- NETTY

精尽 Netty 源码解析 —— ChannelPipeline（二）之添加 ChannelHandler

1. 概述

本文我们来分享，**添加** ChannelHandler 到 pipeline 中的代码具体实现。

在 《精尽 Netty 源码解析 —— ChannelPipeline（一）之初始化》 中，我们看到 ChannelPipeline 定义了一大堆**添加** ChannelHandler 的接口方法：

```
ChannelPipeline addFirst(String name, ChannelHandler handler);
ChannelPipeline addFirst(EventExecutorGroup group, String name, ChannelHandler handler);
ChannelPipeline addLast(String name, ChannelHandler handler);
ChannelPipeline addLast(EventExecutorGroup group, String name, ChannelHandler handler);
ChannelPipeline addBefore(String baseName, String name, ChannelHandler handler);
ChannelPipeline addBefore(EventExecutorGroup group, String baseName, String name, ChannelHandler handl
ChannelPipeline addAfter(String baseName, String name, ChannelHandler handler);
ChannelPipeline addAfter(EventExecutorGroup group, String baseName, String name, ChannelHandler handle
ChannelPipeline addFirst(ChannelHandler... handlers);
ChannelPipeline addFirst(EventExecutorGroup group, ChannelHandler... handlers);
ChannelPipeline addLast(ChannelHandler... handlers);
ChannelPipeline addLast(EventExecutorGroup group, ChannelHandler... handlers);
```

- 考虑到实际当中，我们使用 `#addLast(ChannelHandler... handlers)` 方法较多，所以本文只分享这个方法的具体实现。

2. addLast

`#addLast(ChannelHandler... handlers)` 方法，添加任意数量的 ChannelHandler 对象。代码如下：

```
@Override
public final ChannelPipeline addLast(ChannelHandler... handlers) {
    return addLast(null, handlers);
}

@Override
public final ChannelPipeline addLast(EventExecutorGroup executor, ChannelHandler... handlers) {
    if (handlers == null) {
        throw new NullPointerException("handlers");
```

文章目录

<1>

oup group, String name, ChannelHandler handler) 方法，添加

#addLast(EventExecutorGroup group, String name, ChannelHandler handler) 方法，代码如下：

```
1: @Override
2: @SuppressWarnings("Duplicates")
3: public final ChannelPipeline addLast(EventExecutorGroup group, String name, ChannelHandler handler
4:     final AbstractChannelHandlerContext newCtx;
5:     synchronized (this) { // 同步，为了防止多线程并发操作 pipeline 底层的双向链表
6:         // 检查是否有重复 handler
7:         checkMultiplicity(handler);
8:
9:         // 创建节点名
10:        // 创建节点
11:        newCtx = newContext(group, filterName(name, handler), handler);
12:
13:        // 添加节点
14:        addLast0(newCtx);
15:
16:        // <1> pipeline 暂未注册，添加回调。再注册完成后，执行回调。详细解析，见 {@link #invokeHandlerA
17:        // If the registered is false it means that the channel was not registered on an eventloop
18:        // In this case we add the context to the pipeline and add a task that will call
19:        // ChannelHandler.handlerAdded(...) once the channel is registered.
20:        if (!registered) {
21:            // 设置 AbstractChannelHandlerContext 准备添加中
22:            newCtx.setAddPending();
23:            // 添加 PendingHandlerCallback 回调
24:            callHandlerCallbackLater(newCtx, true);
25:            return this;
26:        }
27:
28:        // <2> 不在 EventLoop 的线程中，提交 EventLoop 中，执行回调用户方法
29:        EventExecutor executor = newCtx.executor();
30:        if (!executor.inEventLoop()) {
31:            // 设置 AbstractChannelHandlerContext 准备添加中
32:            newCtx.setAddPending();
33:            // 提交 EventLoop 中，执行回调 ChannelHandler added 事件
34:            executor.execute(new Runnable() {
35:                @Override
36:                public void run() {
                       0(newCtx);
```

