3

第 3 篇

Netty 核心篇

8

第 8 章

Netty 大动脉 Pipeline

课程目标

1、了解 Netty 服务端的线程池分配规则，线程何时启动。

2、了解 Netty 是如何解决 JDK 空轮训 Bug 的？

3、Netty 是如何实现异步串行无锁化编程的？

内容定位

1、提前阅读了预习资料且完成前面章节学习的人群。

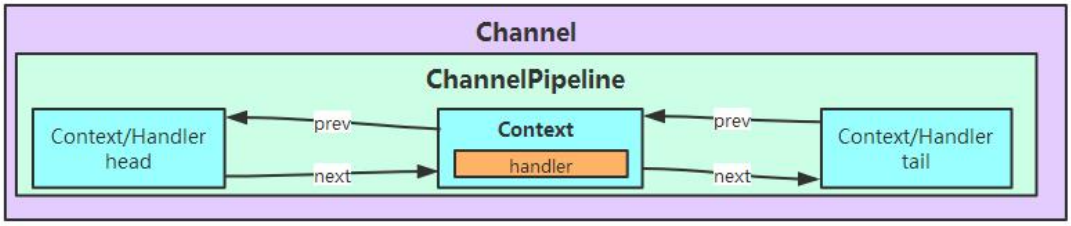
2、希望深入了解 Netty 源码的人群。

3、未来可能参与中间件开发的人群。

8.1 Pipeline 设计原理

8.1.1 Channel 与 ChannelPipeline

相信大家都已经知道，在 Netty 中每个 Channel 都有且仅有一个 ChannelPipeline 与之对应，它们的组成关系如下：



通过上图我们可以看到，一个 Channel 包含了一个 ChannelPipeline，而 ChannelPipeline 中又维护了一个由 ChannelHandlerContext 组成的双向链表。这个链表的头是 HeadContext，链表的尾是 TailContext，并且每个 ChannelHandlerContext 中又关联着一个 ChannelHandler。

上面的图示给了我们一个对 ChannelPipeline 的直观认识，但是实际上 Netty 实现的 Channel 是否真的是这样的呢？

我们继续用源码说话。在前我们已经知道了一个 Channel 的初始化的基本过程，下面我们再回顾一下。下面的代码是 AbstractChannel 构造器：

```
    protected AbstractChannel(Channel parent) {
        this.parent = parent;
        id = newId();
        unsafe = newUnsafe();
        pipeline = newChannelPipeline();
    }
```

AbstractChannel 有一个 pipeline 字段，在构造器中会初始化它为 DefaultChannelPipeline 的实例。这里的代码就印证了一点：每个 Channel 都有一个 ChannelPipeline。接着我们跟踪一下 DefaultChannelPipeline 的初始化过程，首先进入到 DefaultChannelPipeline 构造器中：

```
protected DefaultChannelPipeline(Channel channel) {
        this.channel = ObjectUtil.checkNotNull(channel, "channel");
        succeededFuture = new SucceededChannelFuture(channel, null);
        voidPromise =  new VoidChannelPromise(channel, true);

        tail = new TailContext(this);
        head = new HeadContext(this);
```

```
        head.next = tail;
        tail.prev = head;
}
```

在 DefaultChannelPipeline 构造器中，首先将与之关联的 Channel 保存到字段 channel 中。然后实例化两个 ChannelHandlerContext：一个是 HeadContext 实例 head，另一个是 TailContext 实例 tail。接着将 head 和 tail 互相指向， 构成一个双向链表。

特别注意的是 我们在开始的示意图中 head 和 tail 并没有包含 ChannelHandler ,这是因为 HeadContext 和 TailContext 继承于 AbstractChannelHandlerContext 的同时也实现了 ChannelHandler 接口了，因此它们有 Context 和 Handler 的双重属性。

8.1.2 再探 ChannelPipeline 的初始化

前面我们已经对 ChannelPipeline 的初始化有了一个大致的了解，不过当时重点没有关注 ChannelPipeline，因此没有深入地分析它的初始化过程。那么下面我们就来看一下具体的 ChannelPipeline 的初始化都做了哪些工作吧。先回顾一下，在实例化一个 Channel 时，会伴随着一个 ChannelPipeline 的实例化，并且此 Channel 会与这个 ChannelPipeline 相互关联，这一点可以通过 NioSocketChannel 的父类 AbstractChannel 的构造器予以佐证：

```
protected AbstractChannel(Channel parent) {
        this.parent = parent;
        id = newId();
        unsafe = newUnsafe();
        pipeline = newChannelPipeline();
}
```

当实例化一个 NioSocketChannel 是，其 pipeline 字段就是我们新创建的 DefaultChannelPipeline 对象，那么我们就来看一下 DefaultChannelPipeline 的构造方法。

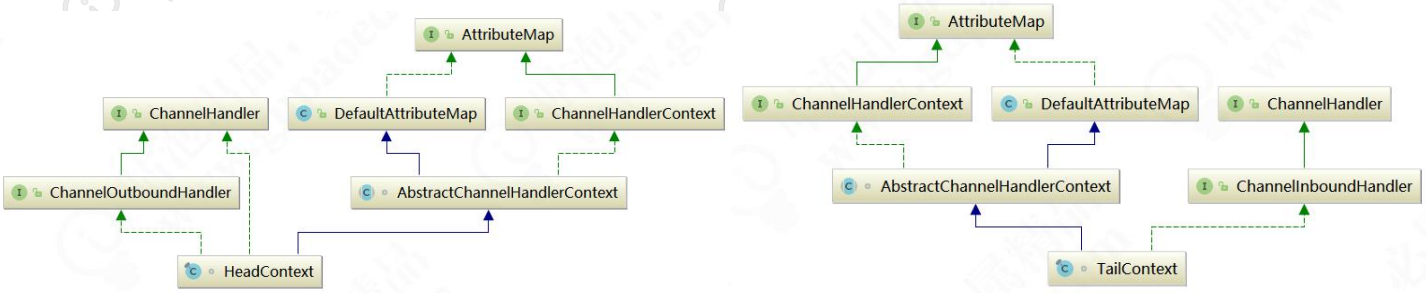
```
protected DefaultChannelPipeline(Channel channel) {
        this.channel = ObjectUtil.checkNotNull(channel, "channel");
        succeededFuture = new SucceededChannelFuture(channel, null);
        voidPromise =  new VoidChannelPromise(channel, true);

        tail = new TailContext(this);
        head = new HeadContext(this);

        head.next = tail;
        tail.prev = head;
}
```

