- NETTY

精尽 Netty 源码解析 —— ChannelPipeline（六）之异常事件的传播

1. 概述

在《精尽 Netty 源码解析 —— ChannelPipeline（四）之 Outbound 事件的传播》和《精尽 Netty 源码解析 —— ChannelPipeline（五）之 Inbound 事件的传播》中，我们看到 Outbound 和 Inbound 事件在 pipeline 中的传播逻辑。但是，无可避免，传播的过程中，可能会发生异常，那是怎么处理的呢？

文章目录

2. notifyOutboundHandlerException

码如下：

```
// AbstractChannelHandlerContext.java

private void invokeBind(SocketAddress localAddress, ChannelPromise promise) {
    if (invokeHandler()) { // 判断是否符合的 ChannelHandler
        try {
            // 调用该 ChannelHandler 的 bind 方法 <1>
            ((ChannelOutboundHandler) handler()).bind(this, localAddress, promise);
        } catch (Throwable t) {
            notifyOutboundHandlerException(t, promise); // 通知 Outbound 事件的传播，发生异常 <2>
        }
    } else {
        // 跳过，传播 Outbound 事件给下一个节点
        bind(localAddress, promise);
    }
}
```

- 在 `<1>` 处，调用 `ChannelOutboundHandler#bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise)` 方法**发生异常**时，会在 `<2>` 处调用 `AbstractChannelHandlerContext#notifyOutboundHandlerException(Throwable cause, ChannelPromise promise)` 方法，通知 Outbound 事件的传播，发生异常。
- 其他 Outbound 事件，大体的代码也是和 `#invokeBind(SocketAddress localAddress, ChannelPromise promise)` 是一致的。如下图所示：

`AbstractChannelHandlerContext#notifyOutboundHandlerException(Throwable cause, ChannelPromise promise)` 方法，通知 Outbound 事件的传播，发生异常。代码如下：

```
private static void notifyOutboundHandlerException(Throwable cause, ChannelPromise promise) {
    // Only log if the given promise is not of type VoidChannelPromise as tryFailure(...) is expected
    // false
    e(promise, cause, promise instanceof VoidChannelPromise ? null :
```

文章目录

onUtil#tryFailure(Promise<?> p, Throwable cause,
nd 事件对应的 Promise 对应的监听者们。代码如下：

```
public static void tryFailure(Promise<?> p, Throwable cause, InternalLogger logger) {
    if (!p.tryFailure(cause) && logger != null) {
        Throwable err = p.cause();
        if (err == null) {
            logger.warn("Failed to mark a promise as failure because it has succeeded already: {}"
        } else {
            logger.warn(
                    "Failed to mark a promise as failure because it has failed already: {}, unnoti
                    p, ThrowableUtil.stackTraceToString(err), cause);
        }
    }
}
```

- 以 bind 事件来举一个监听器的例子。代码如下：

```
ChannelFuture f = b.bind(PORT).addListener(new ChannelFutureListener() { // <1> 监听器就是我!
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        System.out.println("异常: " + future.casue());
    }
}).sync();
```

- `<1>` 处的监听器，就是示例。当发生异常时，就会通知该监听器，对该异常做进一步**自定义**的处理。**也就是说，该异常不会在 pipeline 中传播。**

- 我们再来看看怎么通知监听器的源码实现。调用 `DefaultPromise#tryFailure(Throwable cause)` 方法，通知 Promise 的监听器们，发生了异常。代码如下：

```
@Override
public boolean tryFailure(Throwable cause) {
    if (setFailure0(cause)) { // 设置 Promise 的结果
        // 通知监听器
        notifyListeners();
        // 返回成功
        return true;
    }
    // 返回失败
    return false;
}
```

