Netty4 SEDA 事件驱动原理分析

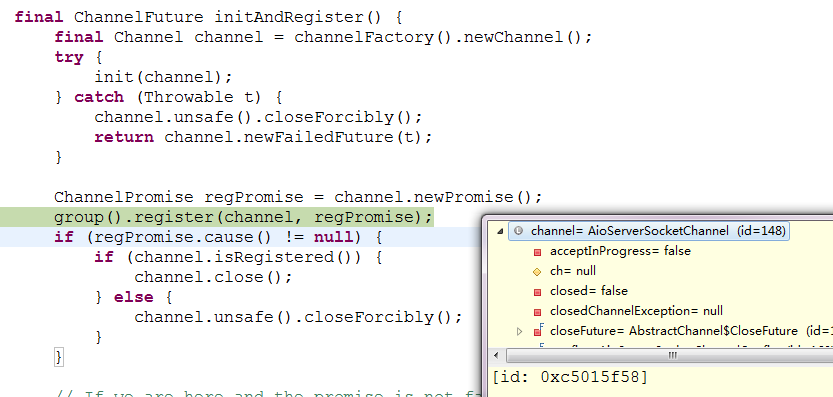
ServerBootstrapAcceptor

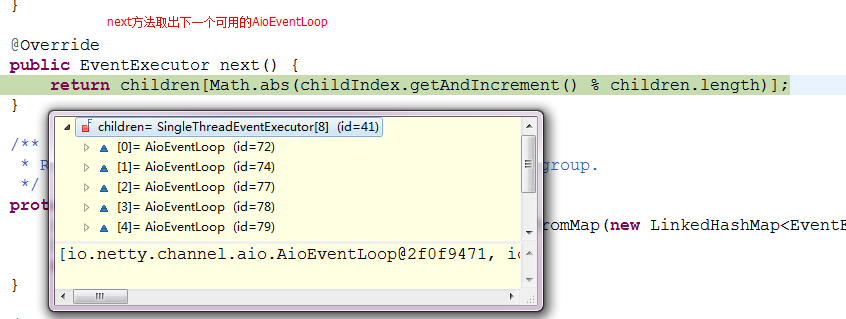
* AioServerSocketChannel的事件处理线程(AioEventLoop)建立过程分析

(Bootstrap启动ServerSocketChannel处理线程的过程分析)

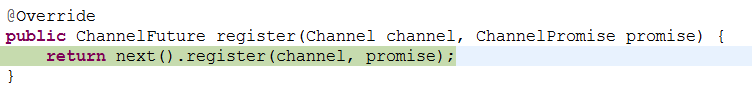
1:[AbstractBootstrap](eclipse-javadoc:%E2%98%82=%20/%3C%7BAbstractBootstrap.java%E2%98%83AbstractBootstrap).initAndRegister(): 向BossGroup(AioEventLoopGroup)

注册初始化后的AioServerSocketChannel,



2,调用[MultithreadEventExecutorGroup](eclipse-javadoc:%E2%98%82=%20/%3C%7BMultithreadEventExecutorGroup.java%E2%98%83MultithreadEventExecutorGroup).next()方法,在BossGroup中取出一个可用的EventLoop

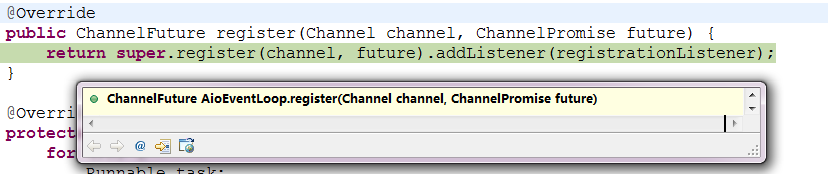
##### 3, 返回到[MultithreadEventLoopGroup](eclipse-javadoc:%E2%98%82=%20/%3C%7BMultithreadEventLoopGroup.java%E2%98%83MultithreadEventLoopGroup).register([Channel](eclipse-javadoc:%E2%98%82=%20/%3C%7BMultithreadEventLoopGroup.java%E2%98%83MultithreadEventLoopGroup~register~QChannel;~QChannelPromise;%E2%98%82Channel) channel, [ChannelPromise](eclipse-javadoc:%E2%98%82=%20/%3C%7BMultithreadEventLoopGroup.java%E2%98%83MultithreadEventLoopGroup~register~QChannel;~QChannelPromise;%E2%98%82ChannelPromise) promise)方法,将AioServerSocketChannel注册到(2)从BossGroup中取出的那个EventLoop.



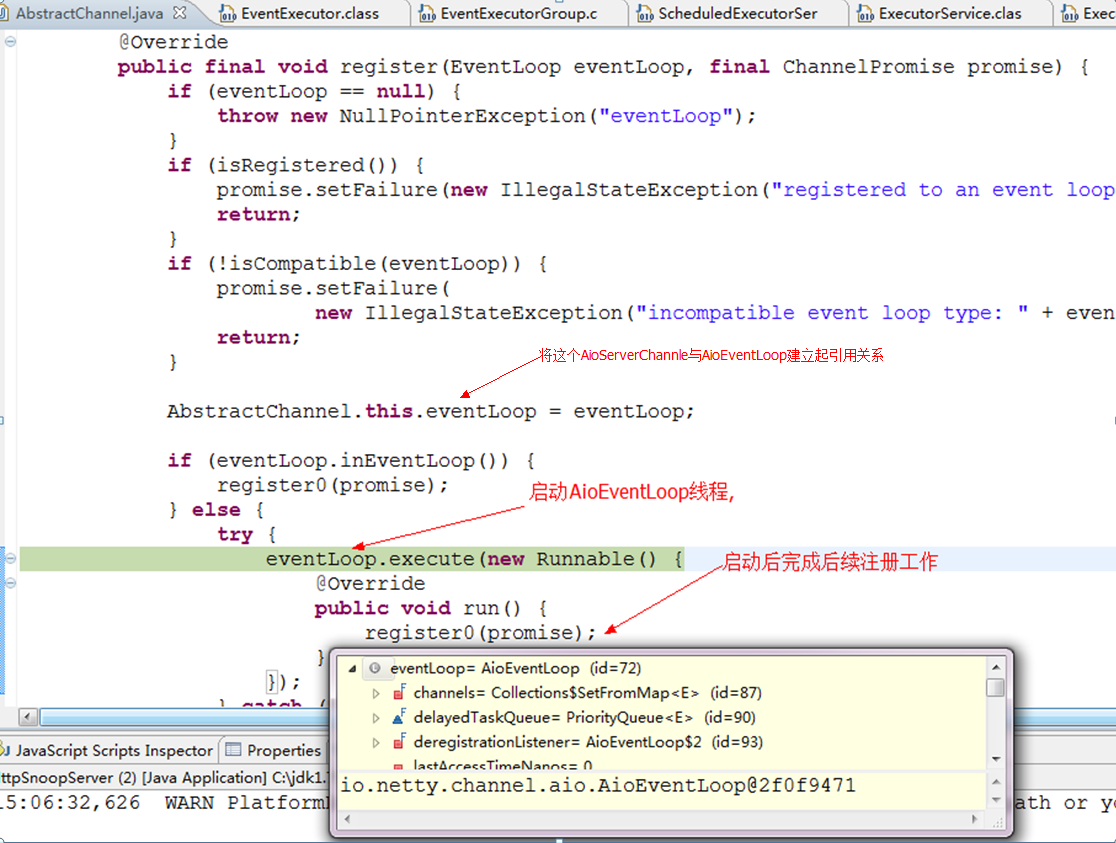
4,进入到[AioEventLoop](eclipse-javadoc:%E2%98%82=%20/%3C%7BAioEventLoop.java%E2%98%83AioEventLoop).register([Channel](eclipse-javadoc:%E2%98%82=%20/%3C%7BAioEventLoop.java%E2%98%83AioEventLoop~register~QChannel;~QChannelPromise;%E2%98%82Channel) channel, [ChannelPromise](eclipse-javadoc:%E2%98%82=%20/%3C%7BAioEventLoop.java%E2%98%83AioEventLoop~register~QChannel;~QChannelPromise;%E2%98%82ChannelPromise) future)方法

[AioEventLoop](eclipse-javadoc:%E2%98%82=%20/%3C%7BAioEventLoop.java%E2%98%83AioEventLoop)实现了JDK的ScheduledExecutorService,持有一条线程,具备任务执行能力,

将AioServerSocketChannel测试到[AioEventLoop](eclipse-javadoc:%E2%98%82=%20/%3C%7BAioEventLoop.java%E2%98%83AioEventLoop)后,由[AioEventLoop](eclipse-javadoc:%E2%98%82=%20/%3C%7BAioEventLoop.java%E2%98%83AioEventLoop)执行ServerSocketChannel的accept新的socket的操作

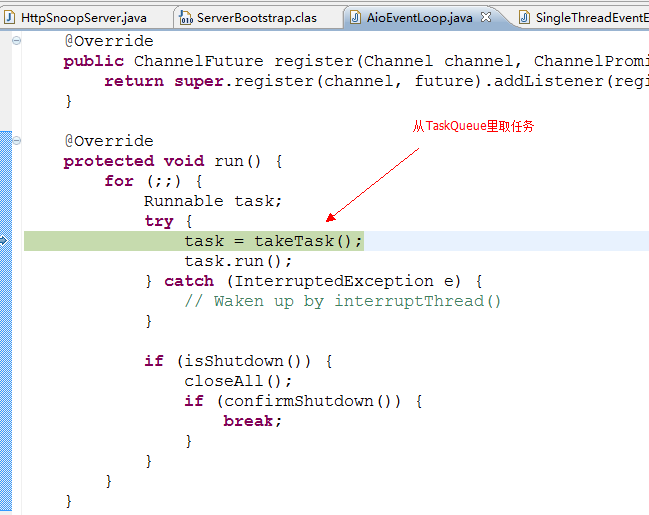


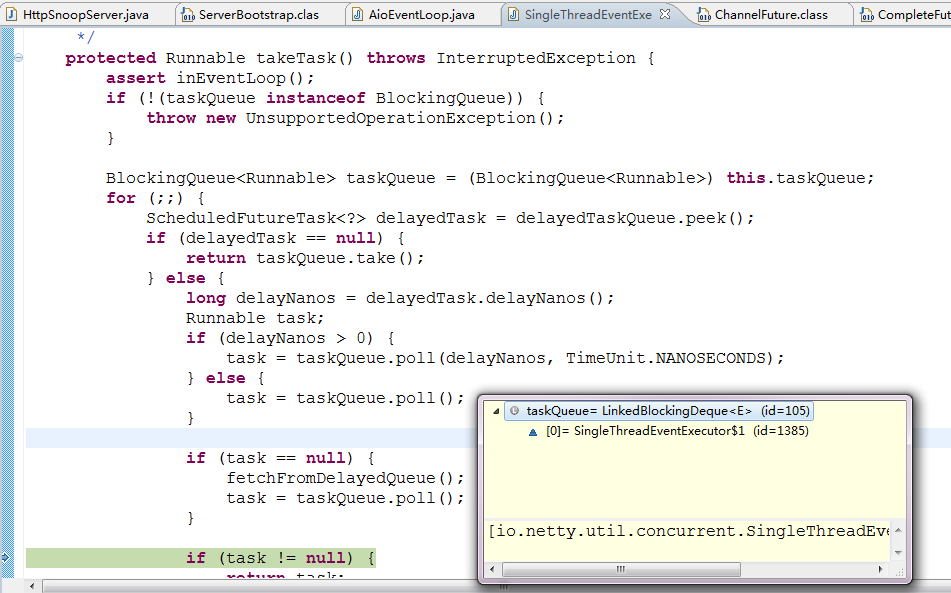
5,进入到[AbstractChannel](eclipse-javadoc:%E2%98%82=%20/%3C%7BAbstractChannel.java%E2%98%83AbstractChannel).[AbstractUnsafe](eclipse-javadoc:%E2%98%82=%20/%3C%7BAbstractChannel.java%E2%98%83AbstractChannel%E2%98%83AbstractUnsafe).register([EventLoop](eclipse-javadoc:%E2%98%82=%20/%3C%7BAbstractChannel.java%E2%98%83AbstractChannel%E2%98%83AbstractUnsafe~register~QEventLoop;~QChannelPromise;%E2%98%82EventLoop) eventLoop, [ChannelPromise](eclipse-javadoc:%E2%98%82=%20/%3C%7BAbstractChannel.java%E2%98%83AbstractChannel%E2%98%83AbstractUnsafe~register~QEventLoop;~QChannelPromise;%E2%98%82ChannelPromise) promise)方法,