可以看到，在 DefaultChannelPipeline 的构造方法中，将传入的 channel 赋值给字段 this.channel，接着又实例化了两

个特殊的字段：tail 与 head，这两个字段是一个双向链表的头和尾。其实在 DefaultChannelPipeline 中，维护了一个以 AbstractChannelHandlerContext 为节点的双向链表 ,这个链表是 Netty 实现 Pipeline 机制的关键。再回顾一下 head 和 tail 的类层次结构：



从类层次结构图中可以很清楚地看到，head 实现了 ChannelInboundHandler，而 tail 实现了 ChannelOutboundHandler 接口 ,并且它们都实现了 ChannelHandlerContext 接口，因此可以说 head 和 tail 即是一个 ChannelHandler，又是一个 ChannelHandlerContext。接着看 HeadContext 构造器中的代码：

```
HeadContext(DefaultChannelPipeline pipeline) {
        super(pipeline, null, HEAD_NAME, false, true);
        unsafe = pipeline.channel().unsafe();
        setAddComplete();
}
```

它调用了父类 AbstractChannelHandlerContext 的构造器，并传入参数 inbound = false，outbound = true。而 TailContext 的构造器与 HeadContext 正好相反，它调用了父类 AbstractChannelHandlerContext 的构造器，并传入参数 inbound = true，outbound = false。也就是说 header 是一个 OutBoundHandler，而 tail 是一个 InboundHandler，关于这一点，大家要特别注意，因为在后面的分析中，我们会反复用到 inbound 和 outbound 这两个属性。

8.1.3 ChannelInitializer 的添加

前面我们已经分析过 Channel 的组成，其中我们了解到，最开始的时候 ChannelPipeline 中含有两个 ChannelHandlerContext（同时也是 ChannelHandler），但是这个 Pipeline 并不能实现什么特殊的功能，因为我们还没有给它添加自定义的 ChannelHandler。通常来说，我们在初始化 Bootstrap，会添加我们自定义的 ChannelHandler，就以我们具体的客户端启动代码片段来举例：

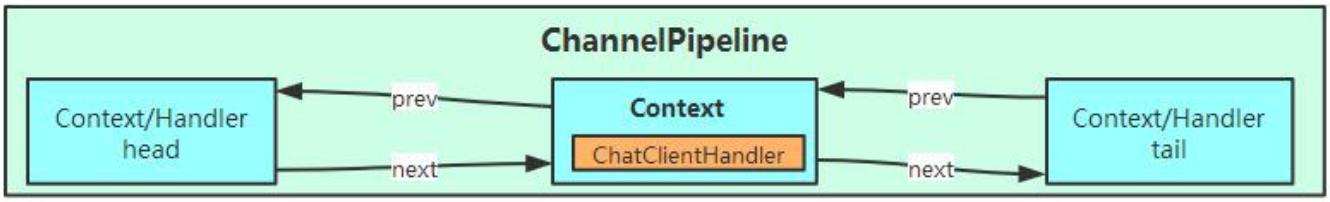
```
        Bootstrap bootstrap = new Bootstrap();
        bootstrap.group(group)
```

```
        .channel(NioSocketChannel.class)
        .option(ChannelOption.SO_KEEPALIVE, true)
        .handler(new ChannelInitializer<SocketChannel>() {
         @Override
         protected void initChannel(SocketChannel ch) throws Exception {
            ChannelPipeline pipeline = ch.pipeline();
            pipeline.addLast(new ChatClientHandler(nickName));
         }
        });
```

上面代码的初始化过程，相信大家都不陌生。在调用 handler 时，传入了 ChannelInitializer 对象，它提供了一个 initChannel()方法给我我们初始化 ChannelHandler。那么这个初始化过程是怎样的呢？下面我们来揭开它的神秘面纱。ChannelInitializer 实现了 ChannelHandler，那么它是在什么时候添加到 ChannelPipeline 中的呢？通过代码跟踪，我们发现它是在 Bootstrap 的 init()方法中添加到 ChannelPipeline 中的，其代码如下：

```
void init(Channel channel) throws Exception {
       ChannelPipeline p = channel.pipeline();
       p.addLast(config.handler());
      //略去 N 句代码
}
```

从上面的代码可见，将 handler()返回的 ChannelHandler 添加到 Pipeline 中，而 handler()返回的其实就是我们在初始化 Bootstrap 时通过 handler()方法设置的 ChannelInitializer 实例，因此这里就是将 ChannelInitializer 插入到了 Pipeline 的末端。此时 Pipeline 的结构如下图所示：



这时候，有小伙伴可能就有疑惑了，我明明插入的是一个 ChannelInitializer 实例，为什么在 ChannelPipeline 中的双向链表中的元素却是一个 ChannelHandlerContext 呢?我们继续去源码中寻找答案。

