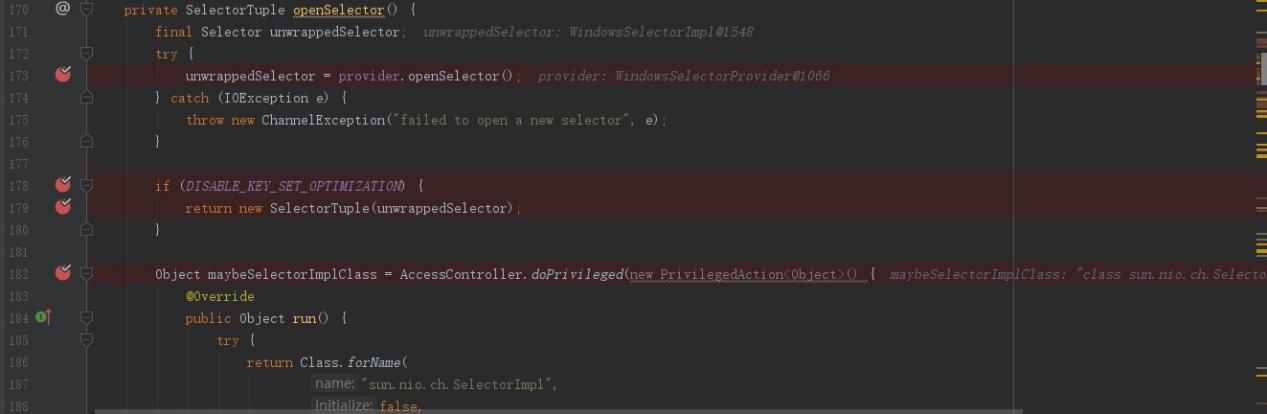
作用待定？？？？

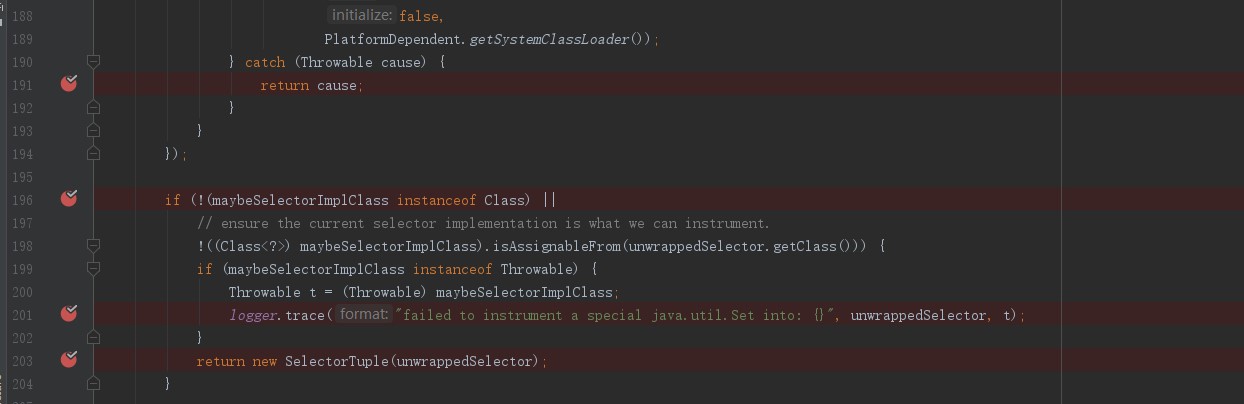
5）rejectedExecutionHandler是RejectedExecutionHandler实现类参考（3.2.2.5 构造方法5）

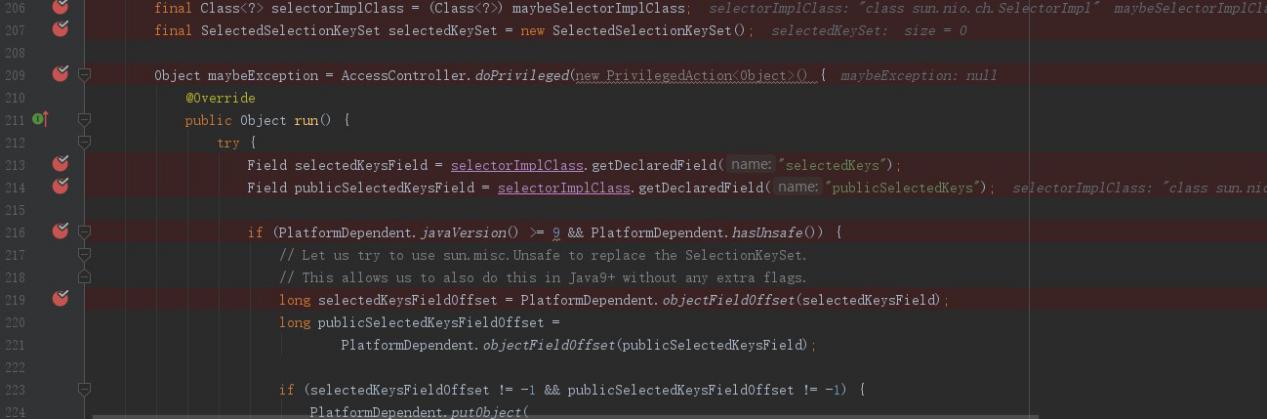
6）queueFactory对为null

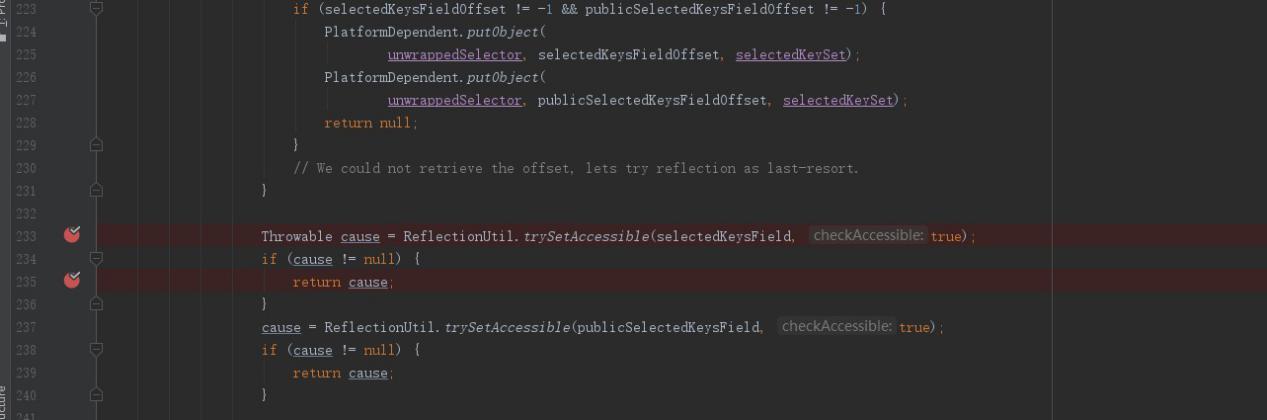
7）SelectorTuple 对象初始化，通过openSelector()方法

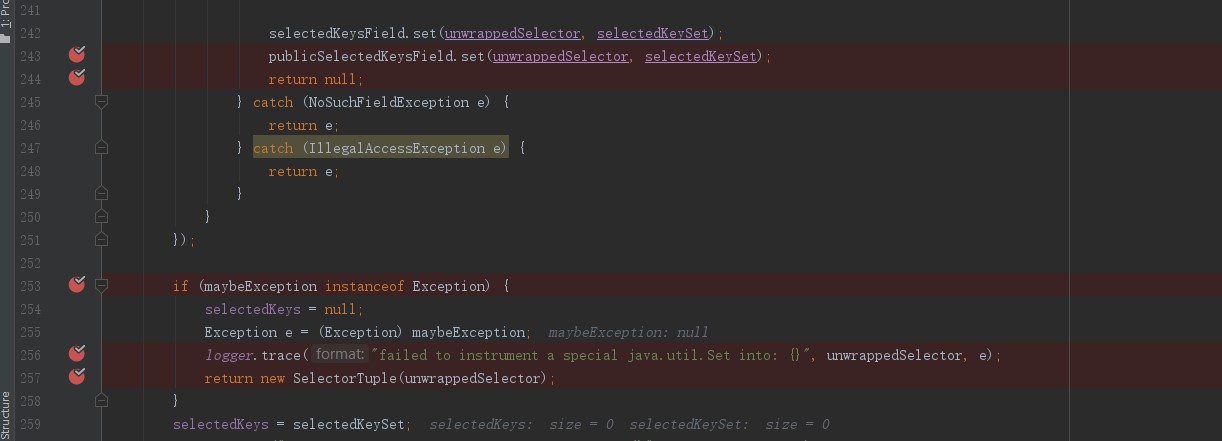
图3-3-2-2-2（下面6张是一个完整NioEventLoop#openSelector()方法）

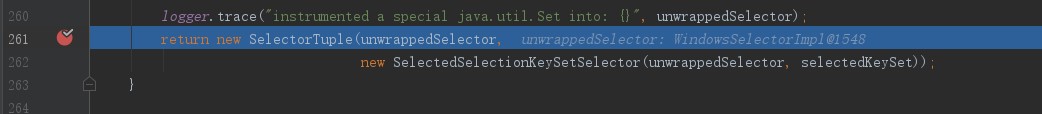












unwrappedSelector 是SelectorProvider.openSelector（）方法返回WindowsSelectorImpl

的实例。(因为是windows环环境）

mybeSelectorImplClass是根据sun.nio.ch.SelectorImpl 获取的Class 实例

图3-3-2-2-3



WindowsSelectorImpl、SelectorImpl继承关系如上。从而导致代码不行进入196行if代码块中，继续向下执行。

从sun.nio.ch.SelectorImpl 类获取selectedKeys、publicSelectedKeys 变量名称Field，

创建SelectedSelectionKeySet 实例，里面封装了java.nio.channels.SelectionKey。

最后通过反射，把对象unwrappedSelector(WindowsSelectorImpl实例) selectedKeys、publicSelectedKeys属性设置SelectedSelectionKeySet 实例。

重新封装SelectorTupe对象返回。SelectorTupe类中包括相关WindowsSelectorImpl、SelectedSelectionKeySet 相关对象。

1. 、this.selector就是 SelectedSelectionKeySetSelector实例
2. 、this.unwrappedSelector就是WindowsSelectorImpl实例