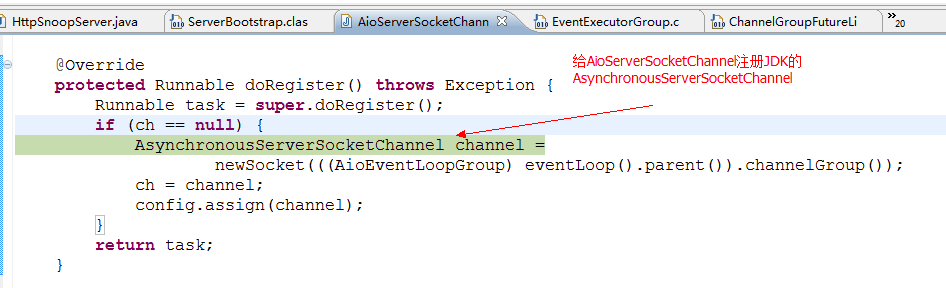
6)进入到[SingleThreadEventExecutor](eclipse-javadoc:%E2%98%82=%20/%3C%7BSingleThreadEventExecutor.java%E2%98%83SingleThreadEventExecutor).execute([Runnable](eclipse-javadoc:%E2%98%82=%20/%3C%7BSingleThreadEventExecutor.java%E2%98%83SingleThreadEventExecutor~execute~QRunnable;%E2%98%82Runnable) task)方法,执行startThread()后,AioServerSocketChannel的事件处理线程(AioEventLoop)启动,将由它来接管AioServerSocketChanel的事件处理(如Accept新的Socket),然后ServerBootstrap线程就进入等待状态,等待ServerSocketChannel绑定端口,然后等待ServerSocketChannel关闭.

7)先看下面





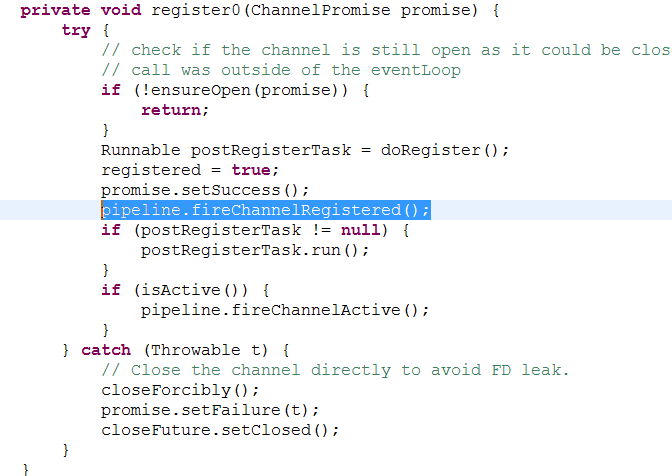
8)AioServerSocketChannel的事件处理线程AioEventLoop从一个BlockingDeque里取出新任务,取出的第一个任务是给AioServerSocketChannel注册JDK的AsynchronousServerSocketChannel(这才是真正和底层操作系统通信的ServerSocketChannel)



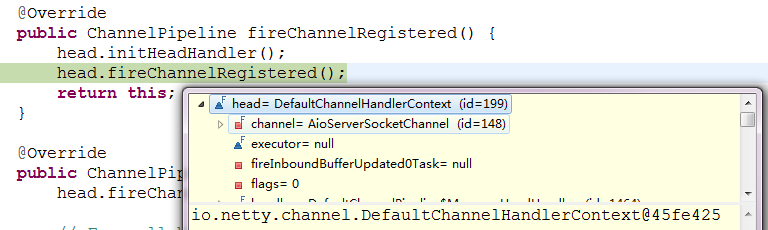
注册了:



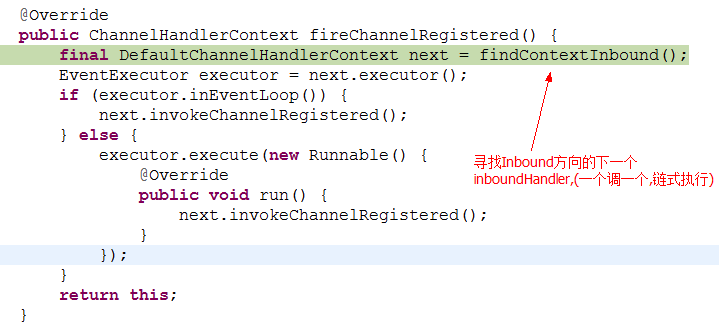
9)Channel成功注册后,调用其pipeline.fireChannelRegistered()方法Fire这个channel注册的事件

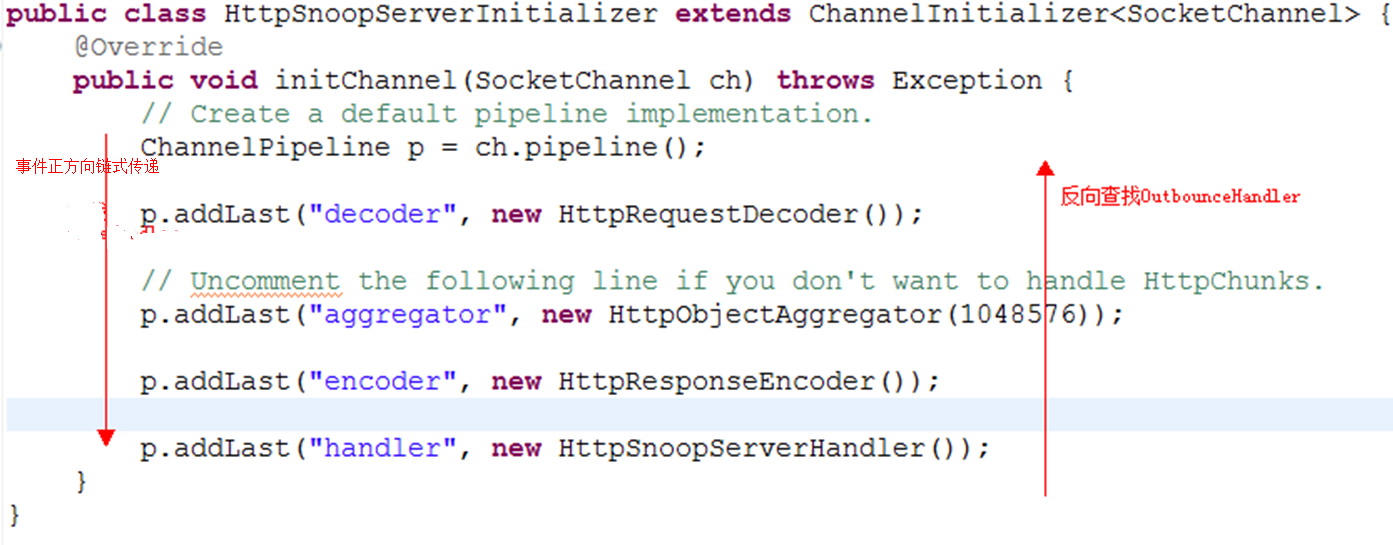


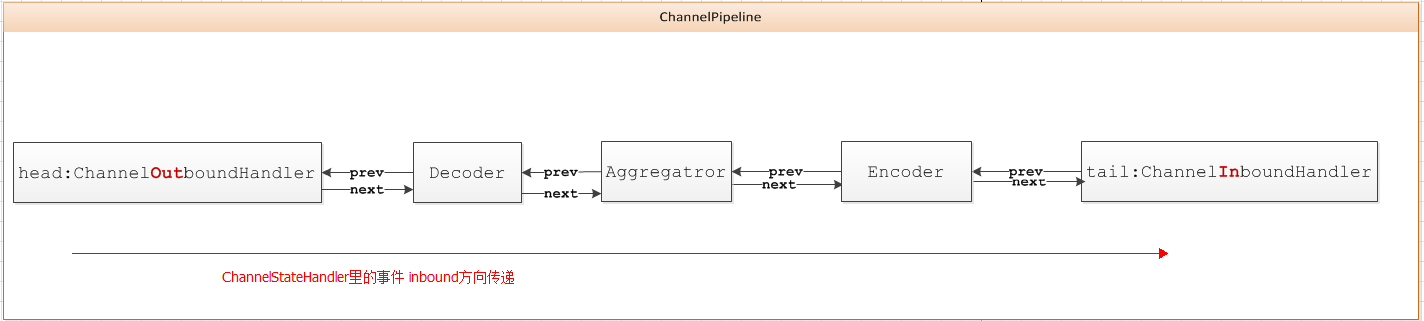
10)ChannelPipeline的fireChannelRegistered方法先初始化 其处理器链的第一个元素HeadHandler,然后再执行HeadHandler的fireChannelRegistered()方法



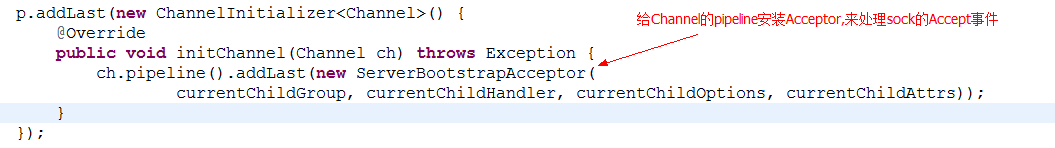
[DefaultChannelHandlerContext](eclipse-javadoc:%E2%98%82=%20/%3C%7BDefaultChannelHandlerContext.java%E2%98%83DefaultChannelHandlerContext).fireChannelRegistered(),Netty4 的DefaultChannelHandlerContext对ChannelInboundInvoker接口的的事件处理方法fireXxxx(),具有链式传递效应,顺序将同一方向上的所有事件



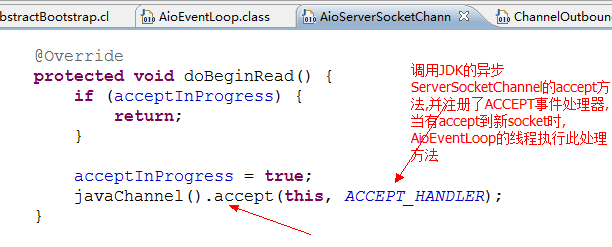




11)然后, AioServerSocketChanel的pipeline的处理器链上的ChannelInitializer响应channelRegistered事件: 给pipeline安装上Acceptor,使得AioServerSocketChanel得以具备处理socket Accept事件的能力.



