第三章：Netty 源码

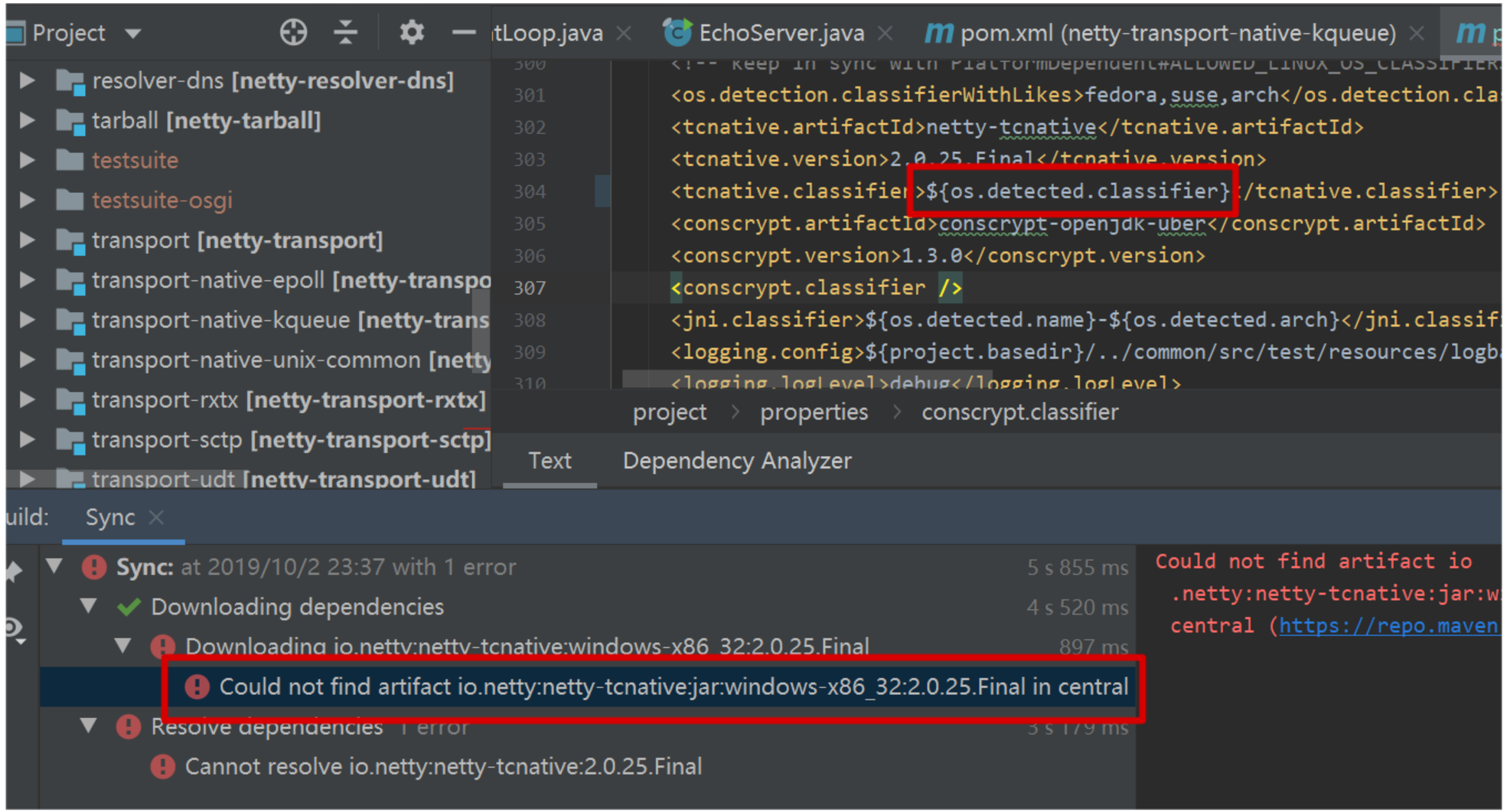
从“线”（请求处理）的角度剖析

Netty 代码编译与总览

- 编译 Netty 常遇问题
- Netty 源码核心包速览

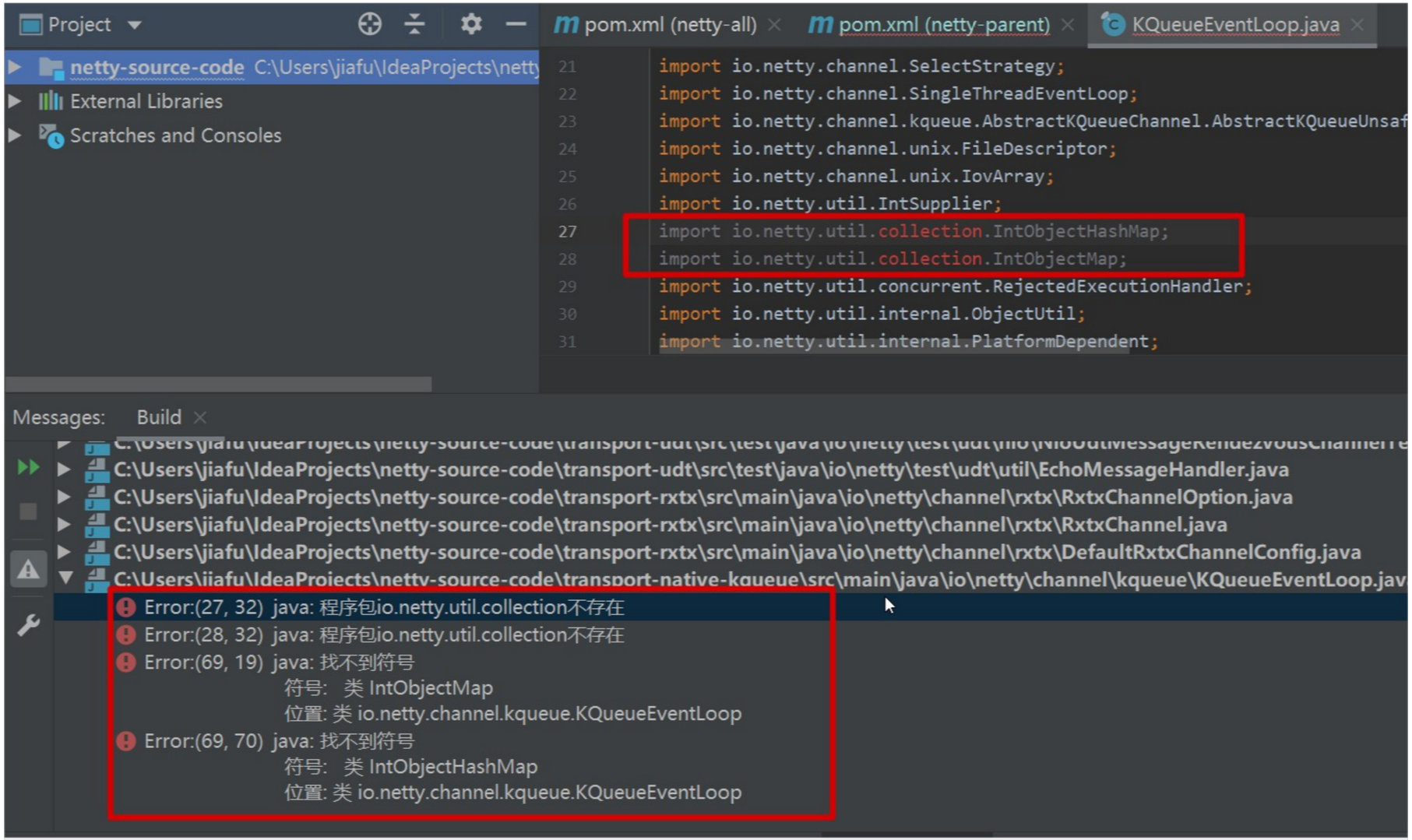
编译 Netty 常遇问题

- 编译 Netty 常见问题 1：



编译 Netty 常遇问题

- 编译 Netty 常见问题 2：



Netty 源码核心包速览

- io.netty.transport
  - io.netty.transport.epoll
  - io.netty.transport.kqueue
  - io.netty.transport.unix.common
- io.netty.transport.sctp
- ~~io.netty.transport.rxtx~~
- ~~io.netty.transport.udt~~

