- NETTY

精尽 Netty 源码解析 —— ChannelPipeline（一）之初始化

1. 概述

在《精尽 Netty 源码分析 —— Netty 简介（二）之核心组件》中，对 EventLoopGroup 和 EventLoop 做了定义，我们再来回顾下：

> ChannelPipeline 为 ChannelHandler 的链，提供了一个容器并定义了用于沿着链传播入站和出站事件流的 API 。一个数据或者事件可能会被多个 Handler 处理，在这个过程中，数据或者事件经流 ChannelPipeline ，由 ChannelHandler 处理。在这个处理过程中，一个 ChannelHandler 接收数据后处理完成后交给下一个 ChannelHandler，或者什么都不做直接交给下一个 ChannelHandler。

因为 ChannelPipeline 涉及的代码量较大，所以笔者会分成好几篇文章分别分享。而本文，我们来分享 ChannelPipeline 的**初始化**。也因此，本文更多是体现 ChannelPipeline 的**整体性**，所以不会过多介绍每个类的具体的**每个方法**的实现。

2. ChannelPipeline

文章目录

nnelInboundInvoker、ChannelOutboundInvoker、Iterable 接口，

```
nelOutboundInvoker, Iterable<Entry<String, ChannelHandler>>

=========
annelHandler handler);
oup group, String name, ChannelHandler handler);
annelHandler handler);
up group, String name, ChannelHandler handler);
e, String name, ChannelHandler handler);
Group group, String baseName, String name, ChannelHandler h
, String name, ChannelHandler handler);
oup group, String baseName, String name, ChannelHandler ha
... handlers);
oup group, ChannelHandler... handlers);
.. handlers);
```

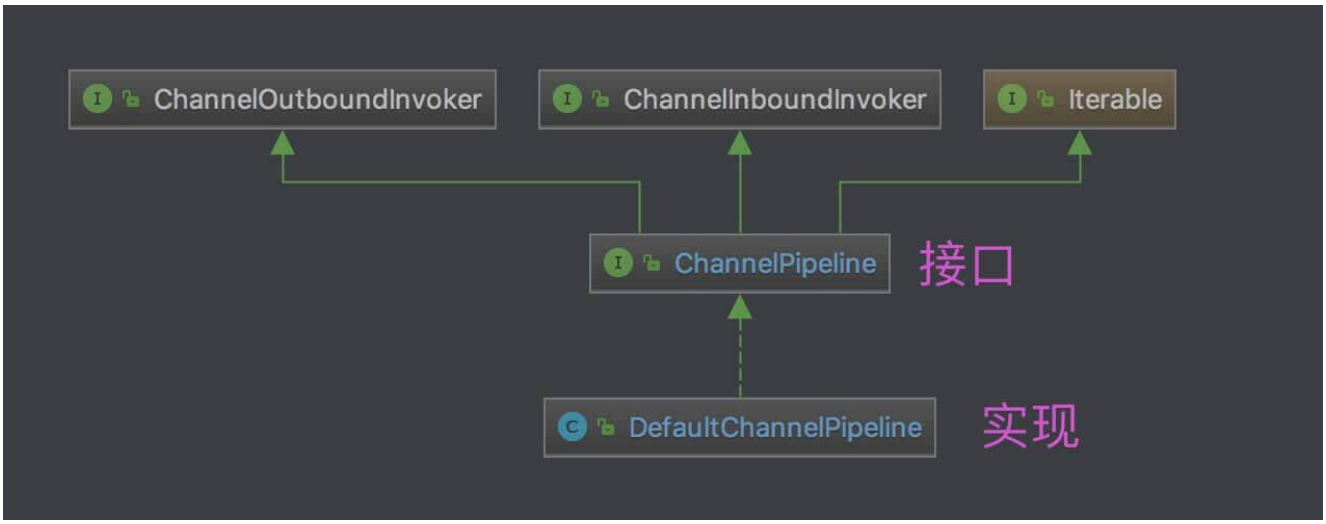
```
oup group, ChannelHandler... handlers);

=========
andler);

ss<T> handlerType);

ChannelHandler removeLast();

// ========== 替换 ChannelHandler 相关 ==========
ChannelPipeline replace(ChannelHandler oldHandler, String newName, ChannelHandler newHandler);
ChannelHandler replace(String oldName, String newName, ChannelHandler newHandler);
<T extends ChannelHandler> T replace(Class<T> oldHandlerType, String newName, ChannelHandler newHa

// ========== 查询 ChannelHandler 相关 ==========
ChannelHandler first();
ChannelHandlerContext firstContext();
ChannelHandler last();
ChannelHandlerContext lastContext();
ChannelHandler get(String name);
<T extends ChannelHandler> T get(Class<T> handlerType);
ChannelHandlerContext context(ChannelHandler handler);
ChannelHandlerContext context(String name);
ChannelHandlerContext context(Class<? extends ChannelHandler> handlerType);
List<String> names();

// ========== Channel 相关 ==========
Channel channel();

// ========== ChannelInboundInvoker 相关 ==========
@Override
ChannelPipeline fireChannelRegistered();
@Override
ChannelPipeline fireChannelUnregistered();
@Override
ChannelPipeline fireChannelActive();
@Override
```