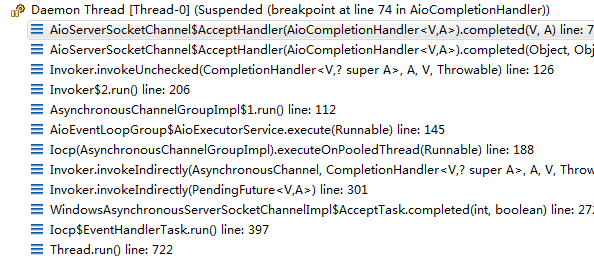
12) AioServerSocketChanel将accept委托给JDK的异步ServerSocketChannel来执行异步accept,并指定了事件处理器,当ACCEPT事件发生的时候, AcceptHandler就会被调用

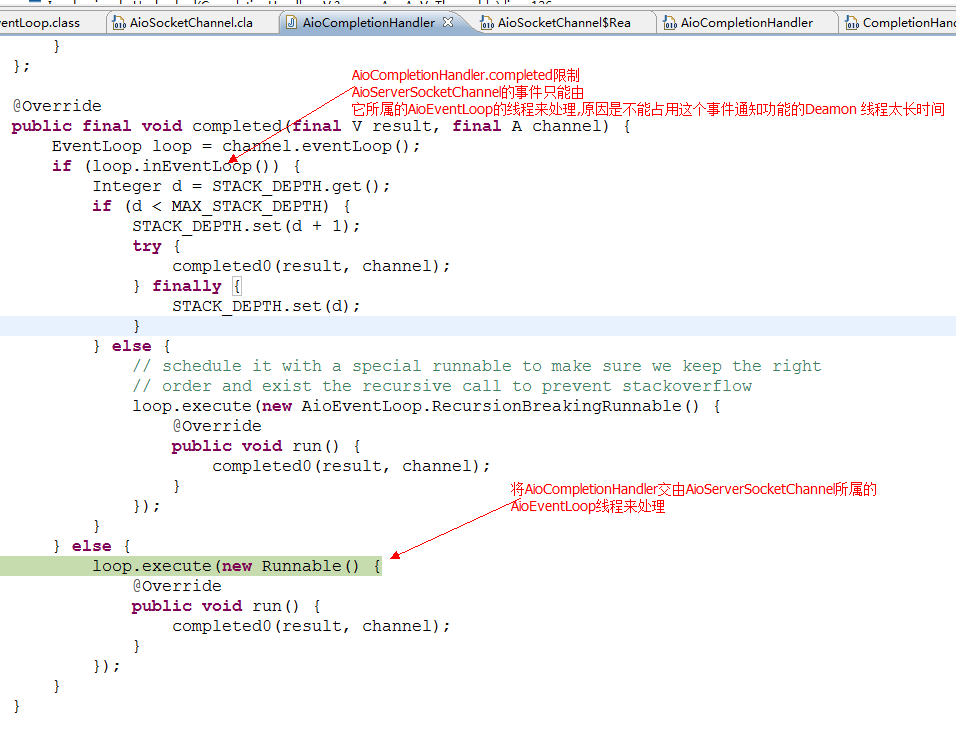


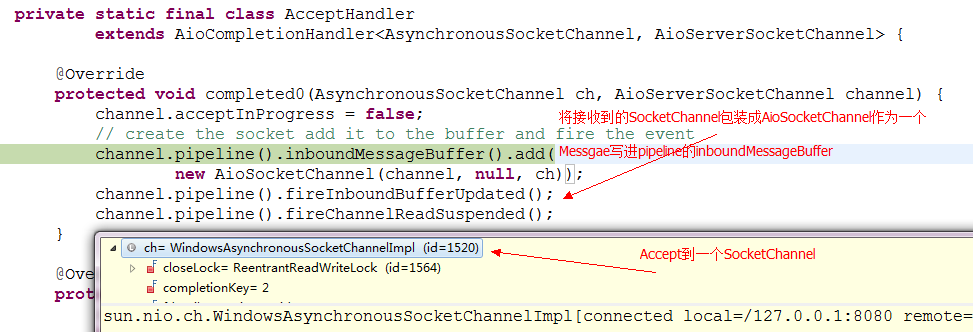
然后AioServerSocketChanel的AioEventLoop就进入等待状态,等待taskQueue有任务到来

13)现在在浏览器输入<http://127.0.0.1:8080/>,向服务器发出请求

当相应事件发生时,JDK的CompletionHandler.complted(..)方法会被一条后台Daemon的线程调用 :

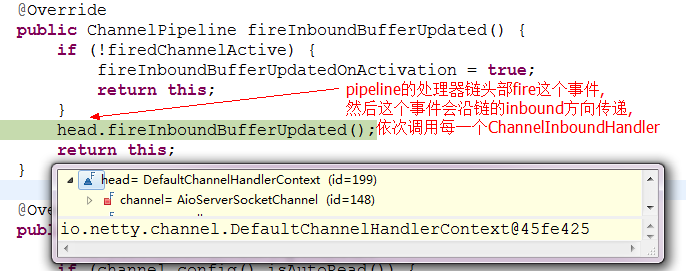




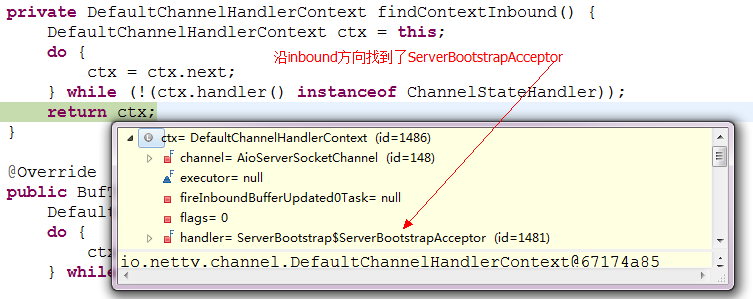


将接收到的SocketChannel包装成AioSocketChannel作为一个Message写进AioServerSocketChannel的ChannelPipeline的inboundMessageBuffer,然后Fire 相关事件inboundBufferUpdated事件

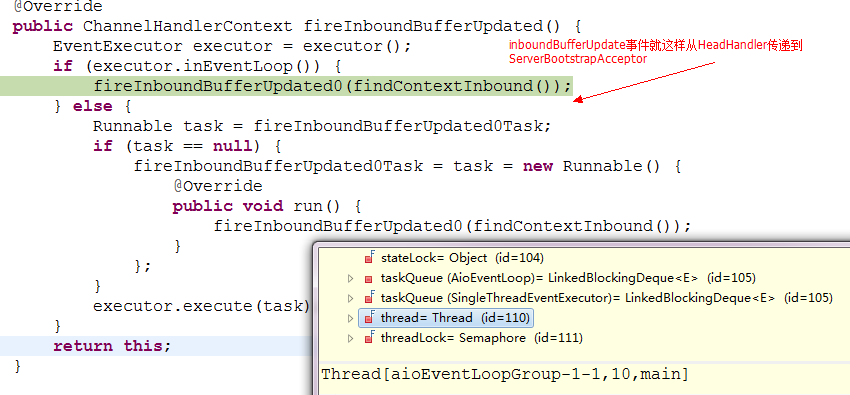
14)AioServerSocketChannel的ChannelPipeline Fire inboundBufferUpdated事件



15)沿inbound方向找到了SererBootstrapAcceptor的ChannelHandlerContext



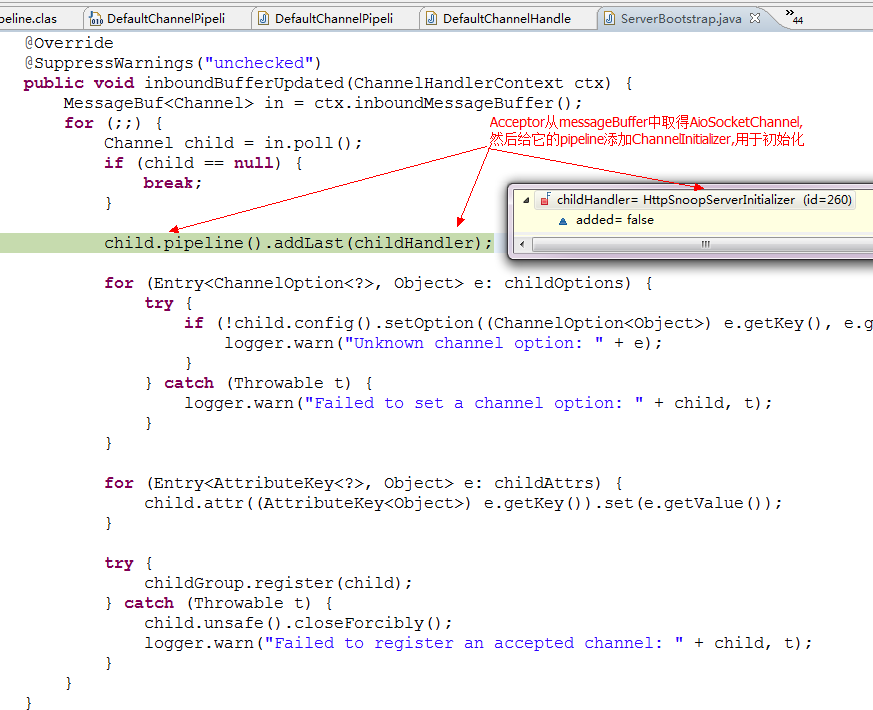
16)inboundBufferUpdate事件传递到ServerBootstrapAcceptor



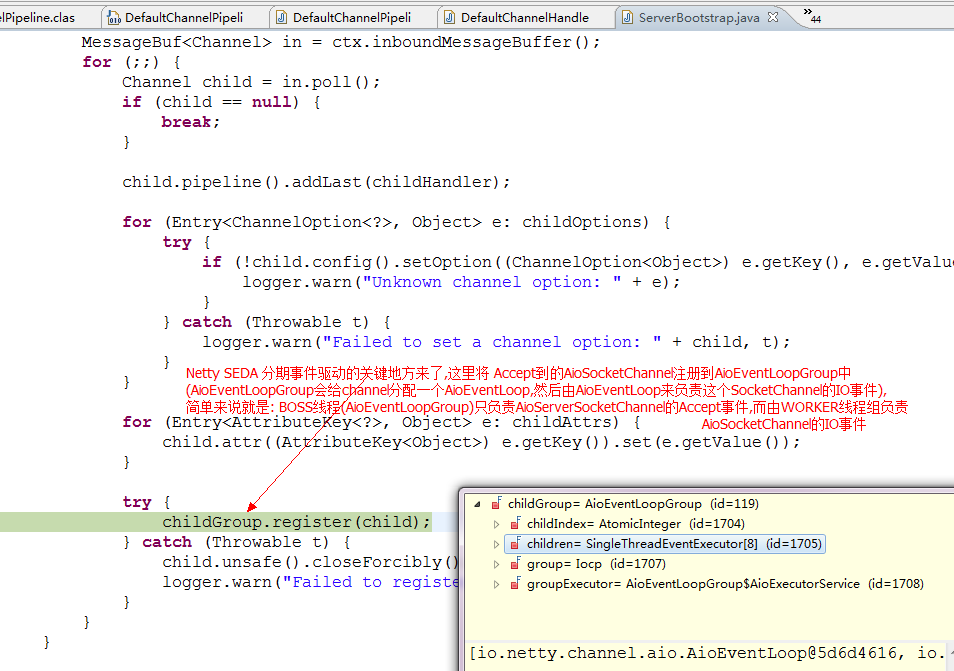
17) [DefaultChannelHandlerContext](eclipse-javadoc:%E2%98%82=%20/%3C%7BDefaultChannelHandlerContext.java%E2%98%83DefaultChannelHandlerContext).invokeInboundBufferUpdated(),调用Acceptor的inboundBufferUpdated方法