刚才，我们提到，在 Bootstrap 的 init()方法中会调用 p.addLast()方法，将 ChannelInitializer 插入到链表的末端：

```
    public final ChannelPipeline addLast(EventExecutorGroup group, String name, ChannelHandler handler) {
        final AbstractChannelHandlerContext newCtx;
        synchronized (this) {
            checkMultiplicity(handler);
            newCtx = newContext(group, filterName(name, handler), handler);
            addLast0(newCtx);
            // 略去 N 句代码
        return this;
    }
    private AbstractChannelHandlerContext newContext(EventExecutorGroup group, String name, ChannelHandler handler) {
        return new DefaultChannelHandlerContext(this, childExecutor(group), name, handler);
```

```
  }
```

addLast()有很多重载的方法，我们只需关注这个比较重要的方法就行。上面的 addLast()方法中，首先检查 ChannelHandler 的名字是否是重复，如果不重复，则调用 newContex()方法为这个 Handler 创建一个对应的 DefaultChannelHandlerContext 实例，并与之关联起来(Context 中有一个 handler 属性保存着对应的 Handler 实例)。为了添加一个 handler 到 pipeline 中，必须把此 handler 包装成 ChannelHandlerContext。因此在上面的代码中我们可以看到新实例化了一个 newCtx 对象，并将 handler 作为参数传递到构造方法中。那么我们来看一下实例化的 DefaultChannelHandlerContext 到底有什么玄机吧。首先看它的构造器：

```
DefaultChannelHandlerContext(DefaultChannelPipeline pipeline, EventExecutor executor, String name, ChannelHandler handler) {
        super(pipeline, executor, name, isInbound(handler), isOutbound(handler));
        if (handler == null) {
            throw new NullPointerException("handler");
        }
        this.handler = handler;
}
```

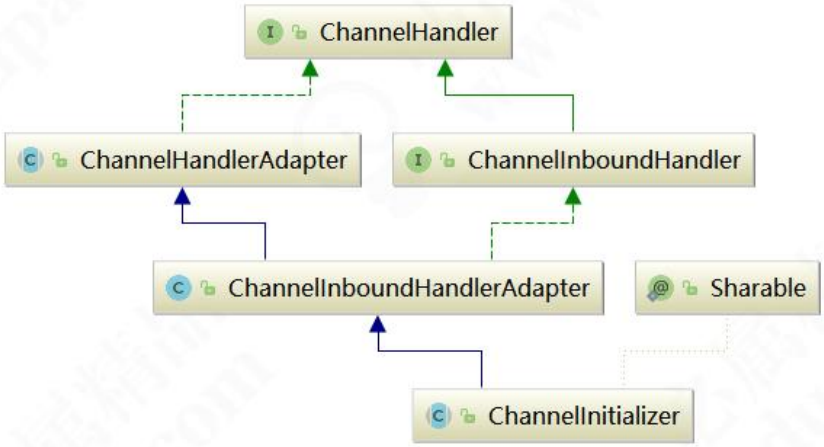
在 DefaultChannelHandlerContext 的构造器中，调用了两个很有意思的方法：isInbound()与 isOutbound()，这两个方法是做什么的呢？

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

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

从源码中可以看到，当一个 handler 实现了 ChannelInboundHandler 接口，则 isInbound 返回 true；类似地，当一个 handler 实现了 ChannelOutboundHandler 接口，则 isOutbound 就返回 true。而这两个 boolean 变量会传递到父类 AbstractChannelHandlerContext 中，并初始化父类的两个字段：inbound 与 outbound。

那么这里的 ChannelInitializer 所对应的 DefaultChannelHandlerContext 的 inbound 与 inbound 字段分别是什么呢？那就看一下 ChannelInitializer 到底实现了哪个接口不就行了？如下是 ChannelInitializer 的类层次结构图：



从类图中可以清楚地看到，ChannelInitializer 仅仅实现了 ChannelInboundHandler 接口，因此这里实例化的 DefaultChannelHandlerContext 的 inbound = true，outbound = false。

兜了一圈，不就是 inbound 和 outbound 两个字段嘛，为什么需要这么大费周折地分析一番？其实这两个字段关系到 pipeline 的事件的流向与分类，因此是十分关键的，不过我在这里先卖个关子，后面我们再来详细分析这两个字段所起的作用。至此， 我们暂时先记住一个结论：ChannelInitializer 所对应的 DefaultChannelHandlerContext 的 inbound = true，outbound = false。

当创建好 Context 之后，就将这个 Context 插入到 Pipeline 的双向链表中，基础较差的可以将下面的逻辑用图画出来：

```
private void addLast0(AbstractChannelHandlerContext newCtx) {
       AbstractChannelHandlerContext prev = tail.prev;
       newCtx.prev = prev;
       newCtx.next = tail;
       prev.next = newCtx;
       tail.prev = newCtx;
}
```

8.1.4 自定义 ChannelHandler 的添加过程

前面我们已经分析了 ChannelInitializer 是如何插入到 Pipeline 中的 接下来就来探讨 ChannelInitializer 在哪里被调用，ChannelInitializer 的作用以及我们自定义的 ChannelHandler 是如何插入到 Pipeline 中的。

先简单复习一下 Channel 的注册过程：

1、首先在 AbstractBootstrap 的 initAndRegister()中，通过 group().register(channel)，调用 MultithreadEventLoopGroup 的 register()方法。

2、在 MultithreadEventLoopGroup 的 register()中调用 next()获取一个可用的 SingleThreadEventLoop，然后调用它的 register()方法。

3、在 SingleThreadEventLoop 的 register()方法中，通过 channel.unsafe().register(this, promise)方法获取 channel 的 unsafe()底层 IO 操作对象，然后调用它的 register()。

4、在 AbstractUnsafe 的 register()方法中，调用 register0()方法注册 Channel 对象。

5、在 AbstractUnsafe 的 register0()方法中，调用 AbstractNioChannel 的 doRegister()方法。

6、AbstractNioChannel 的 doRegister()方法调用 javaChannel().register(eventLoop().selector, 0, this)将 Channel 对应的 Java NIO 的 SockerChannel 对象注册到一个 eventLoop 的 Selector 中，并且将当前 Channel 作为 attachment。