事件

程并发操作 pipeline 底层的双向链表。

lHandler) 方法，校验是否重复的 ChannelHandler 。详细解析，见

- 第 11 行：调用 `#filterName(String name, ChannelHandler handler)` 方法，获得 ChannelHandler 的名字。详细解析，见「4. filterName」。
- 第 11 行：调用 `#newContext(EventExecutorGroup group, String name, ChannelHandler handler)` 方法，创建 **DefaultChannelHandlerContext** 节点。详细解析，见「5. newContext」。
- 第 14 行： `#addLast0(AbstractChannelHandlerContext newCtx)` 方法，添加到最后一个节点。详细解析，见「6. addLast0」。
- ========== 后续分成 3 种情况 ==========
- `<1>`
- 第 20 行：Channel 并未注册。这种情况，发生于 ServerBootstrap 启动的过程中。在 `ServerBootstrap#init(Channel channel)` 方法中，会添加 ChannelInitializer 对象到 pipeline 中，恰好此时 Channel 并未注册。
- 第 22 行：调用 `AbstractChannelHandlerContext#setAddPending()` 方法，设置 AbstractChannelHandlerContext **准备添加中**。
- 第 24 行：调用 `#callHandlerCallbackLater(AbstractChannelHandlerContext, added)` 方法，添加 PendingHandlerAddedTask 回调。在 Channel 注册完成后，执行该回调。详细解析，见「8. PendingHandlerCallback」。
- `<2>`
- 第 30 行：不在 EventLoop 的线程中。
- 第 32 行：调用 `AbstractChannelHandlerContext#setAddPending()` 方法，设置 AbstractChannelHandlerContext **准备添加中**。
- 第 34 至 39 行：提交 EventLoop 中，调用 `#callHandlerAdded0(AbstractChannelHandlerContext)` 方法，执行回调 ChannelHandler 添加完成(added)事件。详细解析，见「7. callHandlerAdded0」。
- `<3>`
- 这种情况，是 `<2>` 在 EventLoop 的线程中的版本。也因为此，已经确认在 EventLoop 的线程中，所以不需要在 `synchronized` 中。
- 第 45 行：和【第 37 行】的代码一样，调用 `#callHandlerAdded0(AbstractChannelHandlerContext)` 方法，执行回调 ChannelHandler 添加完成(added)事件。

3. checkMultiplicity

`#checkMultiplicity(ChannelHandler handler)` 方法，校验是否重复的 ChannelHandler 。代码如下：

```
private static void checkMultiplicity(ChannelHandler handler) {
    if (handler instanceof ChannelHandlerAdapter) {
        ChannelHandlerAdapter h = (ChannelHandlerAdapter) handler;
        // 若已经添加，并且未使用 @Sharable 注解，则抛出异常
        if (!h.isSharable() && h.added) {
            eption(
```

```
() +
 handler, so can't be added or removed multiple times.");
```

象，如果不使用 Netty `@Sharable` 注解，则只能添加到一个 Channel
ChannelHandler 对象(例如在 Spring 环境中)，则必须给这个

ing.transport.netty.NettyHandler 处理器，它就使用了

4. filterName

`#filterName(String name, ChannelHandler handler)` 方法，获得 ChannelHandler 的名字。代码如下：

```
private String filterName(String name, ChannelHandler handler) {
    if (name == null) { // <1>
        return generateName(handler);
    }
    checkDuplicateName(name); // <2>
    return name;
}
```

- `<1>` 处，若**未**传入默认的名字 `name` ，则调用 `#generateName(ChannelHandler)` 方法，根据 ChannelHandler 生成一个**唯一**的名字。详细解析，见 「4.1 generateName」 。
- `<2>` 处，若**已**传入默认的名字 `name` ，则调用 `#checkDuplicateName(String name)` 方法，校验名字唯一。详细解析，见 「4.2 checkDuplicateName」 。

