# 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

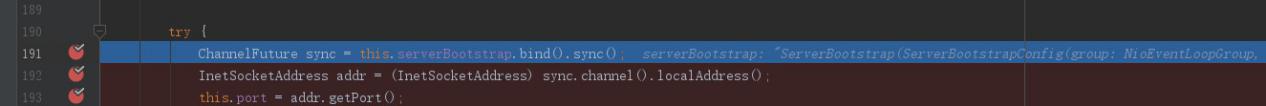


图3-1-1-1 所示，是netty 服务端导向启动类示例（参考rocketmq)

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 启动

图3-1-2-1



### 3.1.3 ServerBootstrap.bind()方法

#### 3.1.3.1执行AbstractBootstrap.bind()方法

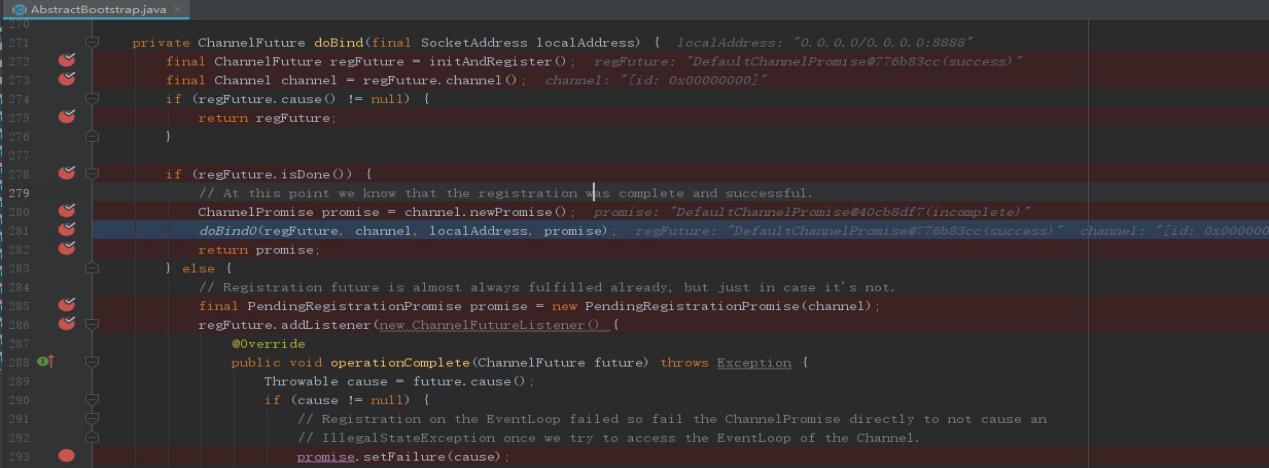
图3-1-3-1-1

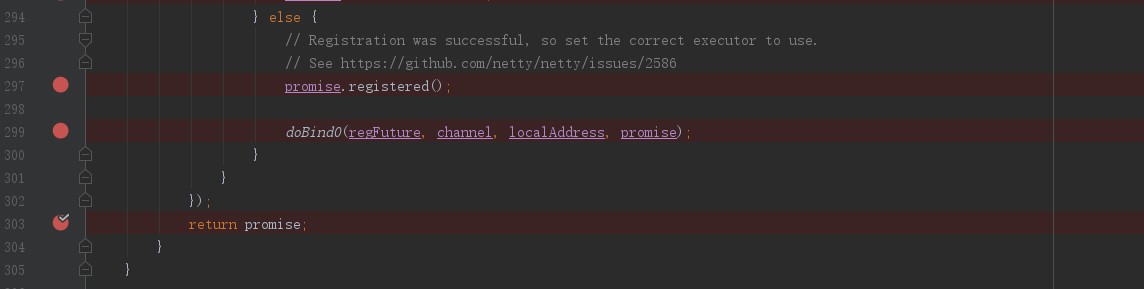


1)、执行3.1.3.2doBind方法

#### 3.1.3.2 doBind方法

图3-1-3-2-1

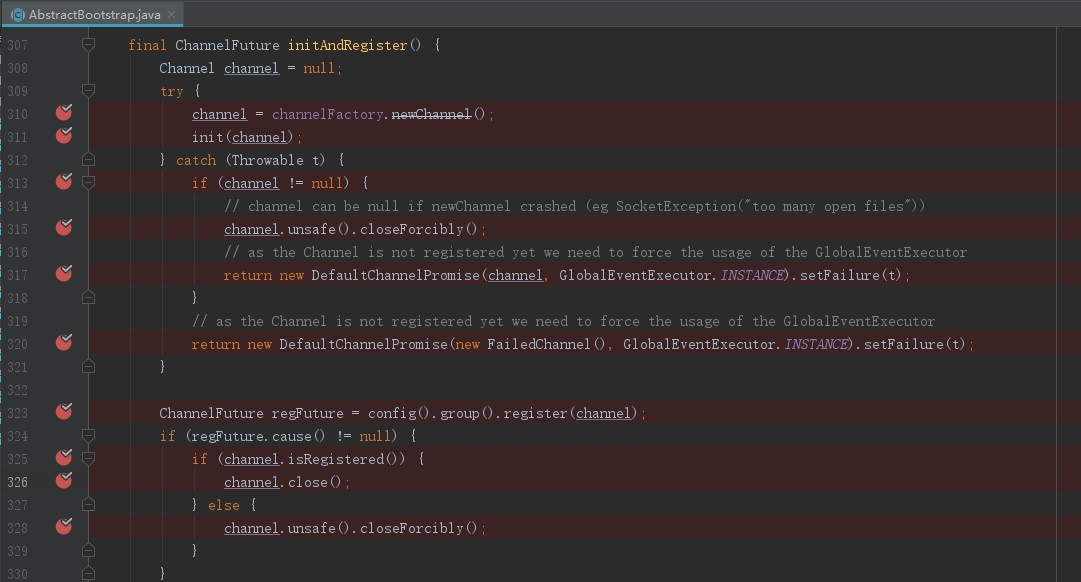


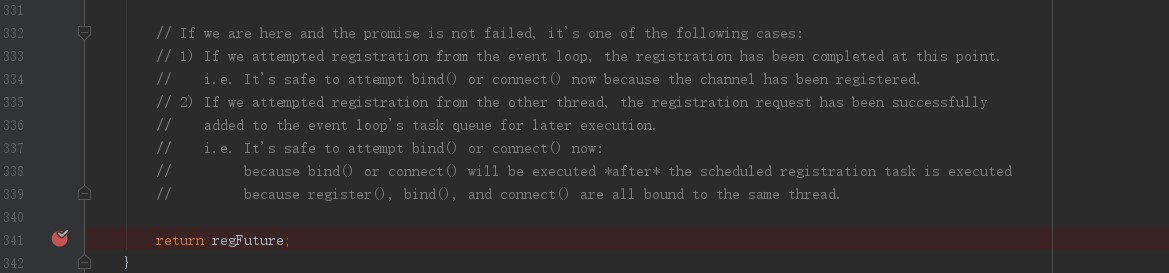


1. 、272 执行3.1.3.3
2. 、281 执行3.1.3.4

#### 3.1.3.3 AbstractBootstrap.initAndReigster()

图3-1-3-3-1

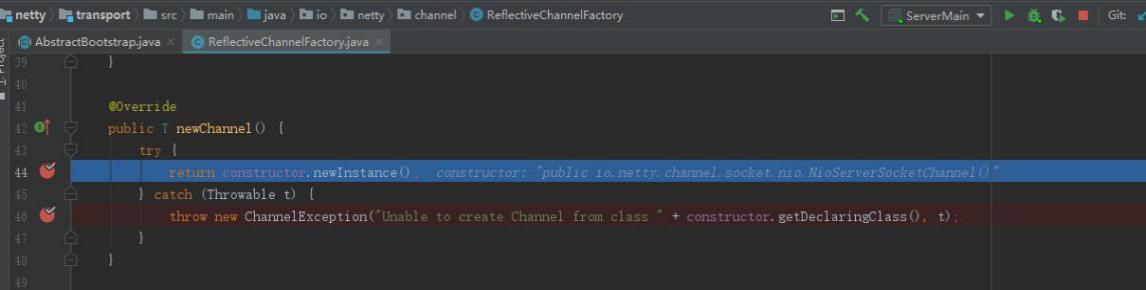




1）图3-1-3-3-1 310行 通过ReflectiveChannelFactory 创建Channel实例。

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

图3-1-3-3-2

其实就是NioServerSocketChannel 的Constructor构造方法实例

构造NioServerSocketChannel 参考(3.6)

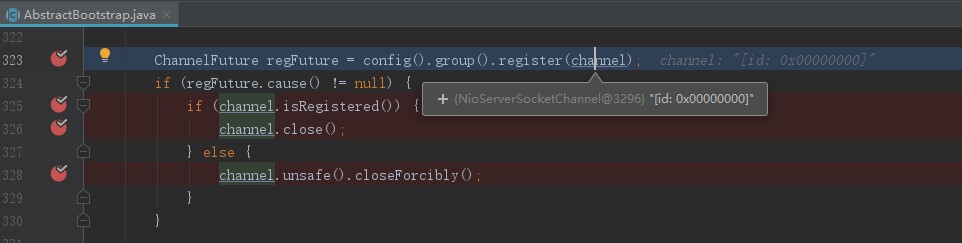
2）、图3-1-3-3-1 310行(参考3.1.4ServerBootstrap.init()方法）

3）、图3-1-3-3-1 323行 config()获取NioServerSocketChannelConfig实例,config().group()

获取NioEventLoopGroup 实例，config().group().register(channel)即

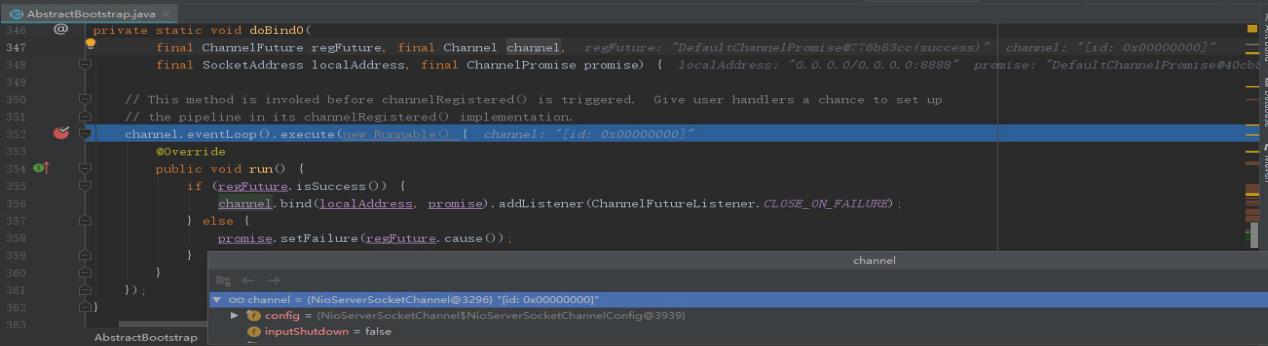
执行NioEventLoopGroup的register()方法，入参为3-1-3-3-2所示NioServerSocketChanne实例。(参考3.2.3)

图3-1-3-3-2



#### 3.1.3.4 doBind0

图3-1-3-4-1



1. 、reqFuture是DefaultChannelPromise实例
2. channel是NioServerSocketChannel实例
3. promise是DefaultChannelPromise实例
4. 352行channel.eventLoop()返回的是AbstractChannel.eventLoop（NioEventLoop），

所以channel.eventLoop().execute方法执行进入图3-1-3-4-2，进而进入3-1-3-4-3

图3-1-3-4-2

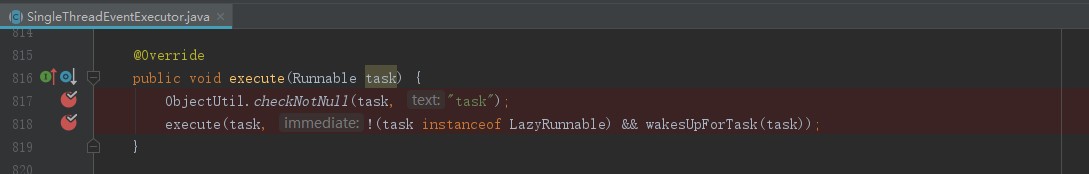
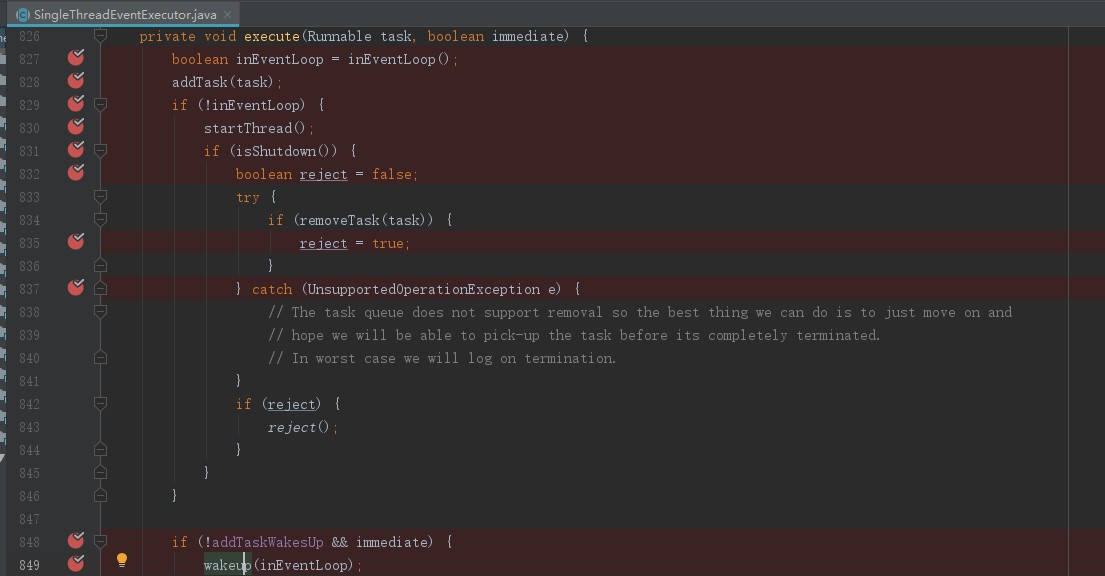


图3-1-3-4-3



2

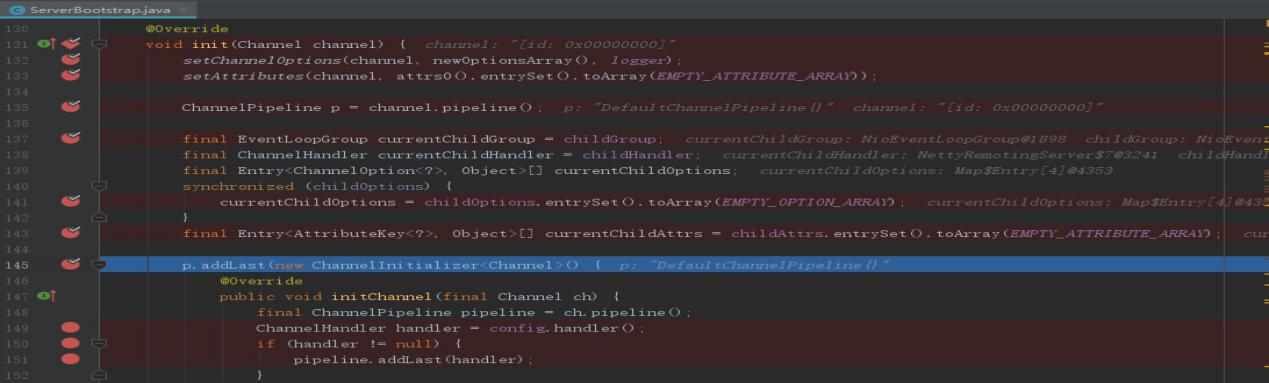
4.1）、当前线程是否与SingleThreadEventExecutor.thread相等

4.2）、828把当前任务添加进SingleThreadEventExecutor.taskQueue。（3.3.8）

4.3、830行参考3.3.5

### 3.1.4ServerBootstrap.init()方法

图3-1-4-1





1）、132行初始化相关option选项。如图3-1-4-2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3-1-4-3、图3-1-4-4、图3-1-4-5所示，当然不同的option

具体设置可能不同，例如3-1-4-6、3-1-4-7。

图3-1-4-2

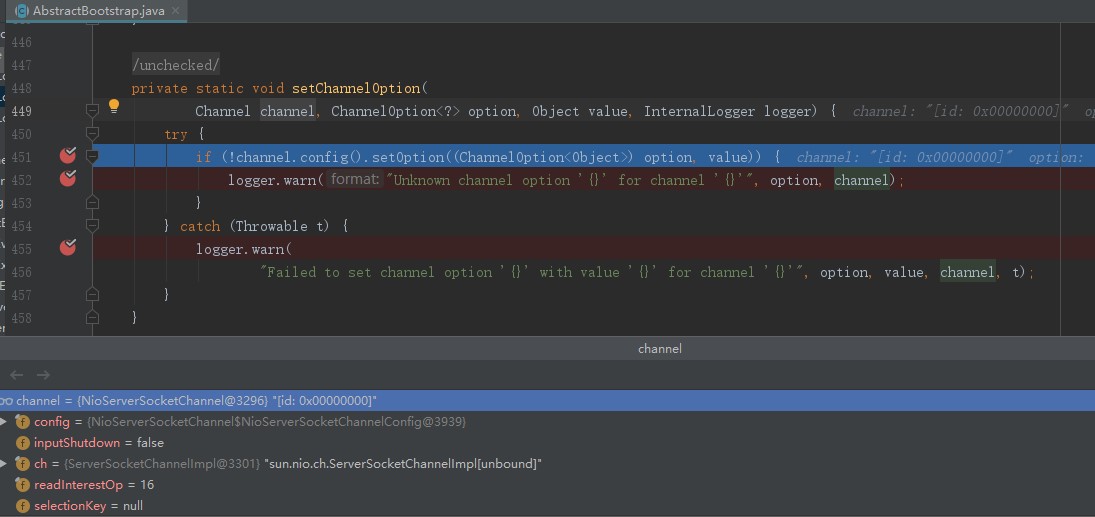


图3-1-4-3

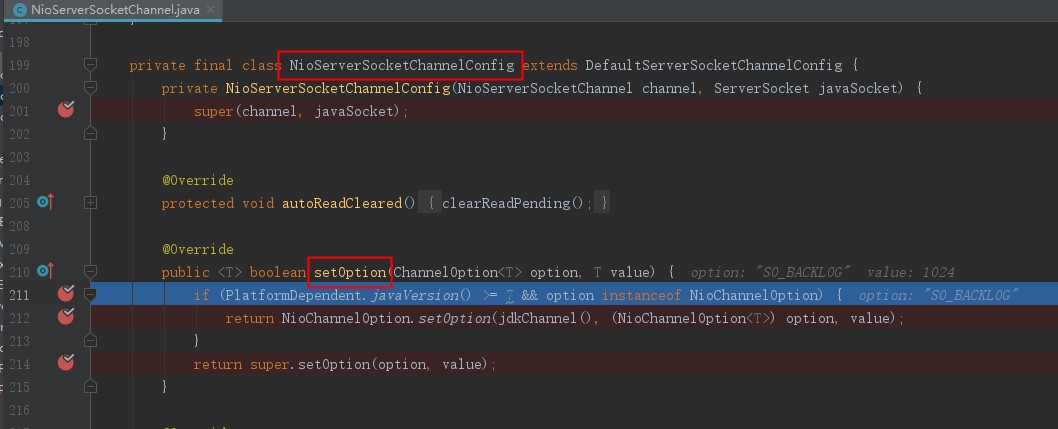


图3-1-4-4

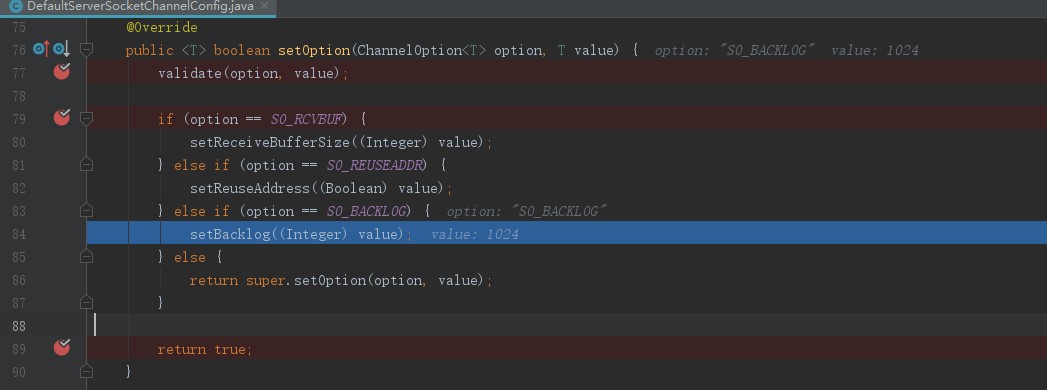


图3-1-4-5

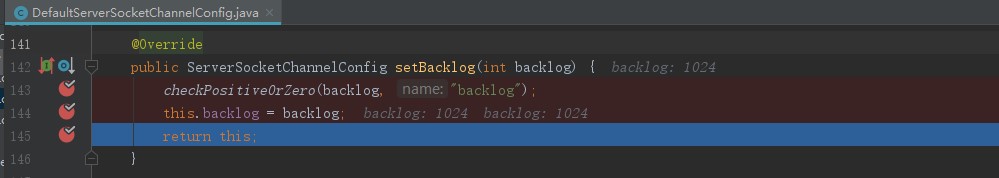


图 3-1-4-6

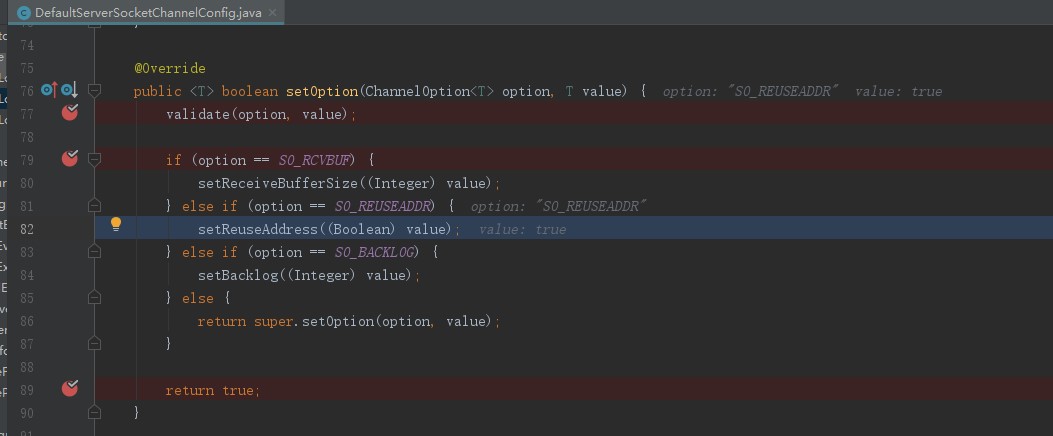
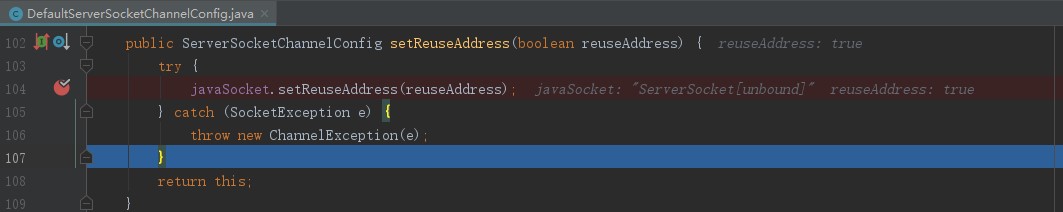