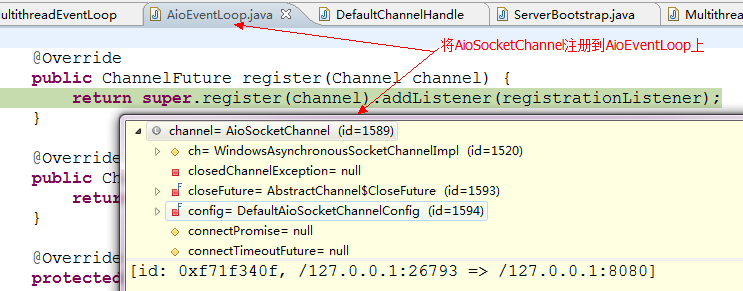
18)Acceptor取得AioSocketChannel,并向其添加ChannelInitializer



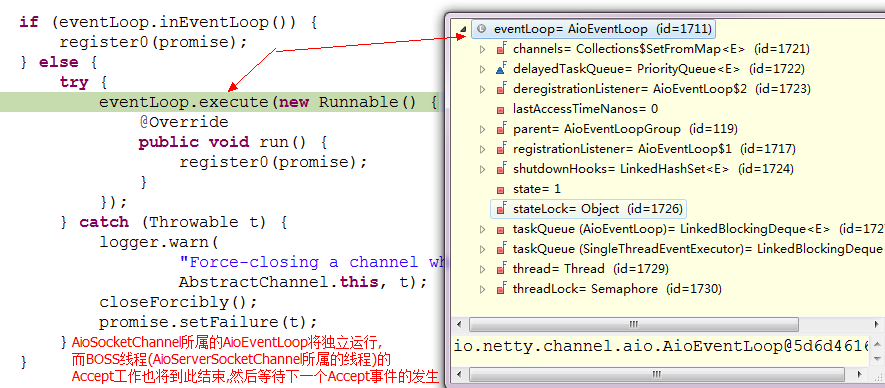
19)向ChildGroup注册AioSocketChannel



20)将AioSocketChannel注册到AioEventLoopGroup分配的AioEventLoop上

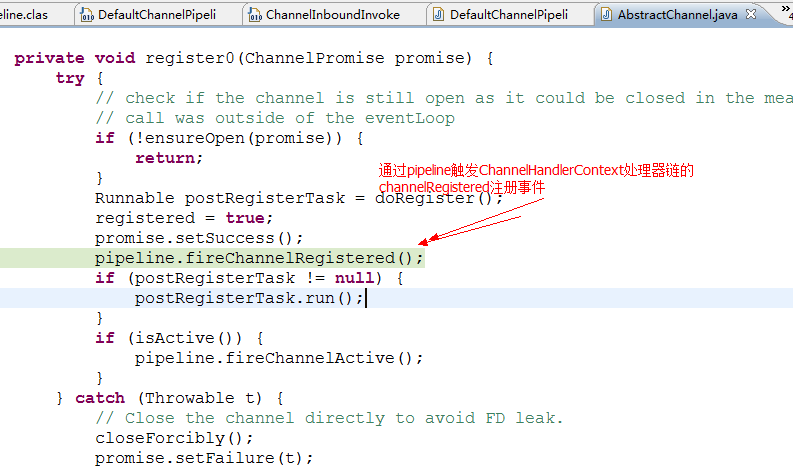


21)BOSS线程(AioServerSocketChannel所属的线程)启动AioSocketChannel所属的AioEventLoop线程,BOSS线程的Accept工作也至此结束,然后就去回去处理其taskQueue中的任务,或进入wait状态,等待下一个Accept事件的发生.

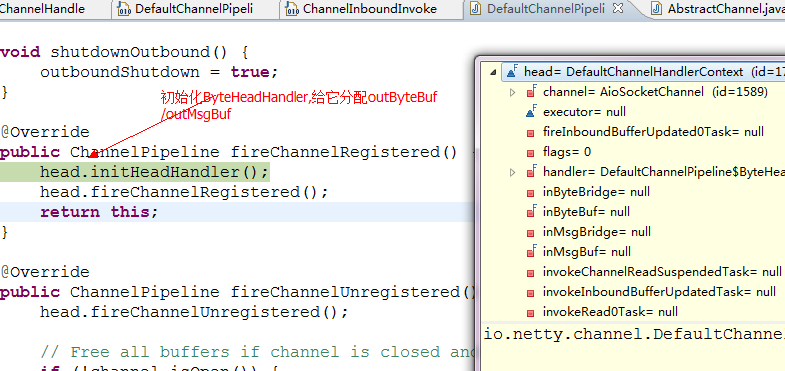


* Worker线程线程的初始化

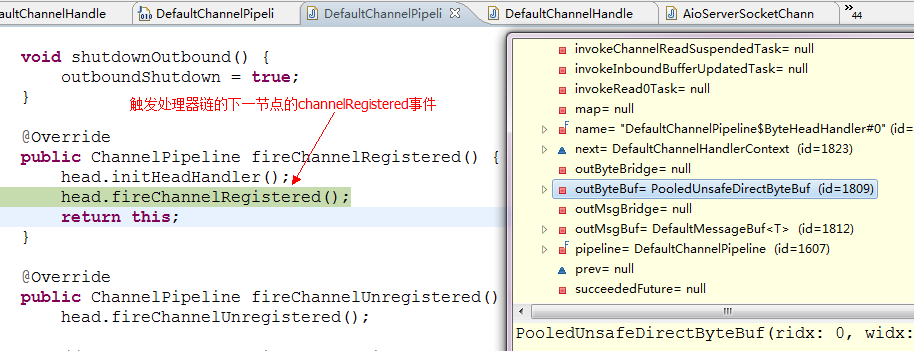
22)Worker线程(AioSocketChannel所属的AioEventLoop)独立运行后,取出其taskQueue中的第一个任务: 通过AioSocketChannel的ChannelPipeline触发pipeline的ChannelHandlerContext处理器链的channelRegistered注册事件



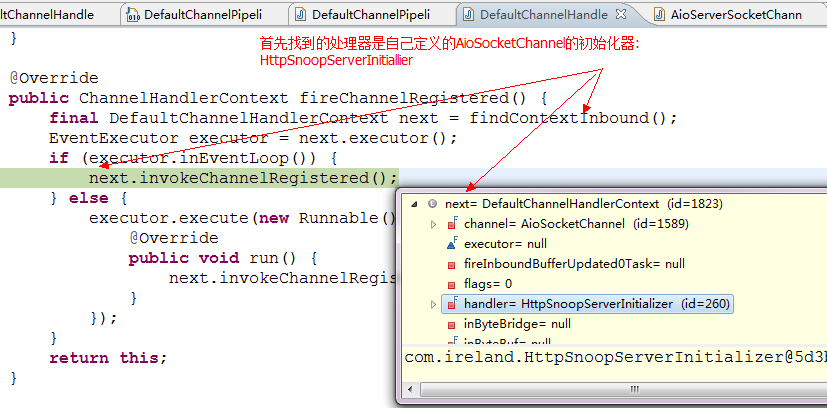
23)初始化ByteHeadHandler(ChannelOutboundHandler的子类),给它分配outByteBuf和outMsgBuf



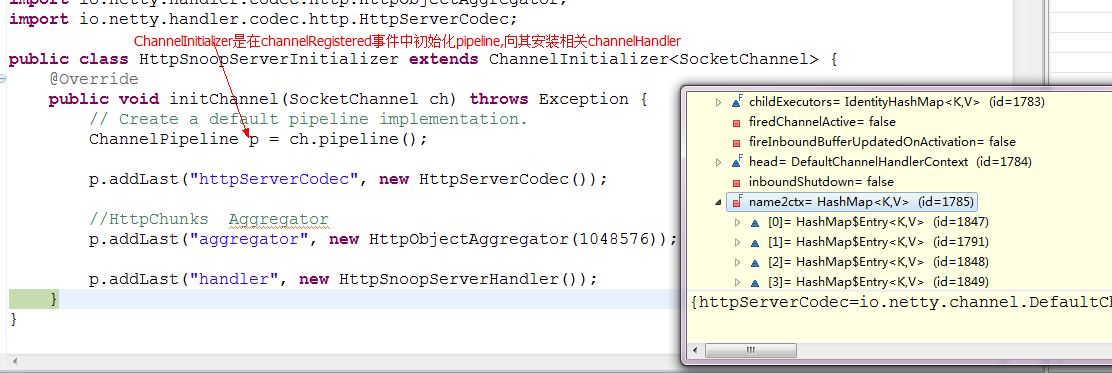
24)触发处理器链的下一节点的channelRegistered事件



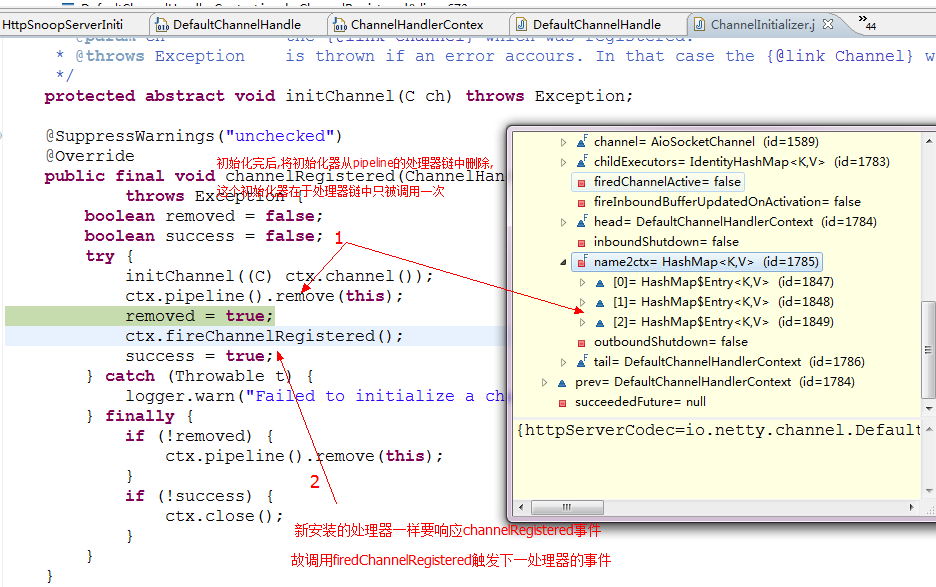
25)沿ChannelHandlerContext处理器链找到了自己定义 的用户AioSocketChannel初始化的HttpSnoopServerInitializer