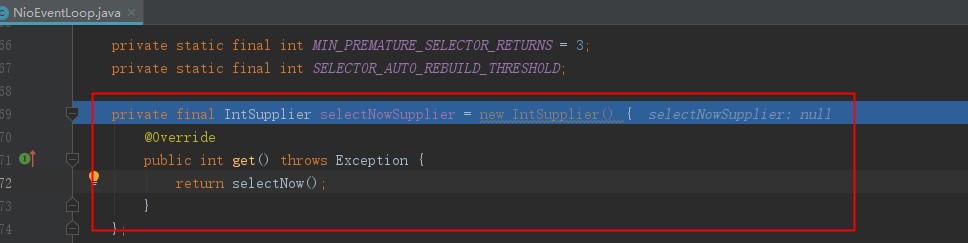
10)、看来NioEventLoop 在构造过程中，初始化了java.nio.channels.Selector，并给

sun.nio.ch.SelectorImpl 中的selectedKeys、publicSelectedKeys 设置新值。

11）、

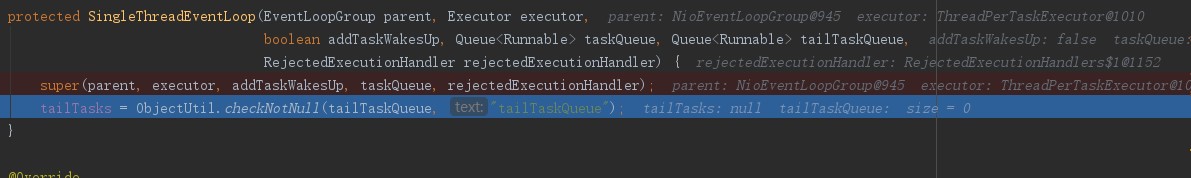
12）成员变量NioEventLoop.selectNowSupplier对象实例化

3-3-2-2-2



#### 3.3.2.3父类构造方法

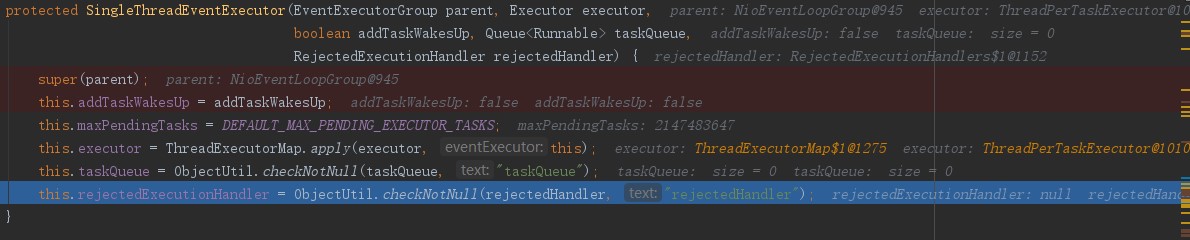
图3-3-2-3



1. parent 是NioEventLoopGroup实例
2. executor是ThreadPerTaskExecutor
3. addTaskWakesUp是 false
4. taskQueue是创建Queue实现类MpscUnboundedArrayQueue的实例
5. tailTaskQueue是创建Queue实现MpscUnboundedArrayQueue的实例

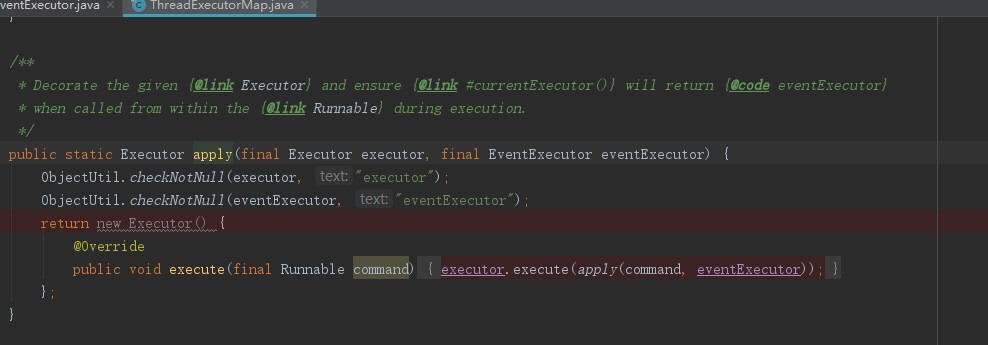
#### 3.3.2.4 父类构造方法

图3-3-2-4



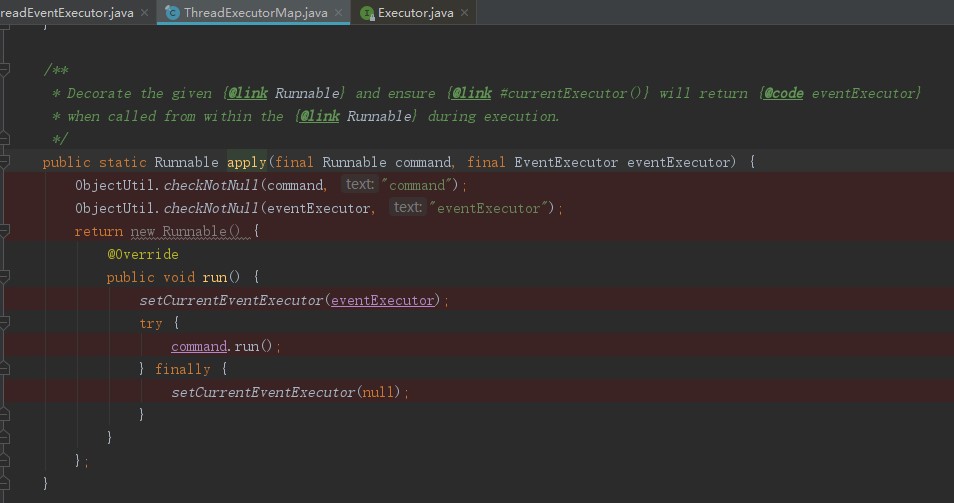
1. parent是NioEventLoopGroup实例
2. 入参executor是ThreadPerTaskExecutor实例
3. addTaskWakesUp false
4. maxPendingTasks 最大挂起任务数量

5）this.executor是图3-3-2-5所示



ThreadExecutorMap#apply方法返回的结果，其实就是java.util.concurrent.Executor的实现实例。Executor#execute实现方法内部，还是通过ThreadPerTaskExecutor#execute方法执行。

至于apply方法看图3-3-2-6所示

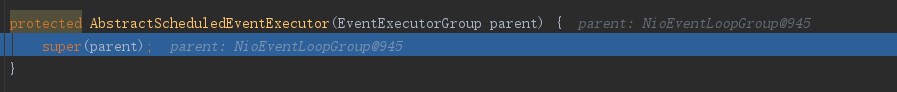


apply方法就是返回一个Runnable实现类实例，最终让ThreadPerTaskExecutor重启线

程执行。

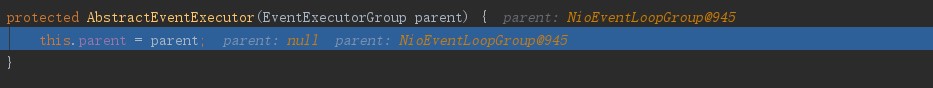
6）taskQueue子构造方法传递过来的

#### 3.3.2.5 父类构造方法



1. 略

#### 3.3.2.6 父类构造方法

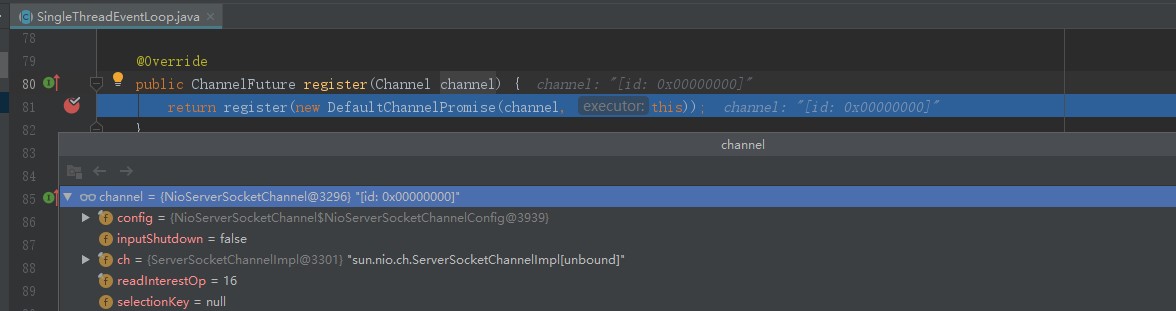


1. parent对象是NioEventLoopGroup对象实例

### 3.3.3 register方法

#### 3.3.3.1 register方法1

图3-3-3-1-1所示



1. 、NioServerSocketChannel实例。
2. 创建DefaultChannelPromise实例，如图3-3-3-1-1所示，channel为NioServerSocketChannel

对象，executor为NioEventLoop实例

把channel赋值给DefaultChannelPromise.channel实例

图3-3-3-1-2所示，把executor赋值给DefaultPromise.executor实例

图3-3-3-1-2

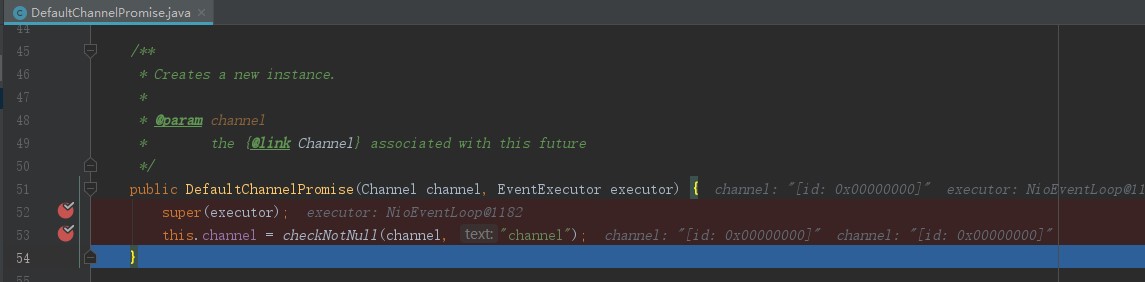
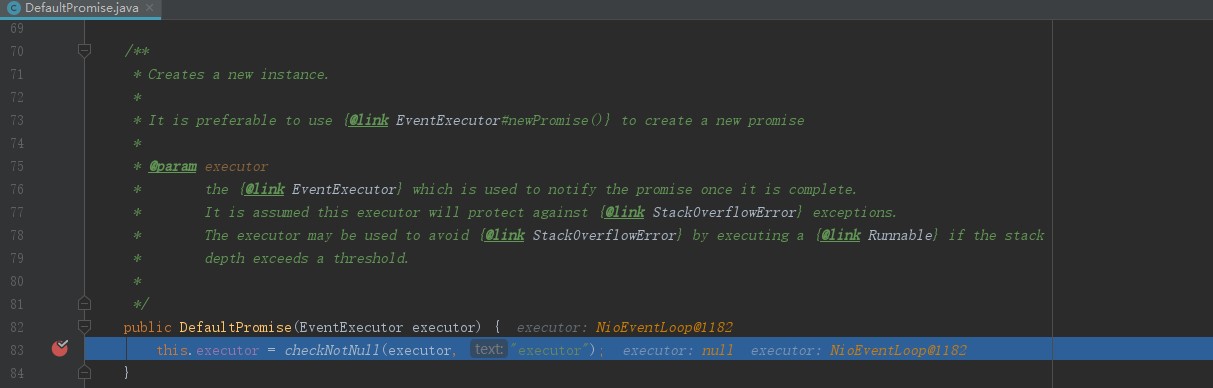
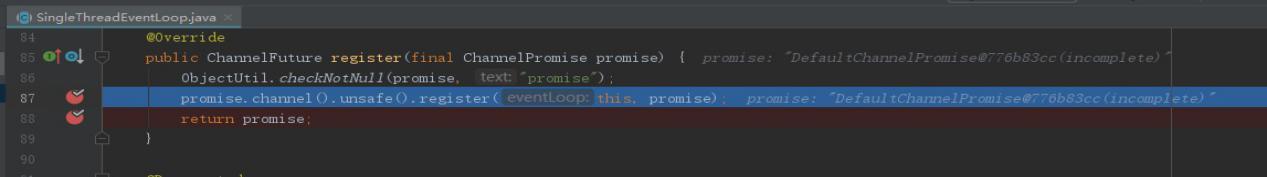


