- NETTY

Pipeline（五）之 Inbound

文章目录

顾下 Inbound 事件的定义：

的编号。

- [x] B01：Inbound 事件是【通知】事件, 当某件事情已经就绪后, 通知上层.

> 老芳芳：B01 = B02 + B03

- [x] B02：Inbound 事件发起者是 Unsafe
- [x] B03：Inbound 事件的处理者是 TailContext, 如果用户没有实现自定义的处理方法, 那么Inbound 事件默认的处理者是 TailContext, 并且其处理方法是空实现.
- [x] B04：Inbound 事件在 Pipeline 中传输方向是 `head（头）-> tail（尾）`
- [x] B05：在 ChannelHandler 中处理事件时, 如果这个 Handler 不是最后一个 Handler, 则需要调用 `ctx.fireIN_EVT` (例如 `ctx.fireChannelActive`) 将此事件继续传播下去. 如果不这样做, 那么此事件的传播会提前终止.
- [x] B06：Inbound 事件流: `Context.fireIN_EVT -> Connect.findContextInbound -> nextContext.invokeIN_EVT -> nextHandler.IN_EVT -> nextContext.fireIN_EVT`

Outbound 和 Inbound 事件十分的镜像，所以，接下来我们来跟着的代码，和《精尽 Netty 源码解析 —— ChannelPipeline（四）之 Outbound 事件的传播》会非常相似。

2. ChannelInboundInvoker

在 `io.netty.channel.ChannelInboundInvoker` 接口中，定义了所有 Inbound 事件对应的方法：

```
ChannelInboundInvoker fireChannelRegistered();
ChannelInboundInvoker fireChannelUnregistered();

ChannelInboundInvoker fireChannelActive();
```

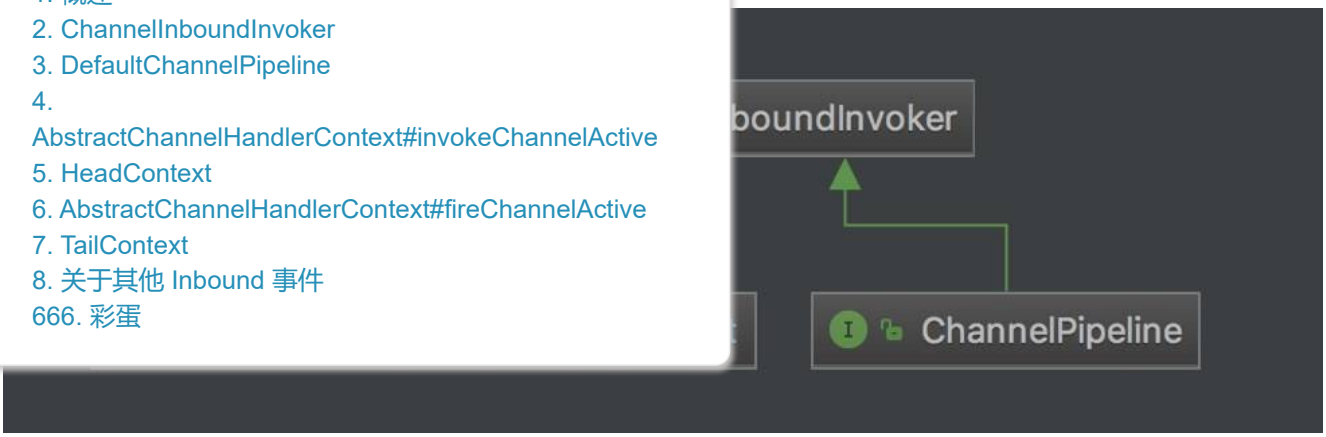
```
ChannelInboundInvoker fireChannelInactive();

ChannelInboundInvoker fireExceptionCaught(Throwable cause);

ChannelInboundInvoker fireUserEventTriggered(Object event);

ChannelInboundInvoker fireChannelRead(Object msg);
ChannelInboundInvoker fireChannelReadComplete();
```

文章目录



- 我们可以看到类图，有 ChannelPipeline、AbstractChannelHandlerContext 都继承/实现了该接口。那这意味着什么呢？我们继续往下看。
- 相比来说，Channel 实现了 ChannelOutboundInvoker 接口，但是**不实现** ChannelInboundInvoker 接口。

在 《精尽 Netty 源码解析 —— 启动（一）之服务端》 中，我们可以看到 Inbound 事件的其中之一 **fireChannelActive** ，本文就以 **fireChannelActive** 的过程，作为示例。调用栈如下：

- `AbstractUnsafe#bind(final SocketAddress localAddress, final ChannelPromise promise)` 方法，代码如下：

```
@Override
public final void bind(final SocketAddress localAddress, final ChannelPromise promise) {
    // 判断是否在 EventLoop 的线程中。
    assertEventLoop();

    // ... 省略部分代码

    // 记录 Channel 是否激活
    boolean wasActive = isActive();

    // 绑定 Channel 的端口
    doBind(localAddress);
```

```
    // 若 Channel 是新激活的，触发通知 Channel 已激活的事件。
    if (!wasActive && isActive()) {
        invokeLater(new Runnable() {
            @Override
            public void run() {
                pipeline.fireChannelActive(); // <1>
            }
        });
```