图3-1-4-7

、

2）、attrs设置

3）、145行，p即DefaltChannelPipeline实例，执行addLast方法，参考3-1-4-8、3-1-4-9所示，3-1-4-9 executor为null。

图3-1-4-8

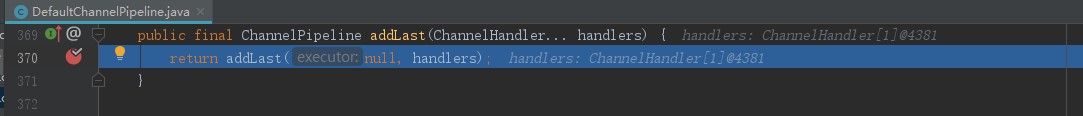


图3-1-4-9



图3-1-4-10

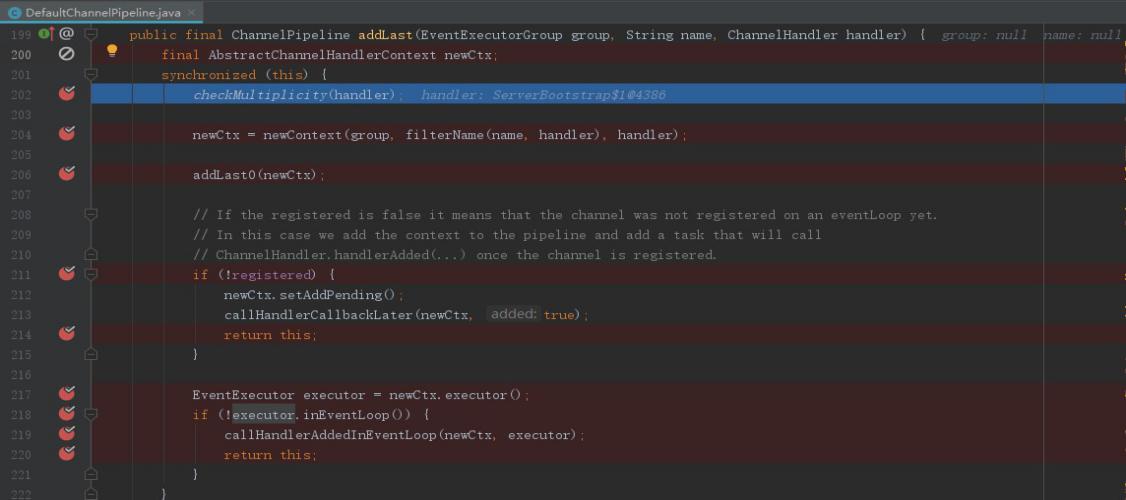




图3-1-4-10 202行校验是否重复添加或没有@Sharable注解 如果直接抛出异常，否则该设置ChannelHandler.added=true。

图3-1-4-10 204行 创建newCtx实例，参考3-1-3-11

图3-1-4-10 206行，把DefaultChannelHandlerContext newCtx实例添加到双向链表尾部，参考图3-1-4-14

图3-1-4-11

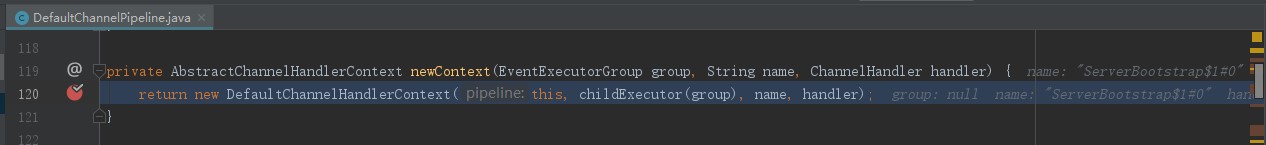
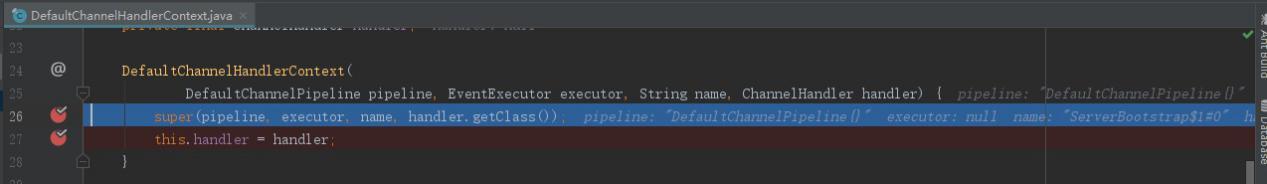


图3-1-4-12



piepeline为DefaultChannelPipeline实例，

executor为null

略

图3-1-4-13

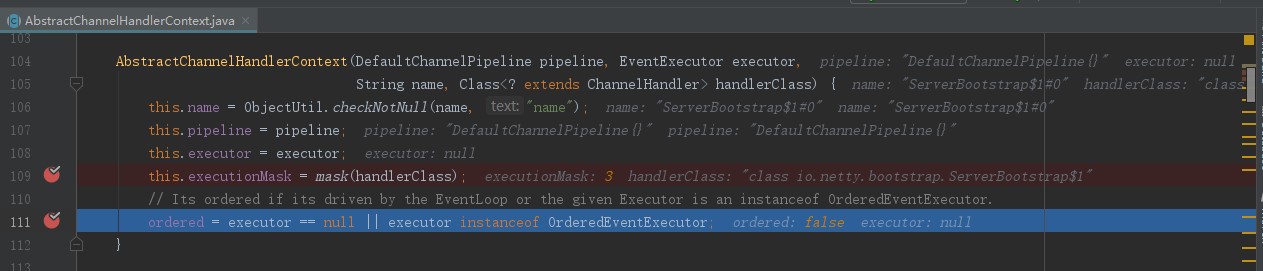
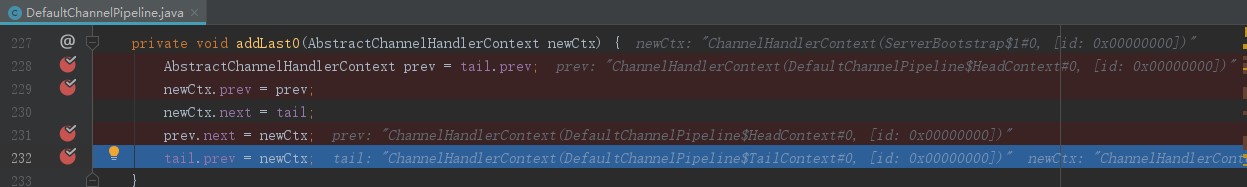


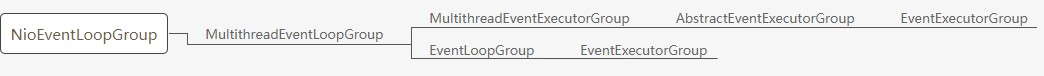
图3-1-4-14



## 3.2 NioEventLoopGroup类

### 3.2.1继承关系

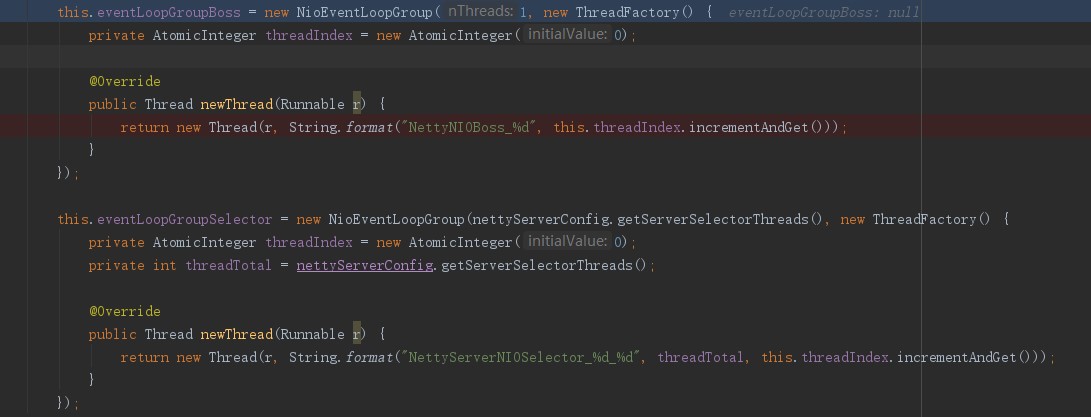
图3-2-1-1



### 3.2.2 构造方法

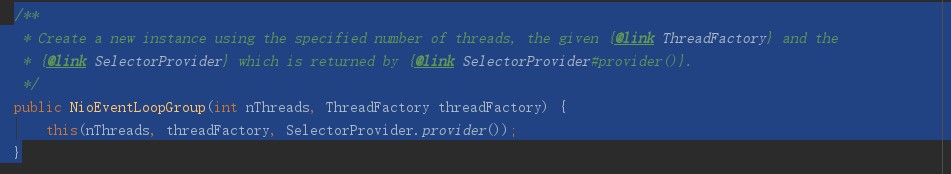
#### 3.2.2.1 构造方法1

图3-2-2-1



#### 3.2.2.2 构造方法2

图3-2-2-2



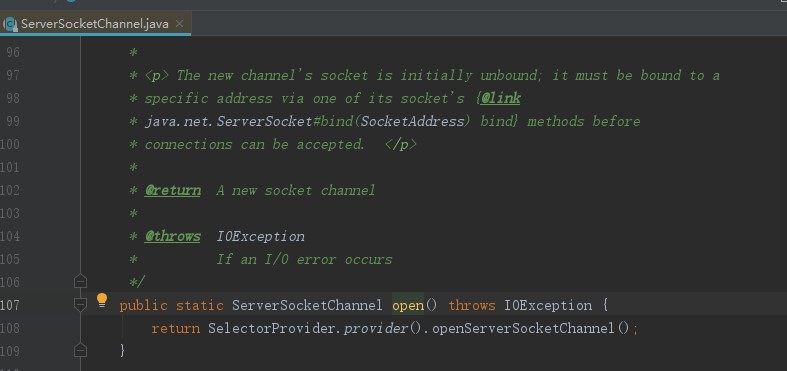
1. nThreads线程数
2. threadFactory用于创建线程的工厂实例（具体代码参考，图3-2-2-1）

#### 3.2.2.3 构造方法3

图3-2-2-3

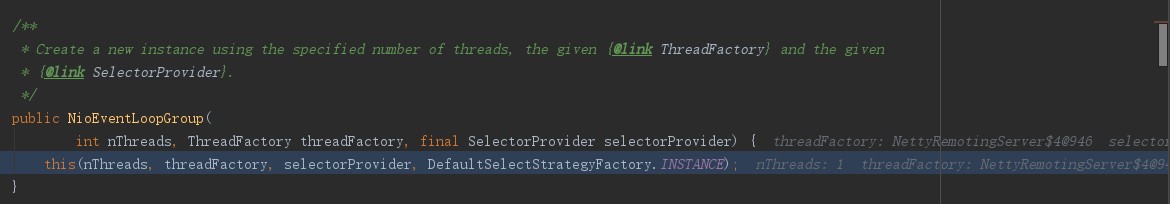


1. SelectorProvider.provider() 返回一个SelectorProvider 的实现类实例。Java.nio.channels.ServerSocketChannel.open()方法中就用到了该对象。如图3-2-2-4



#### 3.2.2.4 构造方法4

图3-2-2-5



1. DefaultSelectStrategyFactory.INSTANCE实例

图3-2-2-6

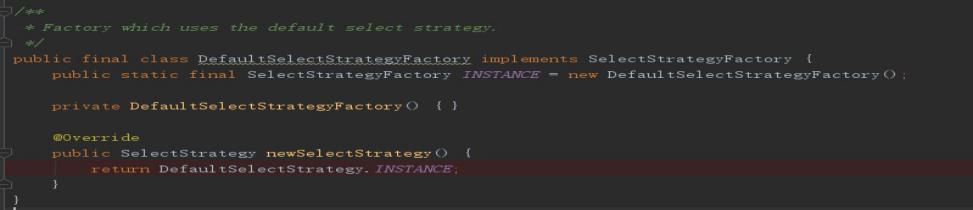
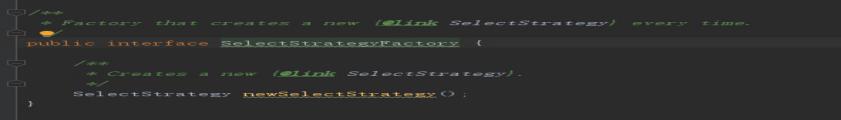


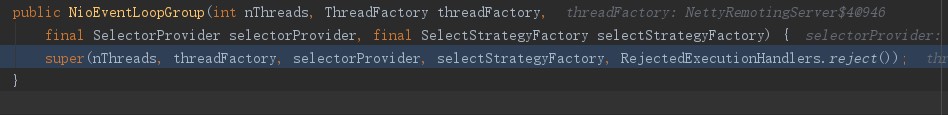
图3-2-2-7



感觉没有啥用

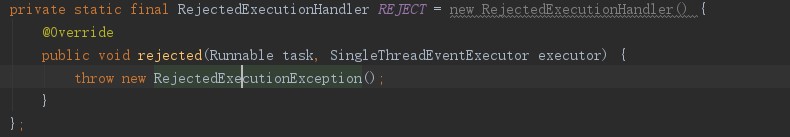
#### 3.2.2.5 构造方法5

图3-2-2-8



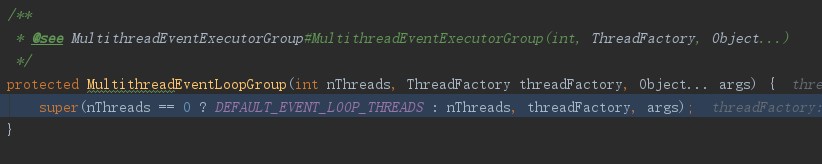
1. RejectedExecutionHandlers.reject()方法返回RejectedExecutionHandler实现类。

图3-2-2-9



#### 3.2.2.6 父构造方法6

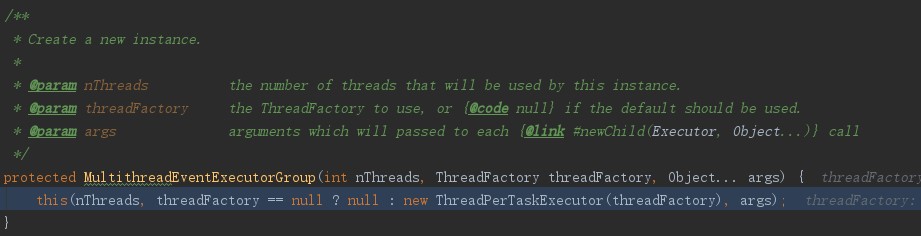
图3-2-2-10



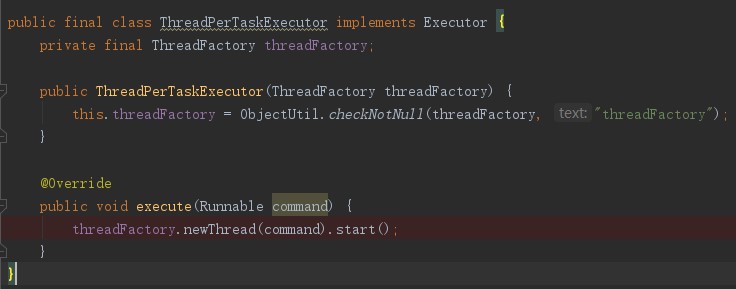
1. 如果nThreads为0，取默认值，默认值最小为1，最大可用核数的2倍。

#### 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

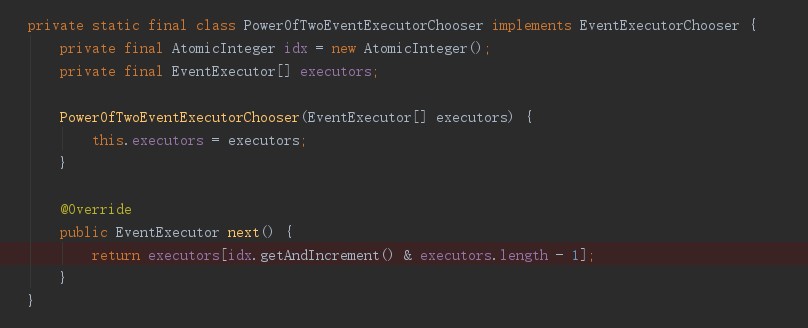
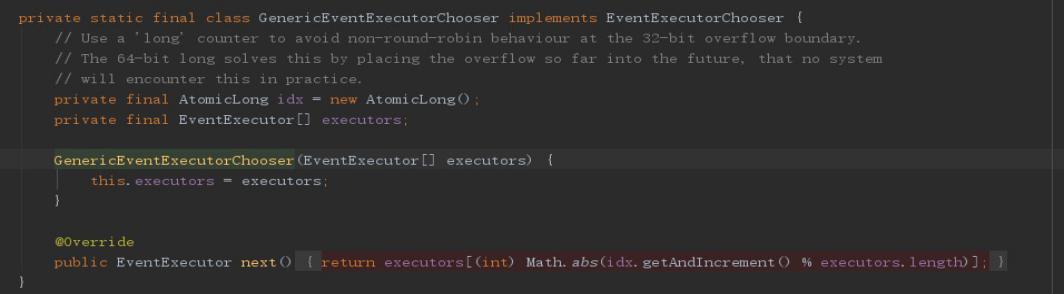


图3-2-2-14



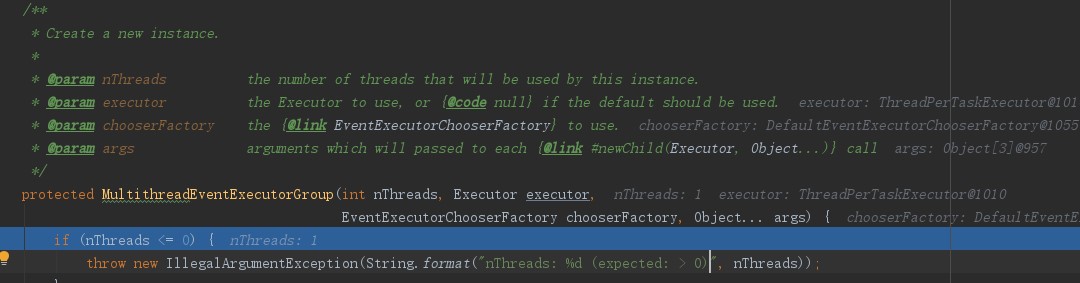
这两个实现类都实现next()方法，只是获取executors数组中的索引方式不一样。

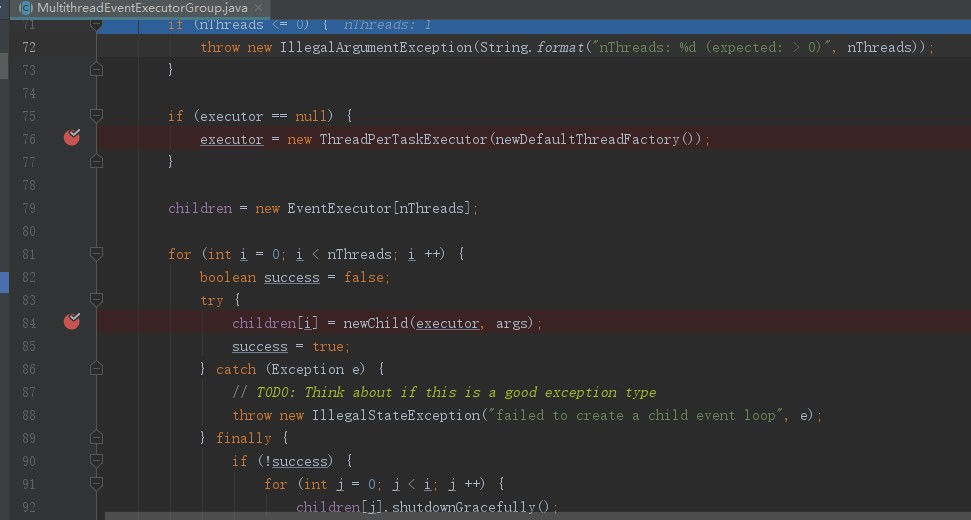
PowerOfTwoEventExecutorChooser#next()方法，计算数组索引方式：增量数字&数据长度-1。GenericEventExecutorChooser#next方法，计算数组索引方式：增量取余数据量长度，返回绝对值。

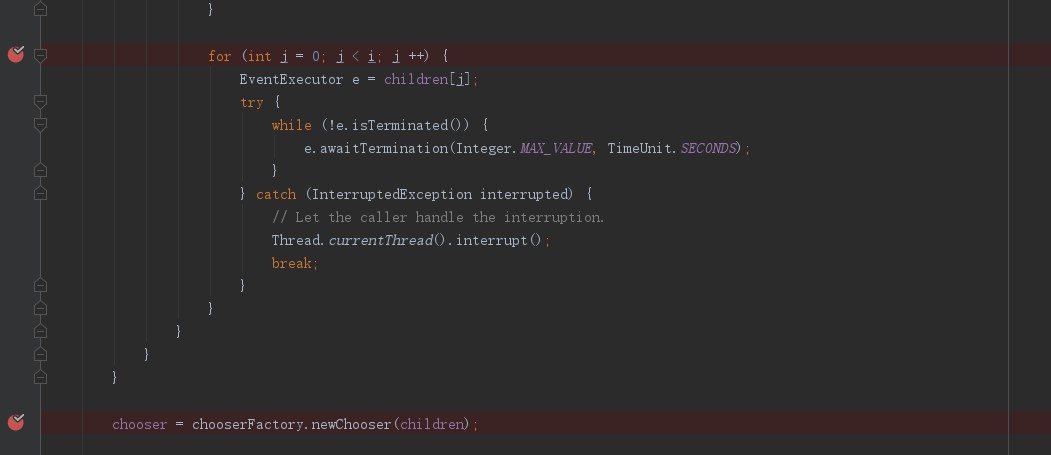
4）EventExecutor对象 其实就是NioEventLoopGroup父类MultithreadEventExecutorGroup中的children，就是NioEventLoop实例。