图3-3-3-1-3



#### 3.3.3.2 register 方法2

图3-3-3-2-1



1. 、promise为 DefaultChannelPromise实例 promise.channel()返回NioServerSocket实例

如图3-3-3-2-2所示，promise.channel().unsafe()进入图3-3-3-2-3、图3-3-3-2-4所示。

promise.channel.unsafe()进入AbstractUnsafe().register方法内，如下图图3-3-3-3-1

图3-3-3-2-2

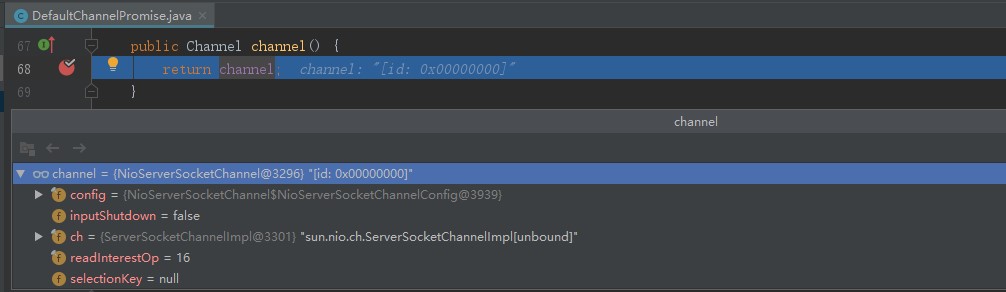


图3-3-3-2-3

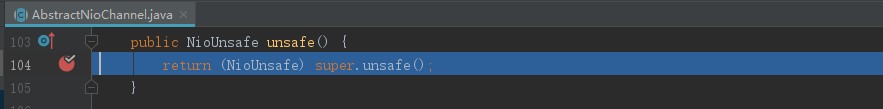
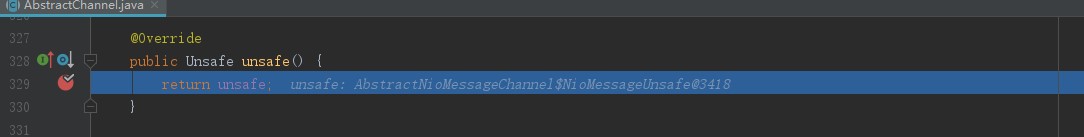
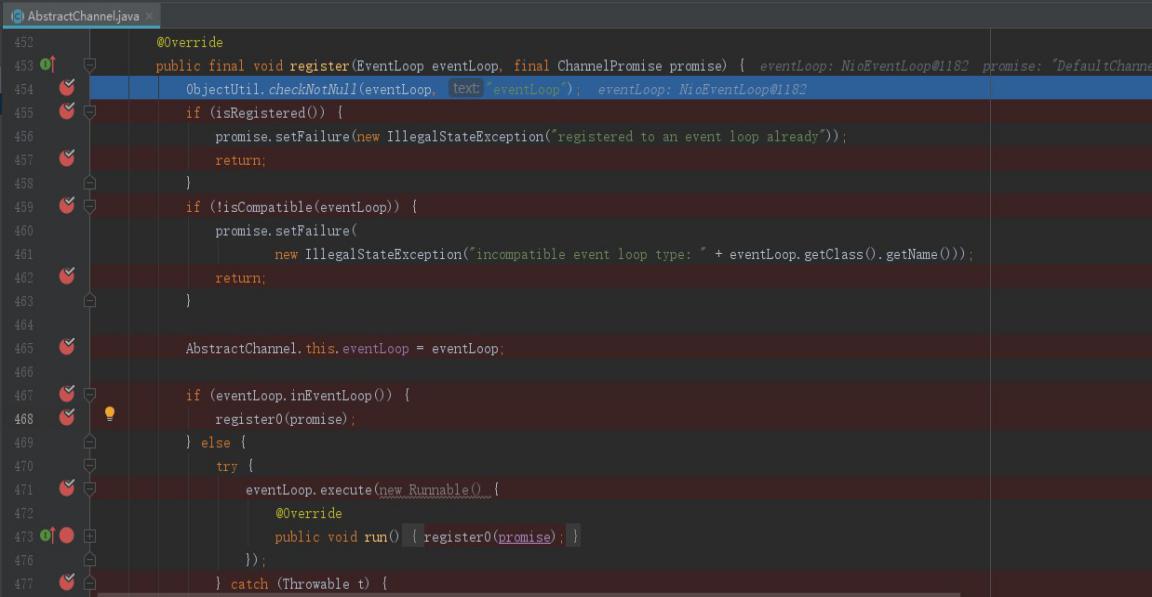


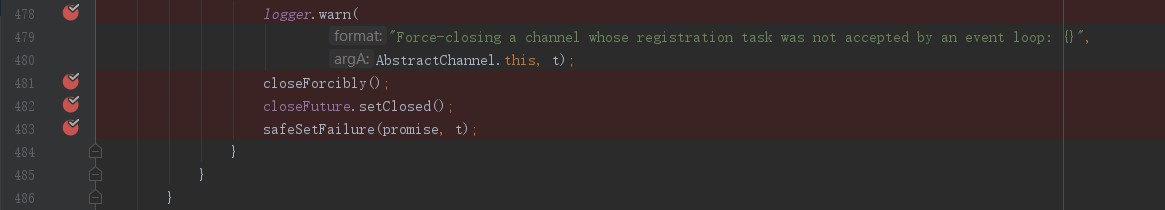
图3-3-3-2-4



#### 3.3.3.3 AbstractUnsafe.register方法

图3-3-3-3-1





1. 、eventloop 为NioEventLoop实例
2. 、promise为 DefaultChannelPromise实例
3. 、455行代表有没有注册过
4. 、459行如图eventLoop参数不是NioEventLoop或子类，就会中止。

## 3.4、DefaultEventExecutorGroup 构造方法

### 3.4.1继承关系

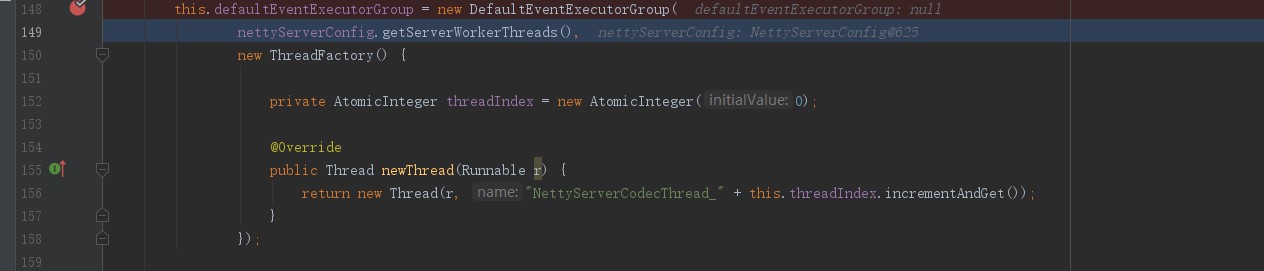
图3-4-1-1



### 3.4.2 构造方法

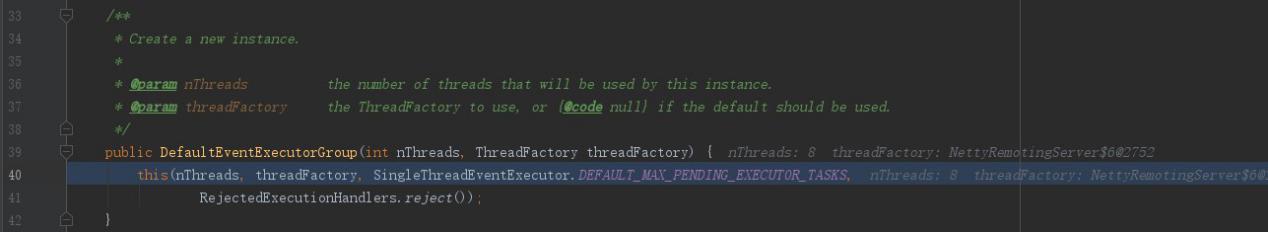
#### 3.4.2.1 构造方法1

图3-4-2-1-1



#### 3.4.2.2 构造方法2

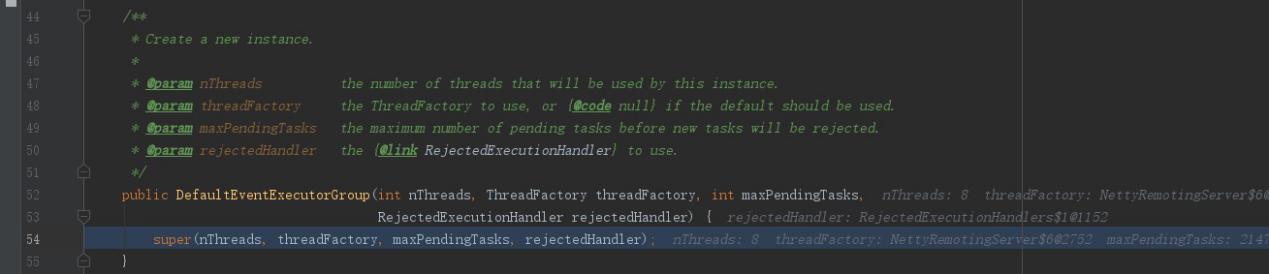
图3-4-2-2-1



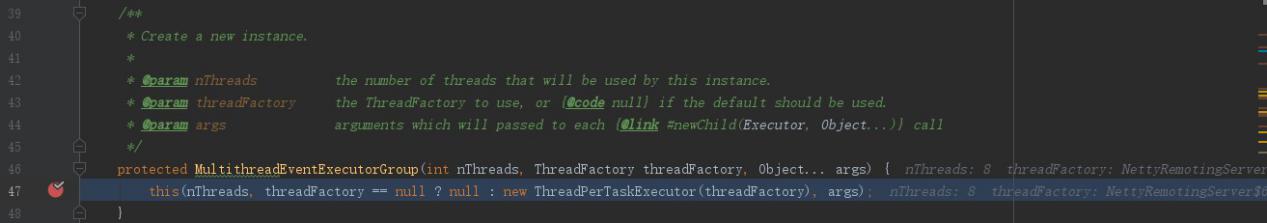
1. 、nThreads线程数
2. threadFactory线程工厂
3. SingleThreadEventExecutor.DEFAULT\_MAX\_PENDING\_EXECUTOR\_TASKS 默认最大挂起任务数量
4. 、参考3.2.2.5 构造方法5