4.1 generateName

`#generateName(ChannelHandler)` 方法，根据 ChannelHandler 生成一个**唯一**名字。代码如下：

```
 1: private String generateName(ChannelHandler handler) {
 2:     // 从缓存中查询，是否已经生成默认名字
 3:     Map<Class<?>, String> cache = nameCaches.get();
 4:     Class<?> handlerType = handler.getClass();
 5:     String name = cache.get(handlerType);
 6:     // 若未生成过，进行生成
 7:     if (name == null) {
 8:         name = generateName0(handlerType);
 9:         cache.put(handlerType, name);
10:     }
11:
12:     // 判断是否存在相同名字的节点
13:     // It's not very likely for a user to put more than one handler of the same type, but make sur
14:     // any name conflicts.  Note that we don't cache the names generated here.
```

```
号，循环生成，直到一个是唯一的
ring(0, name.length() - 1); // Strip the trailing '0'.

e + i;
= null) { // // 判断是否存在相同名字的节点
```

是否已经生成**默认**名字。

- 若未生成过，调用 `#generateName0(ChannelHandler)` 方法，进行生成。而后，添加到缓存 `nameCaches` 中。

- 第 15 行：调用 `#context0(String name)` 方法，判断是否存在相同名字的节点。代码如下：

```
private AbstractChannelHandlerContext context0(String name) {
    AbstractChannelHandlerContext context = head.next;
    // 顺序向下遍历节点，判断是否有指定名字的节点。如果有，则返回该节点。
    while (context != tail) {
        if (context.name().equals(name)) {
            return context;
        }
        context = context.next;
    }
    return null;
}
```

  - 顺序向下遍历节点，判断是否有指定名字的节点。如果有，则返回该节点。
- 第 15 至 25 行：若存在相同名字的节点，则使用**基础**名字 + 编号，循环生成，直到一个名字是**唯一**的，然后结束循环。

4.2 checkDuplicateName

`#checkDuplicateName(String name)` 方法，校验名字唯一。代码如下：

```
private void checkDuplicateName(String name) {
    if (context0(name) != null) {
        throw new IllegalArgumentException("Duplicate handler name: " + name);
    }
}
```

- 通过调用 `#context0(String name)` 方法，获得指定名字的节点。若存在节点，意味着**不唯一**，抛出 IllegalArgumentException 异常。

5. newContext

`#newContext(EventExecutorGroup group, String name, ChannelHandler handler)` 方法，创建 **DefaultChannelHandlerContext** 节点。而这个节点，**内嵌**传入的 ChannelHandler 参数。代码如下：

```
wContext(EventExecutorGroup group, String name, ChannelHandler
ext(this, childExecutor(group) /** <1> **/, name, handler);
```

utorGroup group) 方法，创建**子**执行器。代码如下：

```
ventExecutorGroup group) {

UTOR_PER_GROUP ，每个 Channel 从 EventExecutorGroup 获得不同 E
l.config().getOption(ChannelOption.SINGLE_EVENTEXECUTOR_PER_G
    if (pinEventExecutor != null && !pinEventExecutor) {
        return group.next();
    }
    // <3> 通过 childExecutors 缓存实现，一个 Channel 从 EventExecutorGroup 获得相同 EventExecutor 执
    Map<EventExecutorGroup, EventExecutor> childExecutors = this.childExecutors;
    if (childExecutors == null) {
        // Use size of 4 as most people only use one extra EventExecutor.
        childExecutors = this.childExecutors = new IdentityHashMap<EventExecutorGroup, EventExecut
    }
    // Pin one of the child executors once and remember it so that the same child executor
    // is used to fire events for the same channel.
    EventExecutor childExecutor = childExecutors.get(group);
    // 缓存不存在，进行 从 EventExecutorGroup 获得 EventExecutor 执行器
    if (childExecutor == null) {
        childExecutor = group.next();
        childExecutors.put(group, childExecutor); // 进行缓存
    }
    return childExecutor;
}
```

- 一共有三种情况：
  - `<1>` ，当**不传入** EventExecutorGroup 时，不创建**子**执行器。即，使用 Channel 所注册的 EventLoop 作为执行器。**对于我们日常使用，基本完全都是这种情况。**所以，下面两种情况，胖友不理解也是没关系的。
  - `<2>` ，根据配置项 `ChannelOption.SINGLE_EVENTEXECUTOR_PER_GROUP` ，每个 Channel 从 EventExecutorGroup 获得**不同** EventExecutor 执行器。
  - `<3>` ，通过 `childExecutors` 缓存实现，每个 Channel 从 EventExecutorGroup 获得**相同** EventExecutor 执行器。是否获得相同的 EventExecutor 执行器，这就是 `<2>` 、 `<3>` 的不同。
- **注意**，创建的是 DefaultChannelHandlerContext 对象。