#### 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）









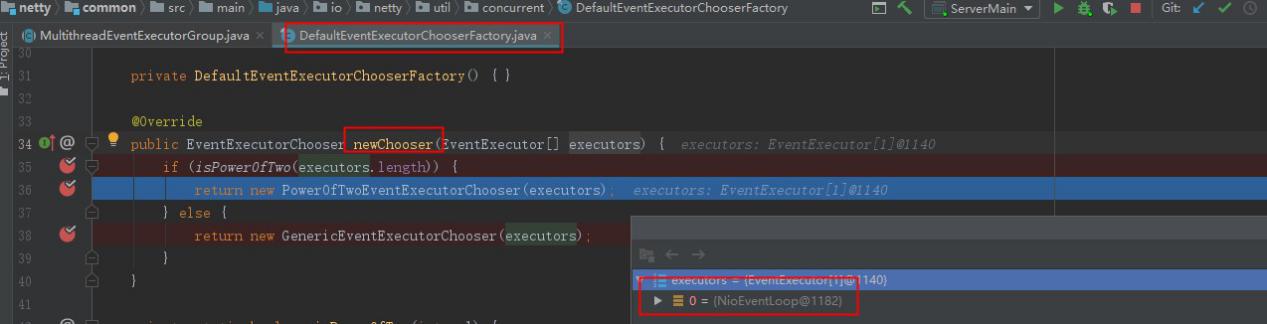
1）nThreads是初始化EventExecutor子类的个数。windows是指NioEventLoop

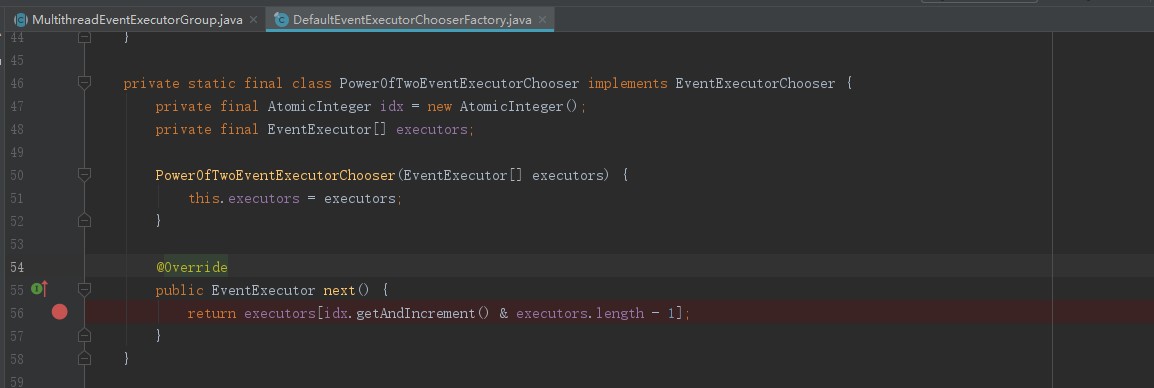
2）如果executor为空，重新创建一个ThreadPerTaskExecutor实例。

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。



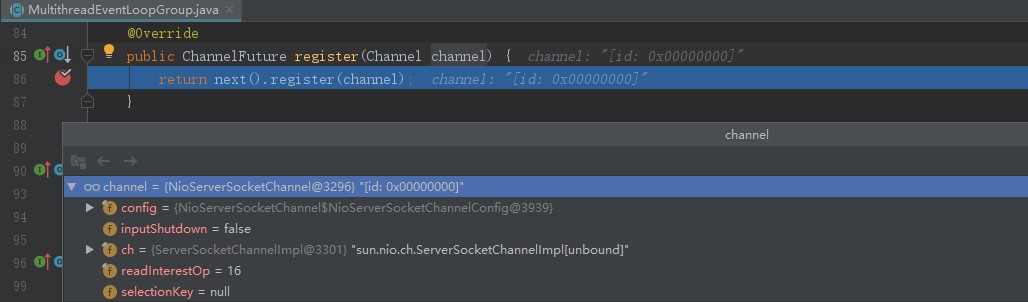


其实就是把NioEventLoop数组 赋值给PowerOfTwoEventExecutorChooser.executors，从而使用NioEventLoopGroup、NioEventLoop产生联系。

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

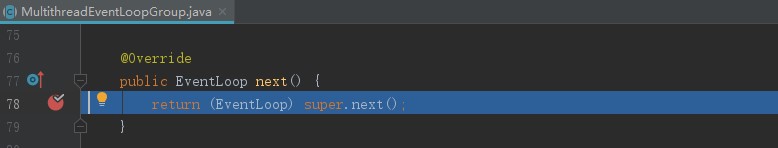


图3-2-3-3

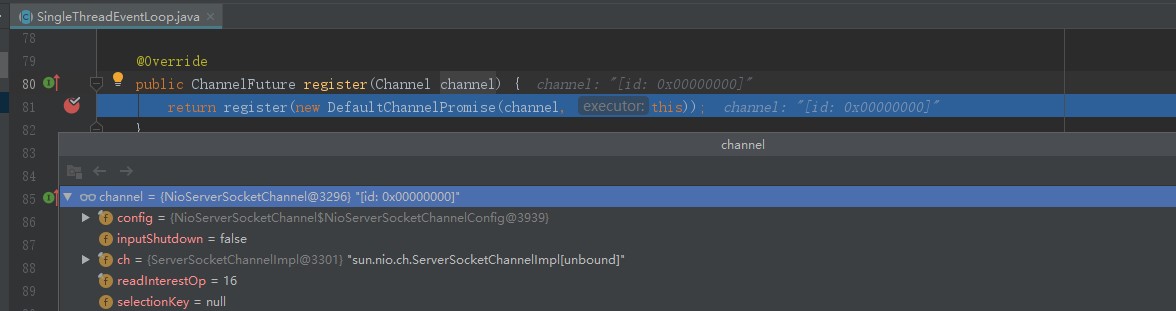


图3-2-3-4



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

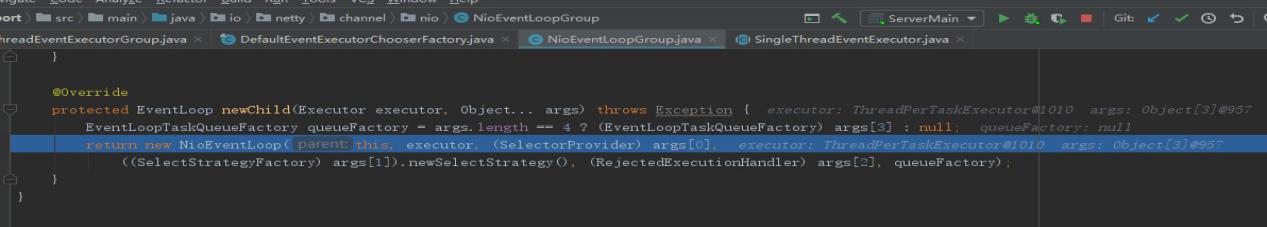
图3-3-1-1



### 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

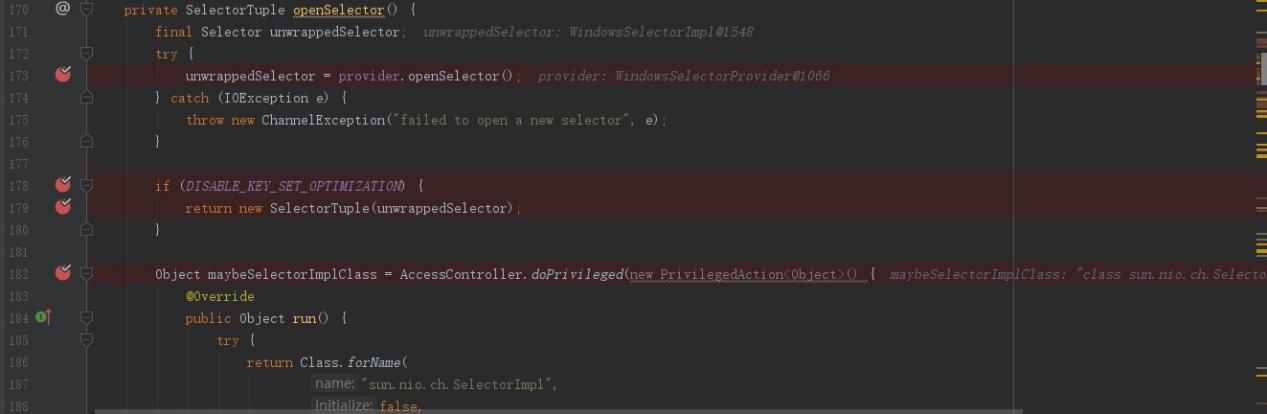
作用待定？？？？

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

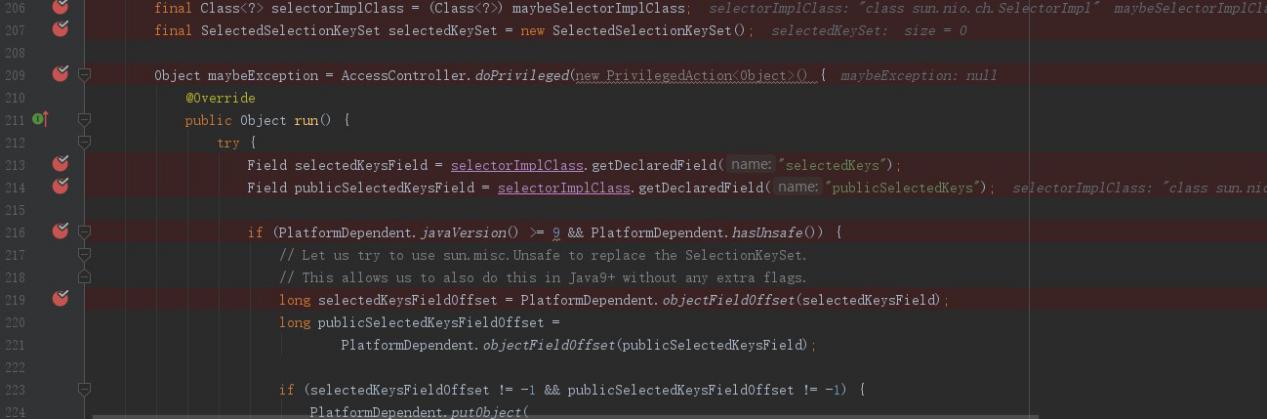
6）queueFactory对为null

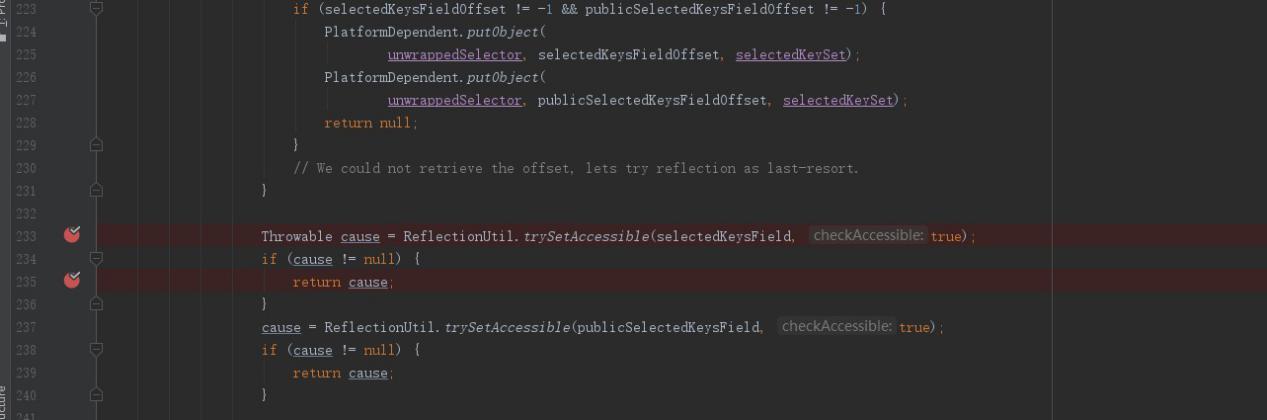
7）SelectorTuple 对象初始化，通过openSelector()方法

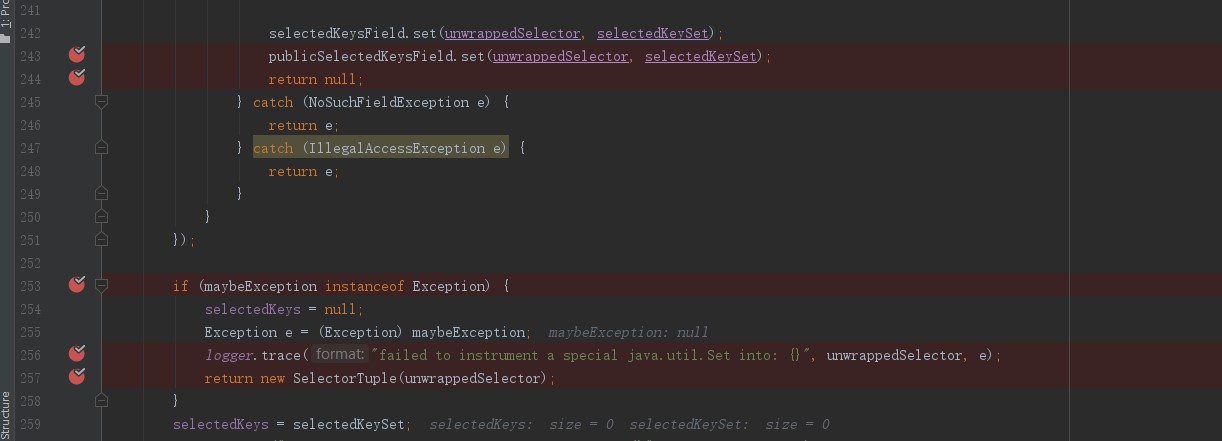
图3-3-2-2-2（下面6张是一个完整NioEventLoop#openSelector()方法）

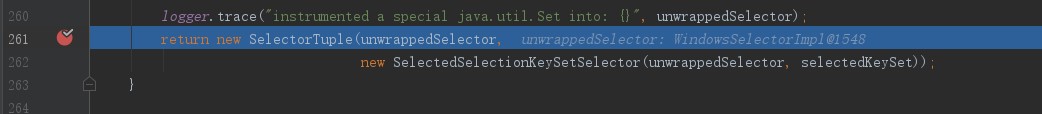












unwrappedSelector 是SelectorProvider.openSelector（）方法返回WindowsSelectorImpl

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

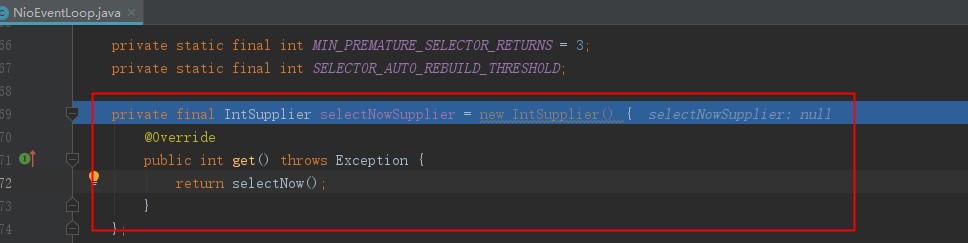
10)、看来NioEventLoop 在构造过程中，初始化了java.nio.channels.Selector，并给

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

11）、

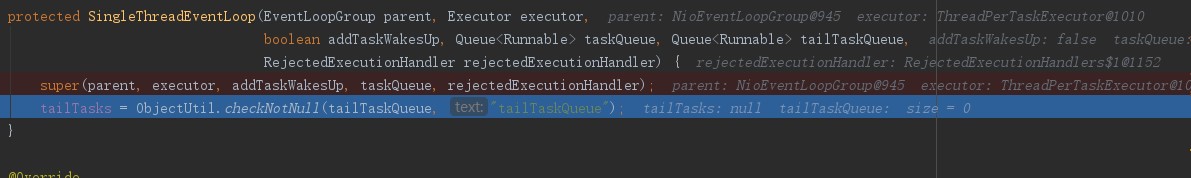
12）成员变量NioEventLoop.selectNowSupplier对象实例化

3-3-2-2-2



#### 3.3.2.3父类构造方法

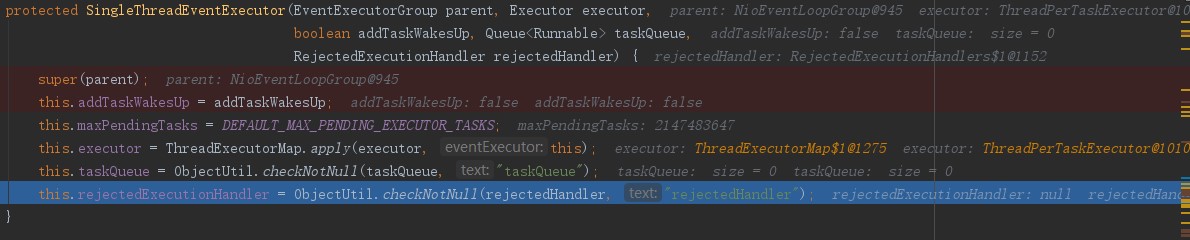
图3-3-2-3



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

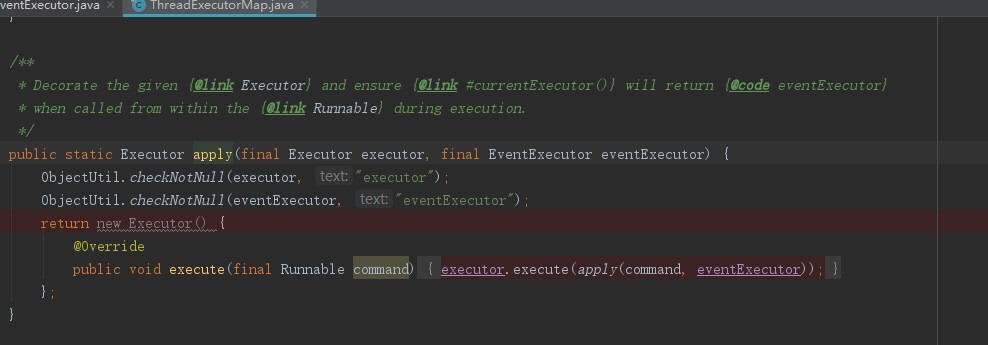
#### 3.3.2.4 父类构造方法

图3-3-2-4



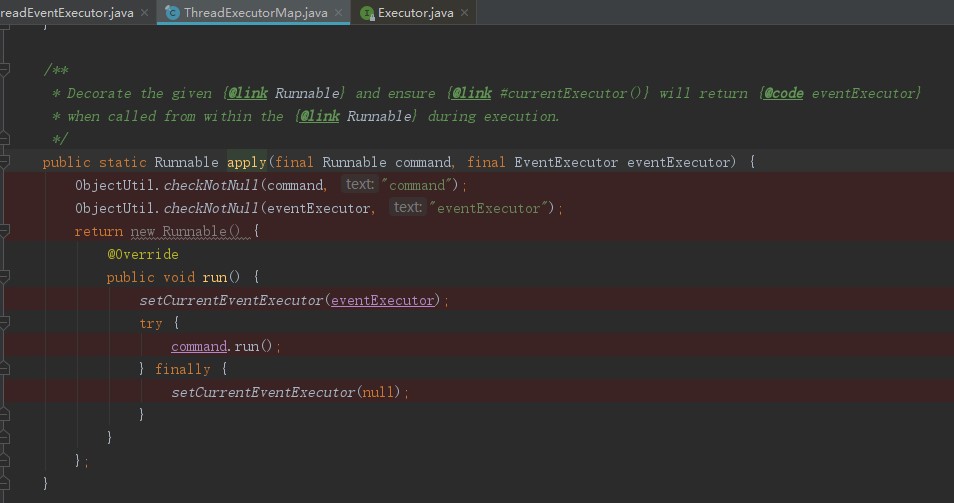
1. parent是NioEventLoopGroup实例
2. 入参executor是ThreadPerTaskExecutor实例
3. addTaskWakesUp false
4. maxPendingTasks 最大挂起任务数量

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

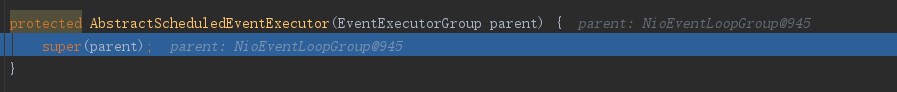


apply方法就是返回一个Runnable实现类实例，最终让ThreadPerTaskExecutor重启线

程执行。

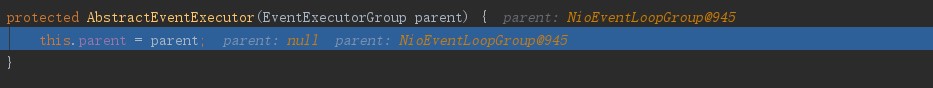
6）taskQueue子构造方法传递过来的

#### 3.3.2.5 父类构造方法



1. 略

#### 3.3.2.6 父类构造方法

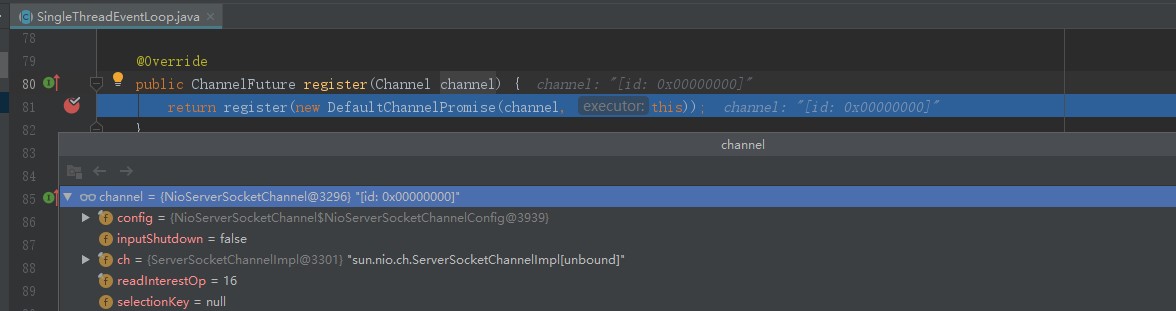


1. parent对象是NioEventLoopGroup对象实例

### 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

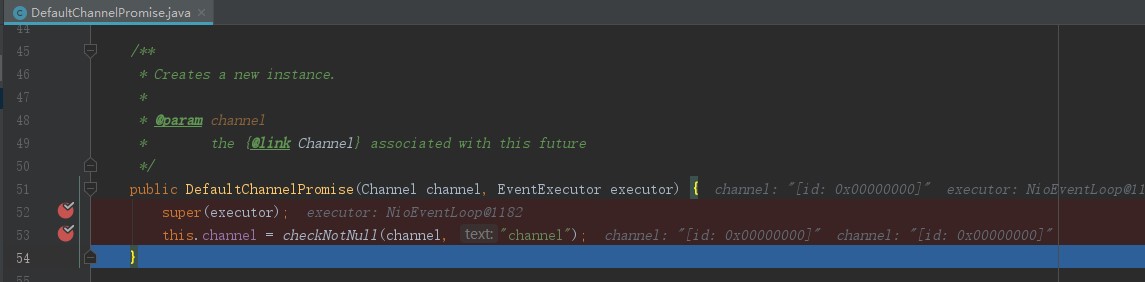
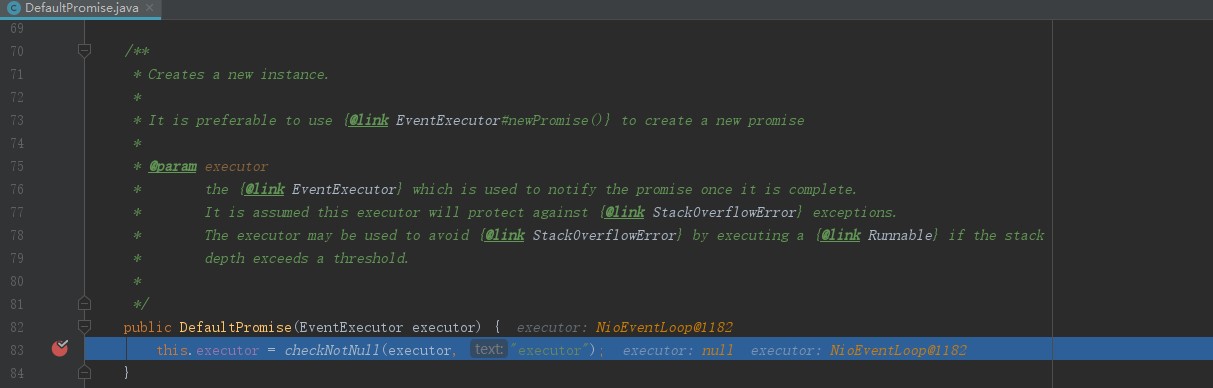
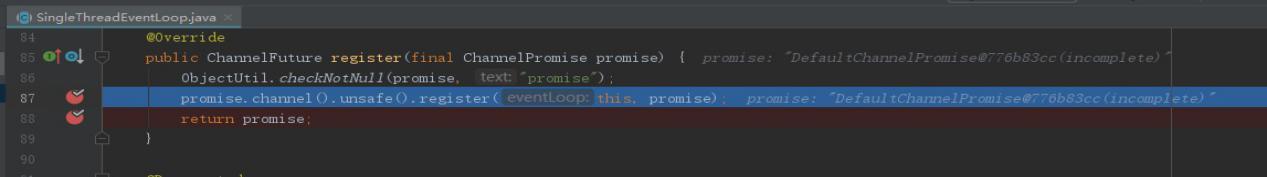