文章目录

) 方法。

事件的定义 B02 。

Active() 方法的具体实现。

DefaultChannelPipeline#fireChannelActive() 方法的实现，代码如下：

```
@Override
public final ChannelPipeline fireChannelActive() {
    AbstractChannelHandlerContext.invokeChannelActive(head);
    return this;
}
```

- 在方法内部，会调用 AbstractChannelHandlerContext#invokeChannelActive(final AbstractChannelHandlerContext next) 方法，而方法参数是 head ，**这符合 Inbound 事件的定义 B04** 。
  - 实际上，HeadContext 的该方法，继承自 AbstractChannelHandlerContext 抽象类，而 AbstractChannelHandlerContext 实现了 ChannelInboundInvoker 接口。*从这里可以看出，对于 ChannelInboundInvoker 接口方法的实现，ChannelPipeline 对它的实现，会调用 AbstractChannelHandlerContext 的对应方法*(有一点绕，胖友理解下)。

4. AbstractChannelHandlerContext#invokeChannelActive

AbstractChannelHandlerContext#invokeChannelActive(final AbstractChannelHandlerContext next) **静态**方法，代码如下：

```
 1: static void invokeChannelActive(final AbstractChannelHandlerContext next) {
 2:     // 获得下一个 Inbound 节点的执行器
 3:     EventExecutor executor = next.executor();
 4:     // 调用下一个 Inbound 节点的 Channel active 方法
 5:     if (executor.inEventLoop()) {
 6:         next.invokeChannelActive();
 7:     } else {
 8:         executor.execute(new Runnable() {
 9:             @Override
10:             public void run() {
```

```
11:                 next.invokeChannelActive();
12:             }
13:         });
14:     }
15: }
```

- 方法参数 `next` ，下一个 Inbound 节点。
- 第 3 行：调用 `AbstractChannelHandlerContext#executor()` 方法，获得下一个 Inbound 节点的执行器。

ChannelActive() 方法，调用下一个 Inbound 节点

个 `next` 方法参数为 `head` (HeadContext) 节

nnelHandler

active 方法
.channelActive(this);

```
 6:         } catch (Throwable t) {
 7:             notifyHandlerException(t);  // 通知 Inbound 事件的传播，发生异常
 8:         }
 9:     } else {
10:         // 跳过，传播 Inbound 事件给下一个节点
11:         fireChannelActive();
12:     }
13: }
```

- 第 2 行：调用 `#invokeHandler()` 方法，判断是否符合的 ChannelHandler 。
- 第 9 至 12 行：若是**不符合**的 ChannelHandler ，则**跳过**该节点，调用 `AbstractChannelHandlerContext#fireChannelActive(` 方法，传播 Inbound 事件给下一个节点。详细解析，见 「6. AbstractChannelHandlerContext#fireChannelActive」 。
- 第 2 至 8 行：若是**符合**的 ChannelHandler ：
  - 第 5 行：调用 ChannelHandler 的 `#channelActive(ChannelHandlerContext ctx)` 方法，处理 Channel active 事件。
    - 😈 实际上，此时节点的数据类型为 DefaultChannelHandlerContext 类。若它被认为是 Inbound 节点，那么他的处理器的类型会是 **ChannelInboundHandler** 。而 `io.netty.channel.ChannelInboundHandler` 类似 ChannelInboundInvoker ，定义了对每个 Inbound 事件的处理。代码如下：

```
void channelRegistered(ChannelHandlerContext ctx) throws Exception;
void channelUnregistered(ChannelHandlerContext ctx) throws Exception;

void channelActive(ChannelHandlerContext ctx) throws Exception;
void channelInactive(ChannelHandlerContext ctx) throws Exception;

void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception

void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception;
void channelReadComplete(ChannelHandlerContext ctx) throws Exception;
```

```
    void channelWritabilityChanged(ChannelHandlerContext ctx) throws Exception;

    @Override
    @SuppressWarnings("deprecation")
    void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exceptio
```

- 胖友自己对比下噢。

- 如果节点的 ChannelInboundHandler#channelActive(ChannelHandlerContext ctx 方法 fireChannelActive() 方法，就不会传 **的定义 B05** 。可能有点绕，我们来看下


```
lInboundHandler, ChannelOutboundHand

Context ctx) throws Exception {

        {
            logger.log(internalLevel, format(ctx, "ACTIVE"));
        }
        // 传递 Channel active 事件，给下一个节点
        ctx.fireChannelActive(); // <1>
    }
}
```

- 如果把 `<1>` 处的代码去掉，Channel active 事件 事件将不会传播给下一个节点！！！ **一定要注意。**
- 这块的逻辑非常重要，如果胖友觉得很绕，一定要自己多调试 + 调试 + 调试。