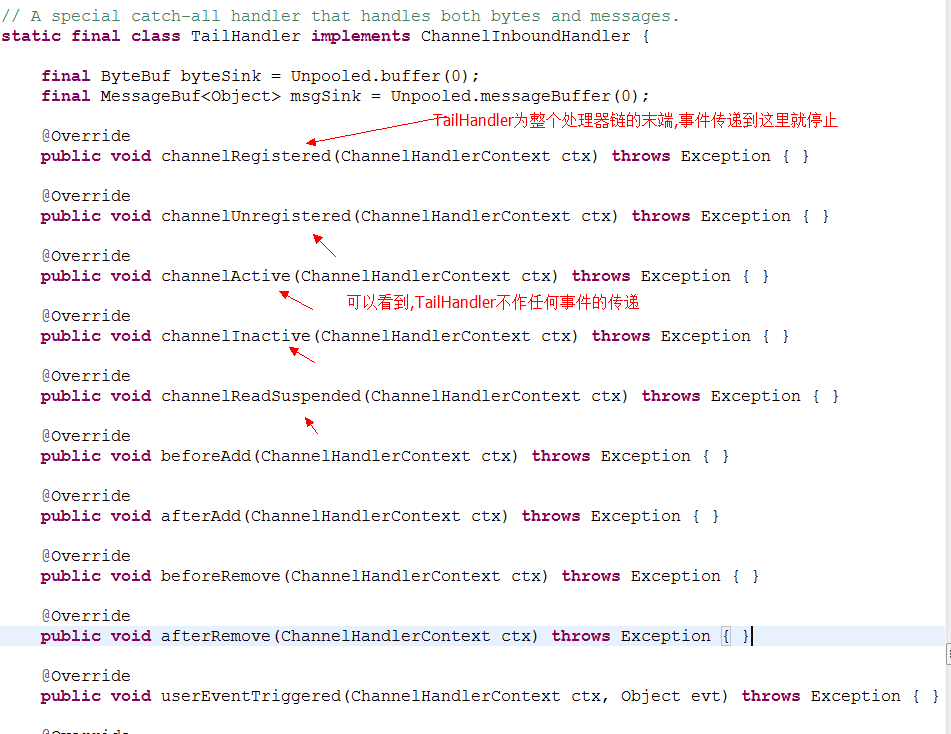
26)ChannelInitializer是利用channelRegistered事件来初始化pipeline,向其安装相关channelHandler



27)从pipeline的处理器链删除初始化器,并触发下一处理器的channelRegistered事件

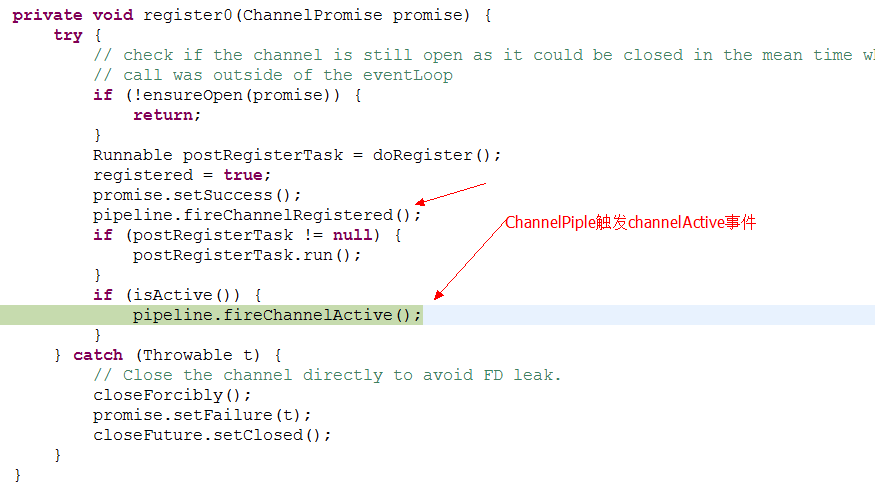


28) channelRegistered事件通过一个一个ChannelHandlerContext的fireChannelRegistered方法沿ByteHeadHandler传递到TailHandler,然后事件就停止传播

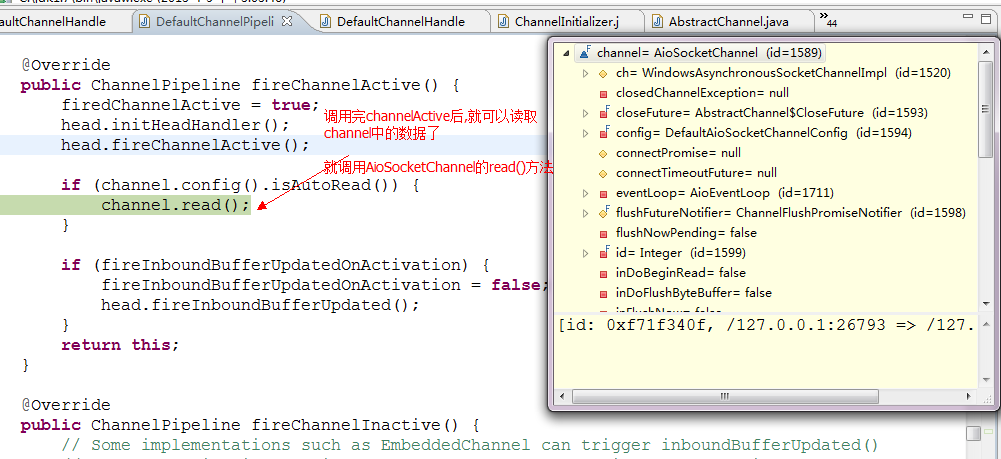


29)ChannelPipeline完成channelRegistered的事件触发后,就触发channelActive事件,

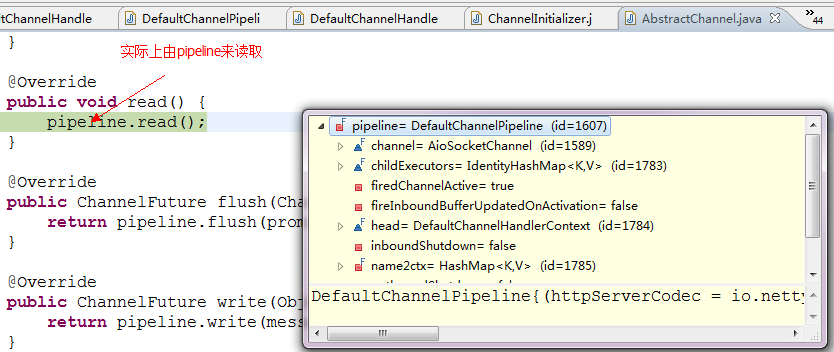
然后channelActive事件就随着ChannelHandlerContext和链式调用而在处理器链中传递,原理上上面差不多,故不详细说.



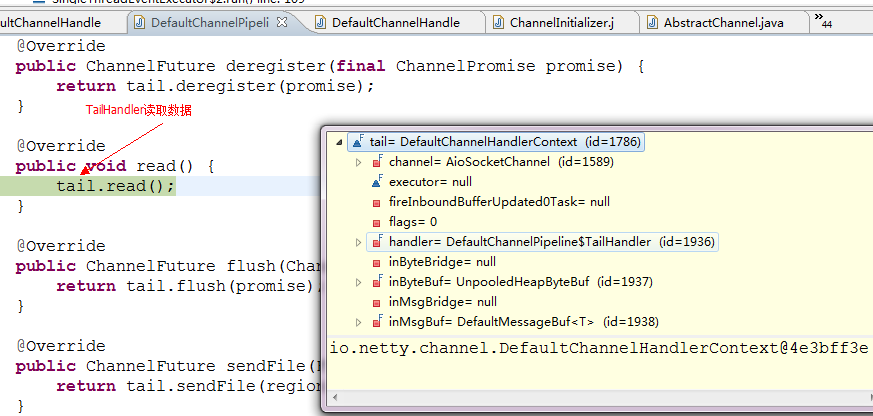
30)AioSocketChannel读取数据



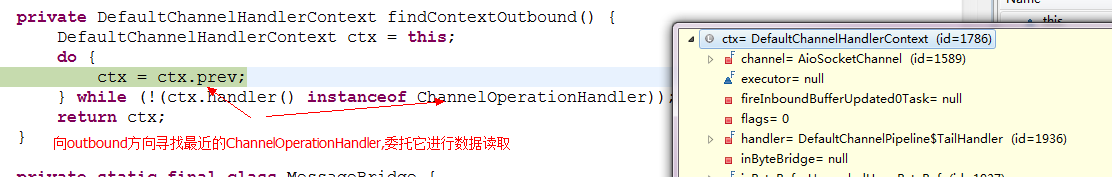
由pipeline来读取数据



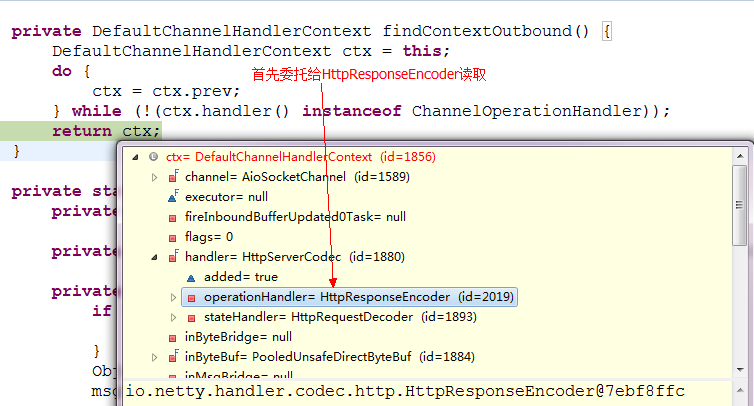
到TailHandler读取数据



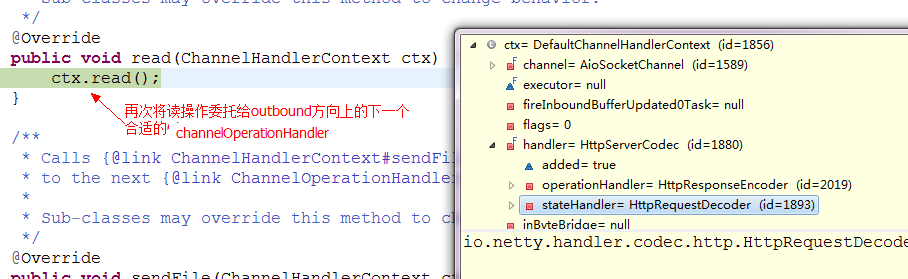
31)向outbound方向上寻找最近的ChannelOperationHandler,委托它进行数据读取