- io.netty.codec.dns
- io.netty.codec.haproxy
- io.netty.codec.http
- io.netty.codec.http2
- io.netty.codec.memcache
- io.netty.codec.mqtt
- io.netty.codec.redis
- io.netty.codec.smtp
- io.netty.codec.socks
- io.netty.codec.stomp
- io.netty.codec.xml

- io.netty.codec

- io.netty.handler
- io.netty.handler.proxy

- io.netty.buffer
- io.netty.common
- io.netty.resolver
- io.netty.resolver.dns

源码剖析：启动服务

- 主线
- 源码演示
- 知识点

主线

our thread

- 创建 selector
- 创建 server socket channel
- 初始化 server socket channel
- 给 server socket channel 从 boss group 中选择一个 NioEventLoop

boss thread

- 将 server socket channel 注册到选择的 NioEventLoop 的 selector
- 绑定地址启动
- 注册接受连接事件（OP_ACCEPT）到 selector 上

知识点

- 启动服务的本质：

```
Selector selector = sun.nio.ch.SelectorProviderImpl.openSelector()

ServerSocketChannel serverSocketChannel = provider.openServerSocketChannel()

selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);

javaChannel().bind(localAddress, config.getBacklog());

selectionKey.interestOps(OP_ACCEPT);
```

知识点

- Selector 是在 new NioEventLoopGroup()（创建一批 NioEventLoop）时创建。
- 第一次 Register 并不是监听 OP_ACCEPT，而是 0:

  selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this) 。
- 最终监听 OP_ACCEPT 是通过 bind 完成后的 fireChannelActive() 来触发的。
- NioEventLoop 是通过 Register 操作的执行来完成启动的。
- 类似 ChannelInitializer，一些 Hander 可以设计成一次性的，用完就移除，例如授权。

源码剖析：构建连接

- 主线
- 源码演示
- 知识点

主线

boss thread

- NioEventLoop 中的 selector 轮询创建连接事件（OP_ACCEPT）：
- 创建 socket channel
- 初始化 socket channel 并从 worker group 中选择一个 NioEventLoop

worker thread

- 将 socket channel 注册到选择的 NioEventLoop 的 selector
- 注册读事件（OP_READ）到 selector 上

知识点

- 接受连接本质：

  selector.select()/selectNow()/select(timeoutMillis) 发现 OP_ACCEPT 事件，处理：

  - SocketChannel socketChannel = serverSocketChannel.accept()
  - selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this);
  - selectionKey.interestOps(OP_READ);

知识点

- 创建连接的初始化和注册是通过 pipeline.fireChannelRead 在 ServerBootstrapAcceptor 中完成的。
- 第一次 Register 并不是监听 OP_READ，而是 0：

  selectionKey = javaChannel().register(eventLoop().unwrappedSelector(), 0, this) 。
- 最终监听 OP_READ 是通过“Register”完成后的fireChannelActive（io.netty.channel.AbstractChannel.AbstractUnsafe#register0中）来触发的
- Worker’s NioEventLoop 是通过 Register 操作执行来启动。
- 接受连接的读操作，不会尝试读取更多次（16次）。

源码剖析：接受数据

- 读数据技巧
- 主线
- 源码演示
- 知识点

读数据技巧

1 自适应数据大小的分配器（AdaptiveRecvByteBufAllocator）：

发放东西时，拿多大的桶去装？小了不够，大了浪费，所以会自己根据实际装的情况猜一猜下次情况，从而决定下次带多大的桶。

2 连续读（defaultMaxMessagesPerRead）：

发放东西时，假设拿的桶装满了，这个时候，你会觉得可能还有东西发放，所以直接拿个新桶等着装，而不是回家，直到后面出现没有装上的情况或者装了很多次需要给别人一点机会等原因才停止，回家。