- 第 7 行：如果发生异常，调用 `#notifyHandlerException(Throwable)` 方法，通知 Inbound 事件的传播，发生异常。详细解析，见《精尽 Netty 源码解析 —— ChannelPipeline（六）之异常事件的传播》。

5. HeadContext

`HeadContext#invokeChannelActive()` 方法，代码如下：

```
@Override
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    // 传播 Channel active 事件给下一个 Inbound 节点 <1>
    ctx.fireChannelActive();

    // 执行 read 逻辑 <2>
    readIfIsAutoRead();
}
```

- `<1>` 处，调用 `AbstractChannelHandlerContext#fireChannelActive()` 方法，传播 Channel active 事件给下一个 Inbound 节点。详细解析，见「6. AbstractChannelHandlerContext」中。
- `<2>` 处，调用 `HeadContext#readIfIsAutoRead()` 方法，执行 read 逻辑。代码如下：

```
// HeadContext.java
private void readIfIsAutoRead() {
    if (channel.config().isAutoRead()) {
        channel.read();
    }
}

// AbstractChannel.java
@Override
```

文章目录

ne 传递该 **read** OutBound 事件，最终调用

- 后续的逻辑，便是《精尽 Netty 源码分析 —— 启动（一）之服务端》的 3.13.3 beginRead 小节，胖友可以自己再去回顾下。
- 这里说的是 **OutBound** 事件，不是本文的 InBound 事件。所以，胖友不要搞混哈。只能说是对《精尽 Netty 源码分析 —— 启动（一）之服务端》的 3.13.3 beginRead 小节的补充。

6. AbstractChannelHandlerContext#fireChannelActive

`AbstractChannelHandlerContext#fireChannelActive()` 方法，代码如下：

```
@Override
public ChannelHandlerContext fireChannelActive() {
    // 获得下一个 Inbound 节点的执行器
    // 调用下一个 Inbound 节点的 Channel active 方法
    invokeChannelActive(findContextInbound());
    return this;
}
```

- 【重要】调用 `AbstractChannelHandlerContext#findContextInbound()` 方法，获得下一个 Inbound 节点的执行器。代码如下：

```
private AbstractChannelHandlerContext findContextInbound() {
    // 循环，向后获得一个 Inbound 节点
    AbstractChannelHandlerContext ctx = this;
    do {
        ctx = ctx.next;
    } while (!ctx.inbound);
    return ctx;
}
```

- 循环，**向后**获得一个 Inbound 节点。

- 循环，**向后**获得一个 Inbound 节点。
- 循环，**向后**获得一个 Inbound 节点。
- 😈 重要的事情说三遍，对于 Inbound 事件的传播，是从 pipeline 的头部到尾部，**这符合 Inbound 事件的定义 B04** 。

- 调用 `AbstractChannelHandlerContext#invokeChannelActive(AbstractChannelHandlerContext)` **静态**方法，调用下一个 Inbound 节点的 Channel active 方法。即，又回到 「4. AbstractChannelHandlerContext#invokeChannelActive」 的开头。

本小节的整个代码实现，**就是 Inbound 事件的定义 B06** 的体现。而随着 Inbound 事件在节点不断从 pipeline 的头部到尾部的

文章目录

法，代码如下：

```
rows Exception {
```

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

```
/**
 * Called once the {@link ChannelInboundHandler#channelActive(ChannelHandlerContext)}event hit
 * the end of the {@link ChannelPipeline}.
 */
protected void onUnhandledInboundChannelActive() {
}
```

- 该方法是个**空**方法，**这符合 Inbound 事件的定义 B03** 。
- 至此，整个 pipeline 的 Inbound 事件的传播结束。

8. 关于其他 Inbound 事件

本文暂时只分享了 **firecChannelActive** 这个 Inbound 事件。剩余的其他事件，胖友可以自己进行调试和理解。例如：**fireChannelRegistered** 事件，并且结合 《精尽 Netty 源码分析 —— 启动（一）之服务端》 一文。

666. 彩蛋

推荐阅读文章：

- 闪电侠 《netty 源码分析之 pipeline(二)》

感觉上来说，Inbound 事件的传播，比起 Outbound 事件的传播，会相对“绕”一点点。简化来说，实际大概是如下：

```
Unsafe 开始 => DefaultChannelPipeline#fireChannelActive

=> HeadContext#invokeChannelActive => DefaultChannelHandlerContext01#fireChannelActive

=> DefaultChannelHandlerContext01#invokeChannelActive => DefaultChannelHandlerContext02#fireChannelAct
```

```
...
=> DefaultChannelHandlerContext99#fireChannelActive => TailContext#fireChannelActive

=> TailContext#invokeChannelActive => 结束
```

笔者觉得可能解释的也有点“绕”，如果不理解或者有地方写的有误解，欢迎来叨叨，以便我们能一起优化这篇文章。

文章目录