图3-3-3-1-3



#### 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

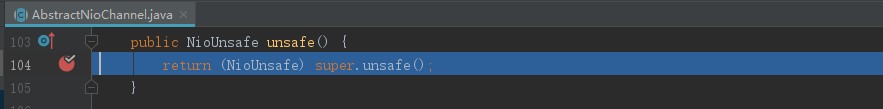
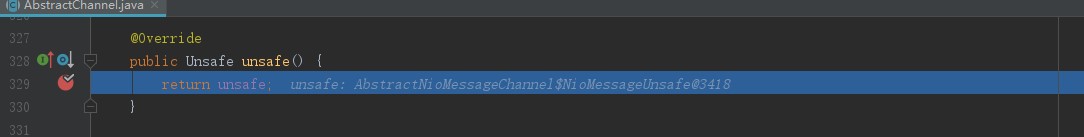
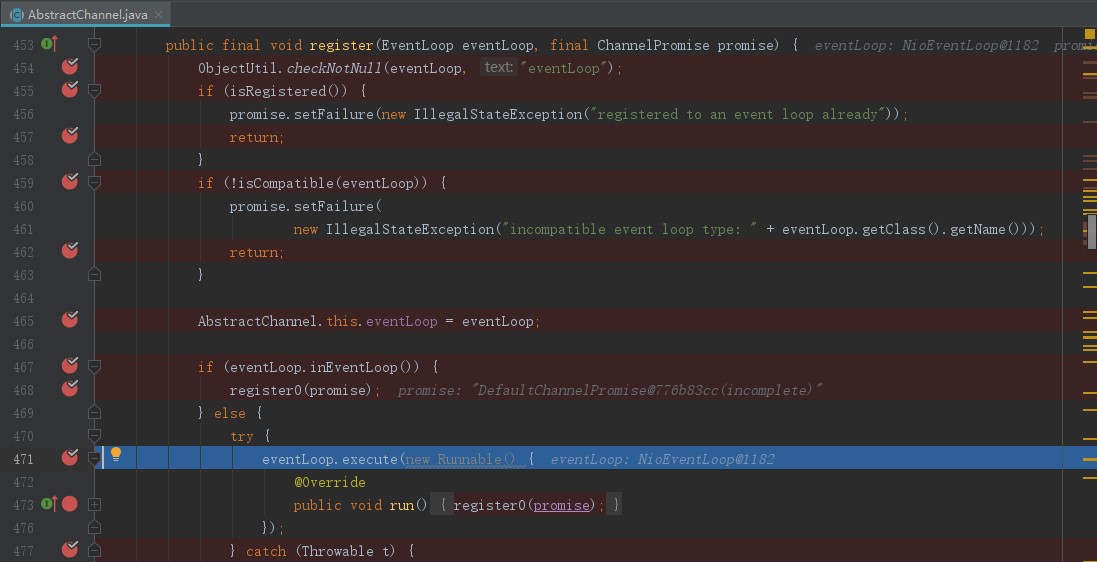


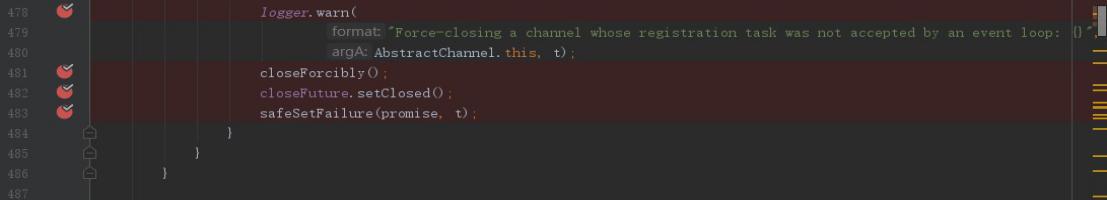
图3-3-3-2-4



#### 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。

5)、465行this.eventLoop = eventLoop，eventLoop来自于图图3-1-1-1

中的eventLoopGroupBoss（AbstractBootstrap.group成员变量）初始化的

NioEventLoop实例。这也是为什么初始化eventLoopGroupBoss时，只创建

一个NioEventLoop实例。（非常重要）

6)、467行，执行inEventLoop()方法如图3-3-3-3-2、3-3-3-3-3所示，判断当前线程与

NioEventLoop父类SingleThreadEventExecutor.thread线程是否相等，相等返回true，

否则返回false。通过图3-3-3-3-3由于NioEventLoop实例中this.thread为null，导致

返回false所示图3-3-3-3-1进代码进入471行

图3-3-3-3-2

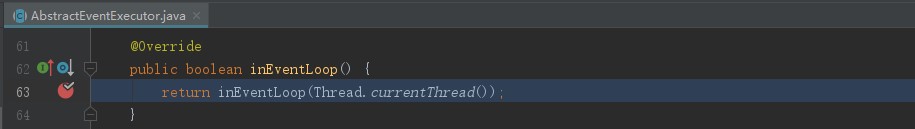
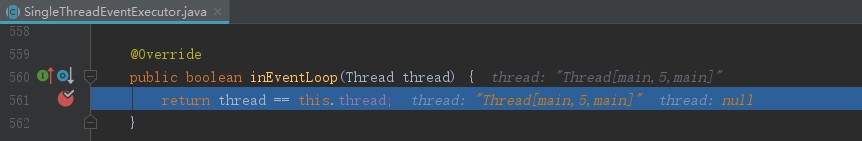


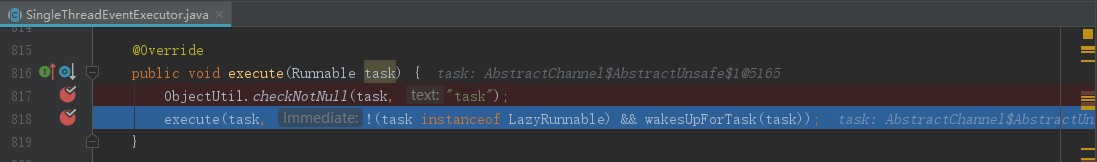
图3-3-3-3-3



7)、471行，进入NioEventLoop 父类execute方法(图3-3-3-3-4），该入参task即为图3-3-3-3-1

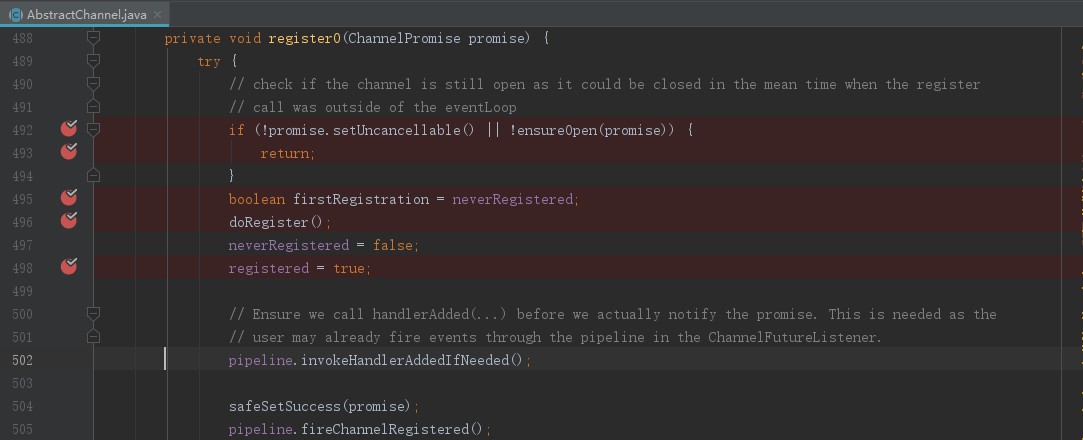
474行register0(promise)。（参考3.3.3.4）。task实现接口run即3.3.3.4register0接口逻辑。关于execute参考3.3.4。

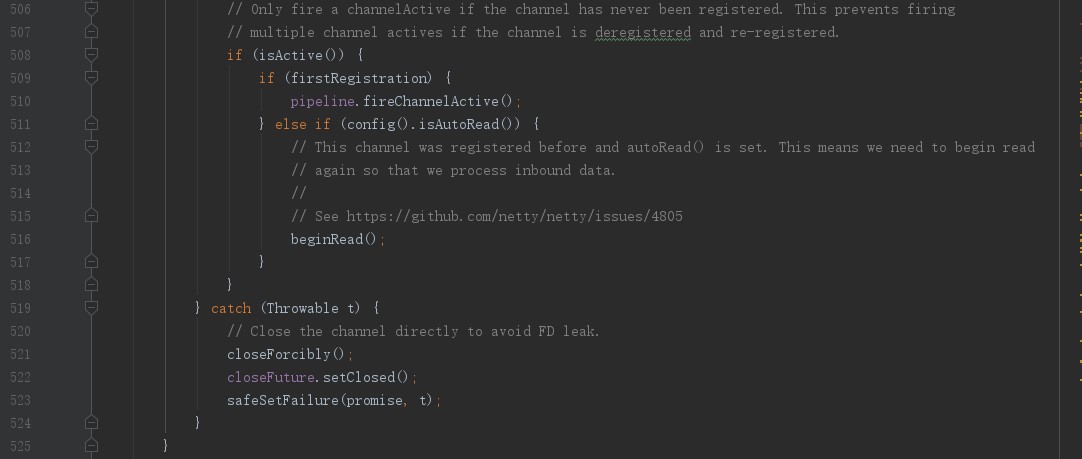
图3-3-3-3-4



#### 3.3.3.4 AbstractUnsafe.register0方法

图3-3-3-4-1

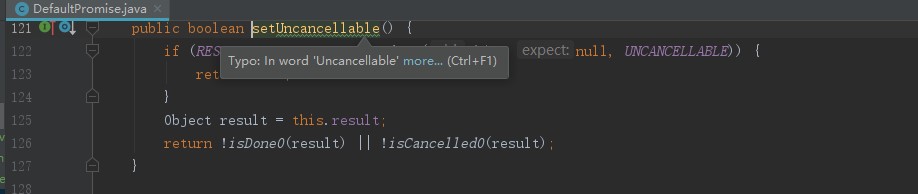




1）492行promise.setUncancellable() 判断是否可以取消，如果可以返回true，否false

参考图3-3-3-4-2

图3-3-3-4-2



1. 、492行ensureOpen方法，进入方法内部如图3-3-3-4-3所示，首页判断

ServerSocketChannel是否打开，如果打开返回true；否则进入图3-3-3-4-5并且

返回false。图3-3-3-4-5promise.tryFailure(cause)参考3.8.4所示

图3-3-3-4-3

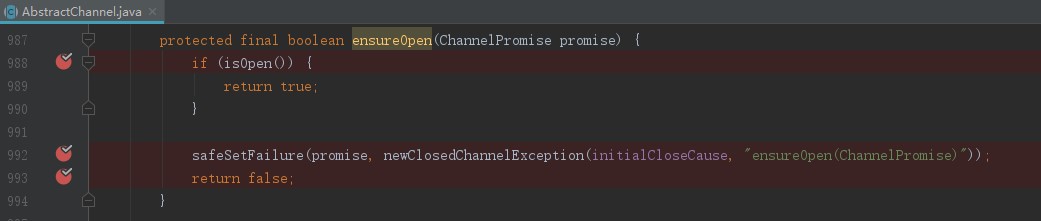


图3-3-3-4-4

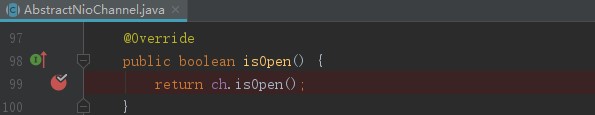
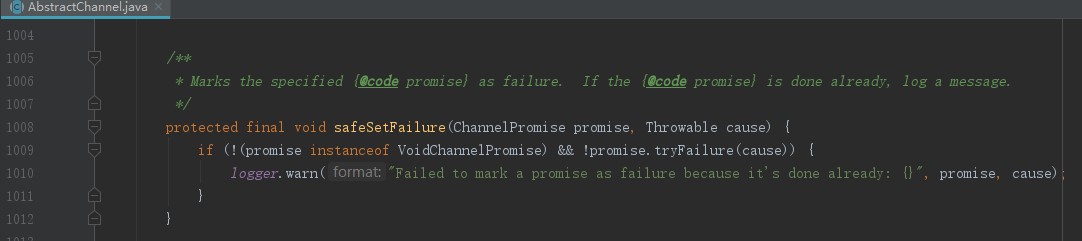


图3-3-3-4-5



1. 、496行 参考3.6.3所示
2. 、497、498 neverRegistered=false, registered=true
3. 、502行，执行pipeline.invokeHandlerAddedIfNeeded()方法（参考3.7.3）
4. 、504行，执行safeSetSuccess，如图3-3-3-4-10,promise.trySuccess()参考

3.8.3 trySuccess方法。

图3-3-3-4-10



7）505行， pipeline.fireChannelRegistered();参考(3.7.4)

8）、508行，如果对应的Channel已经打开并且绑定，isActive()方法返回true，其它返回

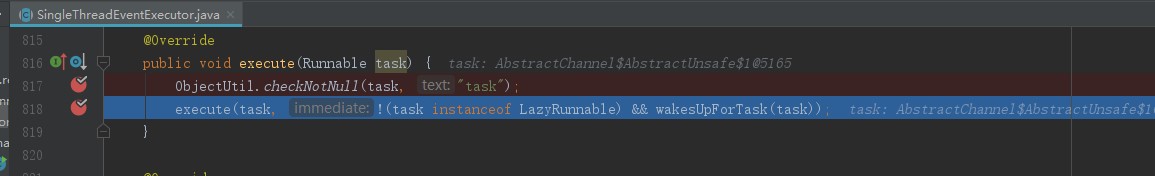
False

1. 、510行，执行pipeline.fireChannelActive参考3.7.5

### 3.3.4 execute方法

#### 3.3.4.1 execute方法1

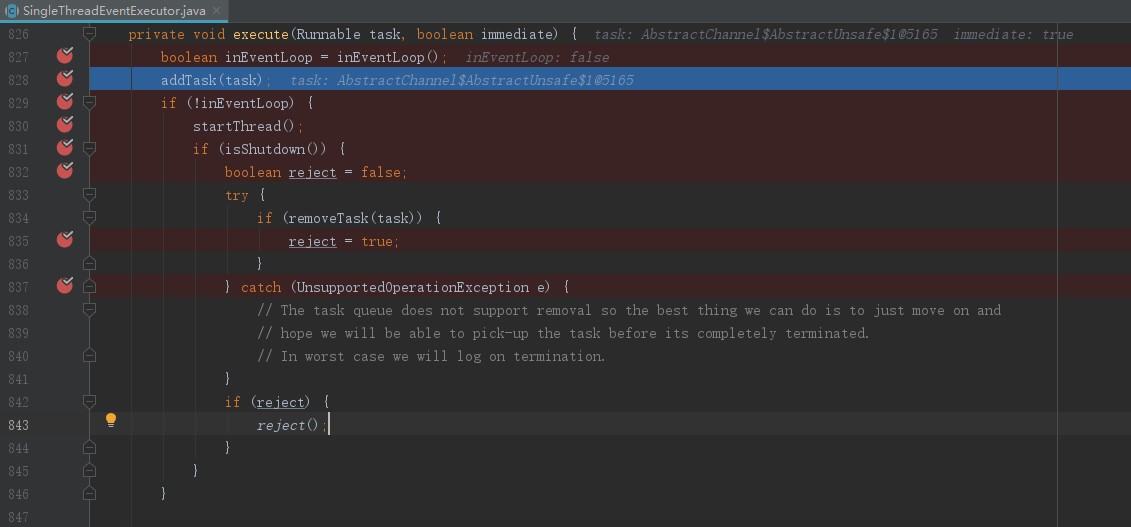
图3-3-4-1

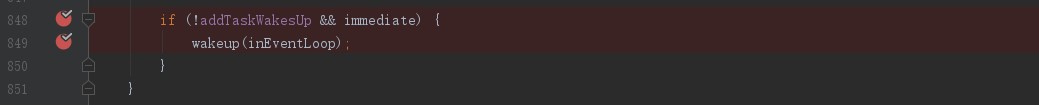


1. 、task核心逻辑是3.3.3.4接口

#### 3.3.4.2 execute方法

图3-3-4-2-1





1）、827为false是因为SingleThreadEventExecutor.thread为null导致的

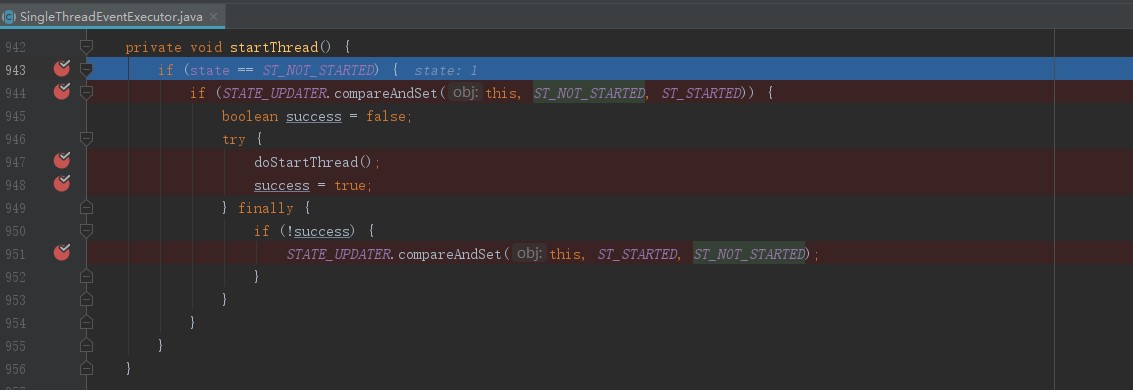
2）、829 addTask方法，即把task任务（3.3.3.4）添加到SingleThreadEventExecutor.taskQueue

队列中

3）、830行，（参考3.3.5）

### 3.3.5 startThread

图3-3-5-1

 1）、943 如果state是未开始状态，即继续执行，否则结束。这是一种锁

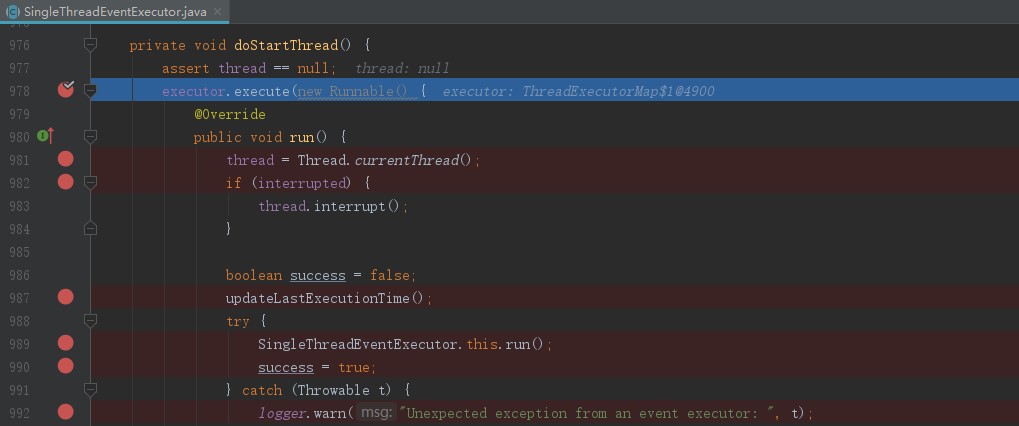
1. 、944 STATE\_UPDATER 是AtomicIntegerFieldUpdater对象，即CAS加锁机制，如果

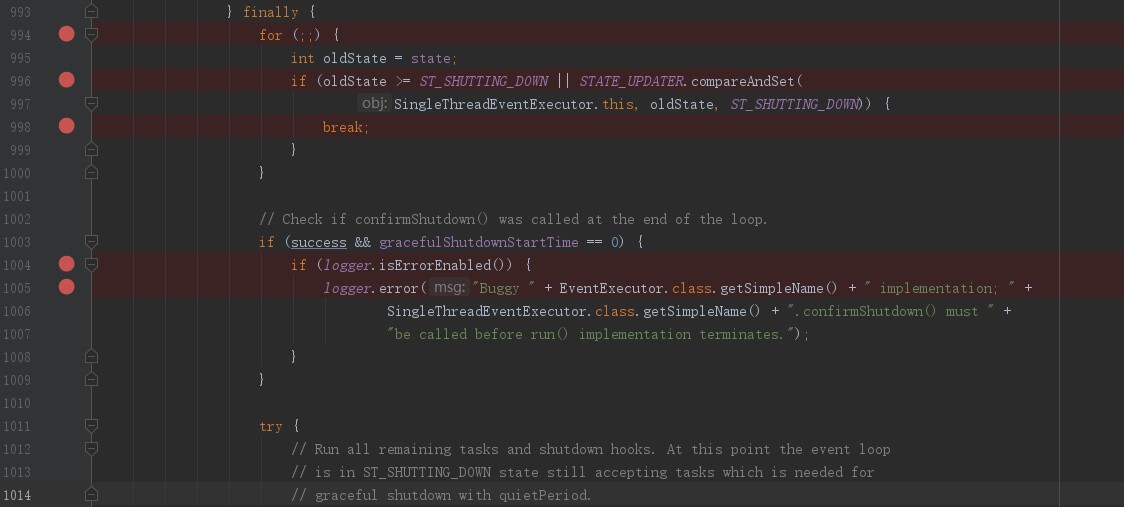
成功state值即变成ST\_STARTED即值为2。

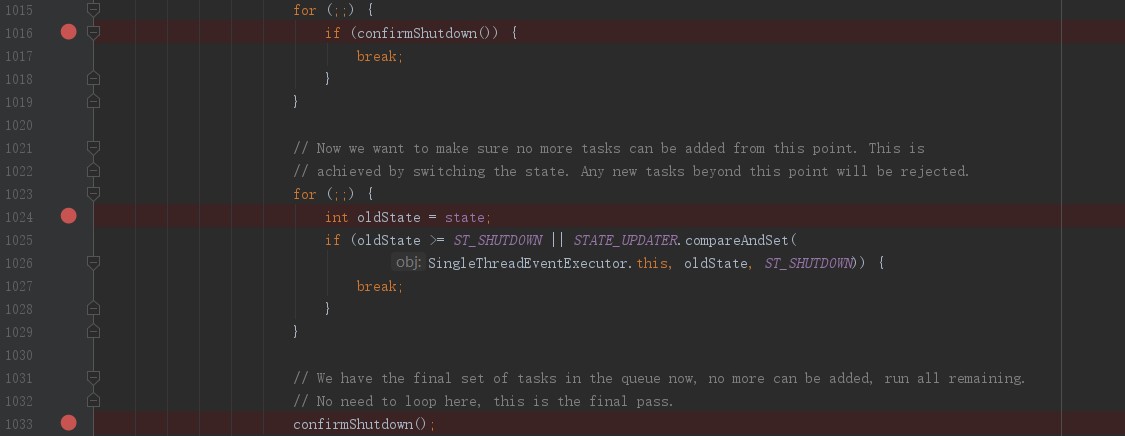
1. 、947执行doStartThread()方法（参考3.3.6）
2. 951 如果success为false，并且state ==2时，再把state变成1。