主线

- 多路复用器（ Selector ）接收到 OP_READ 事件
- 处理 OP_READ 事件：NioSocketChannel.NioSocketChannelUnsafe.read()
  - 分配一个初始 1024 字节的 byte buffer 来接受数据
  - 从 Channel 接受数据到 byte buffer
  - 记录实际接受数据大小，调整下次分配 byte buffer 大小
  - 触发 pipeline.fireChannelRead(byteBuf) 把读取到的数据传播出去
  - 判断接受 byte buffer 是否满载而归：是，尝试继续读取直到没有数据或满 16 次；否，结束本轮读取，等待下次 OP_READ 事件

worker thread

知识点

- 读取数据本质：sun.nio.ch.SocketChannelImpl#read(java.nio.ByteBuffer)
- NioSocketChannel read() 是读数据， NioServerSocketChannel read() 是创建连接
- pipeline.fireChannelReadComplete()；一次读事件处理完成

  pipeline.fireChannelRead(byteBuf)；一次读数据完成，一次读事件处理可能会包含多次读数据操作。
- 为什么最多只尝试读取 16 次？“雨露均沾”
- AdaptiveRecvByteBufAllocator 对 bytebuf 的猜测：放大果断，缩小谨慎（需要连续 2 次判断）

源码剖析：业务处理

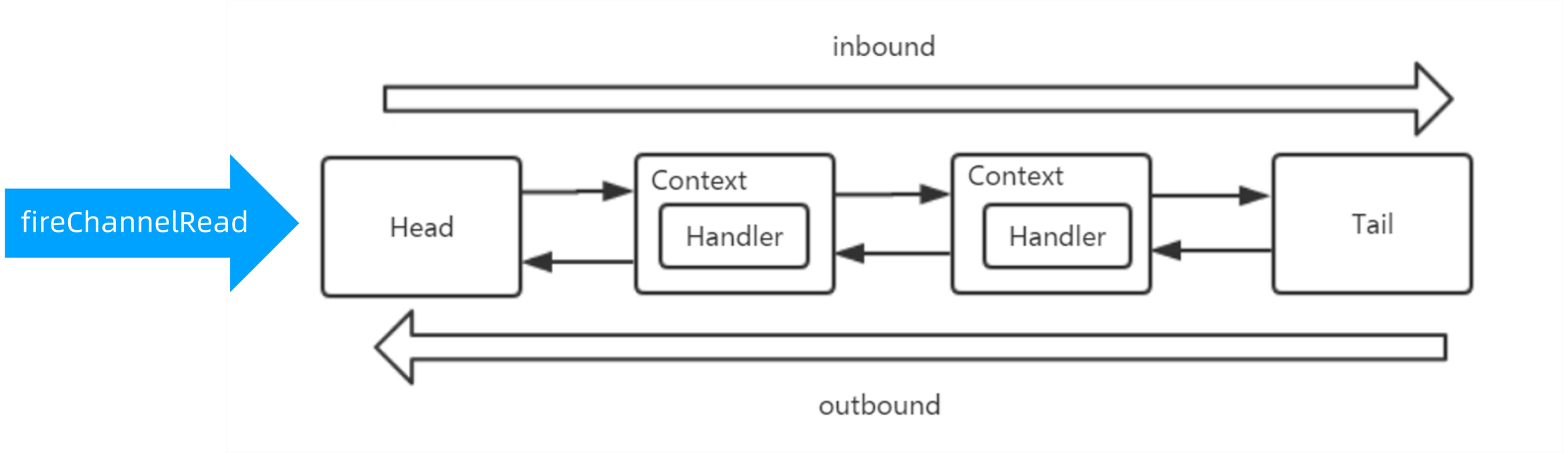
- 主线
- 源码演示
- 知识点

主线

- ~~多路复用器（ Selector ）接收到 OP_READ 事件~~
- ~~处理 OP_READ 事件：NioSocketChannel.NioSocketChannelUnsafe.read()~~
  - ~~分配一个初始 1024 字节的 byte buffer 来接受数据~~
  - ~~从 Channel 接受数据到 byte buffer~~
  - ~~记录实际接受数据大小，调整下次分配 byte buffer 大小~~
  - **触发 pipeline.fireChannelRead(byteBuf) 把读取到的数据传播出去**
  - ~~判断接受 byte buffer 是否满载而归：是，尝试继续读取直到没有数据或满 16 次；否，结束本轮读取，等待下次OP_READ事件~~

worker thread

主线



Handler 执行资格：

- 实现了 ChannelInboundHandler
- 实现方法 channelRead 不能加注解 @Skip

知识点

- 处理业务本质：数据在 pipeline 中所有的 handler 的 channelRead() 执行过程

  Handler 要实现 io.netty.channel.ChannelInboundHandler#channelRead (ChannelHandlerContext ctx, Object msg)，且不能加注解 @Skip 才能被执行到。