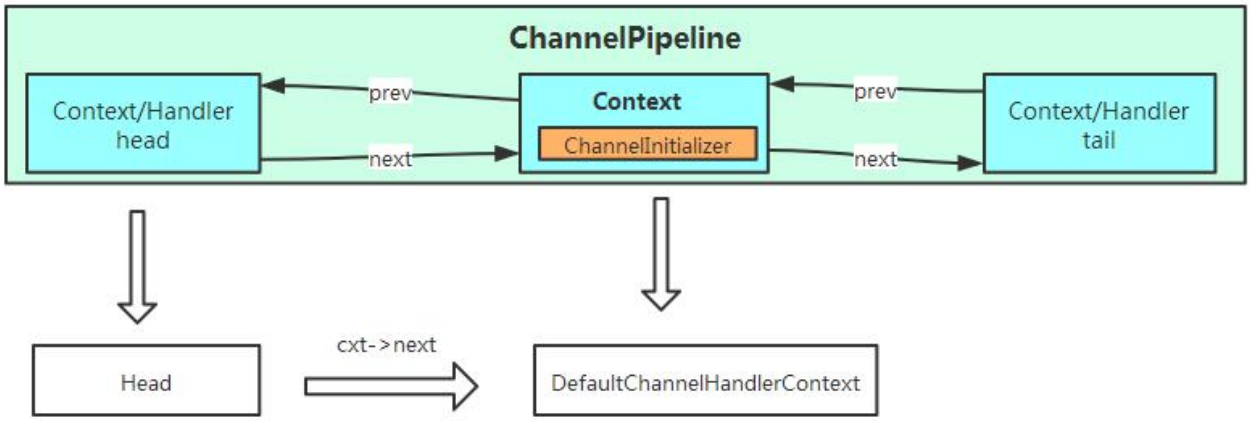
而我们自定义 ChannelHandler 的添加过程，发生在 AbstractUnsafe 的 register0()方法中，在这个方法中调用了 pipeline.fireChannelRegistered()方法，其代码实现如下：

```
public final ChannelPipeline fireChannelRegistered() {
        AbstractChannelHandlerContext.invokeChannelRegistered(head);
        return this;
}
```

再看 AbstractChannelHandlerContext 的 invokeChannelRegistered()方法：

```
static void invokeChannelRegistered(final AbstractChannelHandlerContext next) {
        EventExecutor executor = next.executor();
        if (executor.inEventLoop()) {
            next.invokeChannelRegistered();
        } else {
            executor.execute(new Runnable() {
                @Override
                public void run() {
                    next.invokeChannelRegistered();
                }
            });
        }
}
```

很显然，这个代码会从 head 开始遍历 Pipeline 的双向链表，然后找到第一个属性 inbound 为 true 的 ChannelHandlerContext 实例。想起来了没？我们在前面分析 ChannelInitializer 时，花了大量的篇幅来分析了 inbound 和 outbound 属性，现在这里就用上了。回想一下，ChannelInitializer 实现了 ChannelInboudHandler，因此它所对应的 ChannelHandlerContext 的 inbound 属性就是 true，因此这里返回就是 ChannelInitializer 实例所对应的 ChannelHandlerContext 对象，如下图所示：



当获取到 inbound 的 Context 后，就调用它的 invokeChannelRegistered()方法：

```
private void invokeChannelRegistered() {
       if (invokeHandler()) {
           try {
               ((ChannelInboundHandler) handler()).channelRegistered(this);
           } catch (Throwable t) {
               notifyHandlerException(t);
           }
       } else {
           fireChannelRegistered();
       }
}
```

我们已经知道 ,每个 ChannelHandler 都和一个 ChannelHandlerContext 关联 ,我们可以通过 ChannelHandlerContext 获取到对应的 ChannelHandler。因此很明显，这里 handler()返回的对象其实就是一开始我们实例化的 ChannelInitializer 对象，并接着调用了 ChannelInitializer 的 channelRegistered()方法。看到这里，应该会觉得有点眼熟了。ChannelInitializer 的 channelRegistered()这个方法我们在一开始的时候已经接触到了，但是我们并没有深入地分析这个方法的调用过程。下面我们来看这个方法中到底有什么玄机，继续看代码：

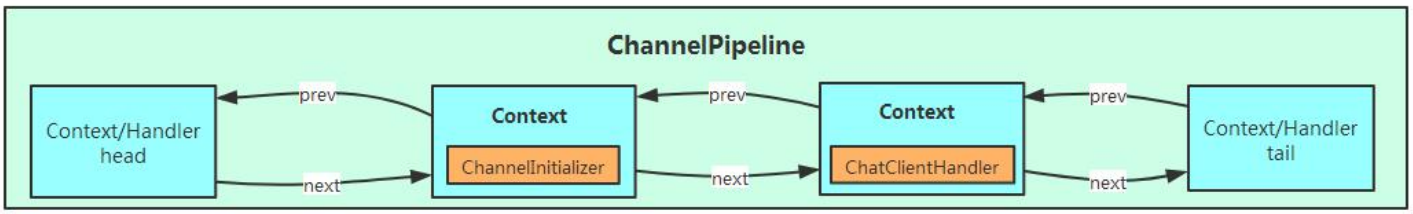
```
    public final void channelRegistered(ChannelHandlerContext ctx) throws Exception {
       if (initChannel(ctx)) {
           ctx.pipeline().fireChannelRegistered();
       } else {
           ctx.fireChannelRegistered();
       }
    }
    private boolean initChannel(ChannelHandlerContext ctx) throws Exception {
       if (initMap.putIfAbsent(ctx, Boolean.TRUE) == null) { // Guard against
           try {
               initChannel((C) ctx.channel());
           } catch (Throwable cause) {
               exceptionCaught(ctx, cause);
           } finally {
               remove(ctx);
           }
```

```
            return true;
        }
        return false;
    }
```