文章目录

- 继承自 ChannelOutboundInvoker 的相关方法。

有可能会疑惑为什么继承 Iterable 接口？因为 ChannelPipeline 是 ChannelHandler 的**链**。

ChannelPipeline 的类图如下：



2.1 ChannelInboundInvoker

`io.netty.channel.ChannelInboundInvoker` ，Channel Inbound Invoker(调用者) 接口。代码如下：

```
ChannelPipeline fireChannelRegistered();
ChannelPipeline fireChannelUnregistered();
ChannelPipeline fireChannelActive();
ChannelPipeline fireChannelInactive();
ChannelPipeline fireExceptionCaught(Throwable cause);
ChannelPipeline fireUserEventTriggered(Ob t event);
ChannelPipeline fireChannelRead(Object ms
                                 ()
                                 ed();
```


hannel Outbound Invoker(调用者) 接口。代码如下：

```
=
ld  ss);
ot  ldress);
ot  ldress, SocketAddress localAddress);
```

```
ChannelFuture write(Object msg);
ChannelFuture write(Object msg, ChannelPromise promise);
ChannelOutboundInvoker flush();
ChannelFuture writeAndFlush(Object msg, ChannelPromise promise);
ChannelFuture writeAndFlush(Object msg);

// ========== Promise 相关 ==========
ChannelPromise newPromise();
ChannelProgressivePromise newProgressivePromise();
ChannelFuture newSucceededFuture();
ChannelFuture newFailedFuture(Throwable cause);
ChannelPromise voidPromise();
```

- 发起 Channel 操作的接口方法。
- 创建 Promise 对象的接口方法。

2.3 Outbound v.s Inbound 事件

在 《Netty 源码分析之 二 贯穿Netty 的大动脉 —— ChannelPipeline (二)》 中，笔者看到一个比较不错的总结：

> 老艿艿：因为要加一些注释，所以暂时不使用引用。

对于 Outbound 事件：

nn t.findContextOutbound -> nextContext.invokeOUT_EVT ->
UT VT

就 后, 通知上层.

- 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.
- Inbound 事件在 Pipeline 中传输方向是 `head（头）-> tail（尾）`

> 旁白：Inbound 翻译为“入站”，所以从 `head（头）`到 `tail（尾）`也合理。

- 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.
- 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, 并且 Context 与 Handler 直接的调用关系是否容易混淆, 因此读者在阅读这里的源码时, 需要特别的注意。

3. DefaultChannelPipeline

`io.netty.channel.DefaultChannelPipeline` ，实现 ChannelPipeline 接口，默认 ChannelPipeline 实现类。😈 实际上，也只有这个实现类。