  中途可退出，不保证执行到 Tail Handler。

- 默认处理线程就是 Channel 绑定的 NioEventLoop 线程，也可以设置其他：

  pipeline.addLast(new UnorderedThreadPoolEventExecutor(10), serverHandler)

源码剖析：发送数据

- 写数据三种方式
- 写数据要点
- 主线
- 源码演示
- 知识点

写数据三种方式

快递场景（包裹）	Netty 写数据（数据）
揽收到仓库	write：写到一个 buffer
从仓库发货	flush：把 buffer 里的数据发送出去
揽收到仓库并立马发货 （加急件）	writeAndFlush：写到 buffer，立马发送
揽收与发货之间有个缓冲的仓库	Write 和 Flush 之间有个 ChannelOutboundBuffer

写数据要点

1 对方仓库爆仓时，送不了的时候，会停止送，协商等电话通知什么时候好了，再送。

Netty 写数据，写不进去时，会停止写，然后注册一个 OP_WRITE 事件，来通知什么时候可以写进去了再写。

2 发送快递时，对方仓库都直接收下，这个时候再发送快递时，可以尝试发送更多的快递试试，这样效果更好。

Netty 批量写数据时，如果想写的都写进去了，接下来的尝试写更多（调整 maxBytesPerGatheringWrite）。

写数据要点

3 发送快递时，发到某个地方的快递特别多，我们会连续发，但是快递车毕竟有限，也会考虑下其他地方。

Netty 只要有数据要写，且能写的出去，则一直尝试，直到写不出去或者满 16 次（writeSpinCount）。

4 揽收太多，发送来不及时，爆仓，这个时候会出个告示牌：收不下了，最好过 2 天再来邮寄吧。

Netty 待写数据太多，超过一定的水位线（writeBufferWaterMark.high()），会将可写的标志位改成 false ，让应用端自己做决定要不要发送数据了。