### 3.3.6 doStartThread

图3-3-6-1









1）979行的executor 已经介绍过了（参考 3.3.2.4 父类构造方法）

1. 、978 行

3.1）、通任务重新封装一个任务，并在72行加了锁，76释放锁（3-3-6-3所示）。

3.2）、图3-3-6-2 57行的executor是ThreadPerTaskExecutor实例，该executor就是创建

一个新线程执行任务。

图3-3-6-2

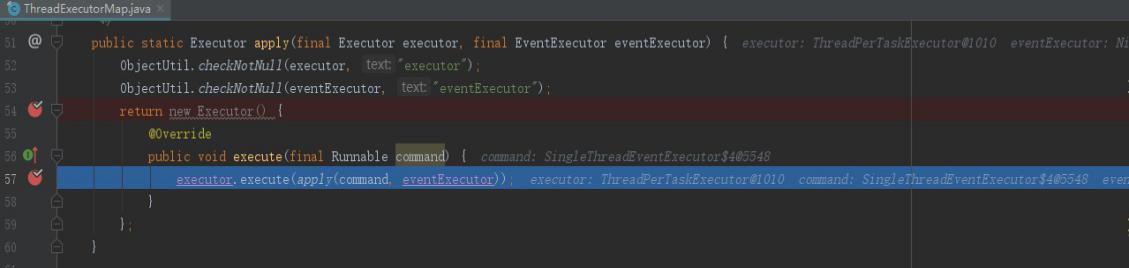
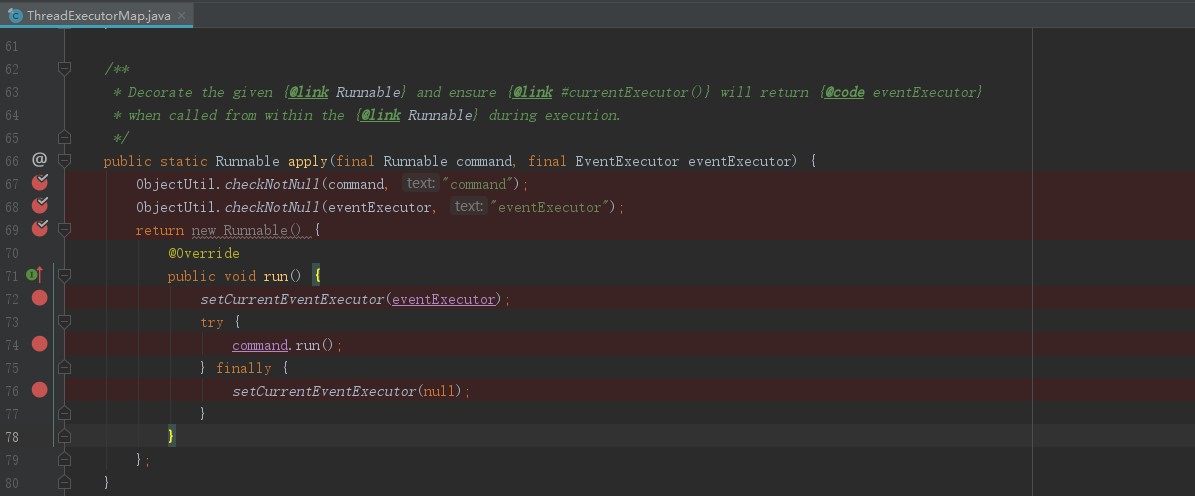


图3-3-6-3



1. 、987 更新SingleThreadEventExecutor.lastExecutionTime =当前时间戳-

ScheduledFutureTask.START\_TIME(ScheduledFutrueTask类第一次加载时间)。

1. 、989 run方法 （参考3.3.7）
2. 、990如果989顺序执行success=true。
3. 、994-1000行代码，如果SingleThreadEventExecutor.state 是ST\_SHUTTING\_DOWN、ST\_SHUTDOWN、ST\_TERMINATED 或者能把ST\_SHUTTING\_DOWN值更新到SingleThreadEventExecutor.state时 跳出循环，继续向下执行。
4. 、1015-1019行，行确认SingleThreadEventExecutor.taskQueue队列是否已经执行完毕。

参考3.3.8

1. 、1023-1029行

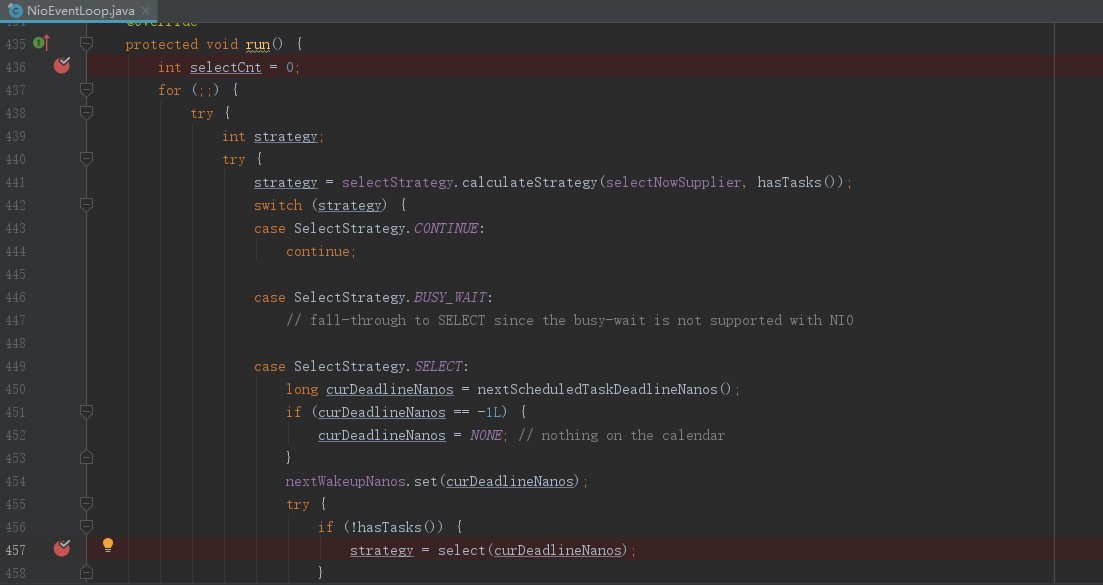
8.1)SingleThreadEventExecutor.state >= 4 ，循环跳出继续执行

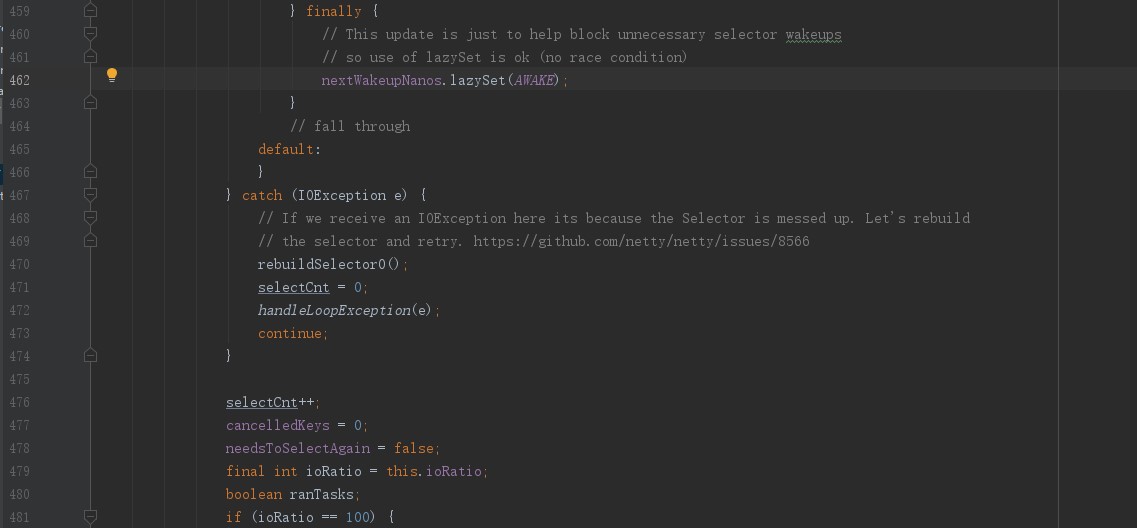
8.2）如果，8.1)是fase。如果SingleThreadEventExecutor.state 是当前状态，并把

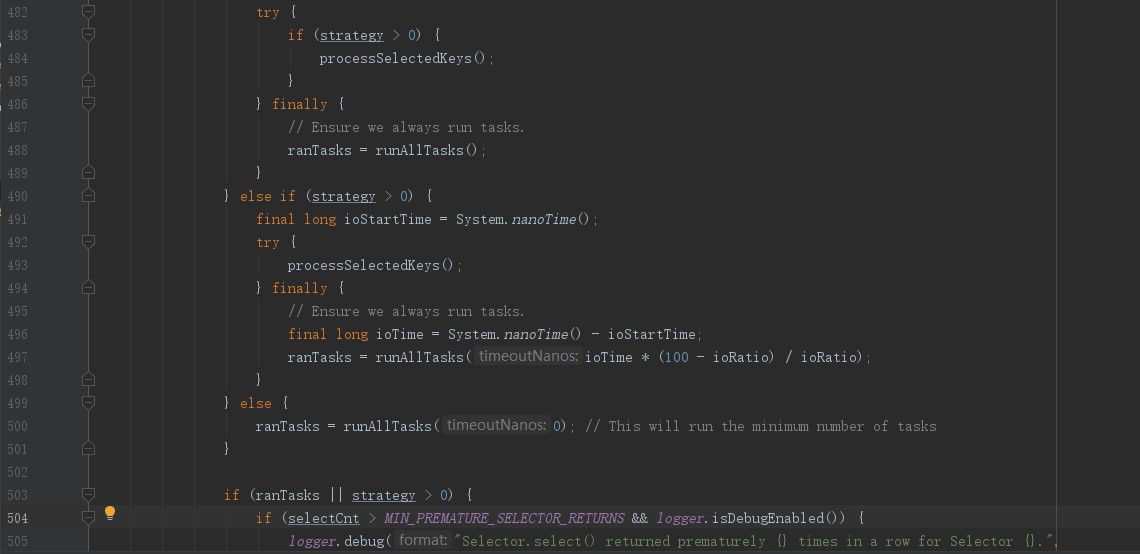
ST\_SHUTDOWN 赋值给SingleThreadEventExecutor.state，同样跳出循环。

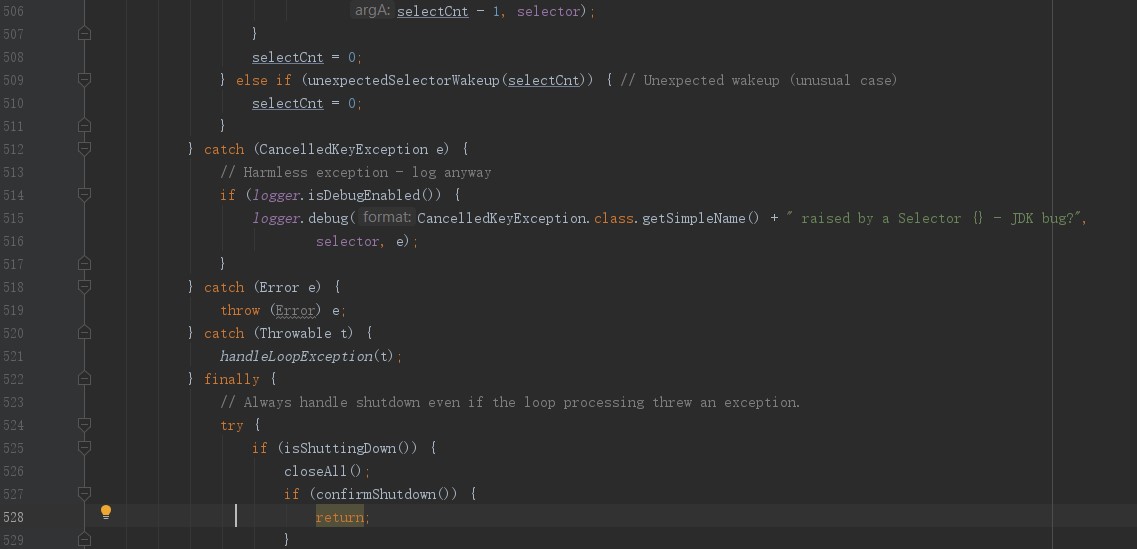
1. 、1033再次执行confirmShutdown()方法。参考(3.3.8)
2. 、1036 空方法体
3. 、1043
4. 、1044行把ST\_TERMINATED 赋值给SingleThreadEventExecutor.state
5. 、1045 唤醒使用threadLock.await方法的线程。
6. 、1046 把SingleThreadEventExecutor.taskQueue队列中剩余的任务全部干掉
7. 、1051 参考3.8.7

### 3.3.7 SingleThreadEventExecutor.run()











1. 、441行解析selectStrategy是初始化NioEventLoopGroup时创建的（也可能是其它的类初始，参考3.2）。
   1. SingleThreadEventExecutor.taskQueue队列非空 或SingleThreadEventLoop.tailTasks

队列非空条件下：strategy是通过java.nio.channels.spi.Selector（实现类sun.nio.ch.WindowsSelectorImpl) selectNow()方法返回的结果

* 1. 、如果1.1条件不成立：

strategy=-1;

1. 、如果strategy ==-2 时中止继续从头开始执行。
2. 、如果strategy==-3 或strategy==-1

3.1）从AbstractScheduledEventExecutor.scheduledTaskQueue 队列中取一个任务（该任务并没有从队列中移除) ，如果存在任务返回任务的截止时间戳，不存在返回-1。

3.2）、如果当前时间戳==-1，当前时间戳设置成Long的最大值

3.3）、NioEventLoop.nextWakeupNanos 设置当前截止时间戳

4)、456行 SingleThreadEventExecutor.taskQueue队列为空 并且

SingleThreadEventLoop.tailTasks队列为空条件下 ：

4.1）当前截止时间戳==-1：

java.nio.channels.spi.Selector（实现类sun.nio.ch.WindowsSelectorImpl) select()方法返回的结果。

4.2)、计算出超时时间戳timeoutMillis，

如果timeoutMillis<=0 ：

java.nio.channels.spi.Selector（实现类sun.nio.ch.WindowsSelectorImpl) selectNow()方法返

如果timeoutMillis>0：

java.nio.channels.spi.Selector（实现类sun.nio.ch.WindowsSelectorImpl) select(timeoutMillis)方法返

5)、再次把NioEventLoop.nextWakeupNanos设置成-1。

6)、481-489行，如果NioEventLoop.ioRatio==100 并且strategy >0条件时：执行7）

7)、processSelectedKeys()方法逻辑。如果NioEventLoop.selectedKeys不是null时：

7.1）、遍历该集合

7.2）、取出集合中元素；集合中元素赋值为null；

7.3）如果元素中附件对象是AbstractNioChannel时：执行8）

7.4）如果元素中附件对象不是AbstractNioChannel时：执行9）

如果NioEventLoop.selectedKey是null:

7.5)、遍历NioEventLoop.selector.selectedKeys()

7.6)、取出集合中元素，删掉相应元素

7.7）如果元素中附件对象是AbstractNioChannel时：执行8）

7.8）如果元素中附件对象不是AbstractNioChannel时：执行9）

8)、入参java.nio.channels.SelectionKey k， io.netty.channel.nio.AbstractNioChannel ch

8.1)、k是有效的时：

当前NioEventLoop.this==ch.eventLoop（即当前NioEventLoop实例是否与AbstractNioChannel 注册的eventLoop是否相等）：

关闭。

8.2）、k.readyOps()获取就绪操作集readyOps。

8.3）、就绪操作集是已连接或等待时(readyOps & SelectionKey.OP\_CONNECT !=0)：

执行java.nio.channels.SocketChannel.finishConnect()方法 ；

执行3.7.5

执行其它关闭方法（3.3.8讲过一部分）

8.4）、就绪操作集是可写时((readyOps & SelectionKey.OP\_WRITE) != 0)：

略

8.5）、(readyOps & (SelectionKey.OP\_READ | SelectionKey.OP\_ACCEPT)) != 0 || readyOps == 0)

读。

9)、488行 参考3.3.8 第5部分。

10)、490-498行

10.1）、 参考7）处

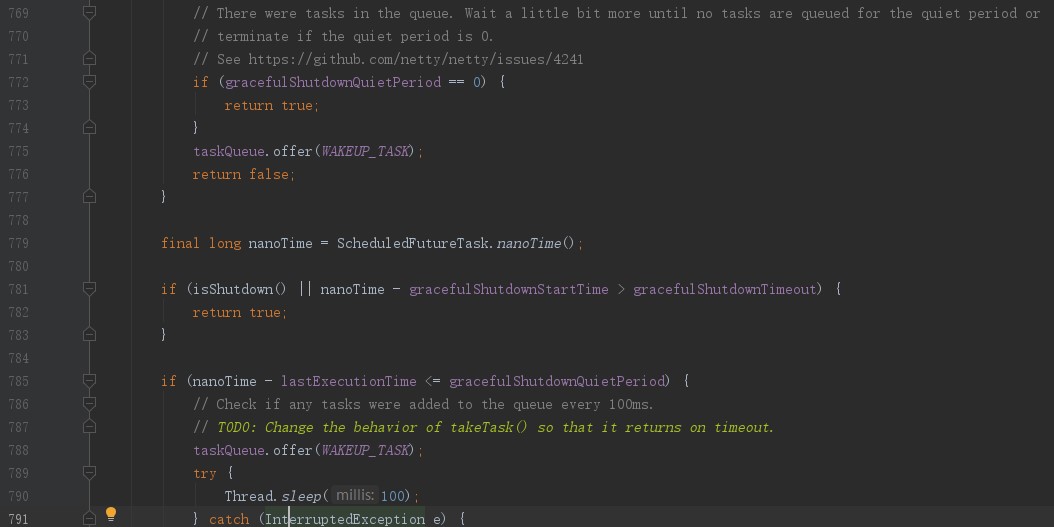
10.2）、runAllTasks方法 跑任务

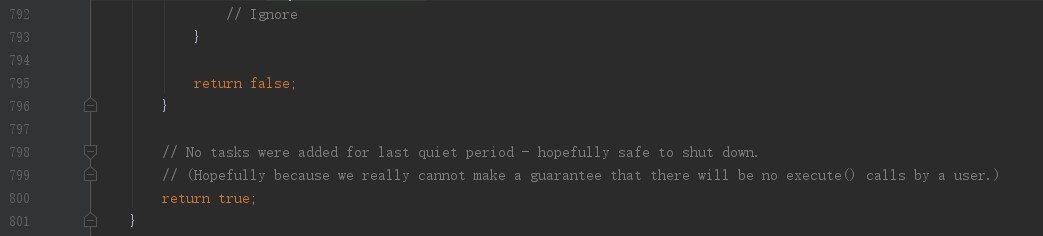
1. 、其它情况执行 10.2)
2. 、526-528 SingleThreadEventExecutor.state >= SingleThreadEventExecutor.ST\_SHTTING\_DOWN :
3. )关掉相关通道
4. )、参考3.3.8）

### 3.3.8SingleThreadEventExecutor.confirmShutdown

图3-3-8-1







1. 、749-751 确认SingleThreadEventExecutor.state是否关闭状态如果否返回false，否则

继续执行。

1. 、753-755 判断当前线程==SingleThreadEventExecutor.thread 如果是返回true继续执行，

否则异常抛出

1. 、757cancelScheduledTask()。AbstractScheduledEventExecutor.scheduledTaskQueue队列中

所有ScheduledFutrueTask.cancelWithRemove方法执行。

1. 、759-761 如果SingleThreadEventExecutor.gracefulShutdownStartTime==0，更新该成员变

量。

1. 、763 runAllTasks()方法执行。

5.1）、把AbstractScheduledEventExecutor.scheduledTaskQueue队列中的未超过截止日期任务拷贝到SingleThreadEventExecutor.taskQueue队列中，并把拷贝成功的任务从AbstractScheduledEventExecutor.scheduledTaskQueue中删除。

5.2）、把SingleThreadEventExecutor.taskQueue队列中的任务，全部执行完。只要有一个

任务执行完，就会更新SingleThreadEventExecutor.lastExecutionTime时间，runAllTasks方法并返回true，其它情况返回false。

6）、763 runAllTasks()如果返回false，即SingleThreadEventExecutor.taskQueue队列中的任务没有执行成功。

runShutdownHooks()方法执行

6.1）、SingleThreadEventExecutor.shutdownHooks list中的钩子任务全部被执行。

6.2）、SingleThreadEventExecutor.shutdownHooks任务全部干掉。

7）、763-767 如果SingleThreadEventExecutor.state 是ST\_SHUTDOWN、ST\_TERMINATED

直接返回true，否则继续执行。

1. 、772-774 SingleThreadEventExecutor.gracefulShutdownQuitePeriod==0 返回true
2. 775-776 把空任务加入到SingleThreadEventExecutor.taskQueue队列中。
3. 781-782 如果 SingleThreadEventExecutor.state 大于等于4 或