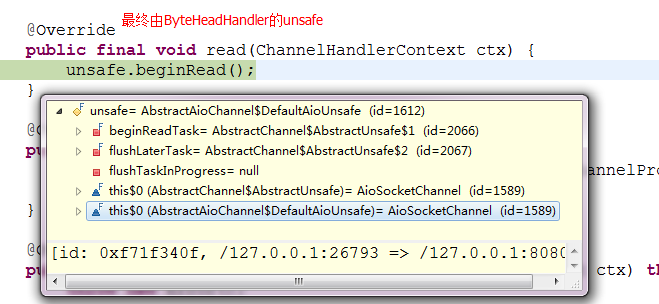
首先委托给HttpResponseEncoder读取



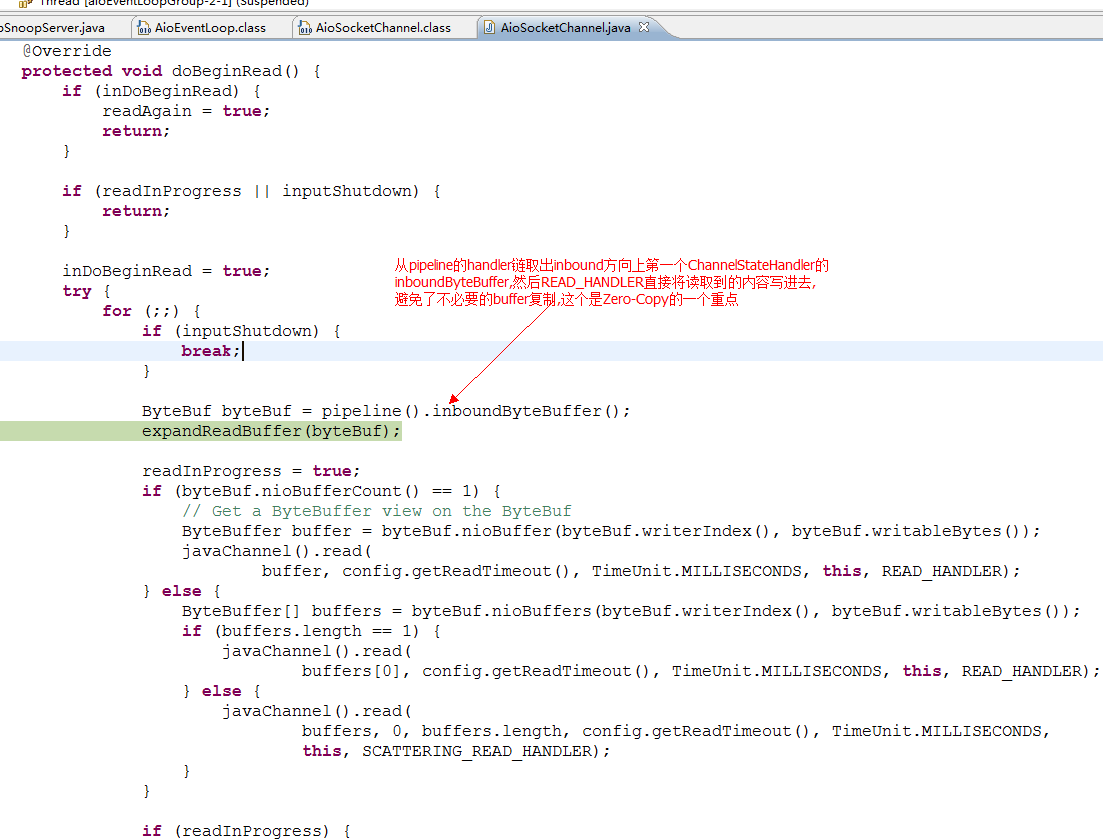
HttpRequestDecoder委托给下一个ChannelOperationHandler读取



最终委托给ByteHeadHandler读取

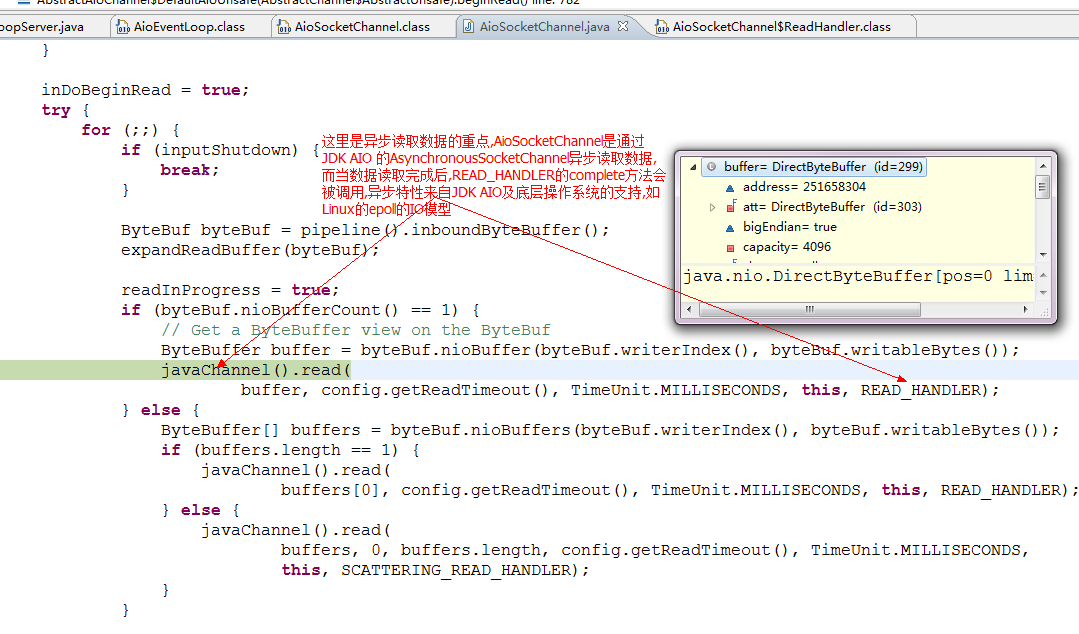


32)最终由AioSocketChannel通过JDK的AsynchronousSocketChannel从底层读取

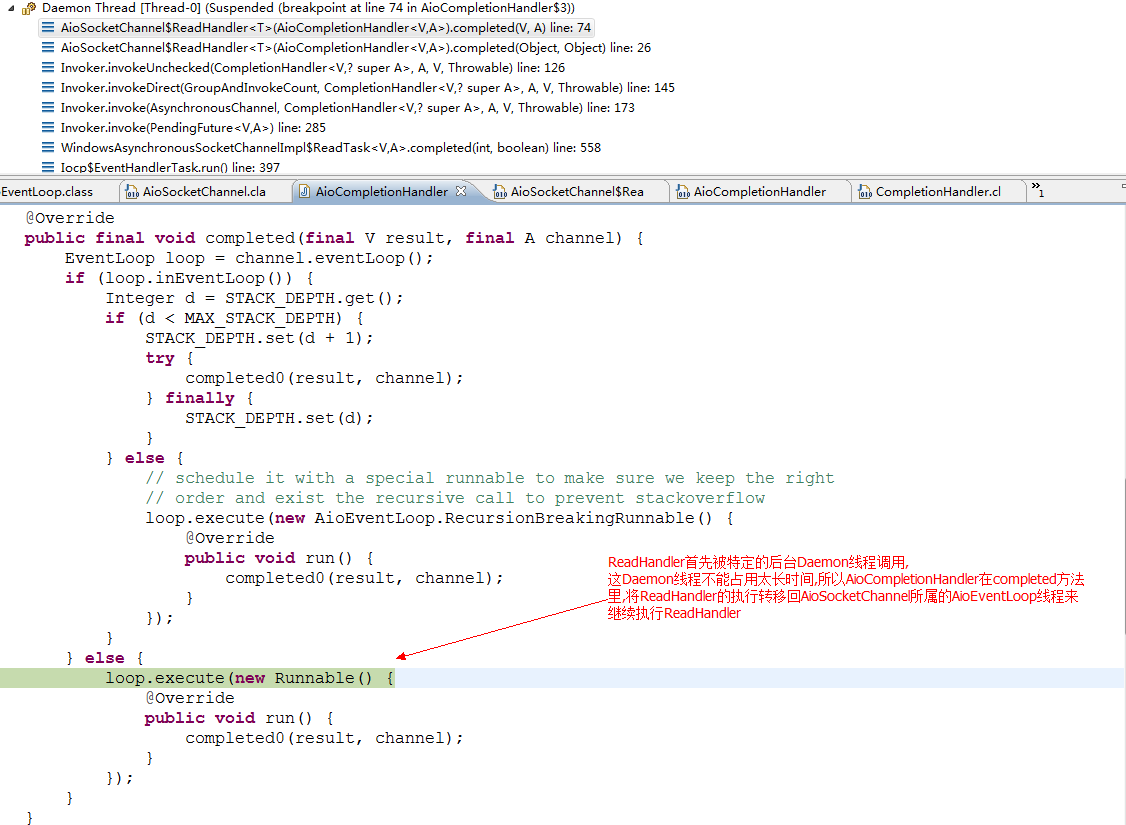


33)AioSocketChannel将读取数据操作委托给JDK的AsynchronousSocketChannel来完成,AioSocketChannel的当前线程不会被阻塞,立马返回处理taskQueue中下一任务或没任务时进入wait等待BlockingQueue的状态.

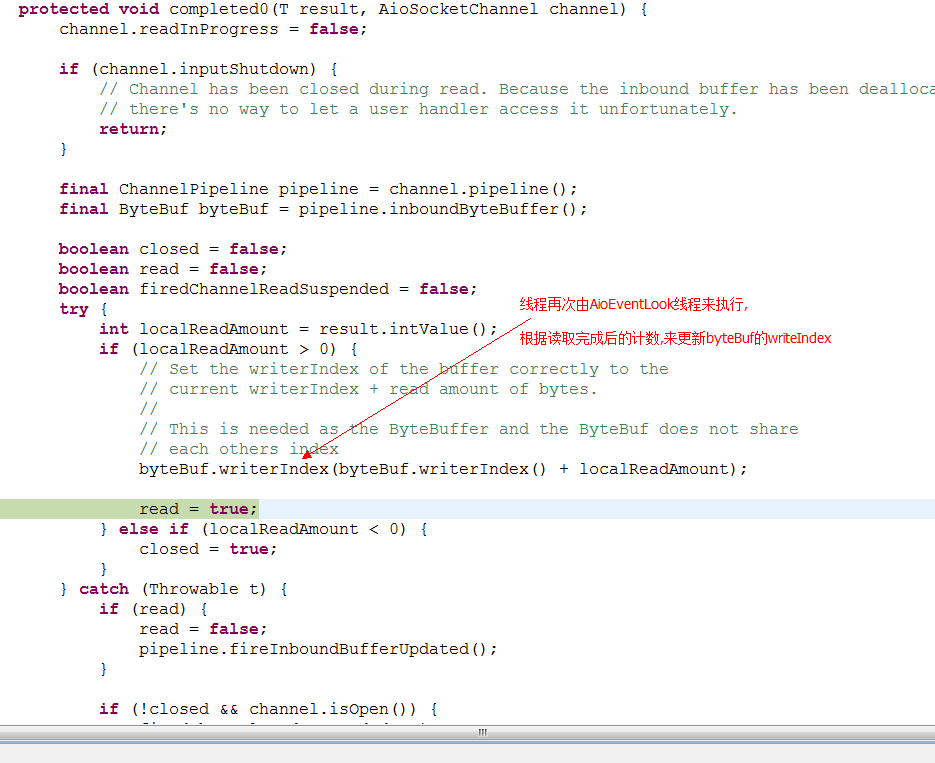
AsynchronousSocketChannel的read操作完成后,ReadHandler会被一条后台Daemon线程调用



