- NETTY

精尽 Netty 源码解析 —— Channel（七）之 close 操作

1. 概述

本文分享 Netty NIO Channel 关闭(**close**)操作的过程，分成客户端和服务端 Channel **两种**关闭：

- 客户端 NioSocketChannel
  - 客户端关闭 NioSocketChannel ，断开和服务端的连接。
  - 服务端关闭 NioSocketChannel ，断开和客户端的连接。
- 服务端 NioServerSocketChannel
  - 服务端关闭 NioServerSocketChannel ，取消端口绑定，关闭服务。

上面的关闭，可能是客户端/服务端主动关闭，也可能是异常关闭。

- 关于 NioSocketChannel 的关闭，在 「2. NioSocketChannel」 详细解析。
- 关于 NioServerSocketChannel 的关闭，在 「3. NioSocketChannel」 详细解析。

2. NioSocketChannel

文章目录

ioSocketChannel 通道。代码如下：

方法实际是 AbstractChannel 实现的。

close 事件在 pipeline 上传播。而 close 事件属于
点，使用 Unsafe 进行关闭。代码如下：

```
@Override
public final ChannelFuture close() {
    return tail.close();
}

// TailContext.java
@Override // FROM AbstractChannelHandlerContext.java 。因为 TailContext 继承 AbstractChannelHandler
public ChannelFuture close() {
```

```
    return close(newPromise());
}

// HeadContext.java
@Override
public void close(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception {
    unsafe.close(promise);
}
```

2.1 AbstractUnsafe#close

`AbstractUnsafe#close()` 方法，关闭 Channel 。代码如下：

```
@Override
public final void close(final ChannelPromise promise) {
    assertEventLoop();

    // 关闭
    close(promise, CLOSE_CLOSED_CHANNEL_EXCEPTION, CLOSE_CLOSED_CHANNEL_EXCEPTION, false);
}
```

```
 1: private void close(final ChannelPromise promise, final Throwable cause, final ClosedChannelExcept
 2:     // 设置 Promise 不可取消
 3:     if (!promise.setUncancellable()) {
 4:         return;
 5:     }
 6:
 7:     // 若关闭已经标记初始化
 8:     if (closeInitiated) {
 9:         // 关闭已经完成，直接通知 Promise 对象
10:         if (closeFuture.isDone()) {
11:             // Closed already.
12:             safeSetSuccess(promise);
13:         // 关闭未完成，添加到 Promise 通知
14:         } else if (!promise.isDone()) { // Only needed if no VoidChannelPromise.
15:             // This means close() was called before so we just register a listener and return
16:             closeFuture.addListener(new ChannelFutureListener() {
17:                 @Override
18:                 public void operationComplete(ChannelFuture future) throws Exception {
19:                     promise.setSuccess();
20:                 }
21:             });
22:         }
23:         return;
24:     }
25:
26:     // 标记关闭已经初始化
27:     closeInitiated = true;
28:
29:     // 获得 Channel 是否激活
30:     final boolean wasActive = isActive();
31:     // 标记 outboundBuffer 为空
32:     final ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
```

文章目录

```
33:     this.outboundBuffer = null; // Disallow adding any messages and flushes to outboundBuffer.
34:     // 执行准备关闭
35:     Executor closeExecutor = prepareToClose();
36:     // 若 closeExecutor 非空
37:     if (closeExecutor != null) {
38:         closeExecutor.execute(new Runnable() {
39:             @Override
40:             public void run() {
41:                 try {
42:                     // 在 closeExecutor 中，执行关闭
43:                     // Execute the close.
44:                     doClose0(promise);
45:                 } finally {
46:                     // 在 EventLoop 中，执行
47:                     // Call invokeLater so closeAndDeregister is executed in the EventLoop again!
48:                     invokeLater(new Runnable() {
49:                         @Override
50:                         public void run() {
51:                             if (outboundBuffer != null) {
52:                                 // 写入数据( 消息 )到对端失败，通知相应数据对应的 Promise 失败。
53:                                 // Fail all the queued messages
54:                                 outboundBuffer.failFlushed(cause, notify);
55:                                 // 关闭内存队列
56:                                 outboundBuffer.close(closeCause);
57:                             }
58:                             // 执行取消注册，并触发 Channel Inactive 事件到 pipeline 中
59:                             fireChannelInactiveAndDeregister(wasActive);
60:                         }
61:                     });
62:                 }
63:             }
```

文章目录

messages in all cases.