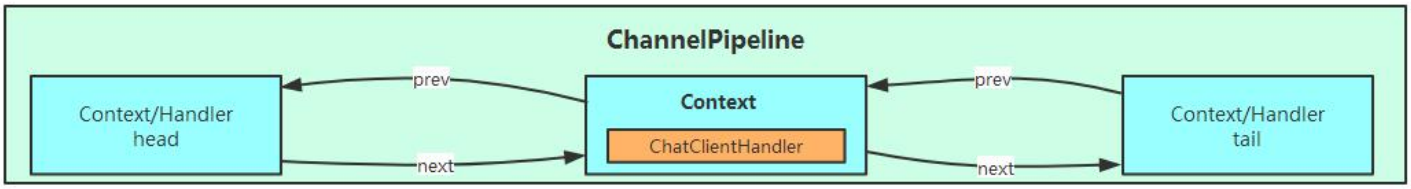
initChannel((C) ctx.channel())这个方法我们也很熟悉，它就是我们在初始化 Bootstrap 时，调用 handler 方法传入的匿名内部类所实现的方法：

```
.handler(new ChannelInitializer<SocketChannel>() {
         @Override
         protected void initChannel(SocketChannel ch) throws Exception {
            ChannelPipeline pipeline = ch.pipeline();
            pipeline.addLast(new ChatClientHandler(nickName));
         }
});
```

因此，当调用这个方法之后，我们自定义的 ChannelHandler 就插入到了 Pipeline，此时 Pipeline 的状态如下图所示：



当添加完成自定义的 ChannelHandler 后，在 finally 代码块会删除自定义的 ChannelInitializer，也就是 remove(ctx)最终调用 ctx.pipeline().remove(this)，因此最后的 Pipeline 的状态如下：



至此，自定义 ChannelHandler 的添加过程也分析得差不多了。

8.1.5 给 ChannelHandler 命名

不知道大家注意到没有，pipeline.addXXX 都有一个重载的方法，例如 addLast()它有一个重载的版本是：

```
ChannelPipeline addLast(String name, ChannelHandler handler);
```

第一个参数指定添加的 handler 的名字(更准确地说是 ChannelHandlerContext 的名字，说成 handler 的名字更便于理解)。那么 handler 的名字有什么用呢？如果我们不设置 name，那么 handler 默认的名字是怎样呢?带着这些疑问，我们依旧还是去源码中找到答案。还是以 addLast()方法为例：

```
public final ChannelPipeline addLast(String name, ChannelHandler handler) {
       return addLast(null, name, handler);
}
```

这个方法会调用重载的 addLast()方法：

```
    public final ChannelPipeline addLast(EventExecutorGroup group, String name, ChannelHandler handler) {
       final AbstractChannelHandlerContext newCtx;
       synchronized (this) {
           checkMultiplicity(handler);

           newCtx = newContext(group, filterName(name, handler), handler);

           addLast0(newCtx);
           // 略去 N 句代码
       }
       return this;
   }
```

第一个参数被设置为 null，我们不用关心它。第二参数就是这个 handler 的名字。看代码可知，在添加一个 handler 之前，需要调用 checkMultiplicity()方法来确定新添加的 handler 名字是否与已添加的 handler 名字重复。

8.1.6 ChannelHandler 默认命名规则

如果我们调用的是如下的 addLast()方法：

```
ChannelPipeline addLast(ChannelHandler... handlers);
```

那么 Netty 就会调用 generateName()方法为新添加的 handler 自动生成一个默认的名字：

```
   private String filterName(String name, ChannelHandler handler) {
       if (name == null) {
           return generateName(handler);
       }
       checkDuplicateName(name);
       return name;
   }
   private String generateName(ChannelHandler handler) {
       Map<Class<?>, String> cache = nameCaches.get();
       Class<?> handlerType = handler.getClass();
       String name = cache.get(handlerType);
       if (name == null) {
           name = generateName0(handlerType);
           cache.put(handlerType, name);
       }
       // 此处省略 N 行代码
       return name;
   }
```

而 generateName()方法会接着调用 generateName0()方法来实际生成一个新的 handler 名字：

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

默认命名的规则很简单，就是用反射获取 handler 的 simpleName 加上"#0"，因此我们自定义 ChatClientHandler 的名字就是 "ChatClientHandler#0"。

8.2 Pipeline 的事件传播机制

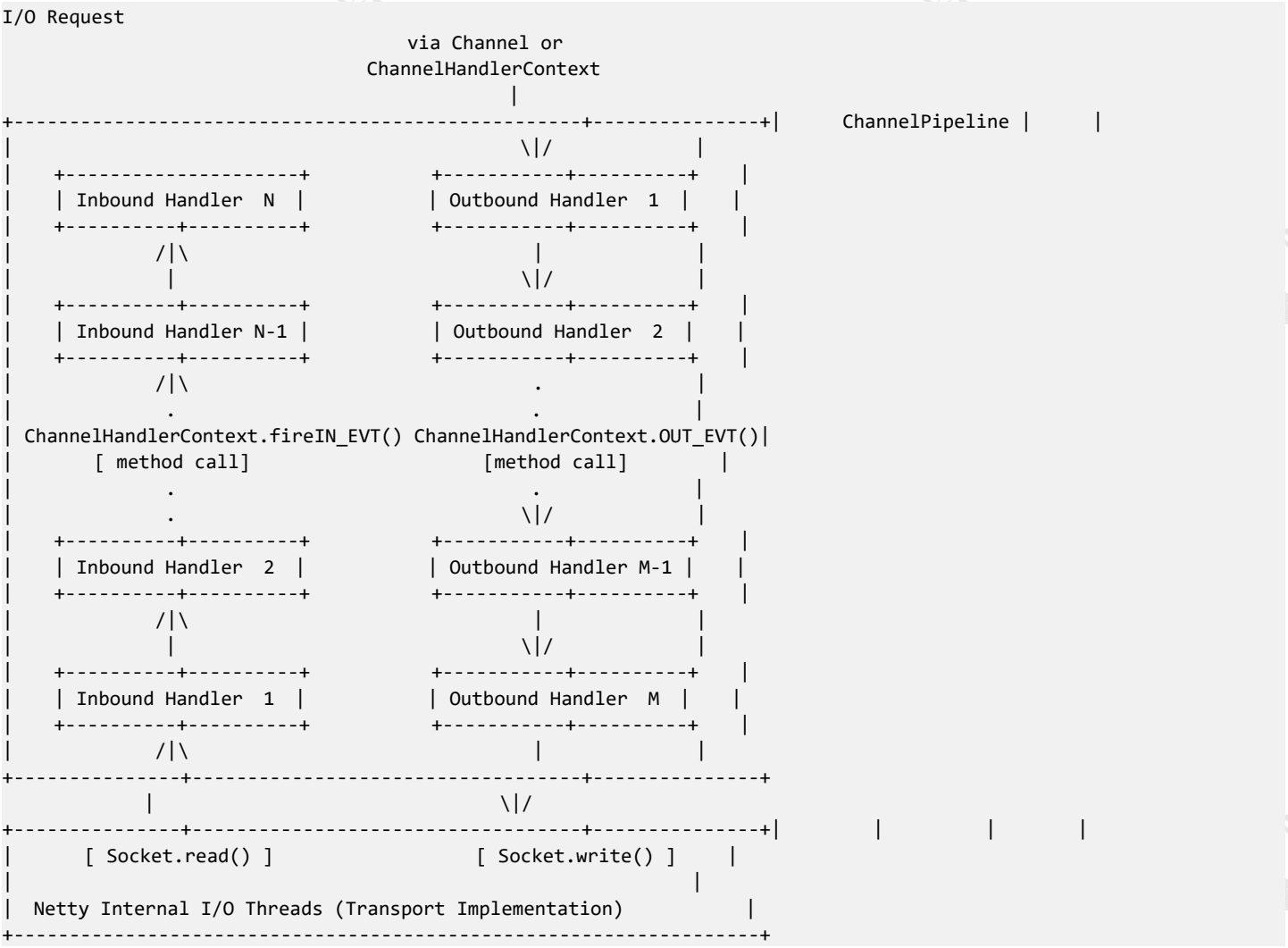
前面章节中，我们已经知道 AbstractChannelHandlerContext 中有 inbound 和 outbound 两个 boolean 变量，分别用于标识 Context 所对应的 handler 的类型，即：