3.1 静态属性

```
/**
 * {@link #head} 的名字
 */
```

文章目录

g erateName0(HeadContext.class);

g erateName0(TailContext.class);

te #name})缓存 ，基于 ThreadLocal ，用于生成在线程中唯一的名字。

>< ass<?>, String>> nameCaches = new FastThreadLocal<Map<Class

:i /alue() throws Exception {
ring>();

```
ldl iater<DefaultChannelPipeline, MessageSizeEstimator.Handle>
Jp ter(
:l s, MessageSizeEstimator.Handle.class, "estimatorHandle");
```

- HEAD_NAME 和 TAIL_NAME 静态属性，通过调用 #generateName0(Class<?> handlerType) 方法，生成对应的名字。代码如下：

```
private static String generateName0(Class<?> handlerType) {
    return StringUtil.simpleClassName(handlerType) + "#0";
}
```

  - 即 HEAD_NAME = "HeadContext#0" ， TAIL_NAME= "TailContext#0" 。
- nameCaches 静态属性，名字(AbstractChannelHandlerContext.name)缓存，基于 ThreadLocal ，用于生成**在线程中唯一的名字**。详细解析，见《精尽 Netty 源码解析 —— Pipeline（二）之添加 ChannelHandler》。
- ESTIMATOR 静态属性， estimatorHandle 属性的**原子**更新器。

3.2 构造方法

```
/**
 * Head 节点
 */
final AbstractChannelHandlerContext head;
/**
 * Tail 节点
 */
final AbstractChannelHandlerContext tail;

/**
 * 所属 Channel 对象
 */
final Channel channel;
```



```
:cu
的 法参数，但是不需要进行通知，就传入该值
l./ stractUnsafe#safeSetSuccess(ChannelPromise)
om e;
用
eal etector.isEnabled();
```

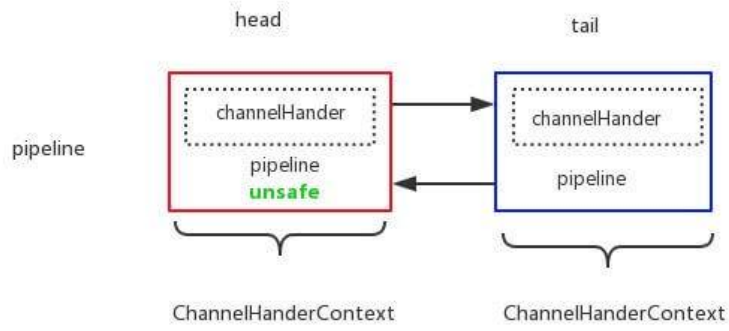
```
1   在的 EventLoop 作为执行器。
解   见 {@link #childExecutor(EventExecutorGroup)} 。
】   通过。
private Map<EventExecutorGroup, EventExecutor> childExecutors;
/**
 * TODO 1008 DefaultChannelPipeline 字段用途
 */
private volatile MessageSizeEstimator.Handle estimatorHandle;
/**
 * 是否首次注册
 */
private boolean firstRegistration = true;

/**
 * This is the head of a linked list that is processed by {@link #callHandlerAddedForAllHandlers()} an
 * all the pending {@link #callHandlerAdded0(AbstractChannelHandlerContext)}.
 *
 * We only keep the head because it is expected that the list is used infrequently and its size is sma
 * Thus full iterations to do insertions is assumed to be a good compromised to saving memory and tail
 * complexity.
 *
 * 准备添加 ChannelHandler 的回调
 */
private PendingHandlerCallback pendingHandlerCallbackHead;

/**
 * Set to {@code true} once the {@link AbstractChannel} is registered.Once set to {@code true} the val
 * change.
 * Channel 是否已注册
 */
private boolean registered;
```

初始化。详细解析，见「4.2 HeadContext」。

始化。详细解析，见「4.3 TailContext」。

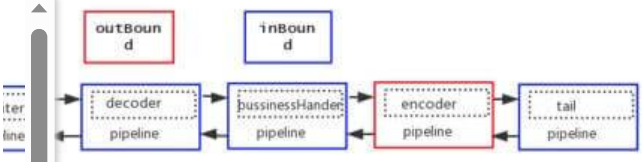
l 节点， tail 节点向**上**指向 head 节点，从而形成**相互**的指