数据对应的 Promise 失败。

tify);

```
80:         // 正在 flush 中，在 EventLoop 中执行执行取消注册，并触发 Channel Inactive 事件到 pipeline 中
81:         if (inFlush0) {
82:             invokeLater(new Runnable() {
83:                 @Override
84:                 public void run() {
85:                     fireChannelInactiveAndDeregister(wasActive);
86:                 }
```

```
87:             });
88:         // 不在 flush 中，直接执行执行取消注册，并触发 Channel Inactive 事件到 pipeline 中
89:         } else {
90:             fireChannelInactiveAndDeregister(wasActive);
91:         }
92:     }
93: }
```

- 方法参数 `cause` 、 `closeCause` ，关闭的“原因”。对于 **close** 操作来说，无论是正常关闭，还是异常关闭，通过使用 **Exception** 来表示**来源**。在 AbstractChannel 类中，枚举了所有来源：

```
// AbstractChannel.java
private static final ClosedChannelException FLUSH0_CLOSED_CHANNEL_EXCEPTION = ThrowableUtil.unknow
        new ClosedChannelException(), AbstractUnsafe.class, "flush0()");
private static final ClosedChannelException ENSURE_OPEN_CLOSED_CHANNEL_EXCEPTION = ThrowableUtil.u
        new ClosedChannelException(), AbstractUnsafe.class, "ensureOpen(...)");
private static final ClosedChannelException CLOSE_CLOSED_CHANNEL_EXCEPTION = ThrowableUtil.unknowr
        new ClosedChannelException(), AbstractUnsafe.class, "close(...)");
private static final ClosedChannelException WRITE_CLOSED_CHANNEL_EXCEPTION = ThrowableUtil.unknowr
        new ClosedChannelException(), AbstractUnsafe.class, "write(...)");
private static final NotYetConnectedException FLUSH0_NOT_YET_CONNECTED_EXCEPTION = ThrowableUtil.u
        new NotYetConnectedException(), AbstractUnsafe.class, "flush0()");
```

- 第 2 至 5 行：调用 `ChannelPromise#setUncancellable()` 方法，设置 Promise 不可取消。
- 第 8 行：若 `AbstractChannel.closeInitiated` 为 `true` 时，表示关闭已经标记初始化，此时**可能**已经关闭完成。
  - 第 10 至 12 行：关闭**已经**完成，直接通知 Promise 对象。
  - 第 13 至 22 行：关闭**并未**完成，通过监听器回调通知 Promise 对象。

文章目录

析，胖友先跳到 「2.2

nsafe#prepareToClose」 中，我们已经看到如果

对象。

，执行 `#doClose0(promise)` 方法，执行关

新看下 「2.2

后续的任务。

详细解析，胖友先跳到 「2.4 doClose0」 中。

able cause, boolean notify) 方法，写入数据(消息)到对端失败，通知相应数据对应的 Promise 失败。详细解析，见《精尽 Netty 源码解析 —— Channel（五）之 flush 操作》 。

- 第 77 行：调用 `ChannelOutboundBuffer#close(Throwable cause)` 方法，关闭内存队列。详细解析，见《精尽 Netty 源码解析 —— Channel（五）之 flush 操作》 。
- 第 81 行：若 `inFlush0` 为 `true` ，**正在** flush 中，**在 EventLoop 中的线程中**，调用 `#fireChannelInactiveAndDeregister(boolean wasActive)` 方法，执行取消注册，并触发 Channel Inactive 事件到 pipeline 中。详细解析，见 「2.5 AbstractUnsafe#fireChannelInactiveAndDeregister」 中。

- 第 90 行：若 `inFlush0` 为 `false` ，**不在** flush 中，**直接**调用 `#fireChannelInactiveAndDeregister(boolean wasActive)` 方法，执行取消注册，并触发 Channel Inactive 事件到 pipeline 中。

2.2 NioSocketChannelUnsafe#prepareToClose

`NioSocketChannelUnsafe#prepareToClose()` 方法，执行准备关闭。代码如下：

```
1: @Override
2: protected Executor prepareToClose() {
3:     try {
4:         if (javaChannel().isOpen() && config().getSoLinger() > 0) {
5:             // We need to cancel this key of the channel so we may not end up in a eventloop spin
6:             // because we try to read or write until the actual close happens which may be later d
7:             // SO_LINGER handling.
8:             // See https://github.com/netty/netty/issues/4449
9:             doDeregister();
10:            // 返回 GlobalEventExecutor 对象
11:            return GlobalEventExecutor.INSTANCE;
12:        }
13:    } catch (Throwable ignore) {
14:        // Ignore the error as the underlying channel may be closed in the meantime and so
15:        // getSoLinger() may produce an exception. In this case we just return null.
16:        // See https://github.com/netty/netty/issues/4449
17:    }
18:    return null;
19: }
```