当前时间戳差-优雅关闭开始时间 > 优雅关闭最大超时时间

返回true

1. 、785-790 (当前时间戳差-最后修改时间) 小于 优雅关闭时间 条件成立下：

把唤醒任务放到SingleThreadEventExecutor.taskQueue队列中去。

否则返回true

## 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

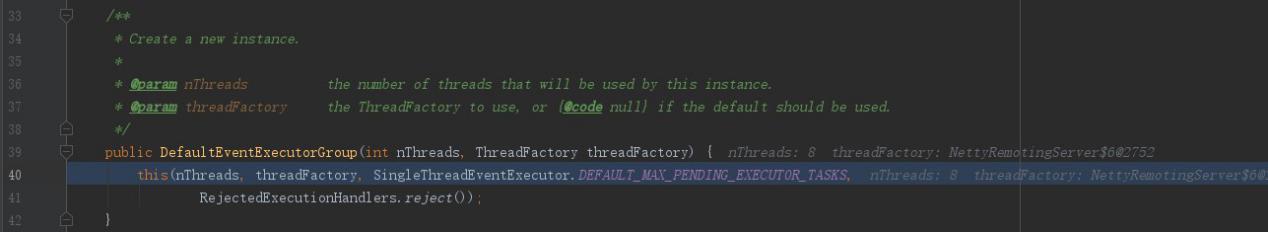
#### 3.4.2.1 构造方法1

图3-4-2-1-1



#### 3.4.2.2 构造方法2

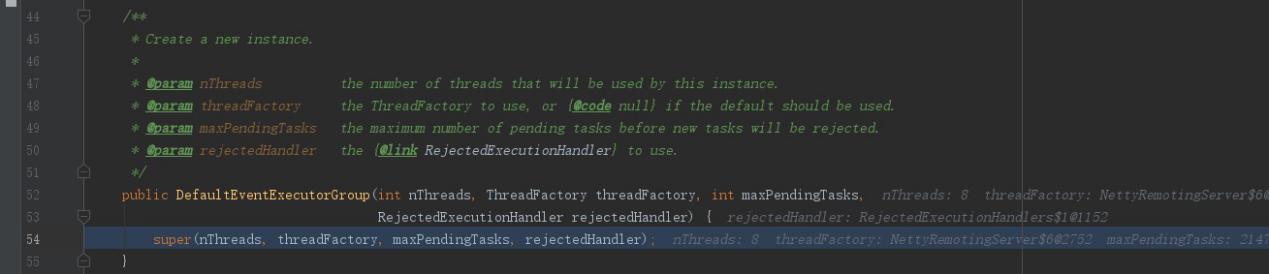
图3-4-2-2-1



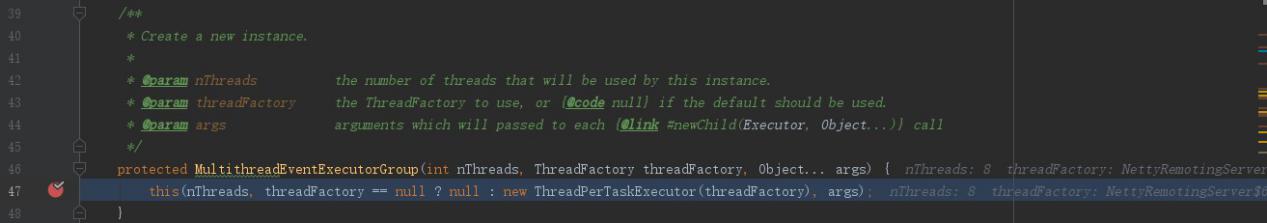
1. 、nThreads线程数
2. threadFactory线程工厂
3. SingleThreadEventExecutor.DEFAULT\_MAX\_PENDING\_EXECUTOR\_TASKS 默认最大挂起任务数量
4. 、参考3.2.2.5 构造方法5

#### 3.4.2.3 构造方法3

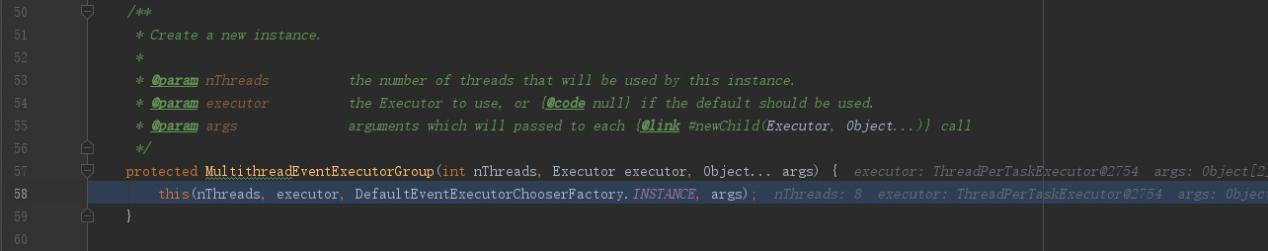
图3-4-2-3-1



#### 3.4.2.4 构造方法4



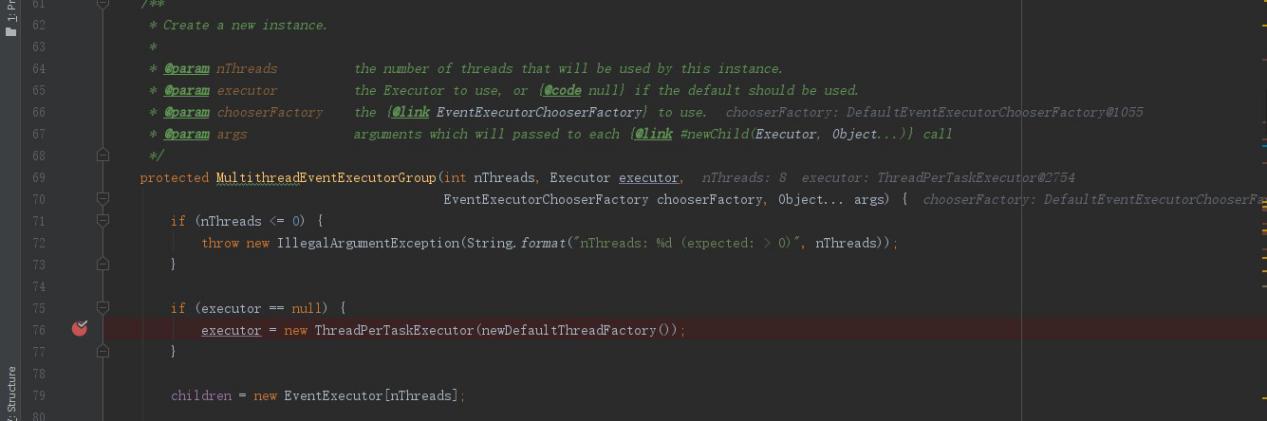
#### 3.4.2.5 构造方法5

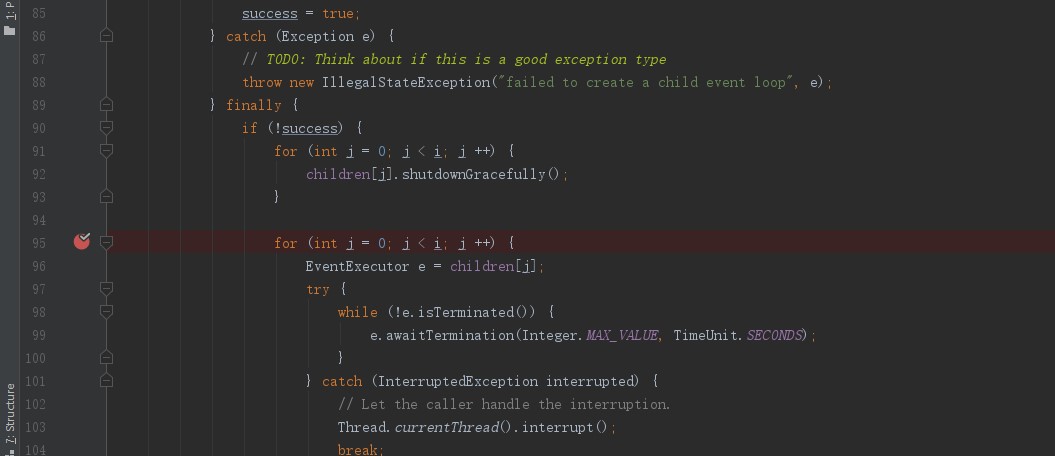


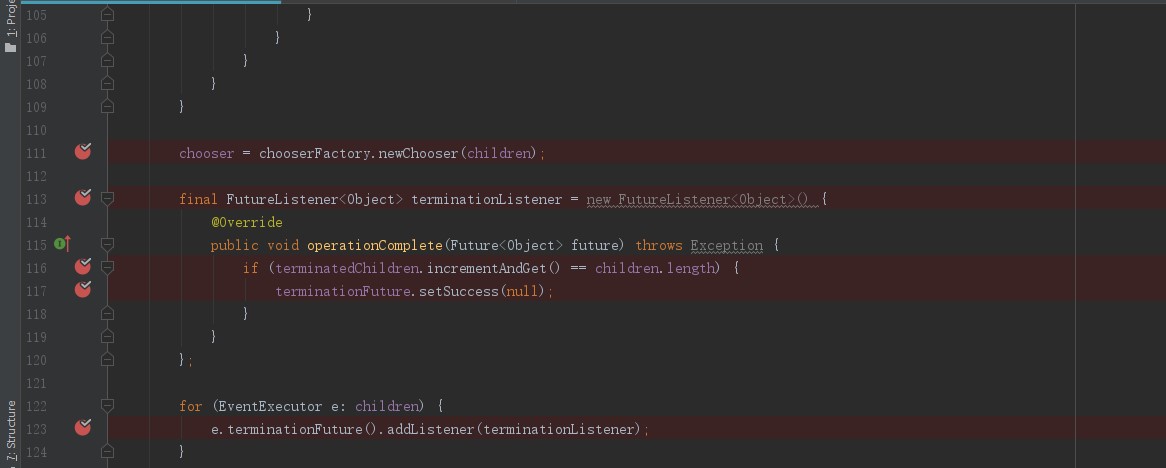
1）

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









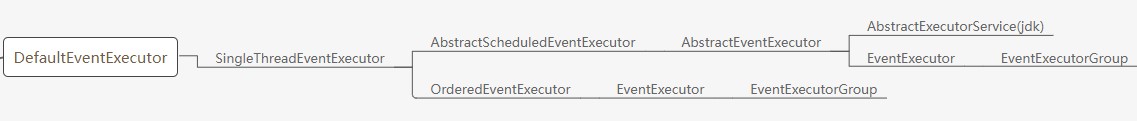
参考3.2

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

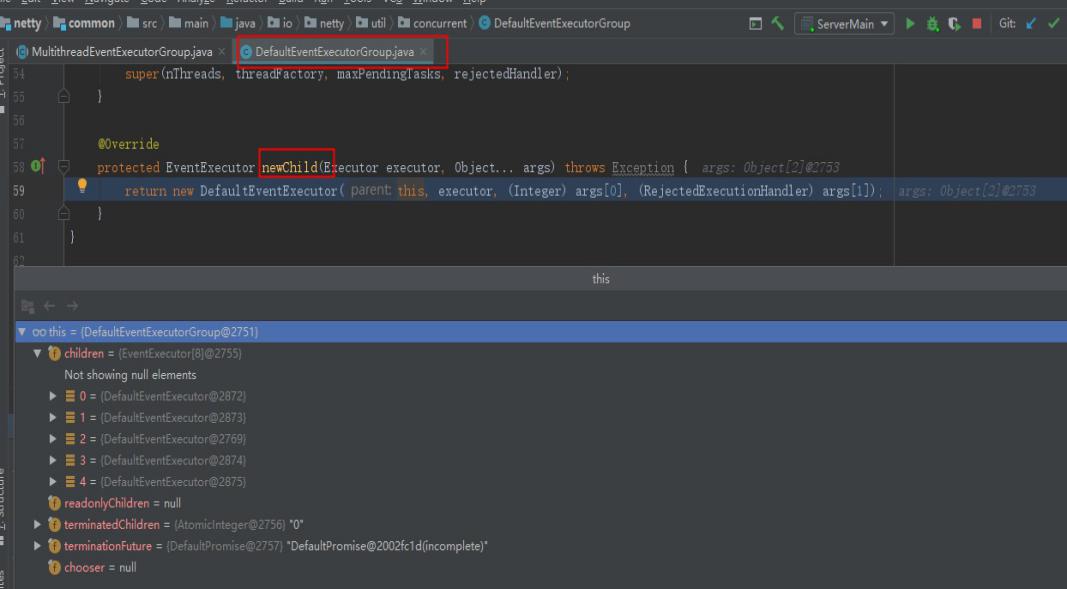
图3-5-1-1



### 3.5.2 构造方法

#### 3.5.2.1 构造方法1

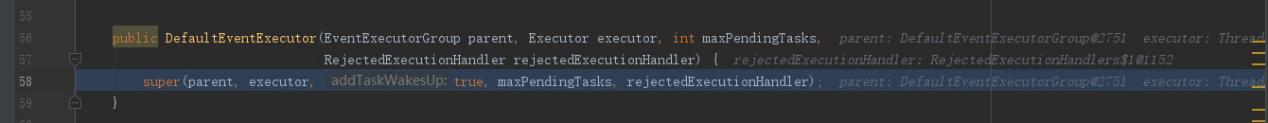
图3-5-2-1-1



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

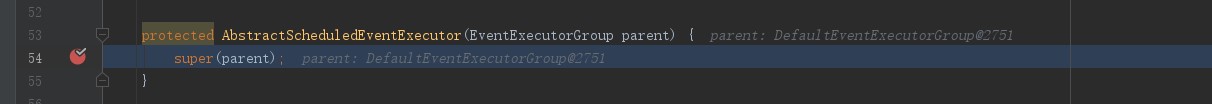
图3-5-2-3-1



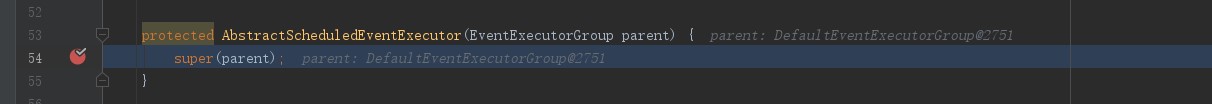
1. 执行父构造方法
2. this.addTaskWakesUp 赋值为true
3. 其它参考（3.3.2.4 父类构造方法）

#### 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

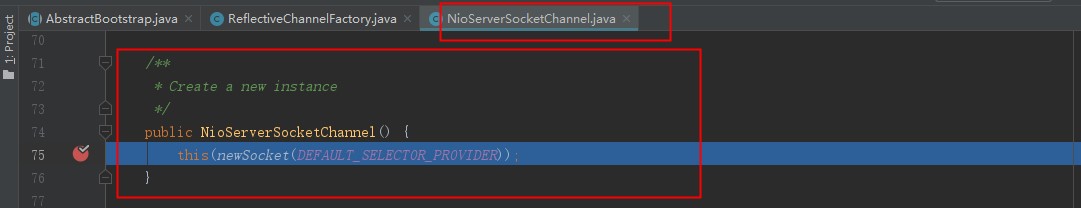
### 3.6.1继承关系

### 3.6.2 NioServerSocketChannel构造方法



#### 3.6.2.1 构造方法1

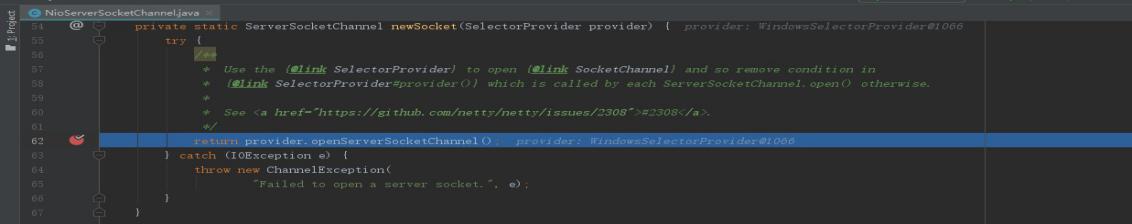
如图3-6-2-1-1



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

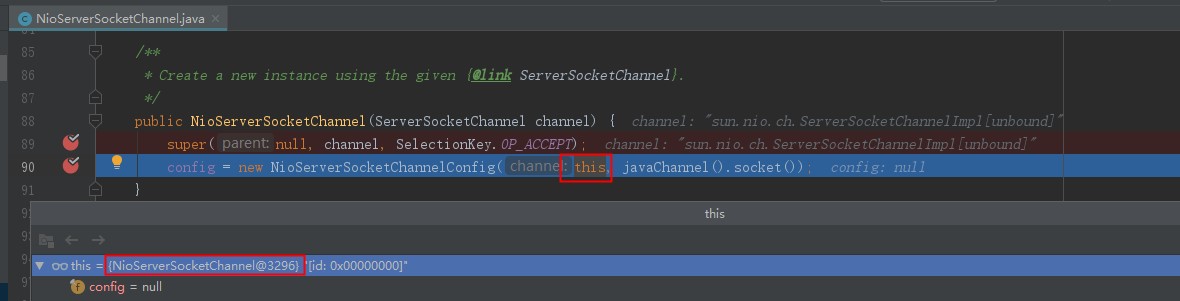
就是通过WindowsSelectorProvider.openServerSocketChannel()方法创建一个ServerSocketChannel对象

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



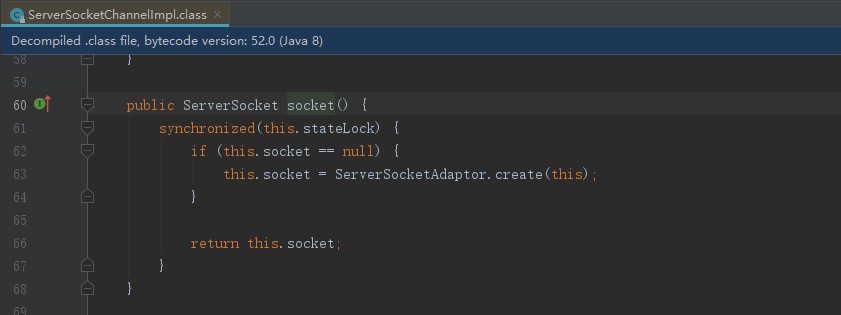
图3-6-2-2-3



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

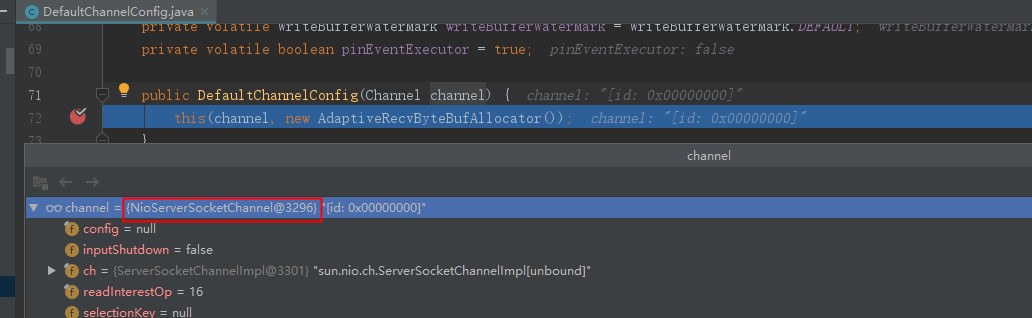


图3-6-2-2-9



图3-6-2-2-10

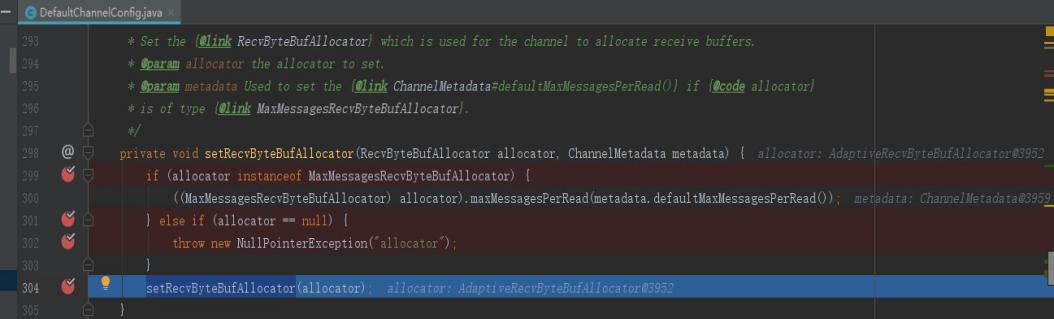
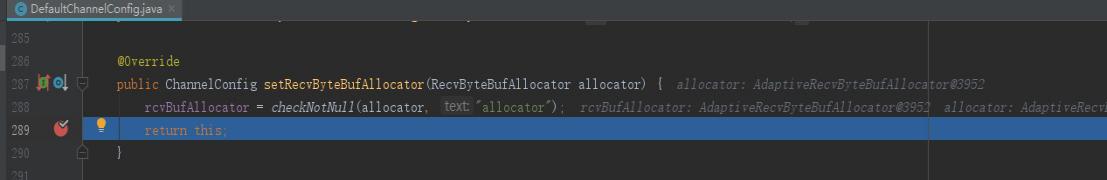
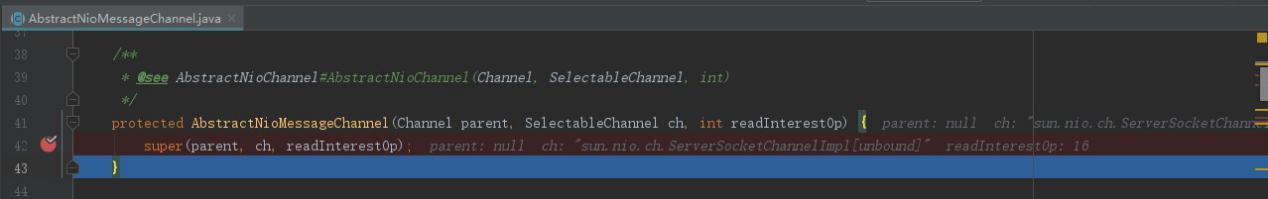