#### 3.4.2.3 构造方法3

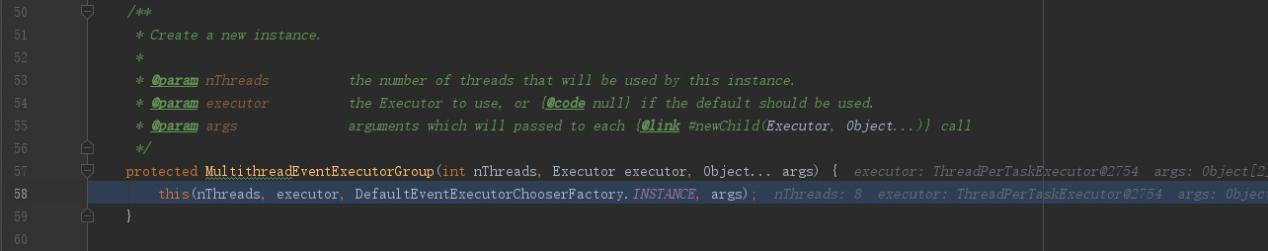
图3-4-2-3-1



#### 3.4.2.4 构造方法4



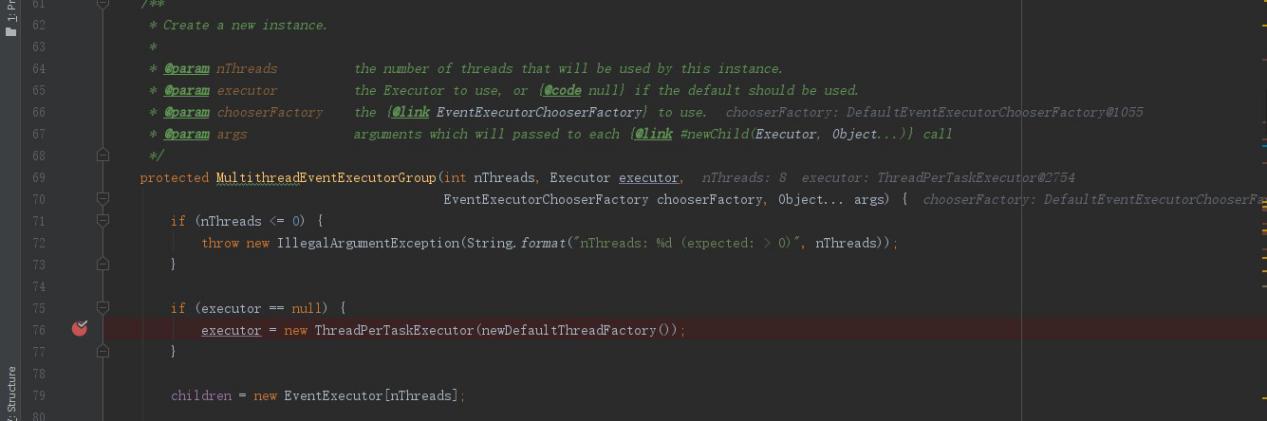
#### 3.4.2.5 构造方法5

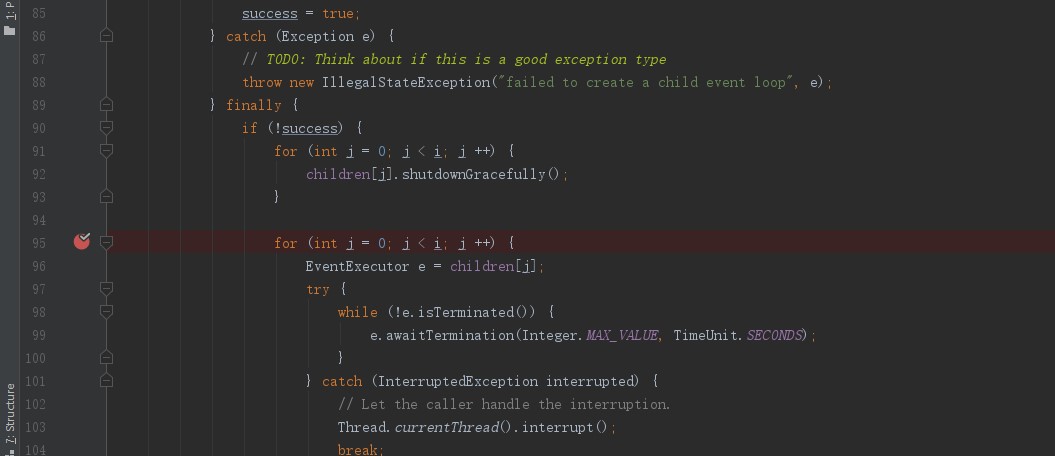


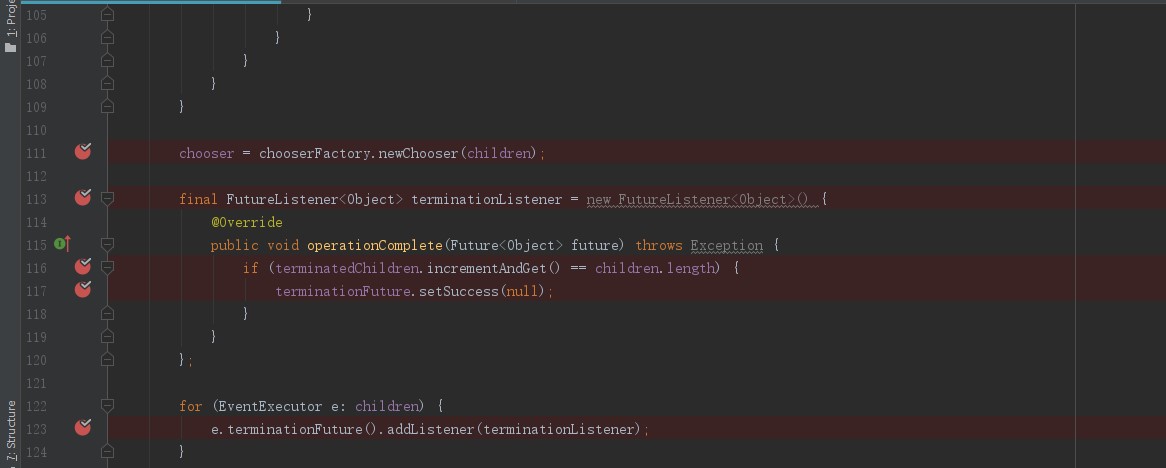
1）

1. executor为ThreadPerTaskExecutor实例，其实之前已经陈述过了

#### 3.4.2.6 构造方法6









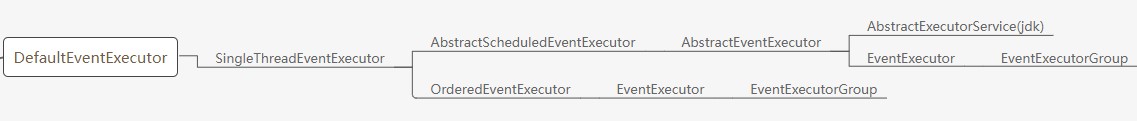
参考3.2

1. 重点看一下newChild()方法，参考3.5
2. 创建FutureListener terminationListener对象
3. 、遍历MultithreadEventExecutorGroup.children（NioEventLoop数组）数组，每个NioEventLoop对象terminationFuture成员变量都加把6)中的terminationListener 变量加到terminationFuture中，从而使用NioEventLoop跟FutrueListener 产生关系。具体是什么关系？不知

## 3.5 DefaultEventExecutor构造方法

#### 3.5.1 继承关系

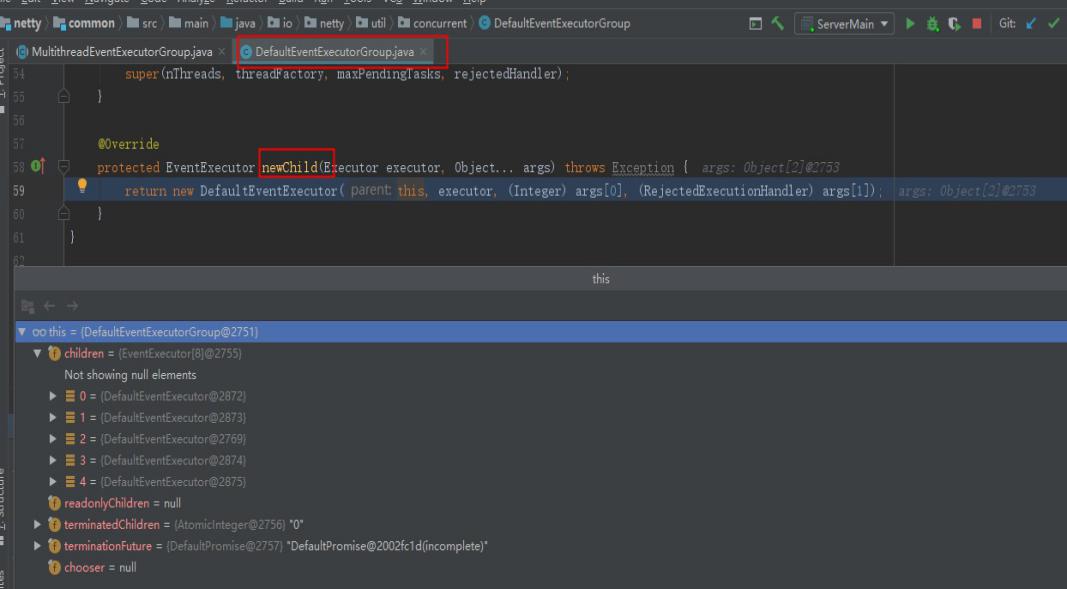
图3-5-1-1



### 3.5.2 构造方法

#### 3.5.2.1 构造方法1

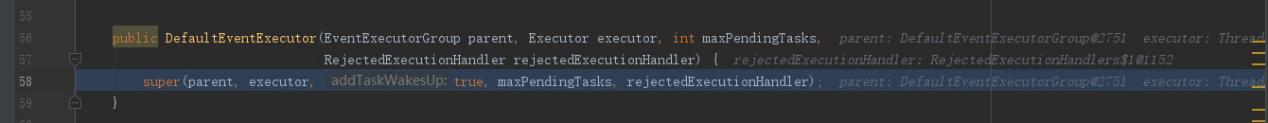
图3-5-2-1-1



1. executor就是ThreadPerTaskExecutor实例（参考3.3.2.1）

#### 3.5.2.2 构造方法2

图3-5-2-2-1



1. parent为DefaultEventExecutorGroup 实例
2. executor为ThreadPerTaskExecutor实例
3. maxPendingTasks 同之前
4. 其它略