1、inbound 为 true 是,表示其对应的 ChannelHandler 是 ChannelInboundHandler 的子类。

2、outbound 为 true 时，表示对应的 ChannelHandler 是 ChannelOutboundHandler 的子类。

这里大家肯定还有很多疑惑，不知道这两个字段到底有什么作用？这还要从 ChannelPipeline 的事件传播类型说起。

Netty 中的传播事件可以分为两种：Inbound 事件和 Outbound 事件。如下是从 Netty 官网针对这两个事件的说明：



从上图可以看出，inbound 事件和 outbound 事件的流向是不一样的，inbound 事件的流行是从下至上，而 outbound 刚好相反，是从上到下。并且 inbound 的传递方式是通过调用相应的 ChannelHandlerContext.fireIN_EVT()方法，而

outbound 方法的的传递方式是通过调用 ChannelHandlerContext.OUT_EVT()方法。例如：ChannelHandlerContext 的 fireChannelRegistered()调用会发送一个 ChannelRegistered 的 inbound 给下一个 ChannelHandlerContext，而 ChannelHandlerContext 的 bind()方法调用时会发送一个 bind 的 outbound 事件给下一个 ChannelHandlerContext。

Inbound 事件传播方法有：

```
public interface ChannelInboundHandler extends ChannelHandler {
    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 channelRead(ChannelHandlerContext ctx, Object msg) throws Exception;
    void channelReadComplete(ChannelHandlerContext ctx) throws Exception;
    void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception;
    void channelWritabilityChanged(ChannelHandlerContext ctx) throws Exception;
    void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception;
}
```

Outbound 事件传播方法有：

```
public interface ChannelOutboundHandler extends ChannelHandler {
    void bind(ChannelHandlerContext ctx, SocketAddress localAddress, ChannelPromise promise) throws Exception;
    void connect(
            ChannelHandlerContext ctx, SocketAddress remoteAddress,
            SocketAddress localAddress, ChannelPromise promise) throws Exception;
    void disconnect(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;
    void close(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;
    void deregister(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;
    void read(ChannelHandlerContext ctx) throws Exception;
    void flush(ChannelHandlerContext ctx) throws Exception;
}
```

大家应该发现了规律：inbound 类似于是事件回调（响应请求的事件），而 outbound 类似于主动触发（发起请求的事件）。注意，如果我们捕获了一个事件，并且想让这个事件继续传递下去，那么需要调用 Context 对应的传播方法 fireXXX，例如：

```
public class MyInboundHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
     System.out.println("连接成功");
     ctx.fireChannelActive();
    }
}

public class MyOutboundHandler extends ChannelOutboundHandlerAdapter {
    @Override
    public void close(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception {
     System.out.println("客户端关闭");
     ctx.close(promise);
    }
}
```

如上面的示例代码 :MyInboundHandler 收到了一个 channelActive 事件 ,它在处理后 ,如果希望将事件继续传播下去，

那么需要接着调用 ctx.fireChannelActive()方法。接下来我们可以用一个代码案例来感受一下，编写如下代码：

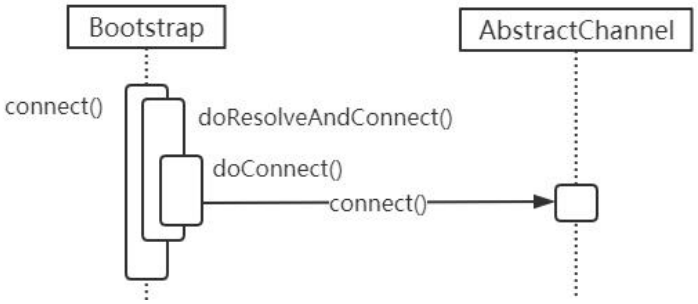
8.2.1 Outbound 事件传播方式

Outbound 事件都是请求事件(request event)，即请求某件事情的发生，然后通过 Outbound 事件进行通知。

Outbound 事件的传播方向是 tail -> customContext -> head。

我们接下来以 connect 事件为例，分析一下 Outbound 事件的传播机制。

首先，当用户调用了 Bootstrap 的 connect()方法时，就会触发一个 Connect 请求事件，此调用会触发如下调用链：



继续跟踪，我们就发现 AbstractChannel 的 connect()其实由调用了 DefaultChannelPipeline 的 connect()方法：

```
public ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise) {
    return pipeline.connect(remoteAddress, promise);
}
```