图3-6-2-2-11



#### 3.6.2.3父构造方法

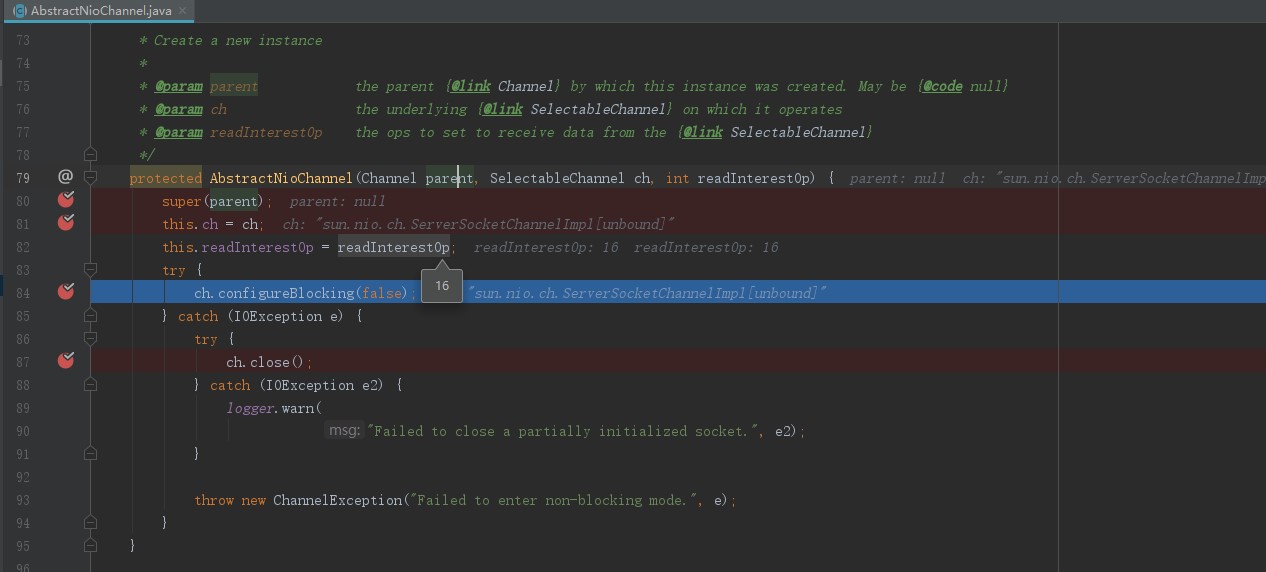
图3-6-2-3-1



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

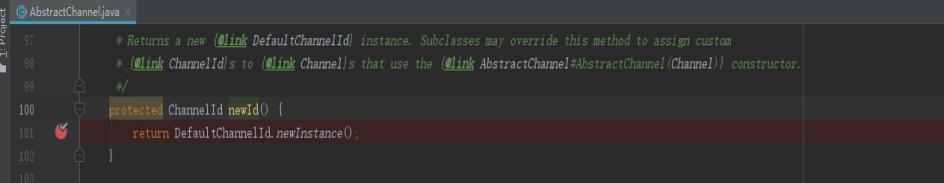


图3-6-2-5-2

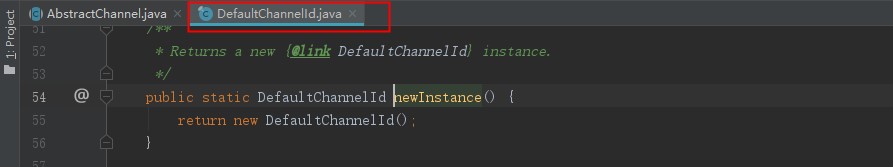
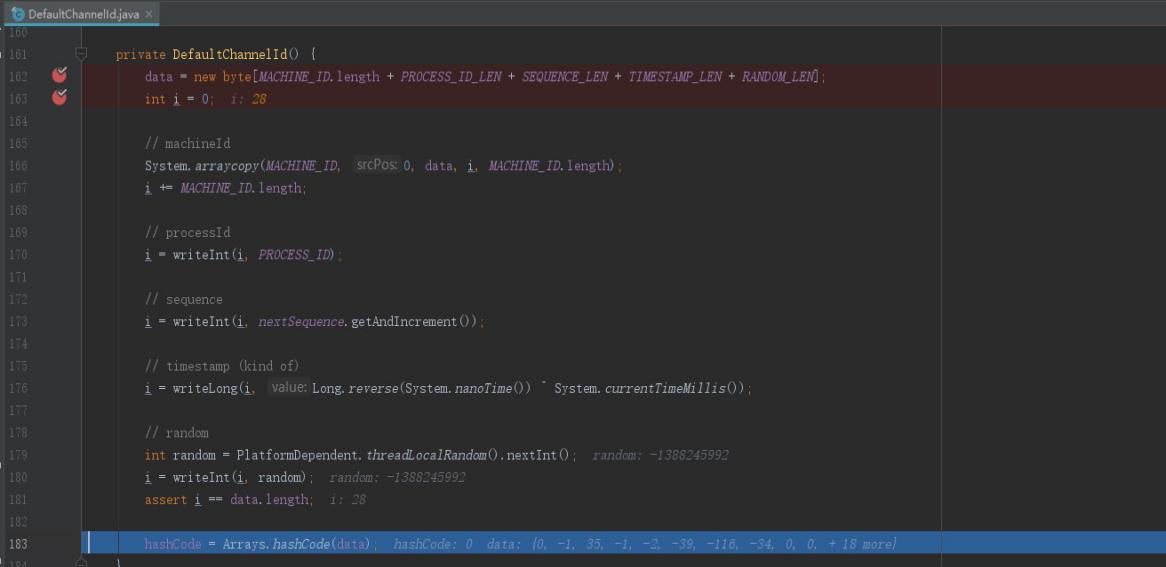


图3-6-2-5-3



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

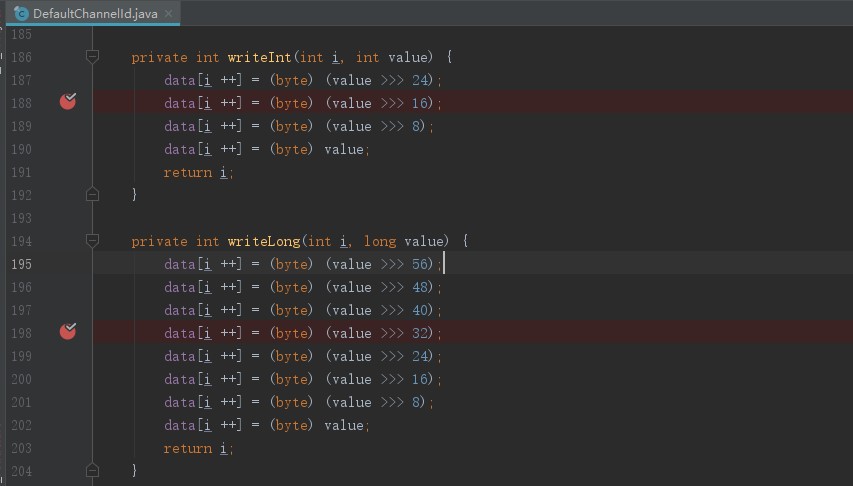
4个字节：进程号

4个字节：序列号（递增）

8个字节：时间戳

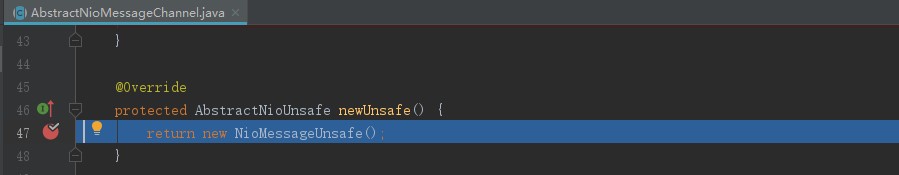
4个字节：随机码

图 3-6-2-5-4



3）、unsafe 是通过newUnsafe()方法初始化，

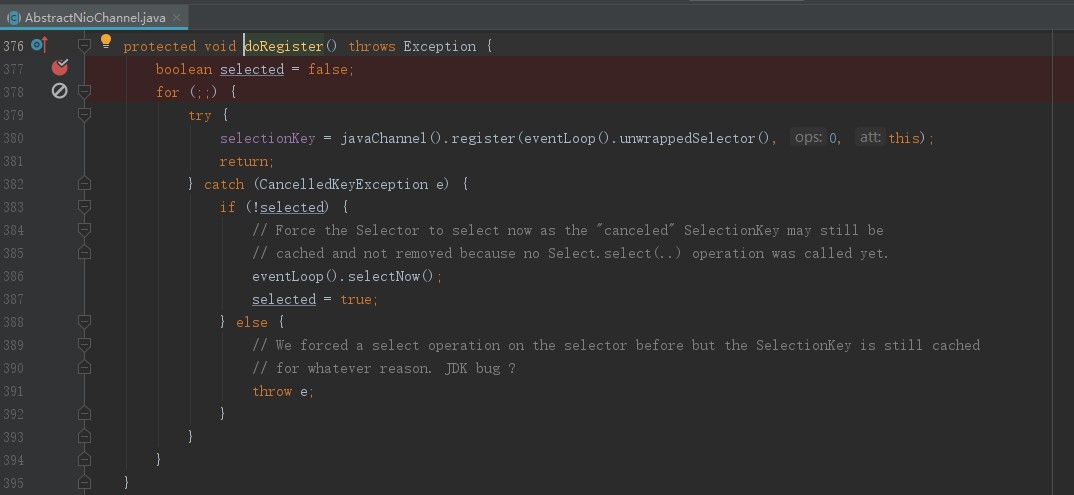
图3-6-2-5-5



1. 、通过newChannelPipeline()方法初始化，参考3.7

### 3.6.3 doRegister()方法

图3-6-3-1



1）380行 javaChannel()返回ServerSocketChannel实例；eventLoop()返回NioEventLoop实例，

eventLoop().unwrappedSelector()方法返回NioEventLoop.unwrappedSelector成员变量(在NioEventLoop构造方法时初始化的参考3.3相关设计）。

javaChannel().register(eventLoop().unwrappedSelector(), 0, this)：就是向ServerSocketChannel中注册selector。为什么Selector初始化放在NioEventLoop初始化中？注册放在Channel初始化中？

2)、

## 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

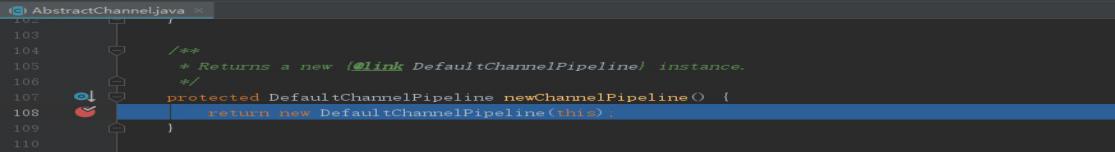
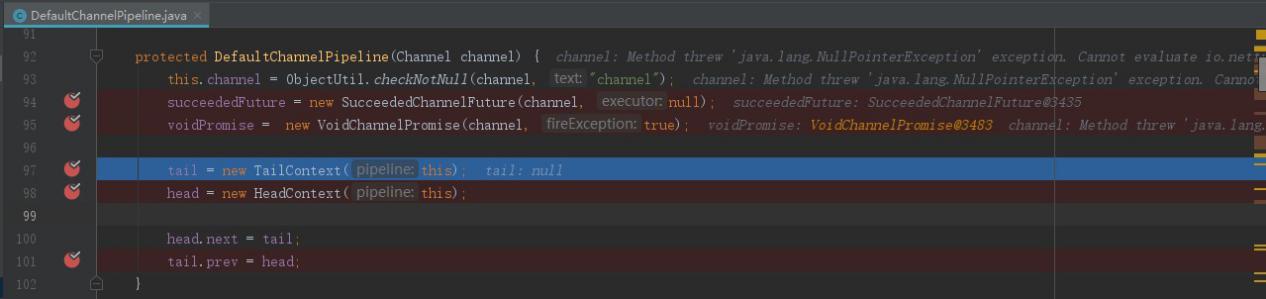


图3-7-2-1-2



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

图3-7-2-1-3

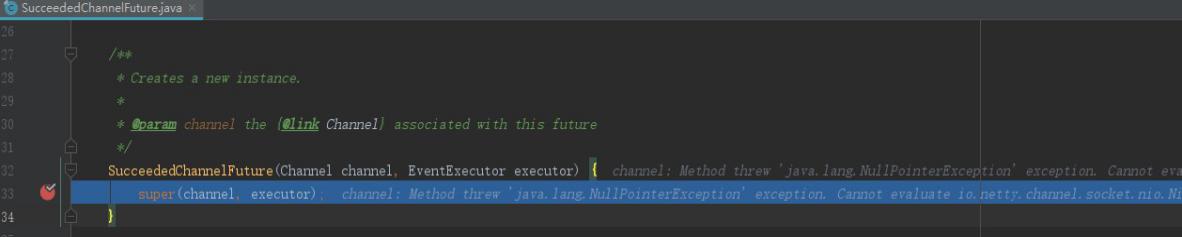


图3-7-2-1-3 中channel为NioServerSocketChannel 实例；executor为null。

图3-7-2-1-4

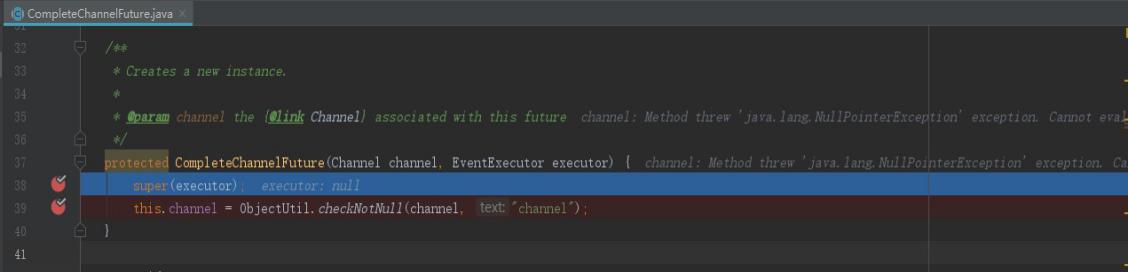
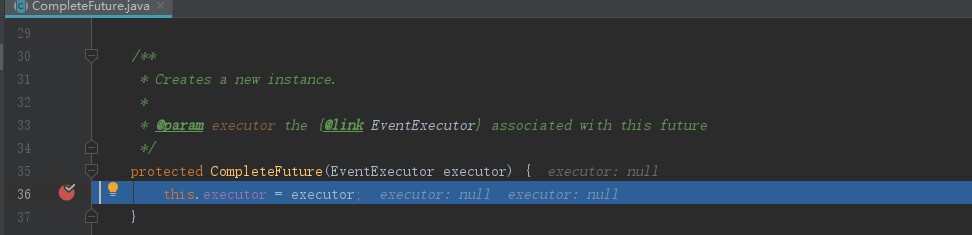


图3-7-2-1-5

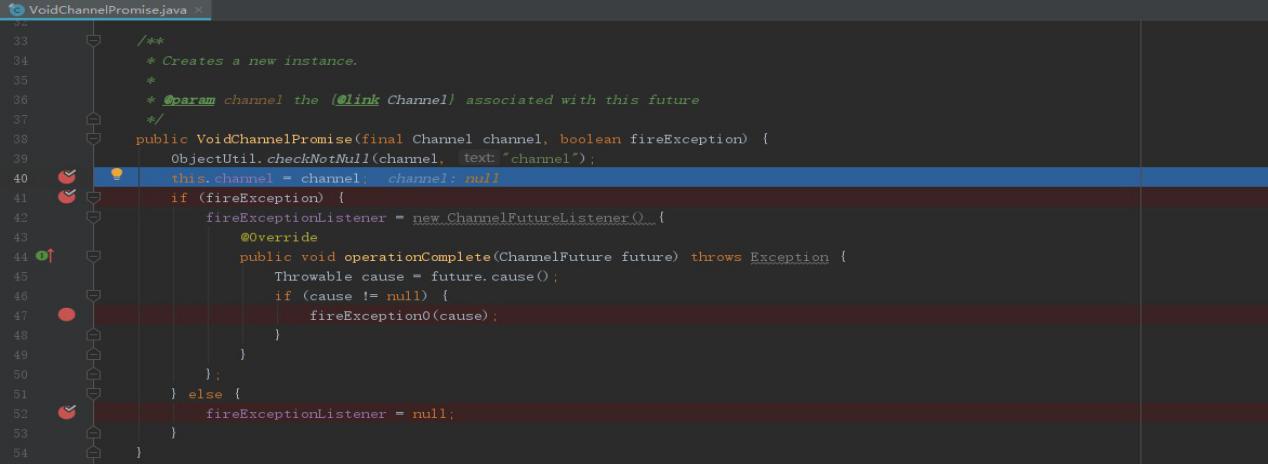


图图3-7-2-1-3、图3-7-2-1-4、图3-7-2-1-5 所示相关继承关系如下：

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

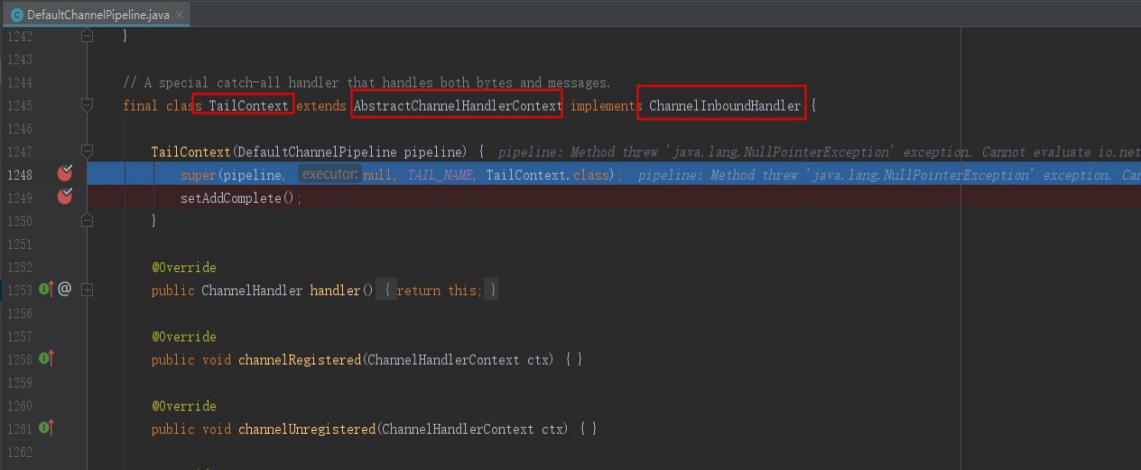
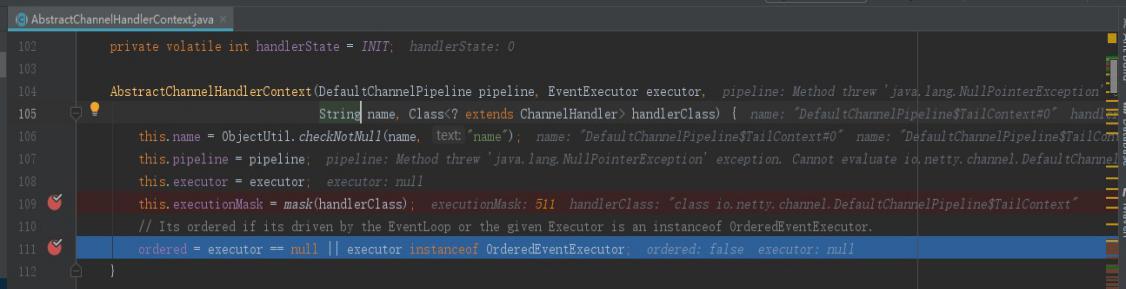


图3-7-2-1-8



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

name为”DefaultChannelPipeline$TailContext#0”；

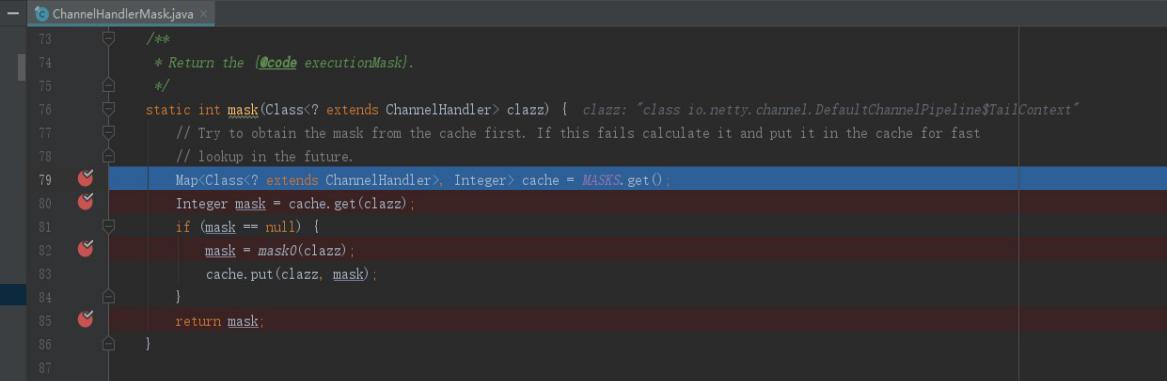
pipleline 为DefaultChannelPipeline实例；

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

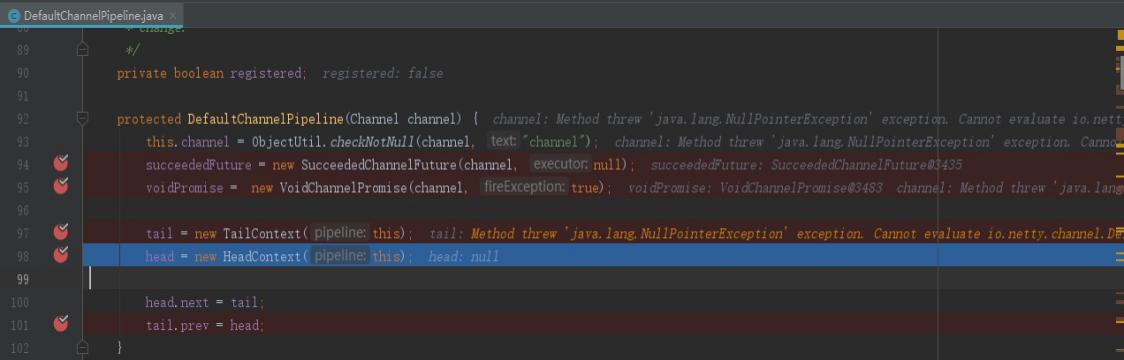
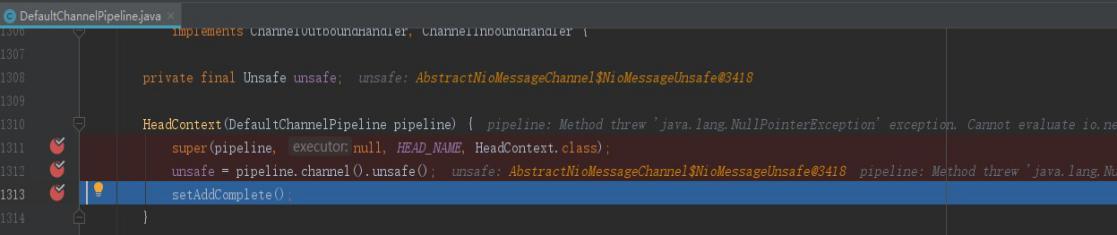