#### 3.5.2.3父亲构造方法3

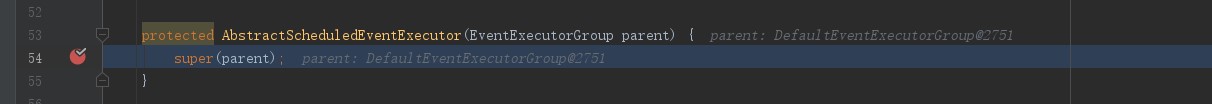
图3-5-2-3-1



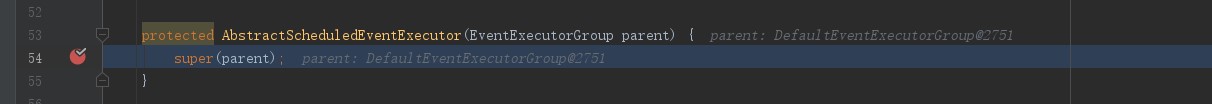
1. 执行父构造方法
2. this.addTaskWakesUp 赋值为true
3. 其它参考（3.3.2.4 父类构造方法）

#### 3.5.2.4 父构造方法4

图3-4-2-5-1



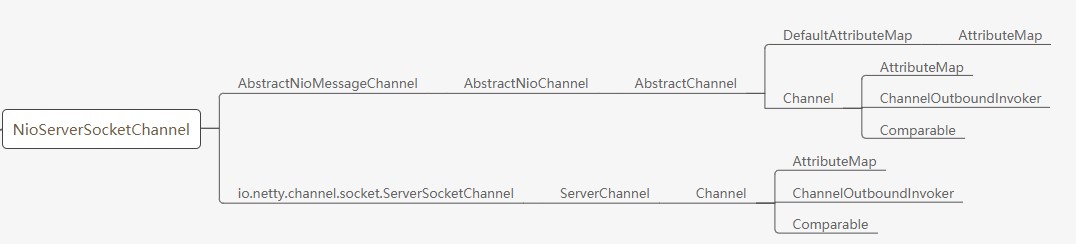
#### 3.5.2.5父构造方法5



## 3.6 NioServerSocketChannel

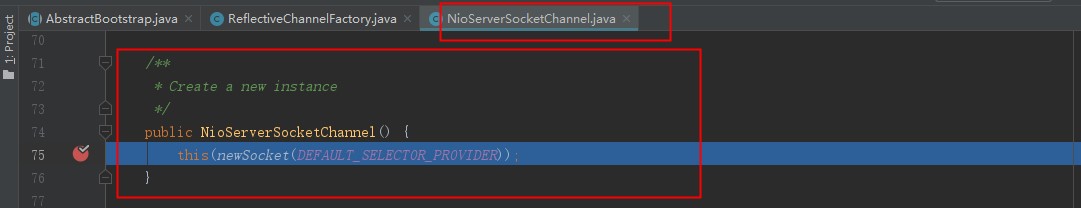
### 3.6.1继承关系

### 3.6.2 NioServerSocketChannel构造方法



#### 3.6.2.1 构造方法1

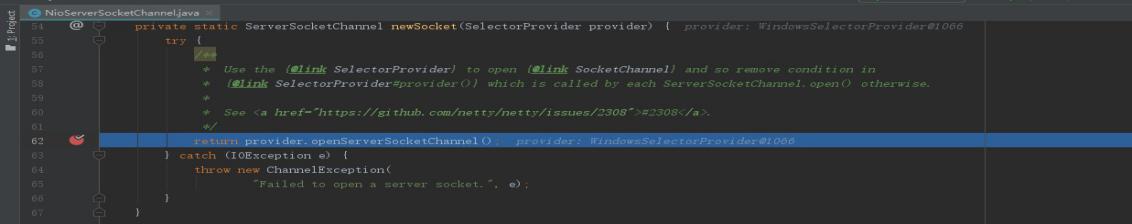
如图3-6-2-1-1



1. 通过newSocket方法创建java.nio.channels.ServerSocketChannel ，参考3-6-2-1-2图所示，

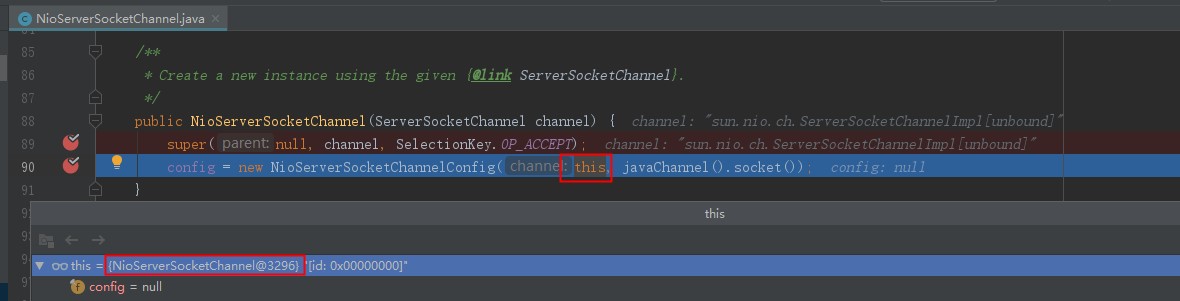
就是通过WindowsSelectorProvider.openServerSocketChannel()方法创建一个ServerSocketChannel对象

如图3-6-2-1-2



#### 3.6.2.2 构造方法2

如图3-6-2-2-1



1. 调用父构造方法（参考3.6.2.3）
2. 90行this即是NioServerSocketChannel

javaChannel()方法参考图3-6-2-2-2、图3-6-2-2-3、图3-6-2-2-4，最终返回 ServerSocketChannel实例，javaChannel().socket()如图3-6-2-2-5 返回ServerSocket实例。

图3-6-2-2-2



图3-6-2-2-3

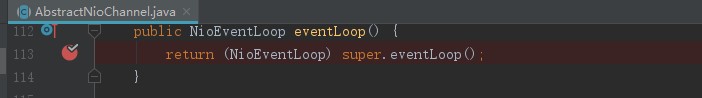
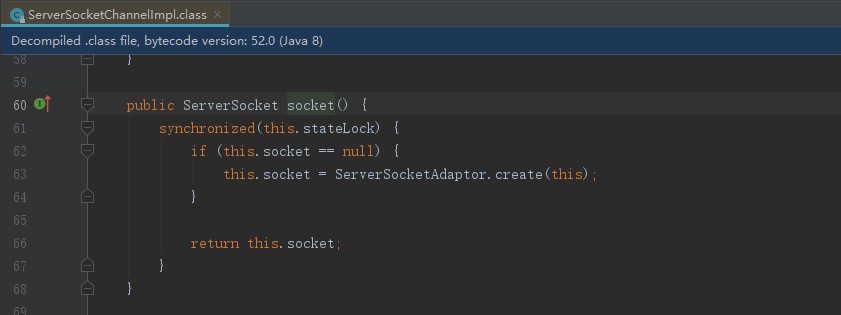