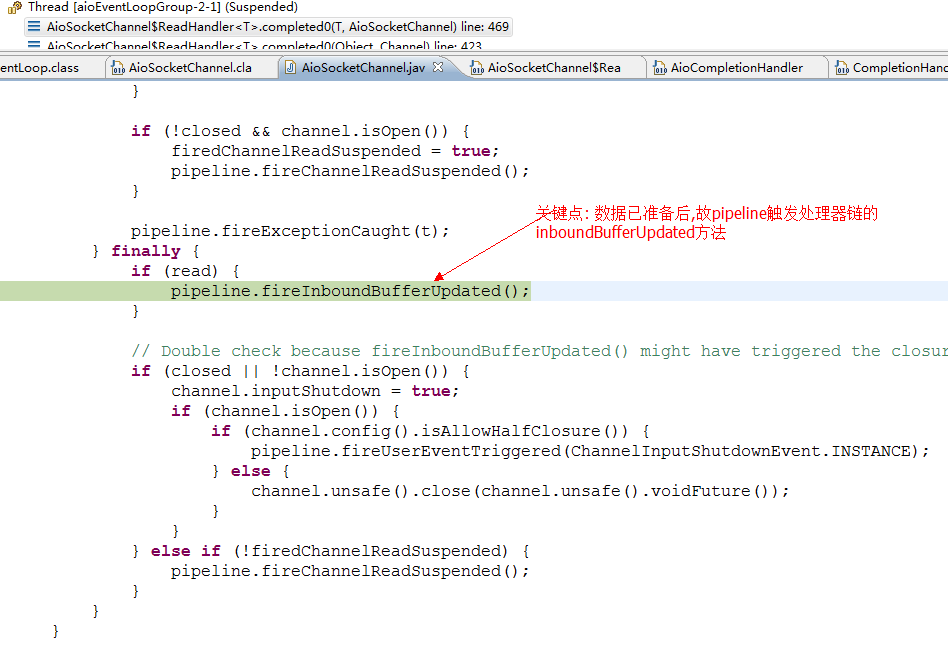
34) ReadHandler的执行从Daemon线程转移回AioSocketChannle所属的AioEventLoop线程来执行



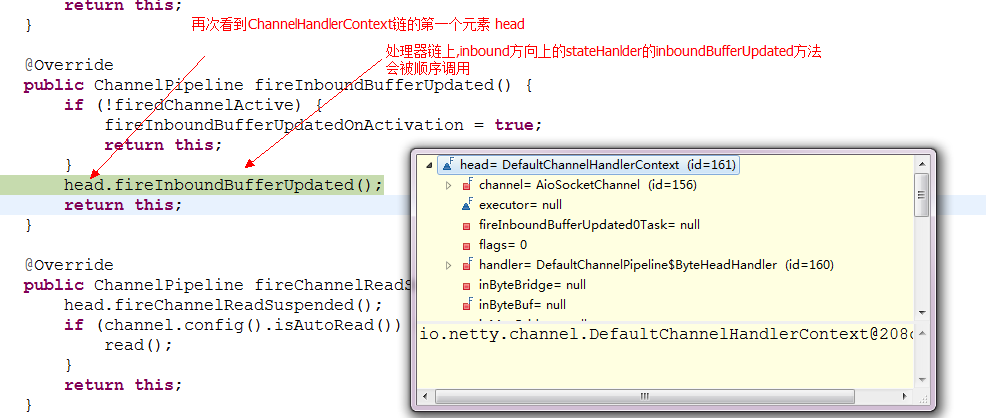
35)由AioEventLoop执行ReadHandler



36)数据已准备好,pipeline触发inboundBufferUpdated方法,然后pipeline中处理器链的inbound方向上的ChannelStateHandler的inboundBufferUpdated方法会被顺度调用



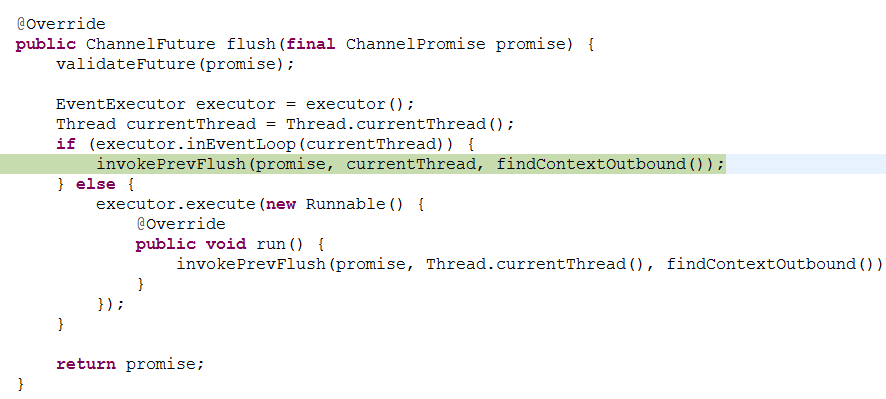
37)head 触发处理器链上inbound方向上的stateHandler的inboudBufferUpdated方法的顺序调用



38) 处理器链上inbound方向上的stateHandler的inboudBufferUpdated方法的顺序调用

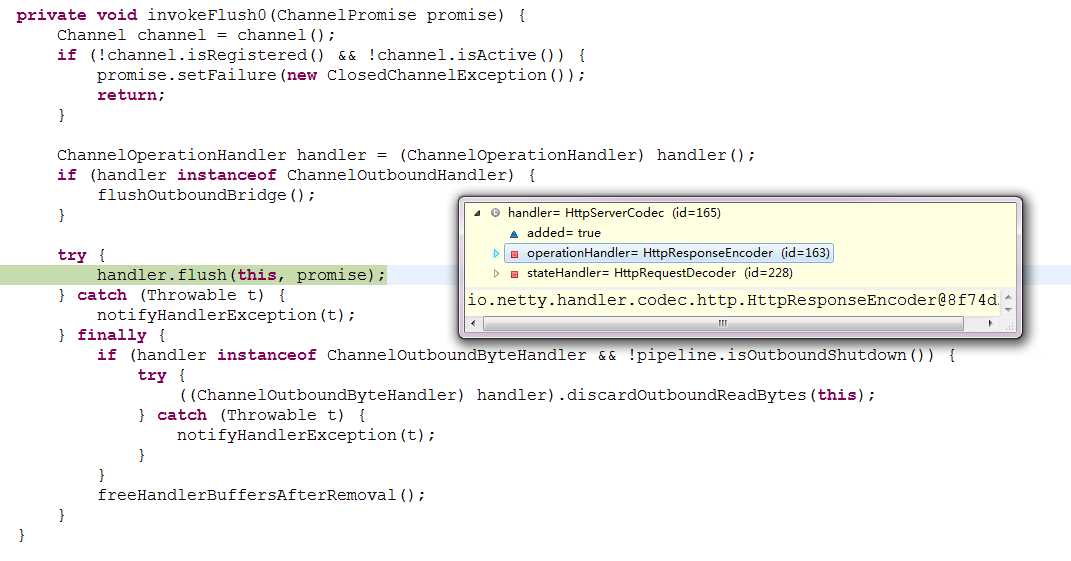
-----这里省略,参考:

39)输入数据: [DefaultChannelHandlerContext](eclipse-javadoc:%E2%98%82=%20/%3C%7BDefaultChannelHandlerContext.java%E2%98%83DefaultChannelHandlerContext).flush([ChannelPromise](eclipse-javadoc:%E2%98%82=%20/%3C%7BDefaultChannelHandlerContext.java%E2%98%83DefaultChannelHandlerContext~flush~QChannelPromise;%E2%98%82ChannelPromise) promise)

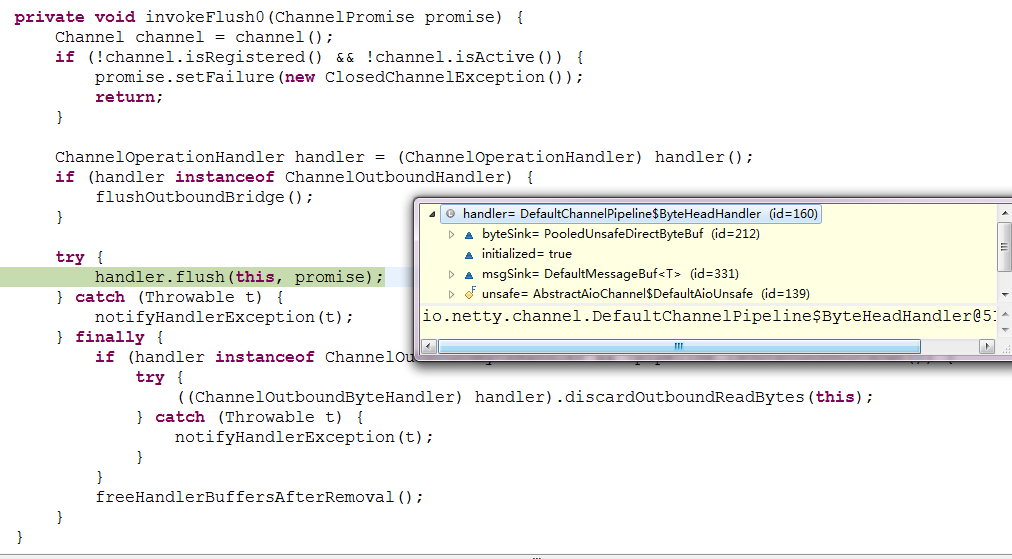


40)由HttpServerCodec中的HttpResponseEncoder将HttpMessage 转换为ByteBuf

DefaultChannelHandlerContext.invokeFlush0(ChannelPromise promise)