而 pipeline.connect()方法的实现如下：

```
public final ChannelFuture connect(SocketAddress remoteAddress, ChannelPromise promise) {
    return tail.connect(remoteAddress, promise);
}
```

可以看到，当 outbound 事件(这里是 connect 事件)传递到 Pipeline 后，它其实是以 tail 为起点开始传播的。

而 tail.connect()其实调用的是 AbstractChannelHandlerContext 的 connect()方法：

```
public ChannelFuture connect(
        final SocketAddress remoteAddress,
        final SocketAddress localAddress, final ChannelPromise promise) {
    //此处省略 N 句
    final AbstractChannelHandlerContext next = findContextOutbound();
    EventExecutor executor = next.executor();
        next.invokeConnect(remoteAddress, localAddress, promise);
    //此处省略 N 句
    return promise;
```

```
    }
```

findContextOutbound()方法顾名思义，它的作用是以当前 Context 为起点，向 Pipeline 中的 Context 双向链表的前端寻找第一个 outbound 属性为 true 的 Context（即关联 ChannelOutboundHandler 的 Context），然后返回。

findContextOutbound()方法代码实现如下：

```
private AbstractChannelHandlerContext findContextOutbound() {
        AbstractChannelHandlerContext ctx = this;
        do {
            ctx = ctx.prev;
        } while (!ctx.outbound);
        return ctx;
}
```

当我们找到了一个 outbound 的 Context 后，就调用它的 invokeConnect()方法，这个方法中会调用 Context 其关联的 ChannelHandler 的 connect()方法：

```
    private void invokeConnect(SocketAddress remoteAddress, SocketAddress localAddress, ChannelPromise promise) {
        if (invokeHandler()) {
            try {
                ((ChannelOutboundHandler) handler()).connect(this, remoteAddress, localAddress, promise);
            } catch (Throwable t) {
                notifyOutboundHandlerException(t, promise);
            }
        } else {
            connect(remoteAddress, localAddress, promise);
        }
    }
```

如果用户没有重写 ChannelHandler 的 connect()方法，那么会调用 ChannelOutboundHandlerAdapter 的 connect()实现：

```
    public void connect(ChannelHandlerContext ctx, SocketAddress remoteAddress,
            SocketAddress localAddress, ChannelPromise promise) throws Exception {
        ctx.connect(remoteAddress, localAddress, promise);
    }
```

我们看到，ChannelOutboundHandlerAdapter 的 connect()仅仅调用了 ctx.connect()，而这个调用又回到了：

Context.connect -> Connect.findContextOutbound -> next.invokeConnect -> handler.connect -> Context.connect

这样的循环中，直到 connect 事件传递到 DefaultChannelPipeline 的双向链表的头节点，即 head 中。为什么会传递到 head 中呢？回想一下，head 实现了 ChannelOutboundHandler，因此它的 outbound 属性是 true。

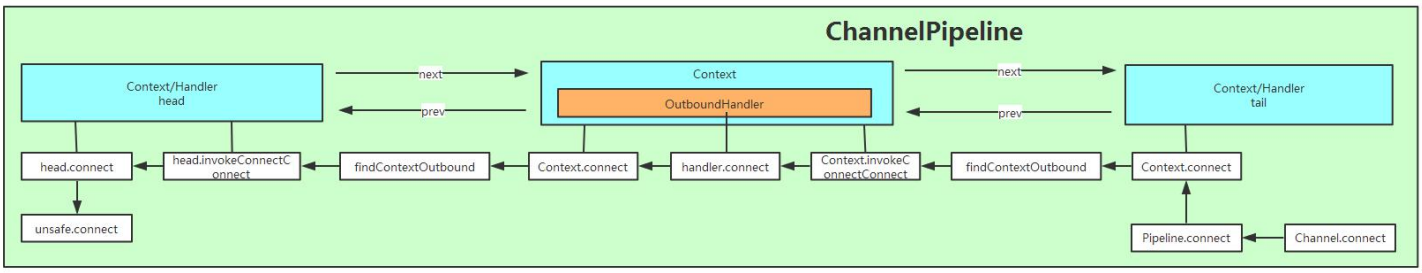
因为 head 本身既是一个 ChannelHandlerContext，又实现了 ChannelOutboundHandler 接口，因此当 connect()消息传递到 head 后，会将消息转递到对应的 ChannelHandler 中处理，而 head 的 handler()方法返回的就是 head 本身：

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

因此最终 connect()事件是在 head 中被处理。head 的 connect()事件处理逻辑如下：

```
public void connect(
              ChannelHandlerContext ctx,
              SocketAddress remoteAddress, SocketAddress localAddress,
              ChannelPromise promise) throws Exception {
          unsafe.connect(remoteAddress, localAddress, promise);
}
```

到这里，整个 connect()请求事件就结束了。下图中描述了整个 connect()请求事件的处理过程：



我们仅仅以 connect()请求事件为例，分析了 outbound 事件的传播过程，但是其实所有的 outbound 的事件传播都遵循着一样的传播规律，小伙伴们可以试着分析一下其他的 outbound 事件，体会一下它们的传播过程。

8.2.2 Inbound 事件传播方式

Inbound 事件和 Outbound 事件的处理过程是类似的，只是传播方向不同。

Inbound 事件是一个通知事件,即某件事已经发生了,然后通过 Inbound 事件进行通知。Inbound 通常发生在 Channel 的状态的改变或 IO 事件就绪。