图3-6-2-2-4



图3-6-2-2-5



1. 、NioServerSocketChannel.config成员变量初始化，图3-6-2-2-6、3-6-2-2-7、3-6-2-2-8

channel为NioServerSocktChannel实例。

图3-6-2-2-8 创建AdaptiveRecvByteBufAllocator 实例初始化、校验。

图3-6-2-2-9所示，channel.metadata()方法返回

图3-6-2-2-10所示，

图3-6-2-2-11 所示，this.rcvBufAllocator 赋值但是不知道有什么用？

图3-6-2-2-6

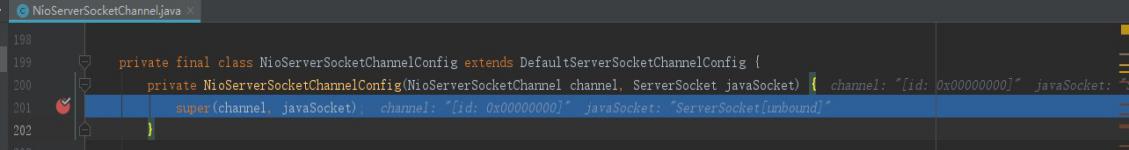


图3-6-2-2-7



图3-6-2-2-8

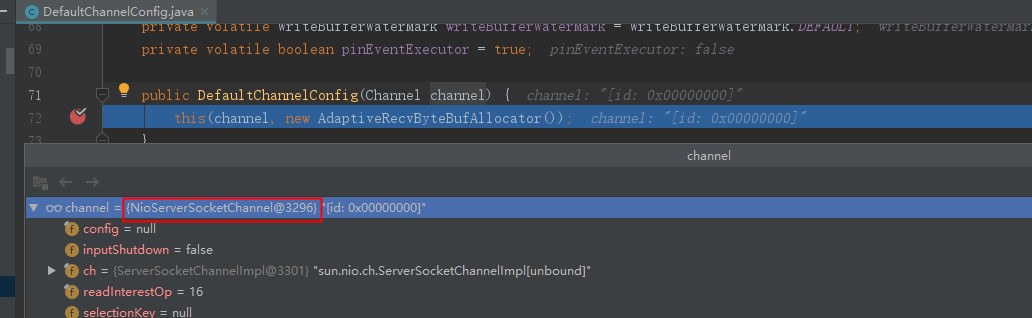


图3-6-2-2-9

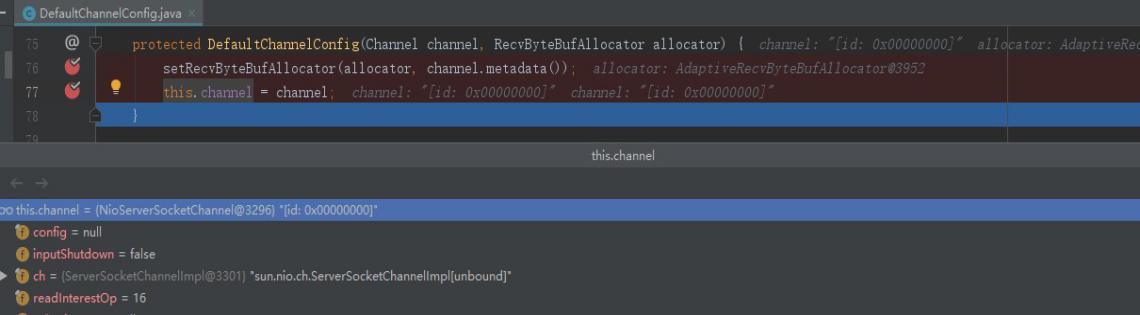


图3-6-2-2-10

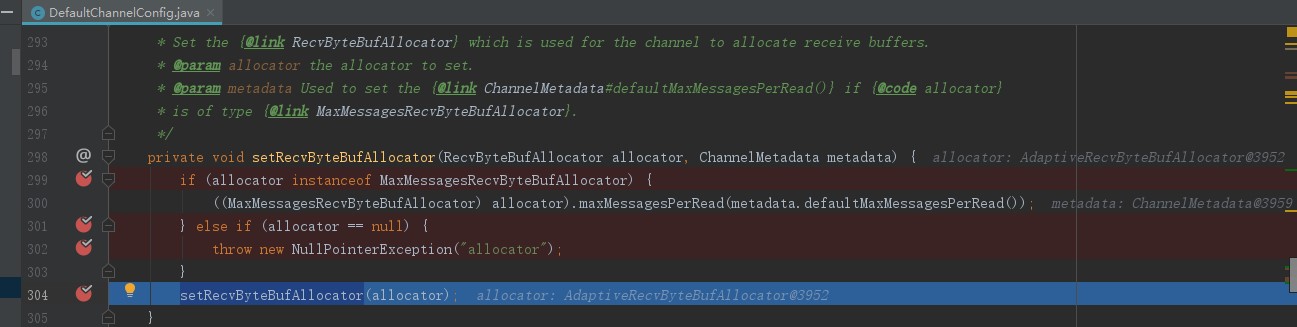
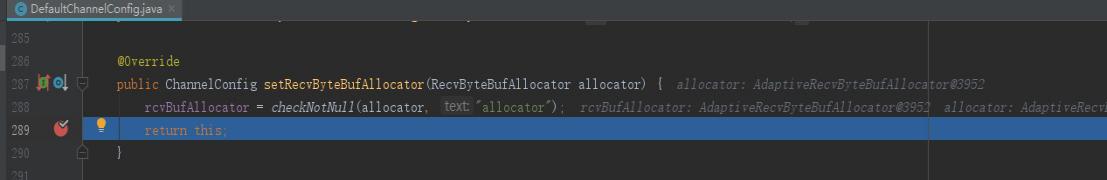
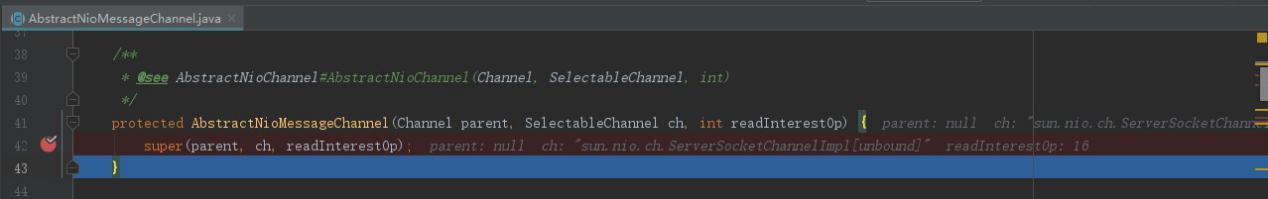


图3-6-2-2-11



#### 3.6.2.3父构造方法

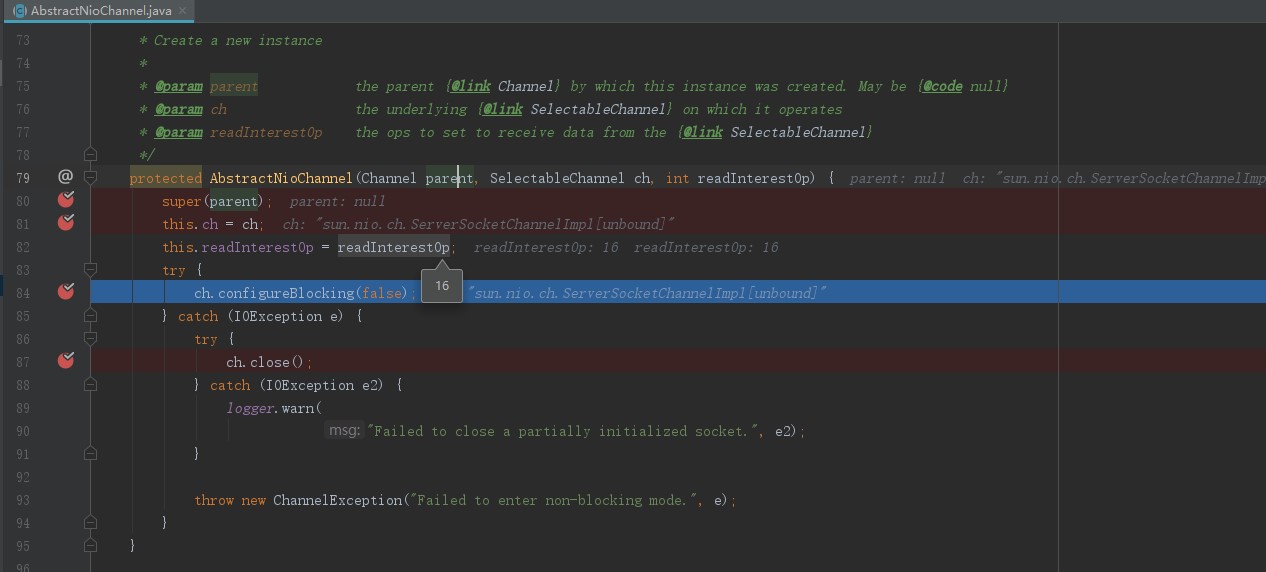
图3-6-2-3-1



1. parent为空
2. ch为sun.nio.ch.ServerSocketChannelImpl对象
3. readInterestOp为16（SelectionKey.OP\_ACCEPT）。

#### 3.6.2.4父构造方法

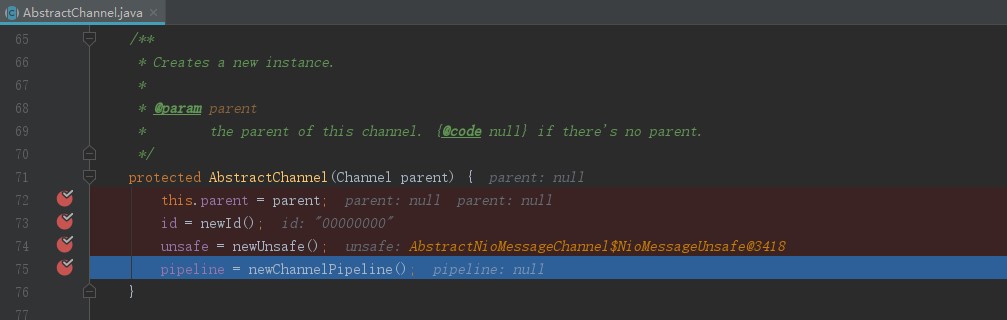
图3-6-2-4-1



1. parent为null
2. ch为ServerSocketChannelImpl 实例（就是java.nio.channels.ServerSocketChannel实现类)
3. 84行设置为非阻塞

#### 3.6.2.5父构造方法

图3-6-2-5-1



1. parent为 null
2. id通过newId()方法初始化

图3-6-2-5-1

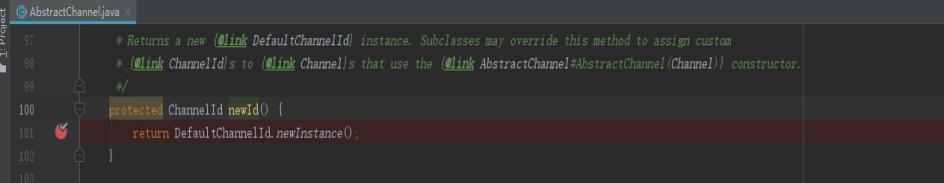


图3-6-2-5-2

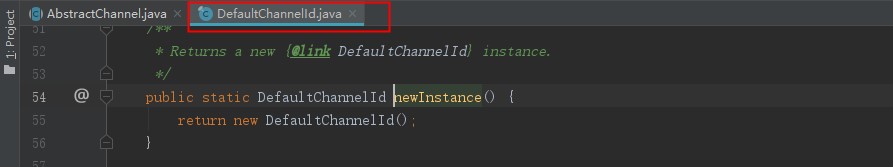
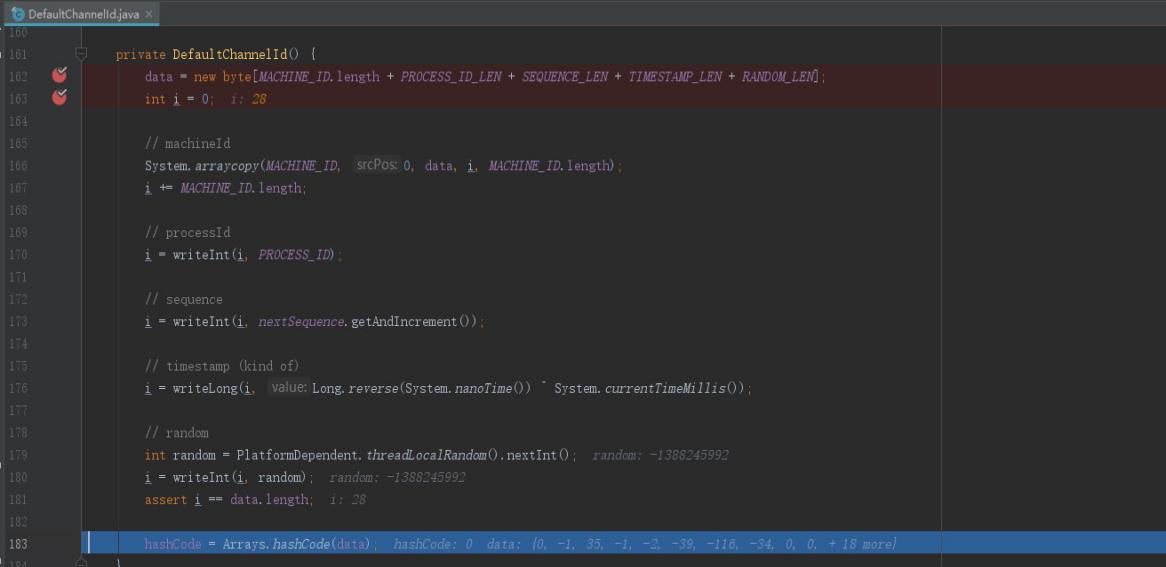