6. addLast0

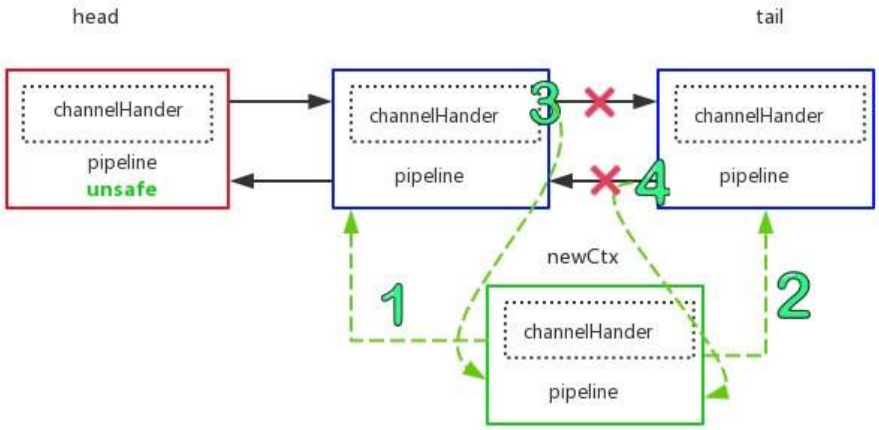
`#addLast0(AbstractChannelHandlerContext newCtx)` 方法，添加到最后一个节点。**注意**，实际上，是添加到 `tail` 节点**之前**。代码如下：

```
private void addLast0(AbstractChannelHandlerContext newCtx) {
    // 获得 tail 节点的前一个节点
    AbstractChannelHandlerContext prev = tail.prev;
```

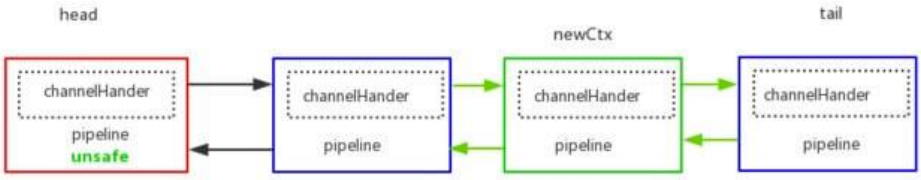
文章目录

析之pipeline(一)》

示这段过程，说白了，其实就是一个双向链表的



- 操作完毕，该节点就加入到 pipeline 中：



7. callHandlerAdded0

`#callHandlerAdded0(AbstractChannelHandlerContext)` 方法，执行回调 ChannelHandler 添加完成(added)事件。代码如下：

```
1: private void callHandlerAdded0(final AbstractChannelHandlerContext ctx) {
2:     try {
3:         // We must call setAddComplete before calling handlerAdded. Otherwise if the handlerAdded
                                                    handler() will miss them because the state will not allow it.
```

```
erContext 已添加

成( added )事件
tx);

erRemoved(ctx); // 回调 ChannelHandler 移除完成( removed )事件

 标记节点已移除

移除成功
20:         } catch (Throwable t2) {
21:             if (logger.isWarnEnabled()) {
22:                 logger.warn("Failed to remove a handler: " + ctx.name(), t2);
23:             }
24:         }
25:
26:         // 触发异常的传播
27:         if (removed) {
28:             fireExceptionCaught(new ChannelPipelineException(
29:                     ctx.handler().getClass().getName() +
30:                     ".handlerAdded() has thrown an exception; removed.", t));
31:         } else {
32:             fireExceptionCaught(new ChannelPipelineException(
33:                     ctx.handler().getClass().getName() +
34:                     ".handlerAdded() has thrown an exception; also failed to remove.", t));
35:         }
36:     }
37: }
```