主线

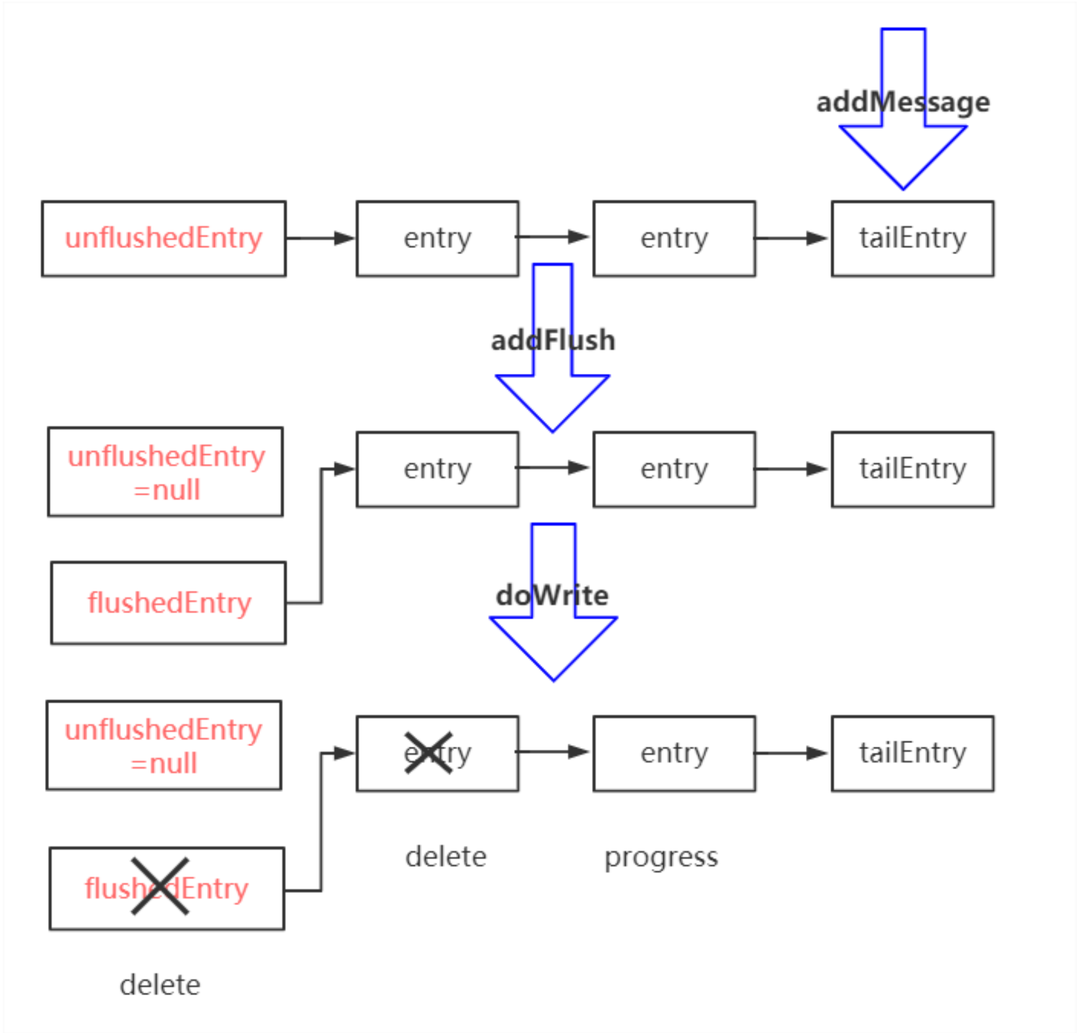
- Write - 写数据到 buffer：

  ChannelOutboundBuffer#addMessage

- Flush - 发送 buffer 里面的数据：

  AbstractChannel.AbstractUnsafe#flush

  - 准备数据: ChannelOutboundBuffer#addFlush
  - 发送：NioSocketChannel#doWrite



知识点

- 写的本质：
  - Single write：sun.nio.ch.SocketChannelImpl#write(java.nio.ByteBuffer)
  - gathering write：sun.nio.ch.SocketChannelImpl#write(java.nio.ByteBuffer[], int, int)
- 写数据写不进去时，会停止写，注册一个 OP_WRITE 事件，来通知什么时候可以写进去了。
- **OP_WRITE 不是说有数据可写，而是说可以写进去**，所以正常情况，不能注册，否则一直触发。

知识点

- 批量写数据时，如果尝试写的都写进去了，接下来会尝试写更多（maxBytesPerGatheringWrite）。
- 只要有数据要写，且能写，则一直尝试，直到 16 次（writeSpinCount），写 16 次还没有写完，就直接 schedule 一个 task 来继续写，而不是用注册写事件来触发，更简洁有力。
- 待写数据太多，超过一定的水位线（writeBufferWaterMark.high()），会将可写的标志位改成 false，让应用端自己做决定要不要继续写。
- channelHandlerContext.channel().write() ： 从 TailContext 开始执行；
  channelHandlerContext.write() : 从当前的 Context 开始。

源码剖析：断开连接

- 主线
- 源码演示
- 知识点

主线

- 多路复用器（Selector）接收到 OP_READ 事件：
- 处理 OP_READ 事件：NioSocketChannel.NioSocketChannelUnsafe.read()：
  - 接受数据
  - 判断接受的数据大小是否 < 0，如果是，说明是关闭，开始执行关闭：
    - 关闭 channel（包含 cancel 多路复用器的 key）。
    - 清理消息：不接受新信息，fail 掉所有 queue 中消息。
    - 触发 fireChannelInactive 和 fireChannelUnregistered 。

知识点

- 关闭连接本质：
  - java.nio.channels.spi.AbstractInterruptibleChannel#close
    - java.nio.channels.SelectionKey#cancel
- 要点：
  - 关闭连接，会触发 OP_READ 方法。读取字节数是 -1 代表关闭。
  - 数据读取进行时，强行关闭，触发 IO Exception，进而执行关闭。
  - Channel 的关闭包含了 SelectionKey 的 cancel 。