图3-6-2-5-3



DefaultChannelId()构造方法用的到writeInt、writeLong方法如图3-6-2-5-4所土示。

DefaultChannelId构造方法主要是初始化一个byte数组data，data数组长度28位，从数组下标0开始，依以是

8个字节：机器id

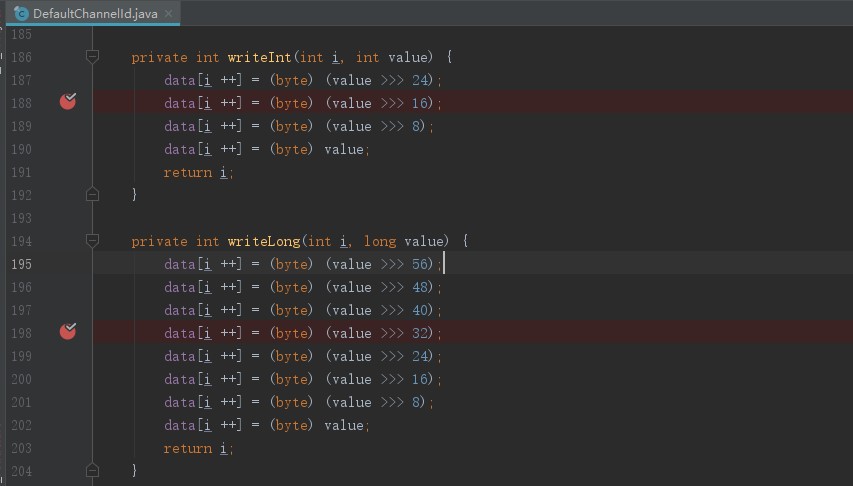
4个字节：进程号

4个字节：序列号（递增）

8个字节：时间戳

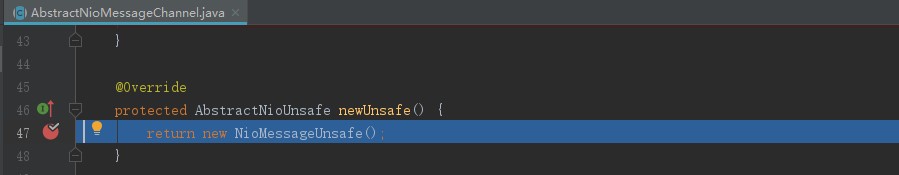
4个字节：随机码

图 3-6-2-5-4



3）、unsafe 是通过newUnsafe()方法初始化，

图3-6-2-5-5



1. 、通过newChannelPipeline()方法初始化，参考3.7

## 3.7 DefaultChannelPipeline

### 3.7.1 继承关系

### 3.7.2 构造方法

#### 3.7.2.1构造方法1

图3-7-2-1-1

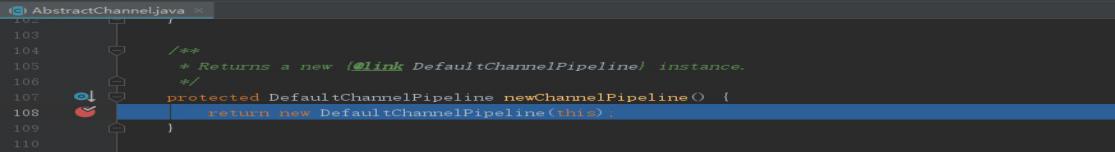
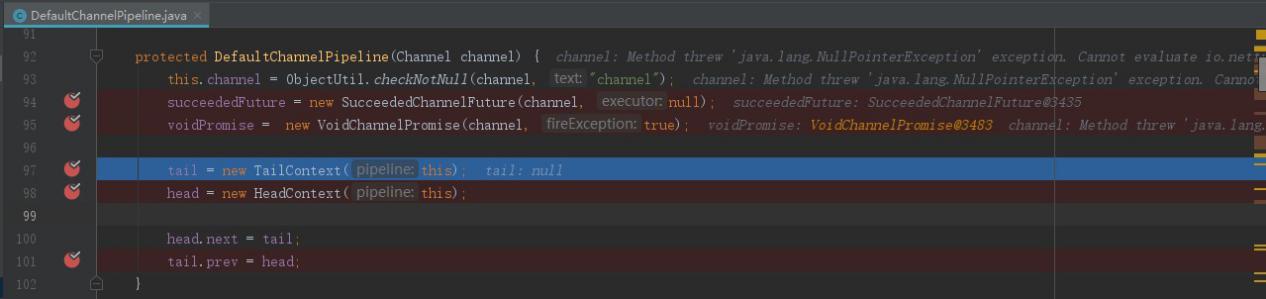


图3-7-2-1-2



1. 、channel为NioServerSocketChannel实例。
2. 创建SucceededChannelFutrue实例

图3-7-2-1-3

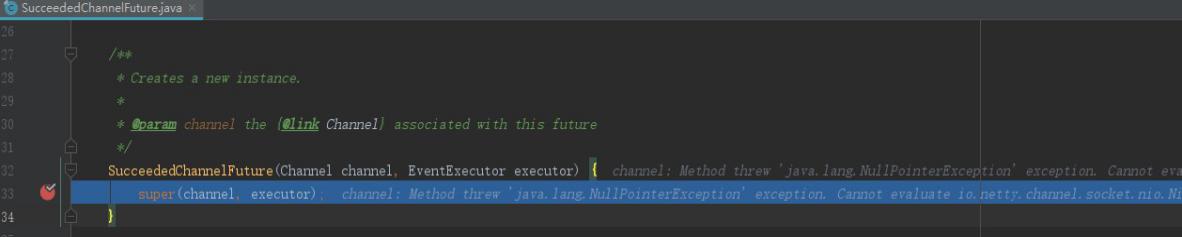


图3-7-2-1-3 中channel为NioServerSocketChannel 实例；executor为null。

图3-7-2-1-4

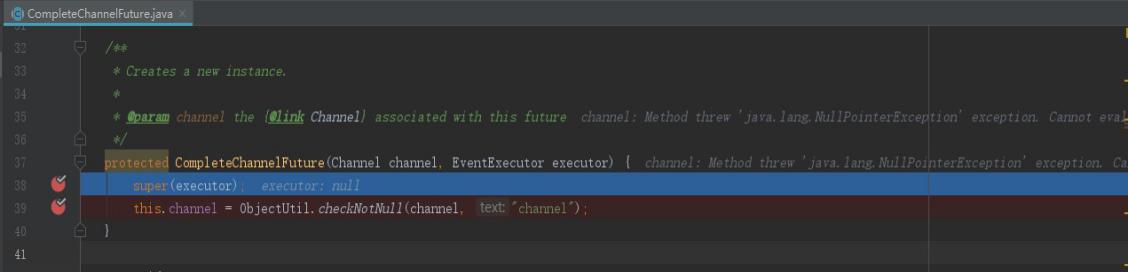
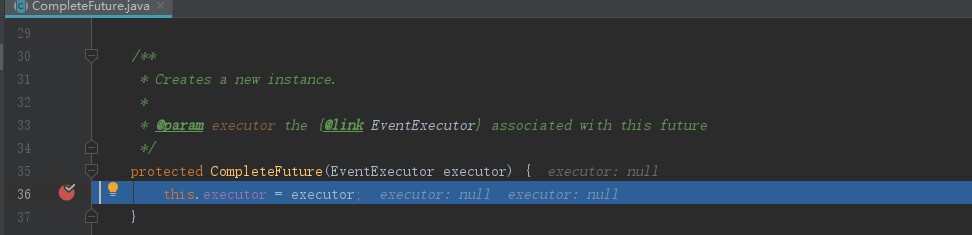


图3-7-2-1-5

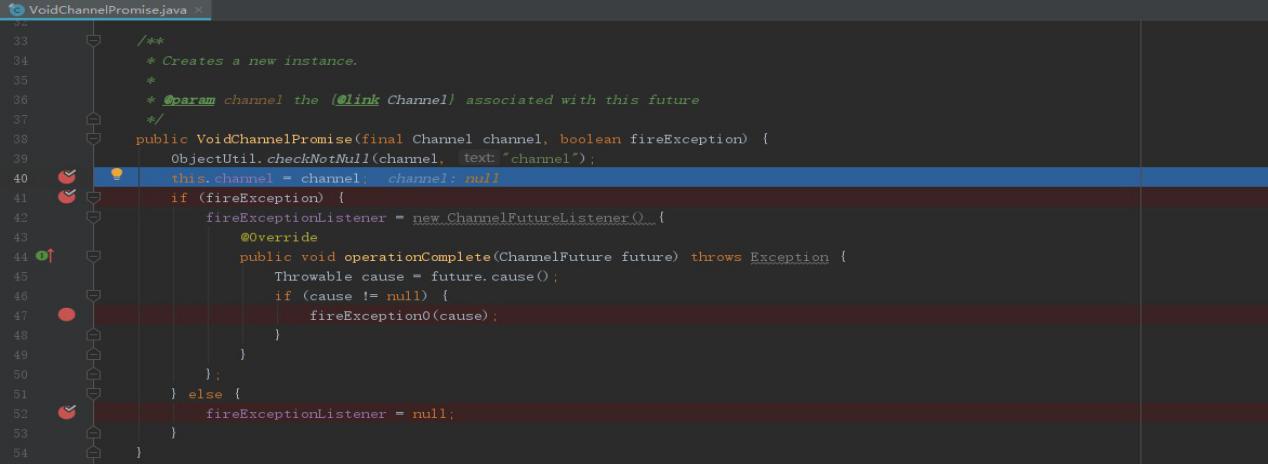


图图3-7-2-1-3、图3-7-2-1-4、图3-7-2-1-5 所示相关继承关系如下：

SucceededChannelFutrue->CompleteChannelFutrue->CompleteFutrue。

3）、创建VoidChannelPromise实例，如图3-7-2-1-6所示

图3-7-2-1-6



创建VoidChannelPromise实例进入构造方法如图3-7-2-1-6所示，fireException为true，

1. 、创建TailContext实例，如图3-7-2-1-7所示，TailContext就是

AbstractChannelHandlerContext的子类。

执行setAddComplete()方法是为加一个标记，标记该AbstractChannelChandlerContext.handlerState属性已经标记为2 添加状态。

图3-7-2-1-7

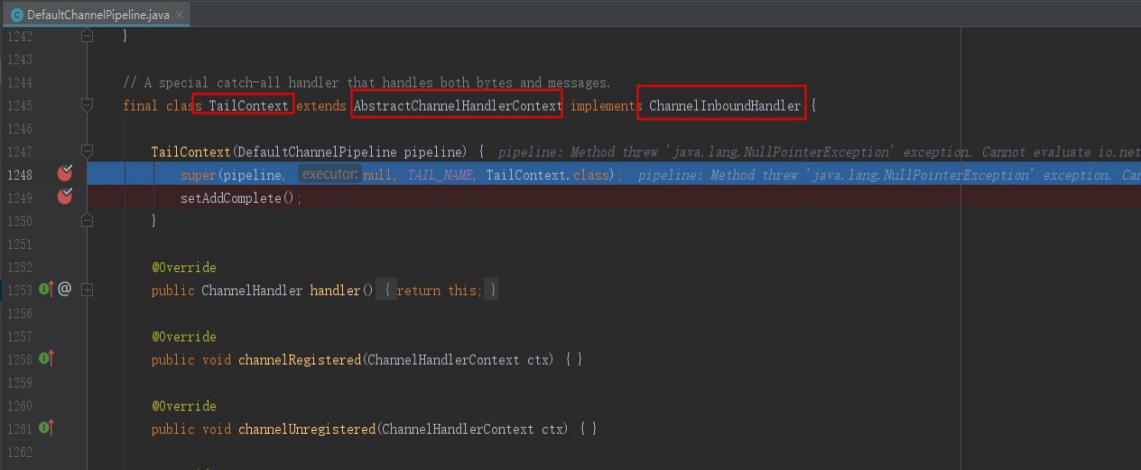
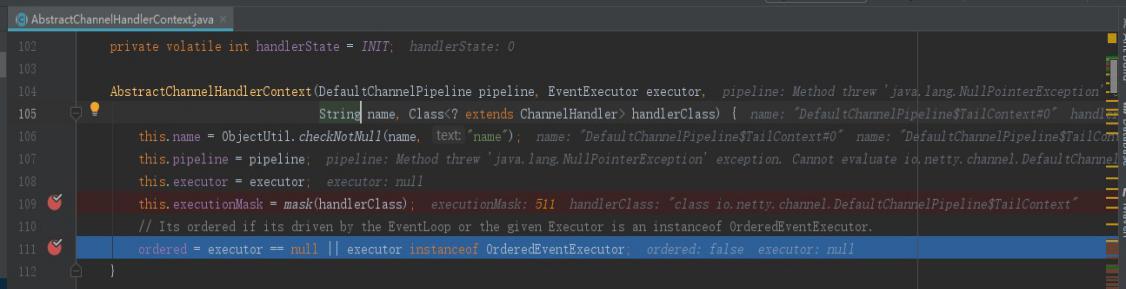