- 第 6 行：调用 `AbstractChannelHandlerContext#setAddComplete()` 方法，设置 AbstractChannelHandlerContext 已添加。
- 第 8 行：调用 `ChannelHandler#handlerAdded(AbstractChannelHandlerContext)` 方法，回调 ChannelHandler 添加完成(added)事件。一般来说，通过这个方法，来初始化 ChannelHandler 。**注意**，因为这个方法的执行在 EventLoop 的线程中，所以要尽量避免执行时间过长。
- 第 9 行：发生异常。
  - 第 10 至 24 行：移除该节点(ChannelHandler)。详细解析，见《精尽 Netty 源码解析 —— ChannelPipeline（三）之移除 ChannelHandler》。
    - 😈 所以， `ChannelHandler#handlerAdded(AbstractChannelHandlerContext)` 方法的执行**异常**时，将被移除。
  - 第 26 至 35 行：触发异常的传播。详细解析，见《精尽 Netty 源码解析 —— ChannelPipeline（六）之异常事件的传播》。

8. PendingHandlerCallback

PendingHandlerCallback ，实现 Runnable 接口，等待添加 ChannelHandler 回调抽象类。代码如下：

efaultChannelPipeline 的内部静态类。

```
dlerCallback implements Runnable {

点

 ctx;

 对象

    PendingHandlerCallback(AbstractChannelHandlerContext ctx) {
        this.ctx = ctx;
    }

    /**
     * 执行方法
     */
    abstract void execute();

}
```

- 通过 `ctx` 和 `next` 字段，形成**回调链**。
- `#execute()` 抽象方法，通过实现它，执行回调逻辑。

为什么会有 PendingHandlerCallback 呢?

因为 ChannelHandler 添加到 pipeline 中，会触发 ChannelHandler 的添加完成(added)事件，并且该事件需要在 Channel 所属的 EventLoop 中执行。

但是 Channel 并未注册在 EventLoop 上时，需要暂时将“触发 ChannelHandler 的添加完成(added)事件”的逻辑，作为一个 PendingHandlerCallback 进行“缓存”。在 Channel 注册到 EventLoop 上时，进行回调执行。

PendingHandlerCallback 有两个实现类：

- PendingHandlerAddedTask
- PendingHandlerRemovedTask

本文只分享 PendingHandlerAddedTask 的代码实现。

8.1 PendingHandlerAddedTask

PendingHandlerAddedTask 实现 PendingHandlerCallback 抽象类，用于回调添加 ChannelHandler 节点。代码如下：

```
private final class PendingHandlerAddedTask extends PendingHandlerCallback {

    PendingHandlerAddedTask(AbstractChannelHandlerContext ctx) {
```

```
cutor();
annelHandler added 事件

调 ChannelHandler added 事件
            try {
                executor.execute(this); // <1>
            } catch (RejectedExecutionException e) {
                if (logger.isWarnEnabled()) {
                    logger.warn(
                            "Can't invoke handlerAdded() as the EventExecutor {} rejected it, removing
                            executor, ctx.name(), e);
                }
                // 发生异常，进行移除
                remove0(ctx);
                // 标记 AbstractChannelHandlerContext 为已移除
                ctx.setRemoved();
            }
        }
    }

}
```

- 在 `#execute()` 实现方法中，我们可以看到，和 `#addLast(EventExecutorGroup group, String name, ChannelHandler handler)` 方法的【第 28 至 45 行】的代码比较类似，目的是，在 EventLoop 的线程中，执行 `#callHandlerAdded0(AbstractChannelHandlerContext)` 方法，回调 ChannelHandler 添加完成(added)事件。
- `<1>` 处，为什么 PendingHandlerAddedTask 可以直接提交到 EventLoop 中呢？因为 PendingHandlerAddedTask 是个 Runnable ，这也就是为什么 PendingHandlerCallback 实现 Runnable 接口的原因。