- 第 4 行：如果配置 `StandardSocketOptions.SO_LINGER` 大于 0 。让我们先来看下它的定义：


```
表示禁用该功能。

缓冲区全部发送到对端。
数据直接向对端发送RST包，对端收到复位错误。
延迟时间到或发送缓冲区中的数据发送完毕，若超时，
```

直到延迟时间到或发送缓冲区中的数据发送完

阻塞 EventLoop 的线程。所以，在【第 11 行】行**真正关闭** Channel 的操作的**执行器**(它也有一

胖友先跳到 「2.2

要调用 `#doDeregister()` 方法呢? 因为 `SO_LINGER` 大于 0 的，**真正关闭** Channel ，需要**阻塞**直到延迟时间到或发送缓冲区中的数据发送完毕。如果不取消该 Channel 的 `SelectionKey.OP_READ` 事件的感兴趣，就会不断触发读事件，导致 CPU 空轮询。为什么呢?在 Channel 关闭时，会**自动**触发 `SelectionKey.OP_READ` 事件。而且，会不断不断不断的触发，如果不进行取消 `SelectionKey.OP_READ` 事件的感兴趣。

- 😈 感叹一句，细思极恐啊，厉害了，Netty 。

- 第 11 行：如果开启 `SO_LINGER` 功能，返回 `GlobalEventExecutor.INSTANCE` 对象。
- 第 18 行：若果关闭 `SO_LINGER` 功能，返回 `null` 对象。
- 😈 胖友，调回 「2.1 AbstractUnsafe#close」 继续把。

2.3 AbstractUnsafe#doDeregister

`AbstractUnsafe#doDeregister()` 方法，执行取消注册。代码如下：

```
@Override
protected void doDeregister() throws Exception {
    eventLoop().cancel(selectionKey());
}
```

- 调用 `EventLoop#cancel(SelectionKey key)` 方法，取消 SelectionKey ，即相当于调用 `SelectionKey#cancel()` 方法。如此，对通道的读写等等 IO 就绪事件不再感兴趣，也不会做出相应的处理。

2.4 AbstractUnsafe#doClose0

`AbstractUnsafe#doClose0(ChannelPromise promise)` 方法，执行**真正的**关闭。代码如下：

```
1: private void doClose0(ChannelPromise promise) {
2:     try {
3:         // 执行关闭
4:         doClose();
5:         // 通知 closeFuture 关闭完成
6:         closeFuture.setClosed();
7:         // 通知 Promise 关闭成功
8:         safeSetSuccess(promise);
9:     } catch (Throwable t) {
10:        // 通知 closeFuture 关闭完成
11:        closeFuture.setClosed();
12:        // 通知 Promise 关闭异常
13:        safeSetFailure(promise, t);
14:    }
```


NioSocketChannel 对它的实现，胖友先跳到

ture 关闭完成。此处就会结束我们在 EchoClient

nise 关闭**成功**。此处就会回调我们对

下：

```
ctx.channel().close().addListener(new ChannelFutureListener() { // 我是一个萌萌哒监听器
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        System.out.println(Thread.currentThread() + "我会被唤醒");
    }
});
```

  - 哟哟哟，要被回调了。
- 若发生异常：
  - 第 11 行：调用 `CloseFuture#setClosed()` 方法，通知 `closeFuture` 关闭完成。
  - 第 13 行: 调用 `#safeSetFailure(promise, t)` 方法，通知 通知 Promise 关闭**异常**。

2.4.1 NioSocketChannel#doClose

`NioSocketChannel#doClose()` 方法，执行 Java 原生 NIO SocketChannel 关闭。代码如下：

```
1: @Override
2: protected void doClose() throws Exception {
3:     // 执行父类关闭方法
4:     super.doClose();
5:     // 执行 Java 原生 NIO SocketChannel 关闭
6:     javaChannel().close();
7: }
```