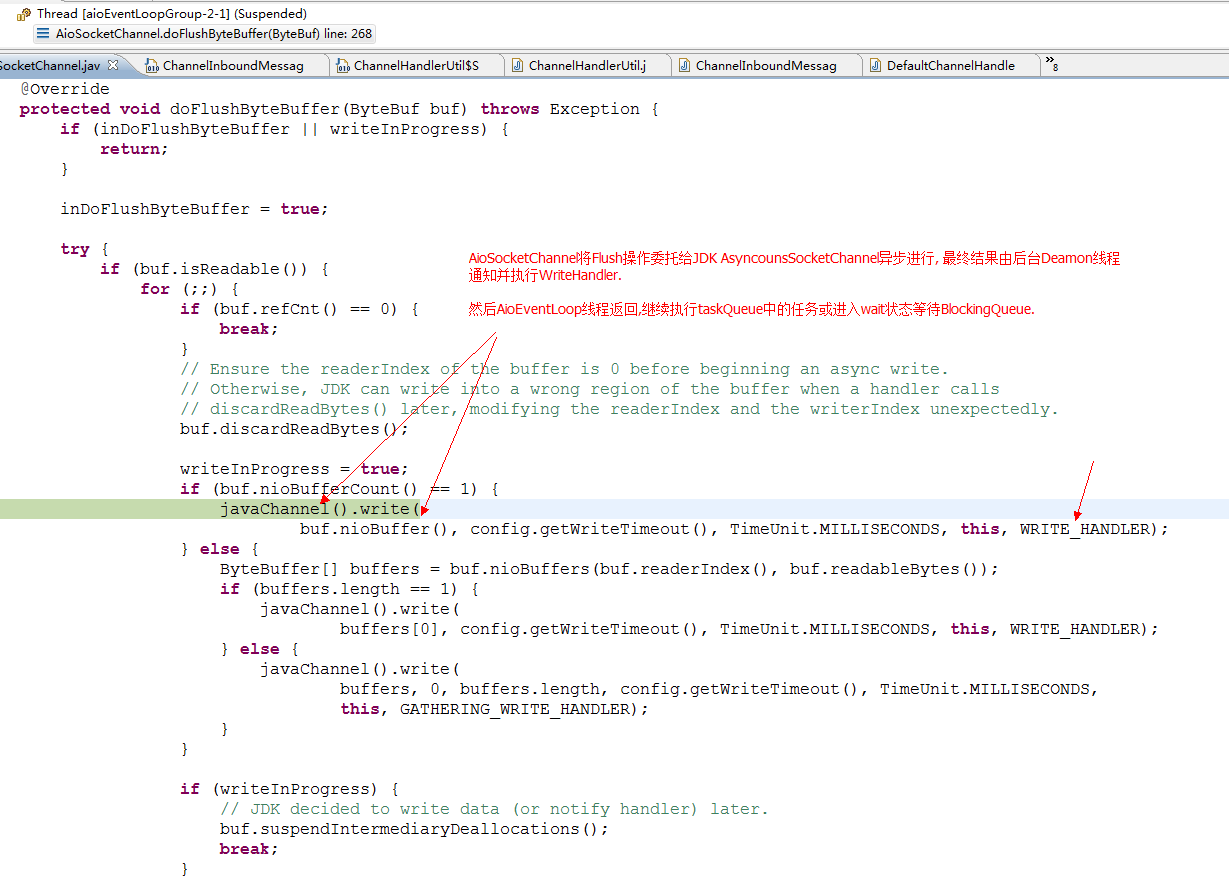


41)由Pipeline的ByteHeadHandler将ByteBuf flush出去

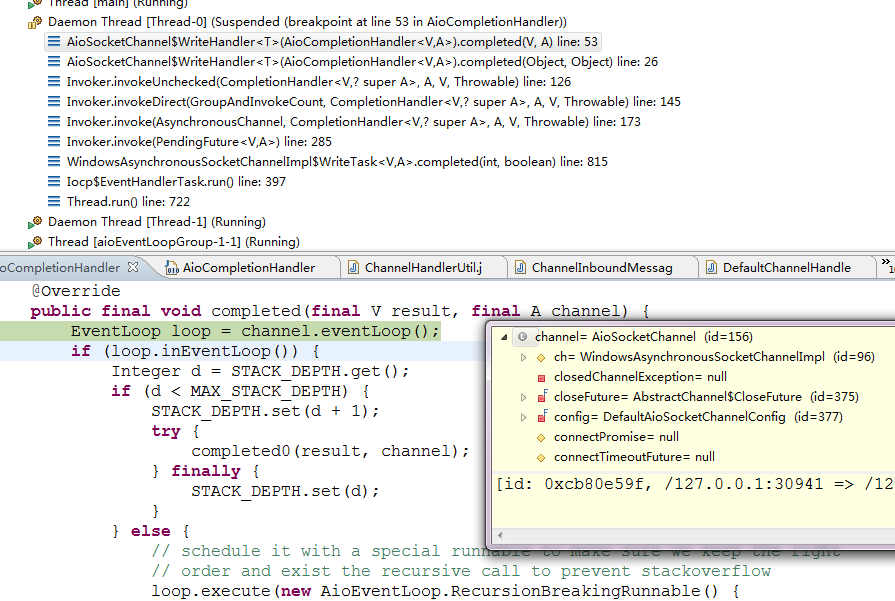


42)AioSocketChannel将Flush操作委托给JDK AsyncounsSocketChannel异步进行,最终结果等写操作完成后由后台Deamon线程通知并执行WriteHandler.

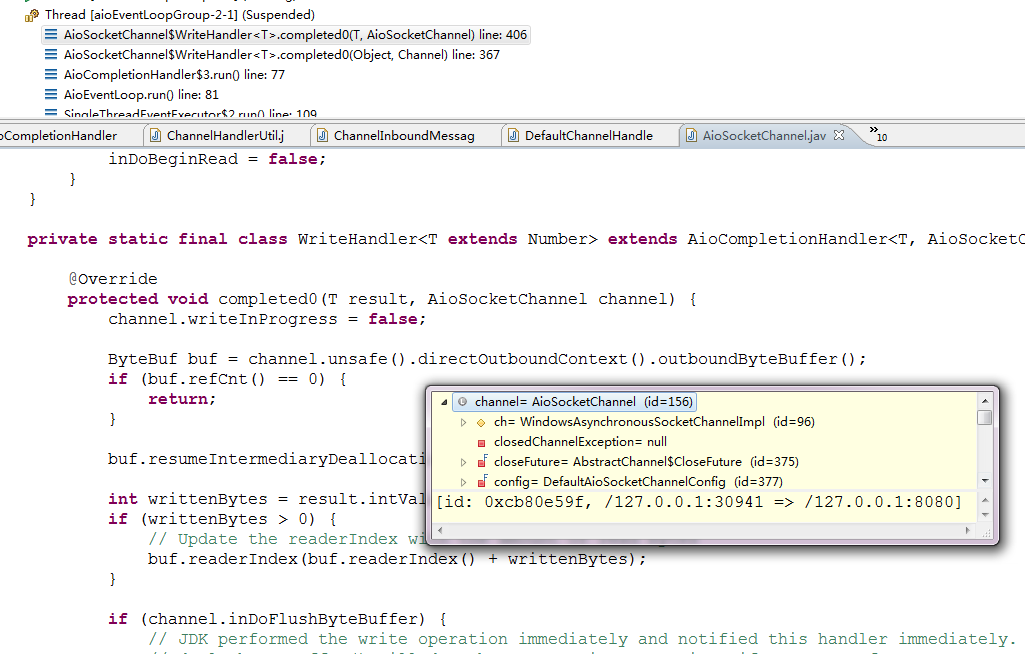
而AioEventLoop线程是马上返回的,不会被write操作阻塞,然后继续执行taskQueue中的任务或进入wait状态等待BlockingQueue.



43)Write操作完成后,WriteHandler被后台Daemon线程调用



44)再次WriteHandler转交由关联的AioSocketChannel所属的AioEventLoop执行



45) AioEventLoop执行完WriteHandler后,整个接收和响应过程就完成了.

Note:

##### [AioServerSocketChannel](eclipse-javadoc:%E2%98%82=%20/%3C%7BAioServerSocketChannel.java%E2%98%83AioServerSocketChannel).AcceptHandler 分析



**private** **static** **final** **class** AcceptHandler **extends** AioCompletionHandler<AsynchronousSocketChannel, AioServerSocketChannel>

{

//以下方法在BOSS线程执行

@Override

**protected** **void** completed0(AsynchronousSocketChannel ch, AioServerSocketChannel channel) {

channel.acceptInProgress = **false**;

// create the socket add it to the buffer and fire the event

channel.pipeline().inboundMessageBuffer().add(**new** AioSocketChannel(channel, **null**, ch));

//触发BufferUpdated事件来通知ServerBootstrapAcceptor 将接收到的SocketChannel注册到Worker线程(AioEventLoop)

channel.pipeline().fireInboundBufferUpdated();

//这里, SocketChannel已被注册到Worker线程,然后触发”读挂起”事件: AioServerSocketChannel再次读取数据,

来处理下一个accept事件

channel.pipeline().fireChannelReadSuspended();

}

. . . . . .

}

**final** **class** DefaultChannelPipeline **implements** ChannelPipeline {

. . . . . .

@Override

**public** ChannelPipeline fireChannelReadSuspended() {

head.fireChannelReadSuspended(); //触发处理链的”读挂起”事件

**if** (channel.config().isAutoRead()) {

read(); // AioServerSocketChannel的pipeline再次读取数据,来处理下一个accept事件

}

**return** **this**;

}

. . . . . .

}

##### [Open Declaration](eclipse-open:%E2%98%82=Netty_13_NettyExample_HTTP_Snoop_ServerBootstrapAcceptor/Z:%5C/netty-4.0.0-CR2%5C/netty-transport-4.0.0.CR2-SNAPSHOT.jar%3Cio.netty.channel.socket.aio(AioSocketChannel$ReadHandler.class%E2%98%83ReadHandler)[AioSocketChannel](eclipse-javadoc:%E2%98%82=Netty_13_NettyExample_HTTP_Snoop_ServerBootstrapAcceptor/Z:%5C/netty-4.0.0-CR2%5C/netty-transport-4.0.0.CR2-SNAPSHOT.jar%3Cio.netty.channel.socket.aio(AioSocketChannel.class%E2%98%83AioSocketChannel).ReadHandler<[T](eclipse-javadoc:%E2%98%82=Netty_13_NettyExample_HTTP_Snoop_ServerBootstrapAcceptor/Z:%5C/netty-4.0.0-CR2%5C/netty-transport-4.0.0.CR2-SNAPSHOT.jar%3Cio.netty.channel.socket.aio(AioSocketChannel$ReadHandler.class%E2%98%83ReadHandler%5dT) extends [Number](eclipse-javadoc:%E2%98%82=Netty_13_NettyExample_HTTP_Snoop_ServerBootstrapAcceptor/Z:%5C/netty-4.0.0-CR2%5C/netty-transport-4.0.0.CR2-SNAPSHOT.jar%3Cio.netty.channel.socket.aio(AioSocketChannel$ReadHandler.class%E2%98%83ReadHandler%5dT%E2%98%82Number)> 分析

**private** **static** **final** **class** ReadHandler<T **extends** Number> **extends** AioCompletionHandler<T, AioSocketChannel> {

//以下方法在Worker线程(AioEventLoop)执行

@Override

**protected** **void** completed0(T result, AioSocketChannel channel) {

channel.readInProgress = **false**;

**if** (channel.inputShutdown) {

// Channel has been closed during read. Because the inbound buffer has been deallocated already,

// there's no way to let a user handler access it unfortunately.

**return**;

}

**final** ChannelPipeline pipeline = channel.pipeline();

**final** ByteBuf byteBuf = pipeline.inboundByteBuffer();

**boolean** closed = **false**;

**boolean** read = **false**;

**boolean** firedChannelReadSuspended = **false**;

**try** {

**int** localReadAmount = result.intValue();

**if** (localReadAmount > 0) {

// Set the writerIndex of the buffer correctly to the

// current writerIndex + read amount of bytes.

//

// This is needed as the ByteBuffer and the ByteBuf does not share

// each others index

byteBuf.writerIndex(byteBuf.writerIndex() + localReadAmount);

read = **true**;

} **else** **if** (localReadAmount < 0) {

closed = **true**;

}

}

//以上主要是inboundByteBuffer的writerIndex的设置

**catch** (Throwable t) {

**if** (read) {

read = **false**;

pipeline.fireInboundBufferUpdated();

}

**if** (!closed && channel.isOpen()) {

firedChannelReadSuspended = **true**;

pipeline.fireChannelReadSuspended();

}

pipeline.fireExceptionCaught(t);

} **finally** {

**if** (read) {

*//触发ChannelHandlerContext链的inboundBufferUpdated事件*

*pipeline.fireInboundBufferUpdated();*

}

// Double check because fireInboundBufferUpdated() might have triggered the closure by a user handler.

**if** (closed || !channel.isOpen()) {

channel.inputShutdown = **true**;

**if** (channel.isOpen()) {

**if** (channel.config().isAllowHalfClosure()) {

pipeline.fireUserEventTriggered(ChannelInputShutdownEvent.*INSTANCE*);

} **else** {

channel.unsafe().close(channel.unsafe().voidFuture());

}

}

}

**else** **if** (!firedChannelReadSuspended) {

//触发channelReadSuspended事件,读取后续到达SocketChannel的数据

*pipeline.fire**ChannelReadSuspended();*

}

}

}

. . . . . . . . . .. …

}

**final** **class** DefaultChannelPipeline **implements** ChannelPipeline {

. . . . . .

@Override

**public** ChannelPipeline fireChannelReadSuspended() {

head.fireChannelReadSuspended(); //触发处理链的”读挂起”事件

**if** (channel.config().isAutoRead()) {

read(); // 数据可能断续到达,让AioSocketChannel的pipeline再次读取数据

}

**return** **this**;

}

. . . . . .

}