FROM 《netty 源码分析之 pipeline(一)》



- pipeline 中的节点的数据结构是 ChannelHandlerContext 类。每个 ChannelHandlerContext 包含**一个** ChannelHandler、它的**上下**节点(**从而形成 ChannelHandler 链**)、以及其他上下文。详细解析，见「4. ChannelHandlerContext」。
- 默认情况下，pipeline 有 head 和 tail 节点，形成默认的 ChannelHandler 链。而我们可以在它们之间，加入自定义的 ChannelHandler 节点。如下图所示:

FROM 《netty 源码分析之 pipeline(一)》



下，ChannelHandler 使用 Channel 所在的 EventLoop 作为执行

，见《精尽 Netty 源码解析 —— Pipeline（二）之添加

通过。

ChannelHandler 的回调。详细解析，见《精尽 Netty 源码解析

。

析，见《精尽 Netty 源码解析 —— Pipeline（二）之添加

详 解析，见《精尽 Netty 源码解析 —— Pipeline（二）之添加

后续的文章。

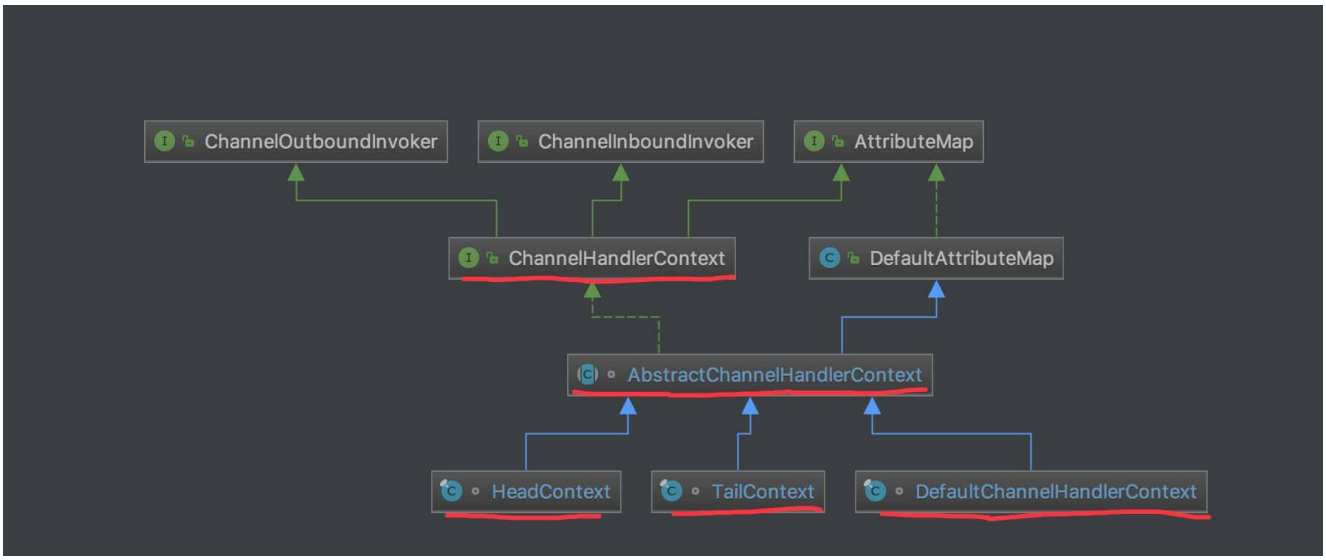
4. ChannelHandlerContext

`io.netty.channel.ChannelHandlerContext` ，继承 ChannelInboundInvoker、ChannelOutboundInvoker、AttributeMap 接口，ChannelHandler Context(上下文)接口，作为 ChannelPipeline 中的**节点**。代码如下：

```
// ========== Context 相关 ==========
String name();
Channel channel();
EventExecutor executor();
ChannelHandler handler();
ChannelPipeline pipeline();
boolean isRemoved(); // 是否已经移除

// ========== ByteBuf 相关 ==========
ByteBufAllocator alloc();

// ========== ChannelInboundInvoker 相关 ==========
@Override
ChannelHandlerContext fireChannelRegistered();
@Override
ChannelHandlerContext fireChannelUnregistered();
@Override
ChannelHandlerContext fireChannelActive();
@Override
ChannelHandlerContext fireChannelInactive();
@Override
ChannelHandlerContext fireExceptionCaught(Throwable cause);
@Override
ChannelHandlerContext fireUserEventTrigge d(Object evt);

 msg);

e();

Changed();

========
```