- 第 4 行：调用 `AbstractNioChannel#doClose()` 方法，执行**父类**关闭方法。代码如下：

```
@Override
protected void doClose() throws Exception {
    // 通知 connectPromise 异常失败
    ChannelPromise promise = connectPromise;
```



```
avoid the race against cancel().
PTION);
```

a 原生 NIO SocketChannel 关闭。

2.5 AbstractUnsafe#fireChannelInactiveAndDeregister

`AbstractUnsafe#fireChannelInactiveAndDeregister(boolean wasActive)` 方法，执行取消注册，并触发 Channel Inactive 事件到 pipeline 中。代码如下：

```
private void fireChannelInactiveAndDeregister(final boolean wasActive) {
    deregister(voidPromise() /** <1> **/, wasActive && !isActive() /** <2> **/);
}
```

```
 1: private void deregister(final ChannelPromise promise, final boolean fireChannelInactive) {
 2:     // 设置 Promise 不可取消
 3:     if (!promise.setUncancellable()) {
 4:         return;
 5:     }
 6:
 7:     // 不处于已经注册状态，直接通知 Promise 取消注册成功。
 8:     if (!registered) {
 9:         safeSetSuccess(promise);
10:         return;
11:     }
12:
13:     // As a user may call deregister() from within any method while doing processing in the Chann
14:     // we need to ensure we do the actual deregister operation later. This is needed as for examp
15:     // we may be in the ByteToMessageDecoder.callDecode(...) method and so still try to do proces
16:     // the old EventLoop while the user already registered the Channel to a new EventLoop. Withou
17:     // the deregister operation this could lead to have a handler invoked by different EventLoop
18:     // threads.
19:     //
20:     // See:
21:     // https://github.com/netty/netty/issues/4435
22:     invokeLater(new Runnable() {
23:         @Override
24:         public void run() {
25:             try {
26:                 // 执行取消注册
27:                 doDeregister();
28:             } catch (Throwable t) {
29:                 logger.warn("Unexpected exception occurred while deregistering a channel.", t);
30:             } finally {
31:                 if (fireChannelInactive) {
32:                     // 触发 Channel Inactive 事件到 pipeline 中
33:                     pipeline.fireChannelInactive();
34:                 }
35:                 // 触发 Channel Unregistered 事件到 pipeline 中
36:                 // Some transports like local and AIO does not allow the deregistration of
37:                 // an open channel.  Their doDeregister() calls close(). Consequently,
38:                 // close() will fire channelUnregistered() - no need to fire channelUnregistered, so che
39:                 if (isOpen()) {
40:                     if (fireChannelInactive) {
41:                         pipeline.fireChannelUnregistered();
42:                     }
43:                     // 触发 Channel Unregistered 事件到 pipeline 中
44:                     pipeline.fireChannelUnregistered();
45:                 }
46:
47:                 // 通知 Promise 取消注册成功。
48:                 safeSetSuccess(promise);
49:             }
50:         }
```

文章目录

```
51:     });
52: }
```

- `<1>` 处，传入 `#deregister(...)` 方法的第一个参数为 `unsafeVoidPromise` ，类型为 VoidChannelPromise **类**，表示需要通知 Promise 。为什么这么说呢？在 `#safeSetSuccess(promise)` 方法中，可以看到：

```
protected final void safeSetSuccess(ChannelPromise promise) {
    if (!(promise instanceof VoidChannelPromise) && !promise.trySuccess()) {
        logger.warn("Failed to mark a promise as success because it is done already: {}", promise);
    }
}
```

  - `!(promise instanceof VoidChannelPromise)` 代码块，表示排除 VoidChannelPromise 类型的 `promise` 。
- `<2>` 处，通过对比新老的 `active` 的值，判断是否 Channel 的状态是否从 Active 变成 Inactive 。
- 第 2 至 5 行：调用 `ChannelPromise#setUncancellable()` 方法，设置 Promise 不可取消。
- 第 7 至 11 行：不处于已经注册状态，直接通知 Promise 取消注册成功，并 `return` 返回。
  - 😈 在当前情况下， `registered = true` ，所以不符合条件。
- 第 22 行：调用 `#invokeLater(Runnable)` 方法，提交任务到 EventLoop 的线程中执行，以避免**一个** Channel 的 ChannelHandler 在**不同**的 EventLoop 或者线程中执行。详细的说明，可以看下【第 13 至 21 行】的英文说明。
  - 😈 实际从目前该方法的调用看下来，有可能不是从 EventLoop 的线程中调用。
- 第 27 行：调用 `AbstractUnsafe#doDeregister()` 方法，执行取消注册。在 「2.3 AbstractUnsafe#doDeregister」中，已经详细解析。
- 第 31 至 34 行：如果 `fireChannelInactive = true` ，调用 `ChannelPipeline#fireChannelInactive()` 方法，触发 Channel Inactive 事件到 pipeline 中。而 Channel Inactive 事件属于 Inbound 事件，所以会从 `head` 节点开始，最终传播到 `tail` 节点，目前并未执行什么逻辑，感兴趣的胖友，可以自己去看看。如果胖友业务上有需要，可以自己添加 ChannelHandler 进行处理。
- 第 40 至 42 行：标记为未注册。
- 第 44 行：调用 `ChannelPipeline#fireChannelUnregistered()` 方法，触发 Channel Unregistered 事件到