图3-7-2-1-8



AbstractChannelHandlerContext子类构造方法如图3-7-2-1-8所示，

name为”DefaultChannelPipeline$TailContext#0”；

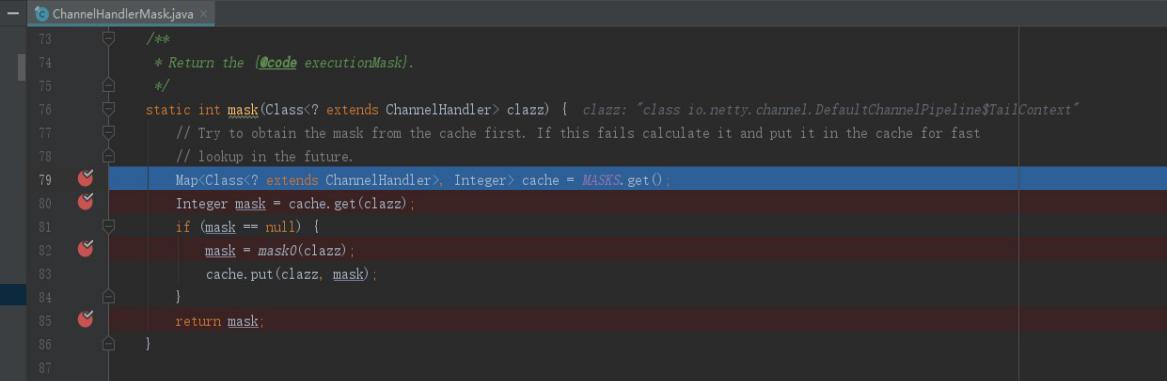
pipleline 为DefaultChannelPipeline实例；

executor为null；

入参handlerClass为TailHandler.class ；

this.executionMask 通过mask方法获取如图3-7-2-1-9所示。

图3-7-2-1-9



1. 、创建HeadContext实例，

图3-7-2-1-10

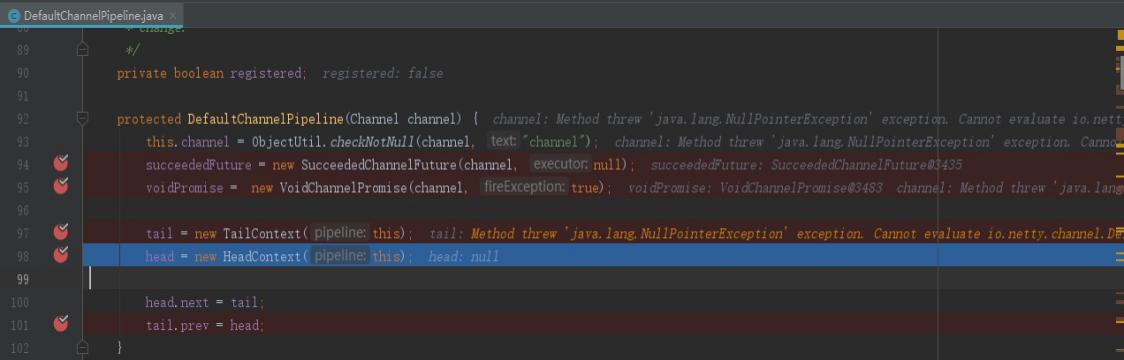
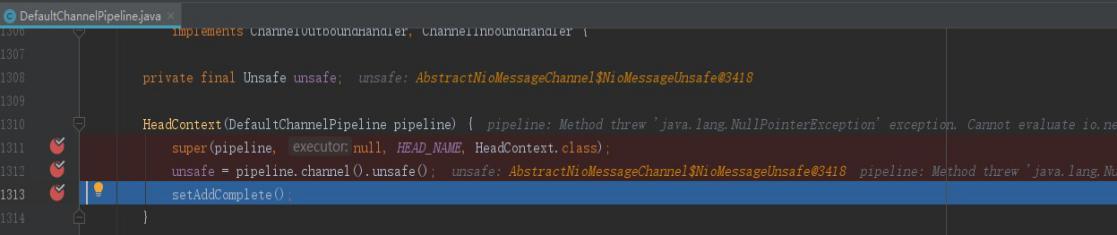


图3-7-2-1-11



如上图3-7-2-1-11与3-7-2-1-7所示HeadContext、TailContext是继承同一个父类

AbstractChannelHandlerContext，所以执行父类接口不再描述，参考3.7.2.1 3）

图3-7-2-1-11 中的pipeline即是图3-7-2-1-12所示，所以pipeline.channel()即图3-7-2-1-13所示返回对象为NioServerSocketChannel实例。Pipeline.channel().unsafe()如图3-7-2-1-14、图3-7-2-1-15所示，最终NioMessageUnsafe实例，这个实例是在初始化NioServerSocketChannel的时候创建的。

图3-7-2-1-12

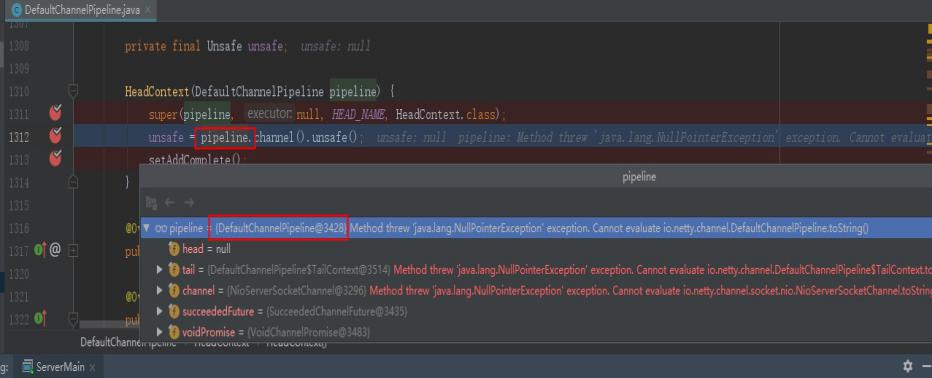


图3-7-2-1-13

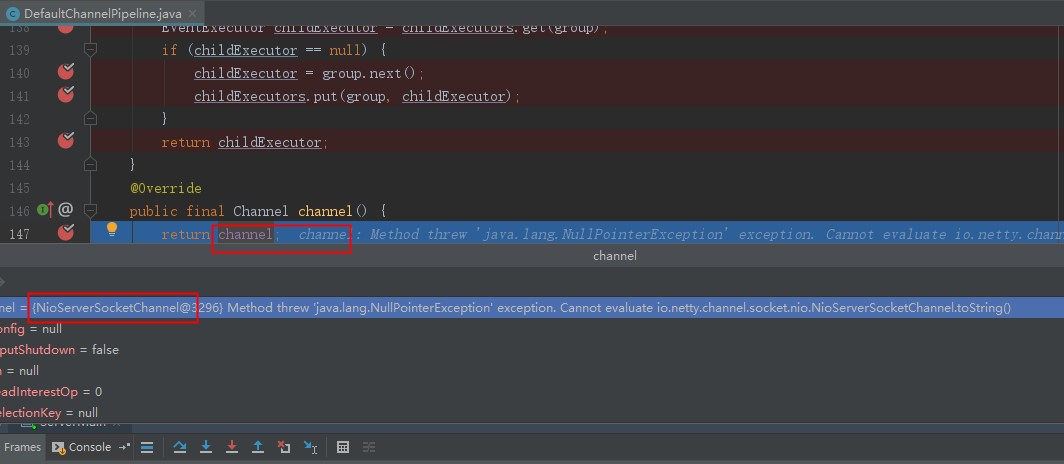


图3-7-2-1-14

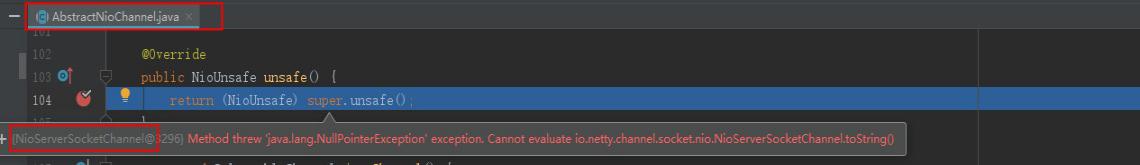
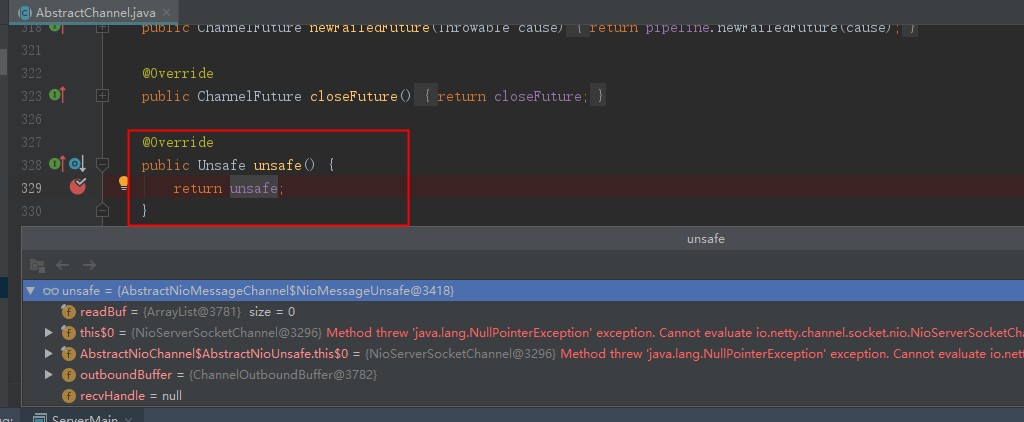


图3-7-2-1-15



5）、DefaultChannelPipeline 100、101行代码代表初始化头、尾。以后存储ChannelHander时使用双向链表数据结构存储，为什么不用java.util现成的？