> 老艿艿：下面开始分享的方法，属于 DefaultChannelPipeline 类。

8.2 callHandlerCallbackLater

`#callHandlerCallbackLater(AbstractChannelHandlerContext ctx, boolean added)` 方法，添加 PendingHandlerCallback 回调。代码如下：

```
/**
 * This is the head of a linked list that is processed by {@link #callHandlerAddedForAllHandlers()} an
 * all the pending {@link #callHandlerAdded0(AbstractChannelHandlerContext)}.
```

```
xpected that the list is used infrequently and its size is sma
 is assumed to be a good compromised to saving memory and tail



ctChannelHandlerContext, boolean)

ndlerCallbackHead;

er(AbstractChannelHandlerContext ctx, boolean added) {

对象
    PendingHandlerCallback task = added ? new PendingHandlerAddedTask(ctx) : new PendingHandlerRe
    PendingHandlerCallback pending = pendingHandlerCallbackHead;
    // 若原 pendingHandlerCallbackHead 不存在，则赋值给它
    if (pending == null) {
        pendingHandlerCallbackHead = task;
    // 若原 pendingHandlerCallbackHead 已存在，则最后一个回调指向新创建的回调
    } else {
        // Find the tail of the linked-list.
        while (pending.next != null) {
            pending = pending.next;
        }
        pending.next = task;
    }
}
```

- `added` 方法参数，表示是否是添加 ChannelHandler 的回调。所以在【第 5 行】的代码，根据 `added` 是否为 `true` ，创建 PendingHandlerAddedTask 或 PendingHandlerRemovedTask 对象。在本文中，当然创建的是 PendingHandlerAddedTask 。
- 第 7 至 17 行：将创建的 PendingHandlerCallback 对象，“添加”到 `pendingHandlerCallbackHead` 中。

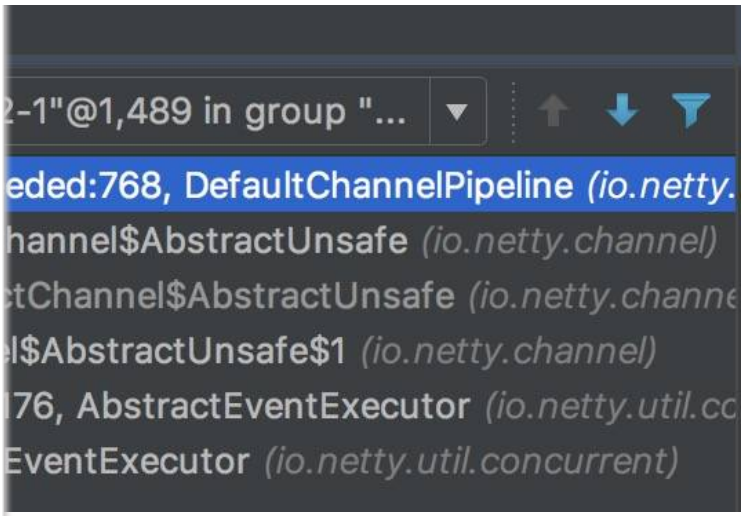
8.3 invokeHandlerAddedIfNeeded

`#invokeHandlerAddedIfNeeded()` 方法，执行**在 PendingHandlerCallback 中**的 ChannelHandler 添加完成(added)事件。它被两个方法所调用：

- `AbstractUnsafe#register0(ChannelPromise promise)` 方法
  - 原因是：

```
// Ensure we call handlerAdded(...) before we actually notify the promise. This is needed as
// user may already fire events through the pipeline in the ChannelFutureListener.
```