会从 `head` 节点开始，最终传播到 `tail` 节
胖友业务上有需要，可以自己添加 ChannelHandler

取消注册成功。

关闭 NioServerSocketChannel 通道。在具体的代码
的实现不同(对应 「2.4.1

erverChannel 关闭。代码如下：

```
protected void doClose() throws Exception {
    javaChannel().close();
}
```

- 调用 `SocketServerChannel#close()` 方法，执行 Java 原生 NIO SocketServerChannel 关闭。

那么可能会有胖友有疑惑了， `#close()` 方法的实现，99.99% 都相似，那么 NioSocketChannel 和 NioServerSocketChannel 差异的关闭逻辑怎么实现呢？答案其实很简单，通过给它们配置不同的 ChannelHandler 实现类即可。

4. Unsafe#closeForcibly

实际上，在 Unsafe 接口上定义了 `#closeForcibly()` 方法，英文注释如下：

```
/**
 * Closes the {@link Channel} immediately without firing any events.  Probably only useful
 * when registration attempt failed.
 */
void closeForcibly();
```

- 立即关闭 Channel ，并且不触发 pipeline 上的任何事件。
- 仅仅用于 Channel 注册到 EventLoop 上失败的情况下。😈 这也就是为什么 `without firing any events` 的原因啦。

AbstractUnsafe 对该接口方法，实现代码如下：

```
@Override
public final void closeForcibly() {
    assertEventLoop();

    try {
        doClose();
    } catch (Exception e) {
        logger.warn("Failed to close a channel.", e);
    }
```

文章目录

Java 原生 NIO SocketServerChannel 或

的事件。

件的就绪，在调用客户端对应在服务端的
客户端的逻辑。在 Netty 的实现，在

```
// 设置最后读取字节数
allocHandle.lastBytesRead(doReadBytes(byteBuf));
// 如果最后读取的字节为小于 0 ，说明对端已经关闭
close = allocHandle.lastBytesRead() < 0;

// 关闭客户端的连接
if (close) {
```

```
    closeOnRead(pipeline);
}
```

- `<1>` 处，读取客户端的 SocketChannel 返回 **-1** ，说明客户端已经关闭。
- `<2>` 处，调用 `#closeOnRead(ChannelPipeline pipeline)` 方法，关闭客户端的连接。代码如下：

```
 1: private void closeOnRead(ChannelPipeline pipeline) {
 2:     if (!isInputShutdown0()) {
 3:         // 开启连接半关闭
 4:         if (isAllowHalfClosure(config())) {
 5:             // 关闭 Channel 数据的读取
 6:             shutdownInput();
 7:             // 触发 ChannelInputShutdownEvent.INSTANCE 事件到 pipeline 中
 8:             pipeline.fireUserEventTriggered(ChannelInputShutdownEvent.INSTANCE);
 9:         } else {
10:             close(voidPromise());
11:         }
12:     } else {
13:         // 标记 inputClosedSeenErrorOnRead 为 true
14:         inputClosedSeenErrorOnRead = true;
15:         // 触发 ChannelInputShutdownEvent.INSTANCE 事件到 pipeline 中
16:         pipeline.fireUserEventTriggered(ChannelInputShutdownReadComplete.INSTANCE);
17:     }
18: }
```

- 第 2 行：调用 `NioSocketChannel#isInputShutdown0()` 方法，判断是否关闭 Channel 数据的读取。代码如下：

```
// NioSocketChannel.java
@Override
                                                    || !isActive();
                                                    ction is closed.
                                                    shutdown
 * @since 1.4
 * @see #shutdownInput
 */
public boolean isInputShutdown() {
    return shutIn;
}
```