源码剖析：关闭服务

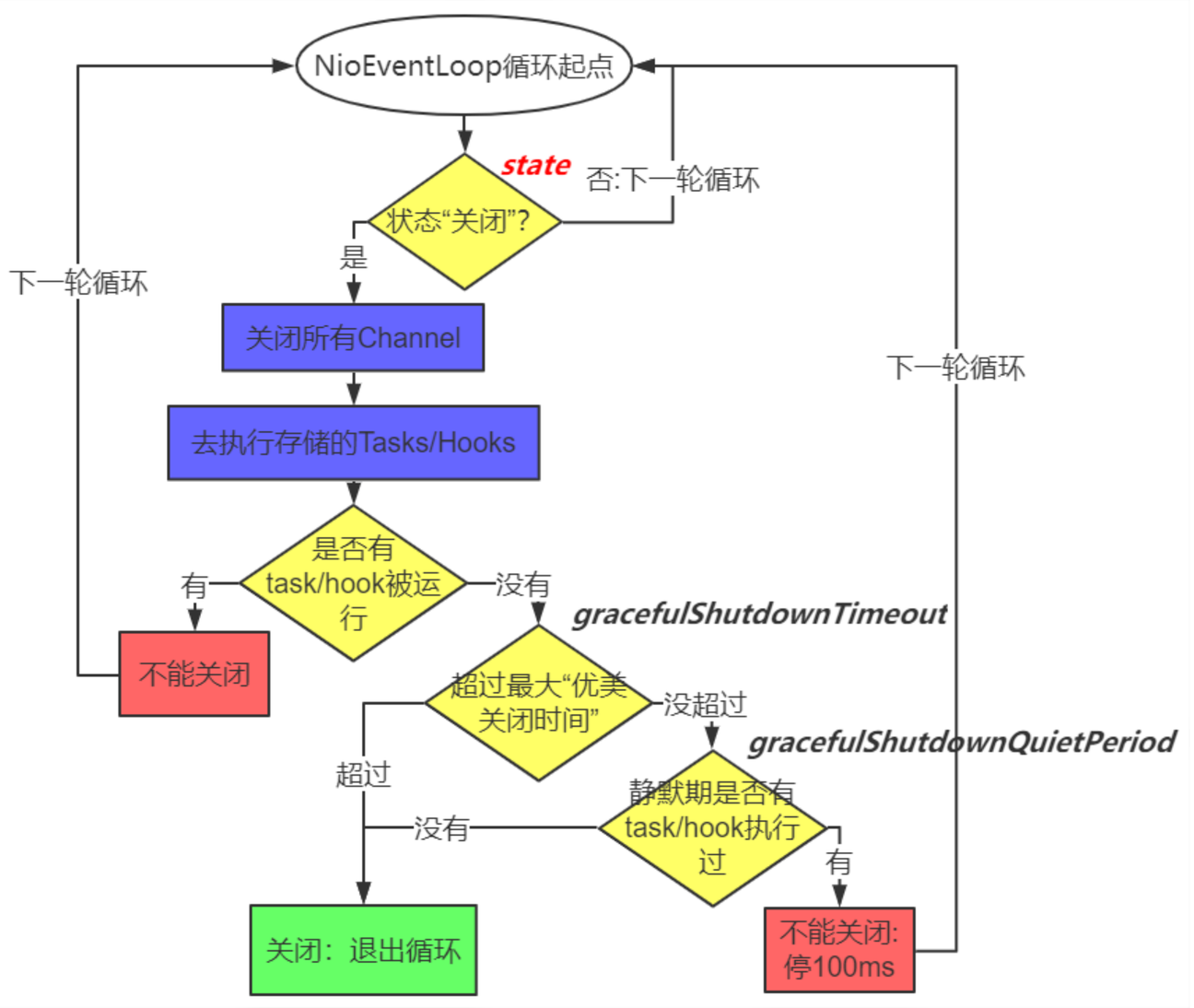
- 主线
- 源码演示
- 知识点

主线

- bossGroup.shutdownGracefully();
  workerGroup.shutdownGracefully();

  关闭所有 Group 中的 NioEventLoop:

  - 修改 NioEventLoop 的 State 标志位
  - NioEventLoop 判断 State 执行退出



知识点

- 关闭服务本质：
  - 关闭所有连接及 Selector ：
    - java.nio.channels.Selector#keys
      - java.nio.channels.spi.AbstractInterruptibleChannel#close
      - java.nio.channels.SelectionKey#cancel
    - selector.close()
  - 关闭所有线程：退出循环体 for (;;)

知识点

- 关闭服务要点：
  - 优雅（DEFAULT_SHUTDOWN_QUIET_PERIOD）
  - 可控（DEFAULT_SHUTDOWN_TIMEOUT）
  - 先不接活，后尽量干完手头的活（先关 boss 后关 worker：不是100%保证）