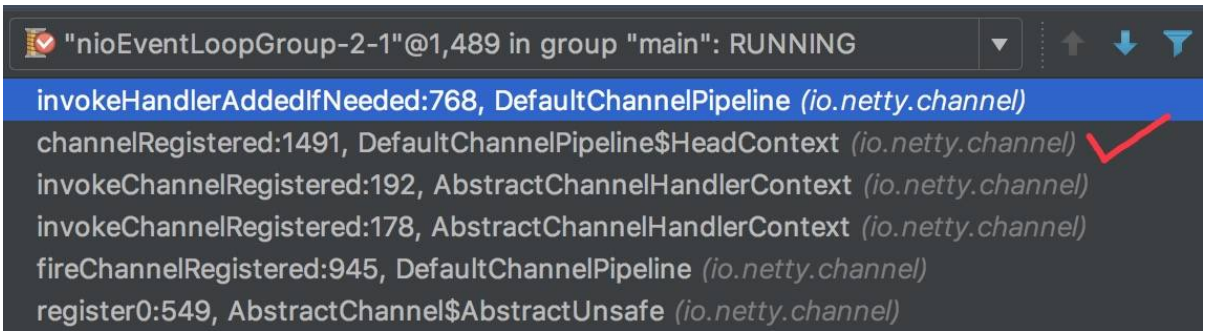
  - 例如 ServerBootstrap 通过 ChannelInitializer 注册自定义的 ChannelHandler 到 pipeline 上的情况。
- 调用栈如下图：



HandlerContext ctx) 方法。

O Client 貌似没啥作用，因为已经在 AbstractUnsafe#register0(ChannelPromise promise) 中触发。胖友也可以自己调试下。

- 调用栈如下图：



#invokeHandlerAddedIfNeeded() 方法，代码如下：

```
/**
 * 是否首次注册
 *
 * {@link #invokeHandlerAddedIfNeeded()}
 */
private boolean firstRegistration = true;

final void invokeHandlerAddedIfNeeded() {
    assert channel.eventLoop().inEventLoop(); // 必须在 EventLoop 的线程中
    // 仅有首次注册有效 <1>
    if (firstRegistration) {
        // 标记非首次注册
        firstRegistration = false;

        // 执行在 PendingHandlerCallback 中的 ChannelHandler 添加完成( added )事件 // <2>
        // We are now registered to the EventLoop. It's time to call the callbacks for the ChannelHand
        // that were added before the registration was done.
        callHandlerAddedForAllHandlers();
    }
}
```

- <1> 处，仅有首次注册有效(firstRegistration = true)时。而后，标记 firstRegistration = false 。

annelRegistered(ChannelHandlerContext ctx) 方法对这个方法

andlers() 方法，执行**在 PendingHandlerCallback 中**的
如下：

```
llHandlers() {
pendingHandlerCallbackHead;
Head


s registered.
已注册

d = this.pendingHandlerCallbackHead;
        // Null out so it can be GC'ed.
        this.pendingHandlerCallbackHead = null; // 置空，help gc
    }

    // 顺序向下，执行 PendingHandlerCallback 的回调
    // This must happen outside of the synchronized(...) block as otherwise handlerAdded(...)
    // holding the lock and so produce a deadlock if handlerAdded(...) will try to add another
    // the EventLoop.
    PendingHandlerCallback task = pendingHandlerCallbackHead;
    while (task != null) {
        task.execute();
        task = task.next;
    }
}
```

- 第 3 至 13 行：获得 `pendingHandlerCallbackHead` 变量。
  - 第 8 行：标记 `registered = true` ，表示已注册。
  - 第 10 至 12 行：置空对象的 `pendingHandlerCallbackHead` 属性，help GC 。
  - 使用 `synchronized` 的原因，和 `#addLast(EventExecutorGroup group, String name, ChannelHandler handler)` 的【第 16 至 26 行】的代码需要对 `pendingHandlerCallbackHead` 互斥，避免并发修改的问题。
- 第 15 至 23 行：顺序循环向下，调用 `PendingHandlerCallback#execute()` 方法，执行 PendingHandlerCallback 的回调，从而将 ChannelHandler 添加到 pipeline 中。
  - 这里不适用 `synchronized` 的原因，看英文注释哈。

666. 彩蛋

添加 ChannelHandler 到 pipeline 中的代码，大部分的比较简单。比较复杂的可能是，「8. PendingHandlerCallback」 中，调用的过程涉及**回调**，所以理解上稍微可能困难。胖友可以多多调试进行解决噢。

推荐阅读文章：

- 闪电侠 《Netty 源码分析之 pipeline(一)》
- Hypercube 《自顶向下深入分析 Netty（七）–ChannelPipeline 源码实现》

文章目录

文章目录