- 继承自 ChannelInboundInvoker 的相关方法，*和 ChannelPipeline 一样*。
- 继承自 ChannelOutboundInvoker 的相关方法，*和 ChannelPipeline 一样*。
- 继承自 AttributeMap 的相关方法，实际上已经废弃(`@Deprecated`)了，不再从 ChannelHandlerContext 中获取，而是从 Channel 中获取。

ChannelHandlerContext 的类图如下：



- 😈 类图中的 AttributeMap 和 DefaultAttributeMap 可以无视。

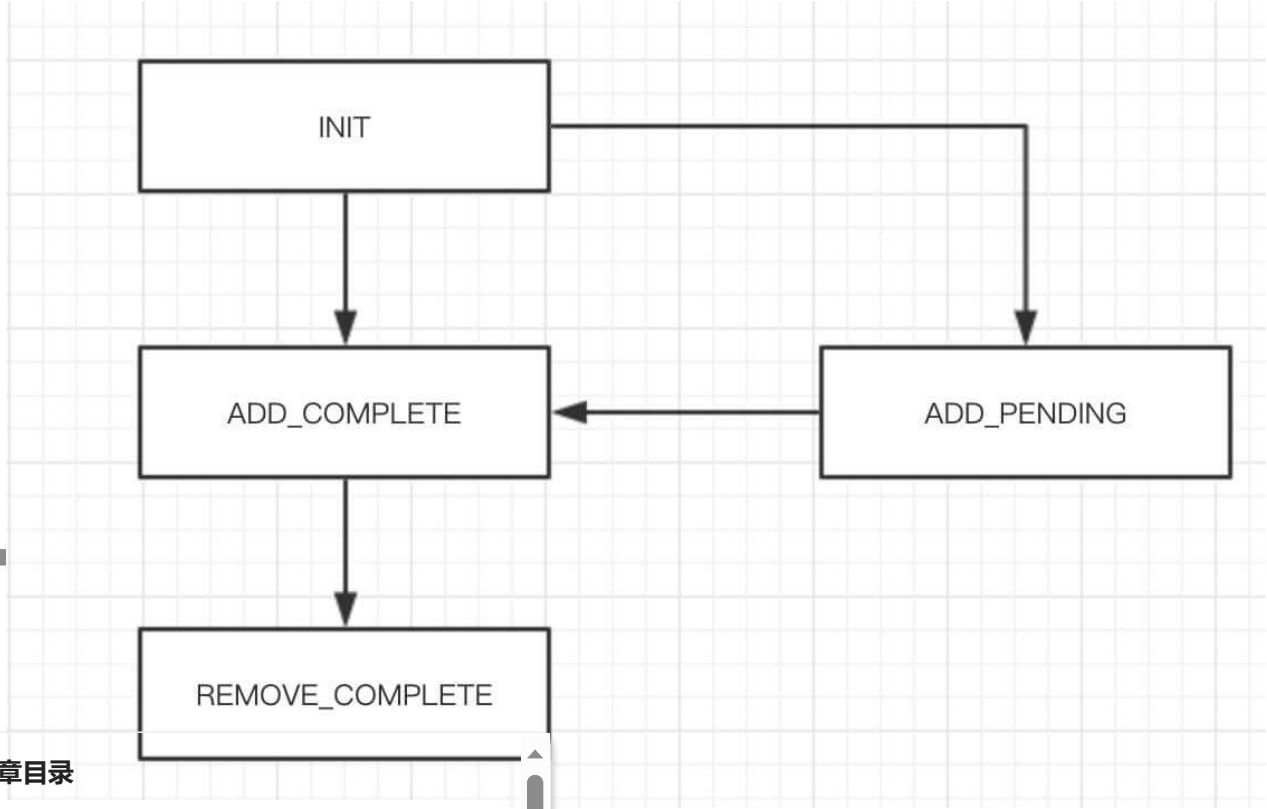
4.1 AbstractChannelHandlerContext

`io.netty.channel.AbstractChannelHandlerContext` ，实现 ChannelHandlerContext、ResourceLeakHint 接口，继承 DefaultAttributeMap 类，ChannelHandlerContext 抽象基类。

静态属性

```
; // 已移除

er<AbstractChannelHandlerContext> HANDLER_STATE_UPDATER = A

// ========== 非静态属性 ==========

/**
 * 处理器状态
 */
private volatile int handlerState = INIT;
```

- `handlerState` 属性(**非静态**属性，放这里主要是为了统一讲)，处理器状态。共有 **4** 种状态。状态变迁如下图：



「.4 setRemoved」、「4.1.5 setAddPending」 中。

tate 的原子更新器。

```
 */
private final DefaultChannelPipeline pipeline;
/**
 * 名字
 */
private final String name;
/**
 * 是否使用有序的 EventExecutor ( {@link #executor} )，即 OrderedEventExecutor
 */
private final boolean ordered;

// Will be set to null if no child executor should be used, otherwise it will be set to the
// child executor.
/**
 * EventExecutor 对象
 */
final EventExecutor executor;
/**
 * 成功的 Promise 对象
 */
private ChannelFuture succeededFuture;

// Lazily instantiated tasks used to trigger events to a handler with different executor. 懒加载
// There is no need to make this volatile as at worse it will just create a few more instances then ne
/**
 * 执行 Channel ReadComplete 事件的任务