- 😈 注意看下英文注释。

文章目录

- <1> 第 4 行：调用 AbstractNioByteChannel#isAllowHalfClosure() 方法，判断是否开启连接**半关闭**的功能。代码如下：

```
// AbstractNioByteChannel.java
private static boolean isAllowHalfClosure(ChannelConfig config) {
    return config instanceof SocketChannelConfig &&
            ((SocketChannelConfig) config).isAllowHalfClosure();
}
```

  - 可通过 ALLOW_HALF_CLOSURE 配置项开启。
    - Netty 参数，一个连接的远端关闭时本地端是否关闭，默认值为 false 。
    - 值为 false 时，连接自动关闭。
    - 值为 true 时，触发 ChannelInboundHandler 的 #userEventTriggered() 方法，事件 ChannelInputShutdownEvent 。

- <1.1> 第 6 行：调用 NioSocketChannel#shutdownInput() 方法，关闭 Channel 数据的读取。代码如下：

```
@Override
public ChannelFuture shutdownInput() {
    return shutdownInput(newPromise());
}

@Override
public ChannelFuture shutdownInput(final ChannelPromise promise) {
    EventLoop loop = eventLoop();
    if (loop.inEventLoop()) {
        shutdownInput0(promise);
    } else {
        loop.execute(new Runnable() {
            @Override
            public void run() {
```

文章目录


```
romise) {
                promise.setFailure(t);
            }
        }

        private void shutdownInput0() throws Exception {
            // 调用 Java NIO Channel 的 shutdownInput 方法
            if (PlatformDependent.javaVersion() >= 7) {
                javaChannel().shutdownInput();
```

```
    } else {
        javaChannel().socket().shutdownInput();
    }
}
```

  - 核心是，调用 Java NIO Channel 的 shutdownInput 方法。
- `<1.1>` 第 8 行：调用 `ChannelPipeline#fireUserEventTriggered(Object event)` 方法，触发 `ChannelInputShutdownEvent.INSTANCE` 事件到 pipeline 中。关于这个事件，胖友可以看看《netty 处理远程主机强制关闭一个连接》。
- `<1.2>` 第 9 至 11 行：调用 `#close(Promise)` 方法，关闭客户端的 Channel 。后续的，就是「2. NioSocketChannel」中。

- 第 12 至 17 行：
  - 第 14 行：标记 `inputClosedSeenErrorOnRead` 为 `true` 。原因如下：

```
/**
 * 通道关闭读取，又错误读取的错误的标识
 *
 * 详细见 https://github.com/netty/netty/commit/ed0668384b393c3502c2136e3cc412a5c8c9056e 提交
 */
private boolean inputClosedSeenErrorOnRead;
```

    - 如下是提交的说明：

```
AbstractNioByteChannel will detect that the remote end of the socket has
been closed and propagate a user event through the pipeline. However if
the user has auto read on, or calls read again, we may propagate the
same user events again. If the underlying transport continuously
notifies us that there is read activity this will happen in a spin loop
which consumes unnecessary CPU
```

文章目录

in when half closed #7801》提供的示

NioByteUnsafe#read() 方法中，会主
码如下：

```
// AbstractNioByteUnsafe.java
public final void read() {
    final ChannelConfig config = config();
    // 若 inputClosedSeenErrorOnRead = true ，移除对 SelectionKey.OP_READ 事件的感兴趣
    if (shouldBreakReadReady(config)) {
        clearReadPending(); // 移除对 SelectionKey.OP_READ 事件的感兴趣
        return;
    }

    // ... 省略其他代码。
}

// AbstractNioByteChannel.java
final boolean shouldBreakReadReady(ChannelConfig config) {
    return isInputShutdown0() && (inputClosedSeenErrorOnRead || !isAllowHalfClosure(
}
```

- x

- 第 16 行：调用 `ChannelPipeline#fireUserEventTriggered(Object event)` 方法，触发 `ChannelInputShutdownEvent.INSTANCE` 事件到 pipeline 中。

666. 彩蛋

比想象中简单的文章。但是，卡了比较久的时间。主要是针对 《High CPU usage with SO_LINGER and sudden connection close (4.0.26.Final+) #4449》 的讨论，中间请教了基友闪电侠和表弟普架。

痛并快乐的过程。如果英文好一点，相信解决的过程，可能更加愉快一些把。

文章目录