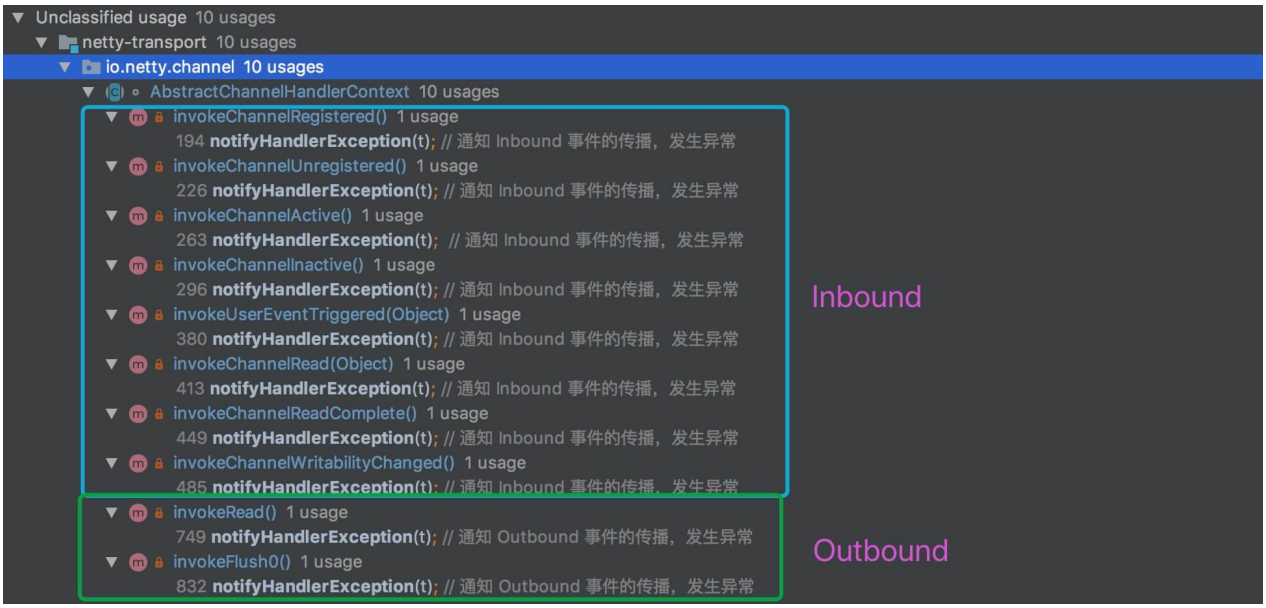
  - 若 `DefaultPromise#setFailure0(Throwable cause)` 方法，设置 Promise 的结果为方法传入的异常。但是有可能会传递失败，例如说，Promise 已经被设置了结果。
  - 如果该方法返回 `false` 通知 Promise 失败，那么 `PromiseNotificationUtil#tryFailure(Promise<?> p, Throwable cause, InternalLogger logger)` 方法的后续，就会使用 `logger` 打印错误日志。

ion

举例子，代码如下：

```
        // 查找匹配符合的 ChannelHandler
        try {
            // 调用该 ChannelHandler 的 Channel active 方法 <1>
            ((ChannelInboundHandler) handler()).channelActive(this);
        } catch (Throwable t) {
            notifyHandlerException(t);  // 通知 Inbound 事件的传播，发生异常 <2>
        }
    } else {
        // 跳过，传播 Inbound 事件给下一个节点
        fireChannelActive();
    }
}
```

- 在 `<1>` 处，调用 `ChannelInboundHandler#channelActive(ChannelHandlerContext ctx)` 方法**发生异常**时，会在 `<2>` 处调用 `AbstractChannelHandlerContext#notifyHandlerException(Throwable cause)` 方法，通知 Inbound 事件的传播，发生异常。
- 其他 Inbound 事件，大体的代码也是和 `#invokeChannelActive()` 是一致的。如下图所示：



- 😈 **注意，笔者在写的时候，突然发现 Outbound 事件中的 read 和 flush 的异常处理方式和 Inbound 事件是一样的。**
- 😈 **注意，笔者在写的时候，突然发现 Outbound 事件中的 read 和 flush 的异常处理方式和 Inbound 事件是一样的。**
- 😈 **注意，笔者在写的时候，突然发现 Outbound 事件中的 read 和 flush 的异常处理方式和 Inbound 事件是一样**