 */
private Runnable invokeChannelReadCompleteTask;
/**
 * 执行 Channel Read 事件的任务
```

```
(n , "name");

E tLoop or the given Executor is an instanceof OrderedEventE
:o instanceof OrderedEventExecutor; // <1>
}
```

- `next` 、 `prev` 属性，分别记录上、下一个节点。
- Handler 相关属性：
  - 在 AbstractChannelHandlerContext 抽象类中，按照我们上文的分享，应该会看到一个类型为 ChannelHandler 的处理器，但是**实际并不是这样**。而是，😈 我们下文 DefaultChannelHandlerContext、TailContext、HeadContext 见。
  - `inbound` 、 `outbound` 属性，分别是否为 Inbound、Outbound 处理器。
  - `name` 属性，处理器名字。
  - `handlerState` 属性，处理器状态，初始为 `INIT` 。
- `executor` 属性，EventExecutor 对象
  - `ordered` 属性，是否使用有序的 `executor` ，即 OrderedEventExecutor ，在构造方法的 `<1>` 处理的初始化。
- `pipeline` 属性，所属 DefaultChannelPipeline 对象。

4.1.3 setAddComplete

`#setAddComplete()` 方法，设置 ChannelHandler 添加完成。完成后，状态有两种结果：

1. `REMOVE_COMPLETE`
2. `ADD_COMPLETE`

代码如下：

```
final void setAddComplete() {
    for (;;) {
        int oldState = handlerState;
        // Ensure we never update when th handlerState is REMOVE_COMPLETE already.
        // oldState is usually ADD_PENDIN ut can also be REMOVE_COMPLETE when an EventExecutor is us

        | HANDLER_STATE_UPDATER.compareAndSet(this, oldState, ADD_COM
```


rS te 属性。

。代码如下：

#setAddPending() 方法，设置 ChannelHandler 准备添加中。代码如下：

```
final void setAddPending() {
    boolean updated = HANDLER_STATE_UPDATER.compareAndSet(this, INIT, ADD_PENDING);
    assert updated; // This should always be true as it MUST be called before setAddComplete() or setR
}
```