图3-7-2-1-11



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

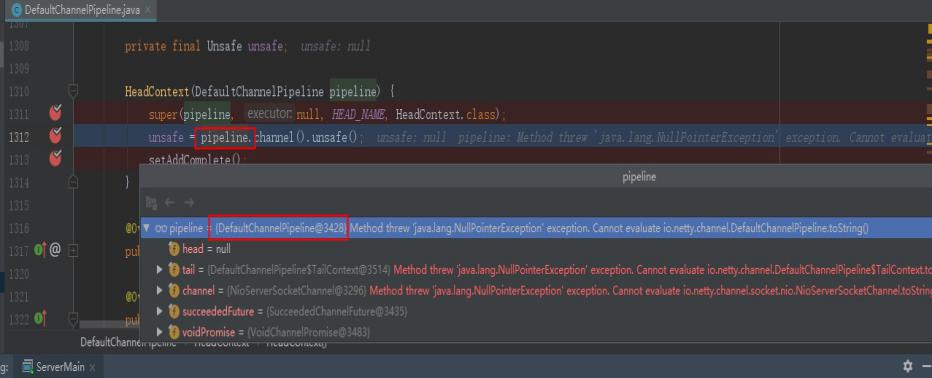


图3-7-2-1-13

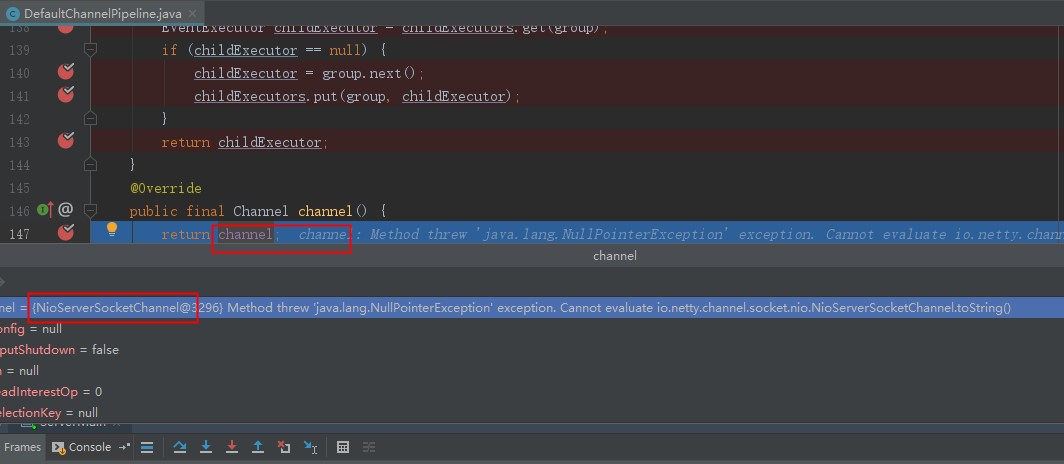


图3-7-2-1-14

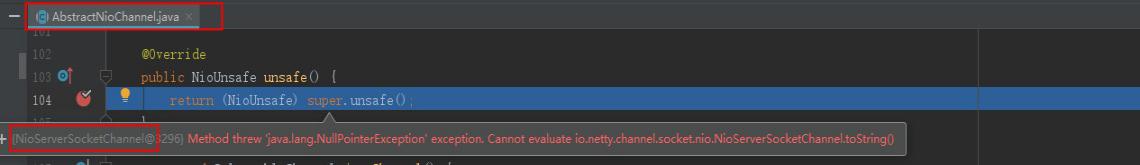


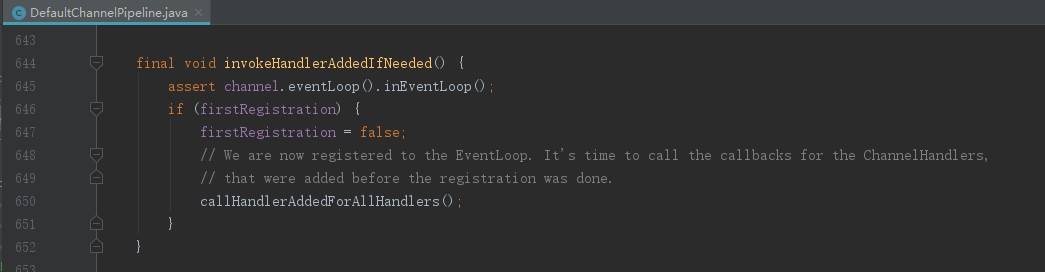
图3-7-2-1-15



1. 、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？

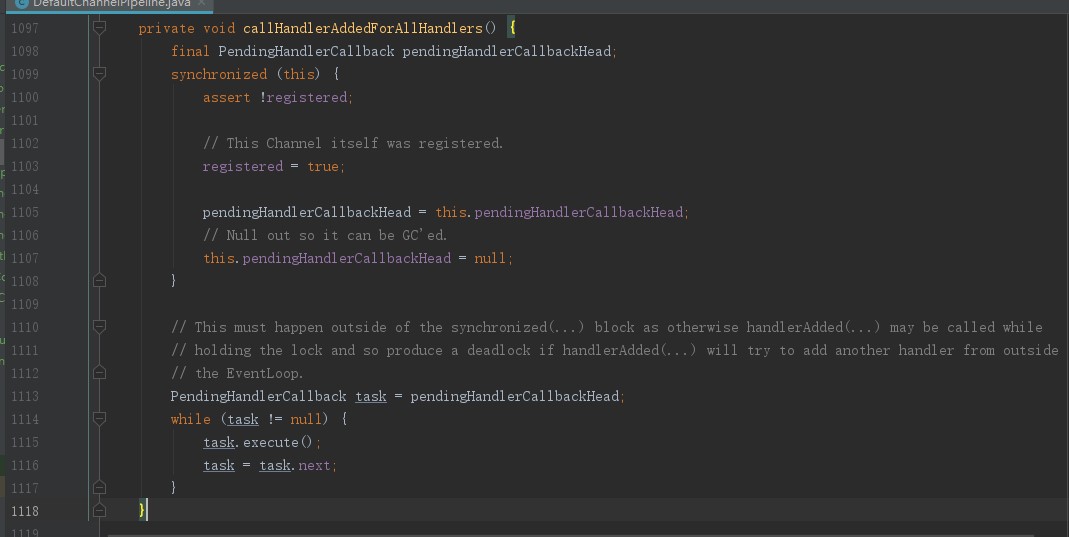
### 3.7.3 invokeHandlerAddedIfNeeded方法

图3-7-3-1



1. 、646行firstRegistration默认为true
2. 、647行firstRegistration=false
3. 、650执行callHandlerAddedForAllHandlers()，如图3-7-3-2

图3-7-3-2

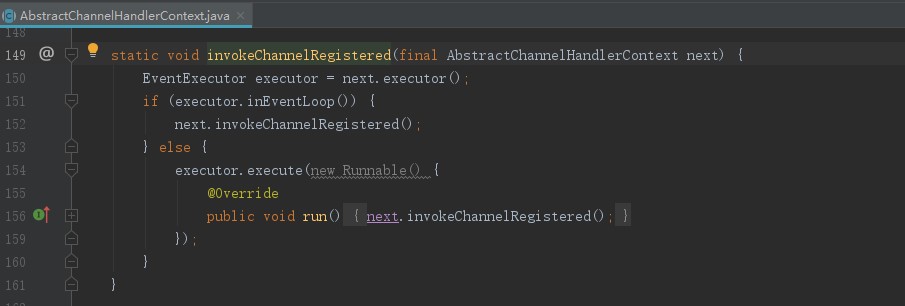


### 3.7.4fireChannelRegistered()

图3-7-4-1



图3-7-4-2



1. next.executor()方法会返回什么呢？

通过图3-7-4-1、3-7-4-2所示，next为DefaultChannelPipeline.head对象，

通过3.7.1可知图150行 executor为AbstractChannel.eventLoop成员变量。

2）、152行执行next.invokeChannelRegistered()方法

图3-7-4-3



图 3-7-4-4

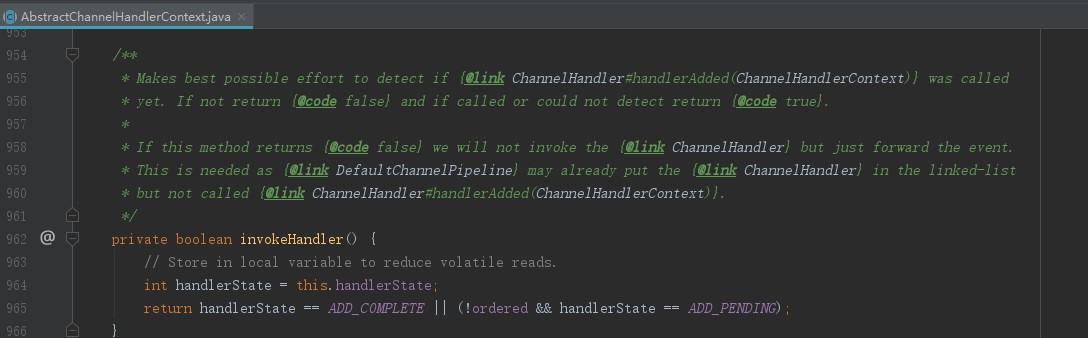


图3-7-4-5

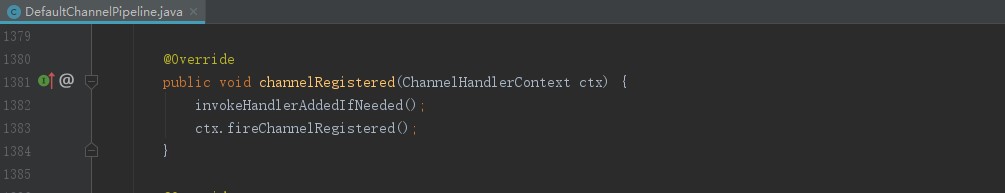


图3-7-4-4 handler()返回HeadContext对象，执行图3-7-4-5

### 3.7.5 fireChannelActive

图3-7-5-1



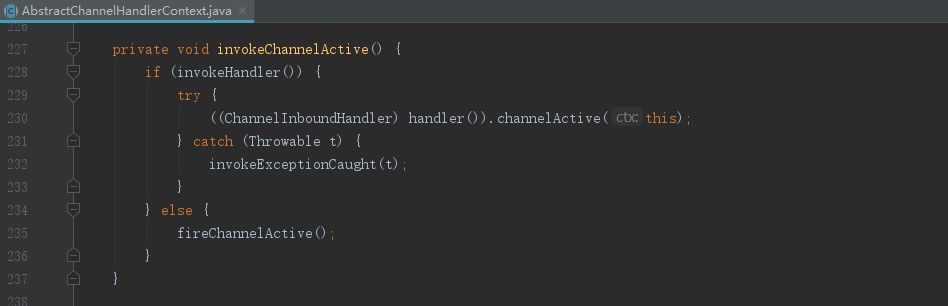
图3-7-5-2



1. 图3-7-5-2 214行返回AbstractChannel.eventLoop对象（结束3.7.1）
2. 图3-7-5-2 216行next.invokeChannelActive()，方法体参考3-7-5-3所示

invokeHandler()、handler()方法可以参考3.7.4，

图3-7-5-3



channelActive()方法？

这块有问题，需要重点确认？

## 3.8DefaultChannelPromise

### 3.8.1继承关系

### 3.8.2构造方法

#### 3.8.2.1构造方法1

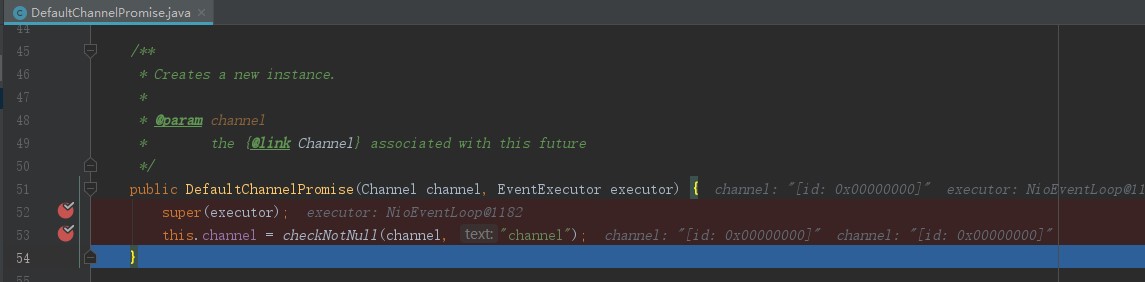
创建DefaultChannelPromise实例，如图3-3-3-1-1所示，channel为NioServerSocketChannel

对象，executor为NioEventLoop实例

把channel赋值给DefaultChannelPromise.channel实例

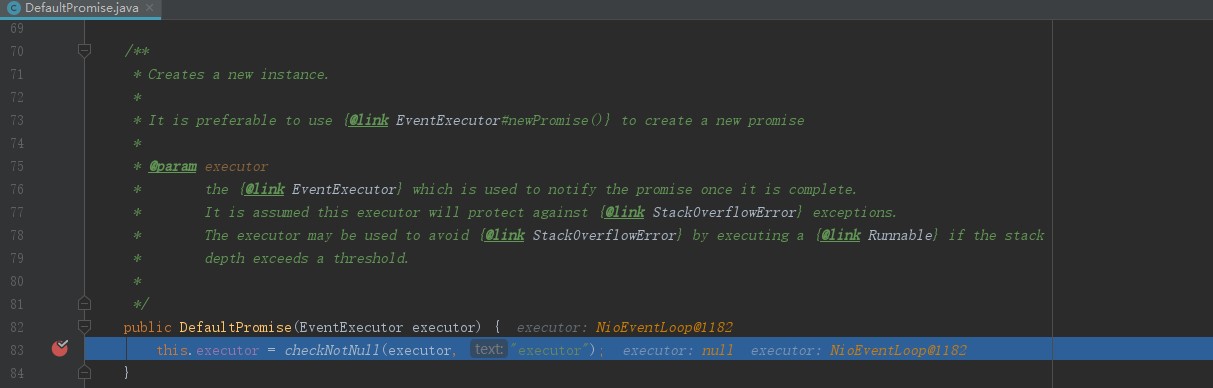
图3-3-3-1-2所示，把executor赋值给DefaultPromise.executor实例

图3-8-2-1-1



#### 3.8.2.2构造方法2

图3-8-2-2-1



### 3.8.3 trySuccess 方法

图3-8-3-1



图3-8-3-2

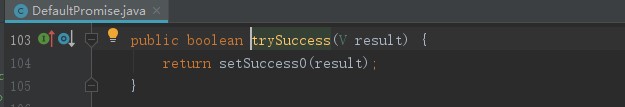


图3-8-3-3

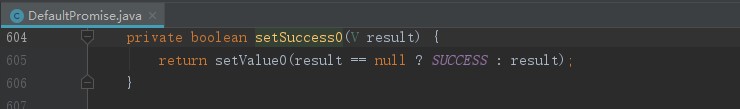
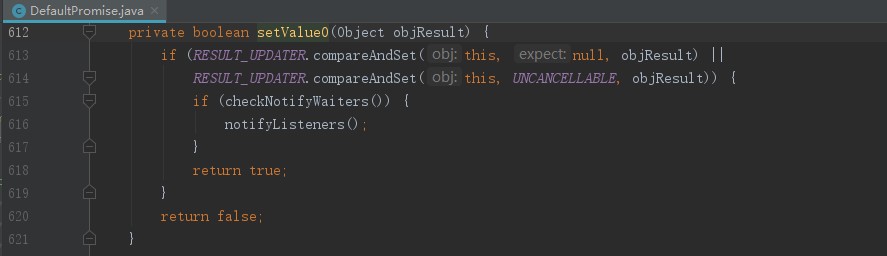


图3-8-3-4



### 3.8.4 trayFailure方法

图3-8-4-1

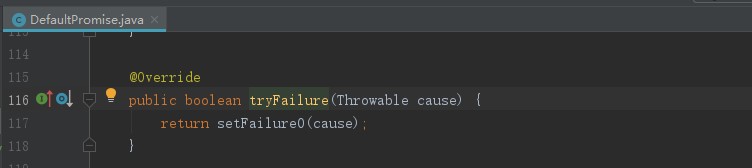
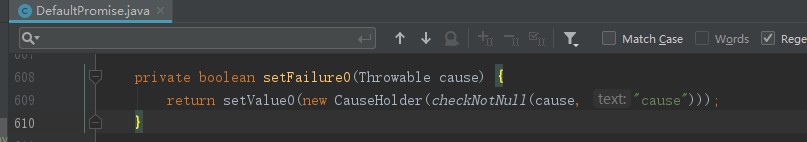


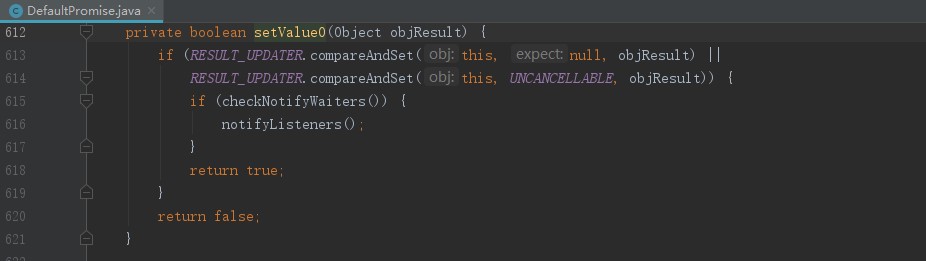
图3-8-4-2



1）setValue0参考3.8.5

### 3.8.5 setValue0

图3-8-5-1



1）、RESULT\_UPDATER =

AtomicReferenceFieldUpdater.newUpdater(DefaultPromise.class, Object.class, "result");

DefaultPromise.result 成员变量初始时为null,

613行compareAndSet方法执行的效果就是：如果DefaultPromise.result为null，把objResult

赋值给DefaultPromise.result并返回true，否则返回false

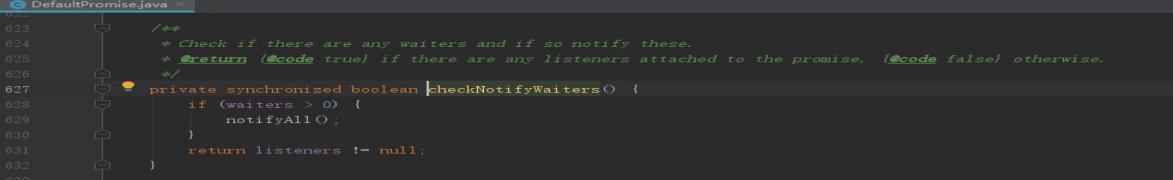
1. 、614行compareAndSet方法执行效果：如果DefaultPromise.result为UNCANCELLABLE

时返回true，并把objResult赋值给DefaultPromise.result。

1. 、615行执行checkNotifyWaiters()方法如3-8-5-2所示 listeners不为null即返回true，否

则返回false。如果DefaultPromise.waiters大于0，直接调用java.lang.Object.notifyAll()接口

图3-8-5-2



1. 、616行，执行notifyListeners()方法如3-8-6所示

### 3.8.6 notifyListeners

图3-8-6-1



1. 、484行，executor() 返回，通过图3-8-6-3所示和3.8.1可推测e为NioEventLoop

实例，即使e为null，也可以返回AbstractChannel.eventLoop成员变量(3.3.3.3)

图3-8-6-2

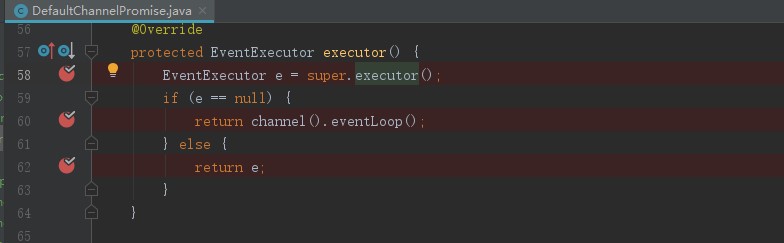


图3-8-6-3



1. 、485 判断当前线程是否是NioEventLoop中的线程，如果是继续执行
2. 、499执行safeExecute方法如图3-8-6-4所示，

图3-8-6-4

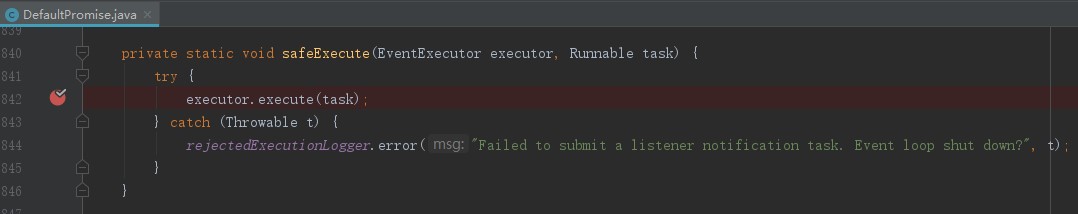


图3-8-6-5

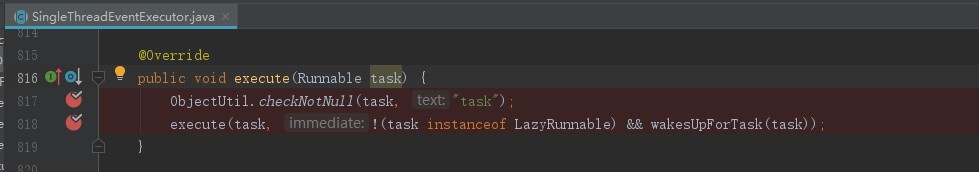
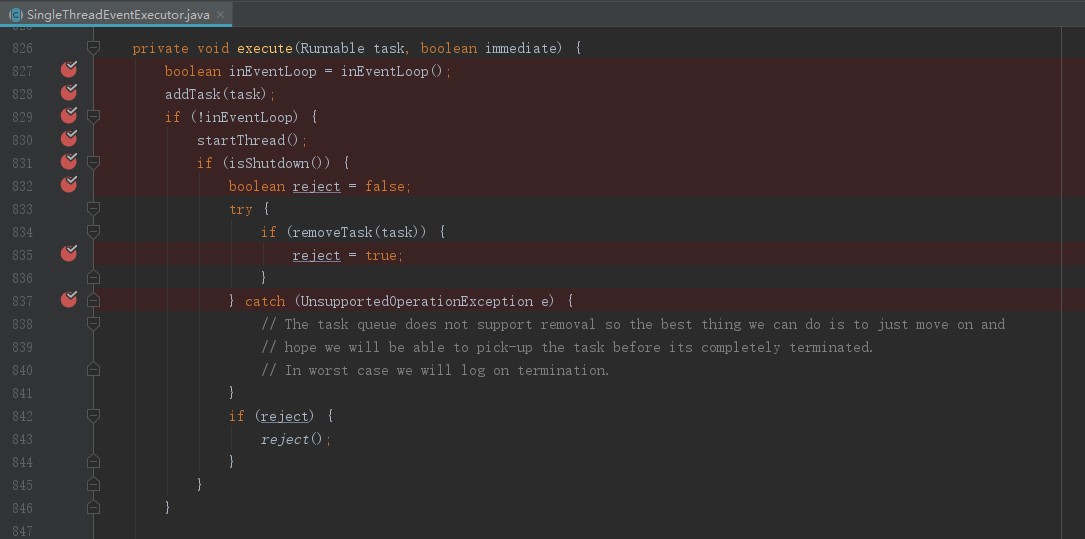


图3-8-6-6





### 3.8.7 DefaultPromise.setSuccess方法

图3-8-7-1

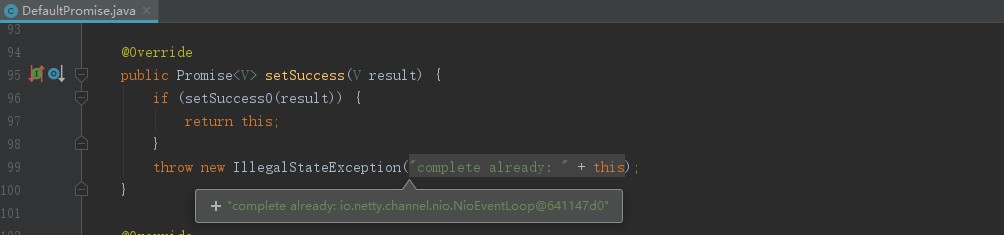
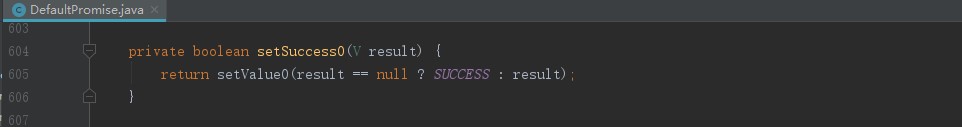


图3-8-7-2



1）setValue0参考3.8.5