文章目录

HandlerException(Throwable cause) 方法，通知 Inbound 事件的传播，

```
Throwable cause) {
    // <1> 如果是在 `ChannelHandler#exceptionCaught(ChannelHandlerContext ctx, Throwable cause)` 方法中
    if (inExceptionCaught(cause)) {
        if (logger.isWarnEnabled()) {
            logger.warn(
                    "An exception was thrown by a user handler " +
                            "while handling an exceptionCaught event", cause);
        }
        return;
    }

    // <2> 在 pipeline 中，传播 Exception Caught 事件
    invokeExceptionCaught(cause);
}
```

- <1> 处，调用 AbstractChannelHandlerContext#inExceptionCaught(Throwable cause) 方法，如果是在 ChannelHandler#exceptionCaught(ChannelHandlerContext ctx, Throwable cause) 方法中，**发生异常**，仅打印错误日志，**并 return 返回**。否则会形成死循环。代码如下：

```
private static boolean inExceptionCaught(Throwable cause) {
    do {
        StackTraceElement[] trace = cause.getStackTrace();
        if (trace != null) {
            for (StackTraceElement t : trace) { // 循环 StackTraceElement
                if (t == null) {
                    break;
```

```
                }
                if ("exceptionCaught".equals(t.getMethodName())) { // 通过方法名判断
                    return true;
                }
            }
        }
        cause = cause.getCause();
    } while (cause != null); // 循环异常的 cause() ，直到到没有

    return false;
}
```

- 通过 StackTraceElement 的方法名来判断，是不是 `ChannelHandler#exceptionCaught(ChannelHandlerContext ctx, Throwable cause)` 方法。

- `<2>` 处，调用 `AbstractChannelHandlerContext#invokeExceptionCaught(Throwable cause)` 方法，在 pipeline 中，传递 Exception Caught 事件。在下文中，我们会看到，和《精尽 Netty 源码解析 —— ChannelPipeline（五）之 Inbound 事件的传播》的逻辑(`AbstractChannelHandlerContext#invokeChannelActive()`)是**一致**的。

- 比较特殊的是，Exception Caught 事件在 pipeline 的起始节点，不是 `head` 头节点，而是**发生异常的当前节点开**
  播的 Inbound **xxx** 事件，在发生异常后，转化成 **Exception Caught** 事

文章目录

中的传播过程中，一直没有处理掉该异常的节点，最终会到达尾节点
annelHandlerContext ctx, Throwable cause)` 方法的实现，代码如

```
@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
    onUnhandledInboundException(cause);
}
```

- 在方法内部，会调用 `DefaultChannelPipeline#onUnhandledInboundException(Throwable cause)` 方法，代码如下：

```
/**
 * Called once a {@link Throwable} hit the end of the {@link ChannelPipeline} without bee
 * in {@link ChannelHandler#exceptionCaught(ChannelHandlerContext, Throwable)}.
 */
protected void onUnhandledInboundException(Throwable cause) {
    try {
        logger.warn(
                "An exceptionCaught() event was fired, and it reached at the tail of the
                        "It usually means the last handler in the pipeline did not handle
                cause);
    } finally {
        ReferenceCountUtil.release(cause);
    }
}
```

- 打印**告警**日志，并调用 `ReferenceCountUtil#release(Throwable)` 方法，释放需要释放的资源。
- 从英文注释中，我们也可以看到，这种情况出现在**使用者**未定义合适的 ChannelHandler 处理这种异常，所以对于这种情况下， `tail` 节点只好打印**告警**日志。

- 实际使用时，笔者建议胖友一定要定义 ExceptionHandler ，能够处理掉所有的异常，而不要使用到 `tail` 节点的异常处理。😈
- 好基友【闪电侠】对尾节点 `tail` 做了很赞的总结

> 总结一下，tail 节点的作用就是结束事件传播，并且对一些重要的事件做一些善意提醒

666. 彩蛋

推荐阅读文章：

- 闪电侠 《netty 源码分析之 pipeline(二)》

文章目录

次