- 当且仅当 `INIT` 可修改为 `ADD_PENDING` 。理论来说，这是一个绝对会成功的操作，原因见英文注释。

4.1.6 其他方法

AbstractChannelHandlerContext 中的其他方法，详细解析，见后续的文章。

4.2 HeadContext

HeadContext ，实现 ChannelOutboundHandler、ChannelInboundHandler 接口，继承 AbstractChannelHandlerContext 抽象类，**pipe 头节点** Context 实现类。

> HeadContext 是 DefaultChannelPipeline 的内部类。

4.2.1 构造方法

```
private final Unsafe unsafe;

HeadContext(DefaultChannelPipeline pipeli ) {
    ls  true); // <1>
    );  / <2>
```


nte 的构造方法，设置 `inbound = false` 、 `outbound = true` 。

af 属性。HeadContext 实现 ChannelOutboundHandler 接口的方法，

```
t   x, SocketAddress localAddress, ChannelPromise promise) th
);
```

```
te  ctx, SocketAddress remoteAddress, SocketAddress localAdc
```

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

@Override
public void deregister(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception {
    unsafe.deregister(promise);
}

@Override
public void read(ChannelHandlerContext ctx) {
    unsafe.beginRead();
}

@Override
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception
    unsafe.write(msg, promise);
}

@Override
public void flush(ChannelHandlerContext ctx) throws Exception {
    unsafe.flush();
}
```

- 这也就是为什么设置 `outbound = true` 的原因。

- `<3>` 处，调用 `#setAddComplete()` 方法，设置 ChannelHandler 添加完成。此时， `handlerStatus` 会变成 `ADD_COMPLETE` 状态。

文章目录

，继承 AbstractChannelHandlerContext 抽象类，**pipe 尾节点** Context 实现

TailContext 是 DefaultChannelPipeline 的内部类。

4.3.1 构造方法

```
TailContext(DefaultChannelPipeline pipeline) {
    super(pipeline, null, TAIL_NAME, true, false); // <1>
    setAddComplete(); // <2>
}
```

- `<1>` 处，调用父 AbstractChannelHandlerContext 的构造方法，设置 `inbound = true` 、 `outbound = false` ，和 HeadContext **相反**。
- `<2>` 处，调用 `#setAddComplete()` 方法，设置 ChannelHandler 添加完成。此时， `handlerStatus` 会变成 `ADD_COMPLETE` 状态。

4.3.2 handler

`#handler()` 方法，返回自己作为 Context 的 **ChannelHandler** 。代码如下：

```
@Override
public ChannelHandler handler() {
    return this;
}
```

- 因为 HeadContext ，实现 ChannelInboundHandler 接口，而它们本身就是 ChannelHandler 。

4.3.3 其他方法

or xt ，实现 AbstractChannelHandlerContext 抽象类。代码如下：

```
: ends AbstractChannelHandlerContext {

;

li , EventExecutor executor, String name, ChannelHandler handl
i bound(handler), isOutbound(handler)); // <1>

lo 'handler");
```

```
    private static boolean isInbound(ChannelHandler handler) {
        return handler instanceof ChannelInboundHandler;
    }

    private static boolean isOutbound(ChannelHandler handler) {
        return handler instanceof ChannelOutboundHandler;
    }

}
```

- 不同于 HeadContext、TailContext，它们自身就是一个 Context 的同时，也是一个 ChannelHandler 。而 DefaultChannelHandlerContext 是**内嵌** 一个 ChannelHandler 对象，即 `handler` 。这个属性通过构造方法传入，在 `<2>` 处进行赋值。
- `<1>` 处，调用父 AbstractChannelHandlerContext 的构造方法，通过判断传入的 `handler` 是否为 ChannelInboundHandler 和 ChannelOutboundHandler 来分别判断是否为 `inbound` 和 `outbound` 。

666. 彩蛋

推荐阅读如下文章：

- 闪电侠 《netty 源码分析之 pipeline(一)》
- 永顺 《Netty 源码分析之 二 贯穿Netty 的大动脉 —— ChannelPipeline (一)》
- 占小狼 《Netty 源码分析之 ChannelPipeline》