Inbound 的特点是它传播方向是 head -> customContext -> tail。

上面我们分析了 connect()这个 Outbound 事件,那么接着分析 connect()事件后会发生什么 Inbound 事件 ,并最终找到 Outbound 和 Inbound 事件之间的联系。当 connect()这个 Outbound 传播到 unsafe 后 ,其实是在 AbstractNioUnsafe 的 connect()方法中进行处理的：

```
public final void connect(
              final SocketAddress remoteAddress,
              final SocketAddress localAddress, final ChannelPromise promise) {

              if (doConnect(remoteAddress, localAddress)) {
                  fulfillConnectPromise(promise, wasActive);
              } else {
                  ...
              }
}
```

在 AbstractNioUnsafe 的 connect()方法中，首先调用 doConnect()方法进行实际上的 Socket 连接，当连接上后会调用 fulfillConnectPromise()方法：

```
private void fulfillConnectPromise(ChannelPromise promise, boolean wasActive) {
          if (!wasActive && active) {
              pipeline().fireChannelActive();
          }
}
```

我们看到,在 fulfillConnectPromise()中 ,会通过调用 pipeline().fireChannelActive()方法将通道激活的消息(即 Socket 连接成功)发送出去。而这里，当调用 pipeline.fireXXX 后，就是 Inbound 事件的起点。因此当调用 pipeline().fireChannelActive()后，就产生了一个 ChannelActive Inbound 事件，我们就从这里开始看看这个 Inbound 事件是怎么传播的？

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

果然，在 fireChannelActive()方法中，调用的是 head.invokeChannelActive()，因此可以证明 Inbound 事件在 Pipeline 中传输的起点是 head。那么,在 head.invokeChannelActive()中又做了什么呢？

```
static void invokeChannelActive(final AbstractChannelHandlerContext next) {
       EventExecutor executor = next.executor();
       if (executor.inEventLoop()) {
           next.invokeChannelActive();
       } else {
           executor.execute(new Runnable() {
               @Override
               public void run() {
                   next.invokeChannelActive();
               }
           });
       }
}
```

上面的代码应该很熟悉了。 回想一下在 Outbound 事件(例如 connect()事件)的传输过程中时，我们也有类似的操作：

1、首先调用 findContextInbound()，从 Pipeline 的双向链表中中找到第一个属性 inbound 为 true 的 Context，然后将其返回。

2、调用 Context 的 invokeChannelActive()方法，invokeChannelActive()方法源码如下：

```
    private void invokeChannelActive() {
       if (invokeHandler()) {
           try {
               ((ChannelInboundHandler) handler()).channelActive(this);
           } catch (Throwable t) {
               notifyHandlerException(t);
           }
```

```
        } else {
            fireChannelActive();
        }
    }
```

这个方法和 Outbound 的对应方法(如：invokeConnect()方法)如出一辙。与 Outbound 一样，如果用户没有重写 channelActive() 方法，那就会调用 ChannelInboundHandlerAdapter 的 channelActive()方法：

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {
        ctx.fireChannelActive();
}
```

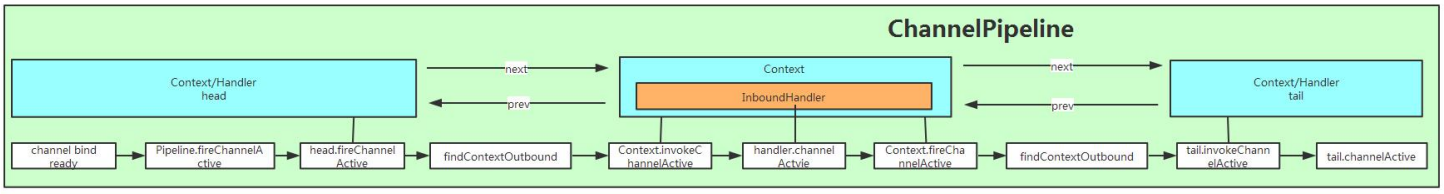
同样地，在 ChannelInboundHandlerAdapter 的 channelActive()中，仅仅调用了 ctx.fireChannelActive()方法，因此就会进入 Context.fireChannelActive() -> Connect.findContextInbound() -> nextContext.invokeChannelActive() -> nextHandler.channelActive() -> nextContext.fireChannelActive()这样的循环中。同理，tail 本身既实现了 ChannelInboundHandler 接口，又实现了 ChannelHandlerContext 接口，因此当 channelActive()消息传递到 tail 后，会将消息转递到对应的 ChannelHandler 中处理，而 tail 的 handler()返回的就是 tail 本身：

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

因此 channelActive Inbound 事件最终是在 tail 中处理的，我们看一下它的处理方法：

```
public void channelActive(ChannelHandlerContext ctx) throws Exception {
}
```

TailContext 的 channelActive()方法是空的。如果大家自行查看 TailContext 的 Inbound 处理方法时就会发现，它们的实现都是空的。可见，如果是 Inbound,当用户没有实现自定义的处理器时，那么默认是不处理的。下图描述了 Inbound 事件的传输过程：



8.2.3 Pipeline 事件传播小结

Outbound 事件总结:

1、Outbound 事件是请求事件（由 connect()发起一个请求，并最终由 unsafe 处理这个请求）。

2、Outbound 事件的发起者是 Channel。

3、Outbound 事件的处理者是 unsafe。

4、Outbound 事件在 Pipeline 中的传输方向是 tail -> head。

5、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 的方法（如：ctx.connect()方法)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。

6、Outbound 事件流 :Context.OUT_EVT() -> Connect.findContextOutbound() -> nextContext.invokeOUT_EVT() -> nextHandler.OUT_EVT() -> nextContext.OUT_EVT()

Inbound 事件总结:

1、Inbound 事件是通知事件，当某件事情已经就绪后，通知上层。

2、Inbound 事件发起者是 unsafe。

3、Inbound 事件的处理者是 Channe，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。

4、Inbound 事件在 Pipeline 中传输方向是 head -> tail。

5、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()事件（如：ctx.fireChannelActive()方法）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。

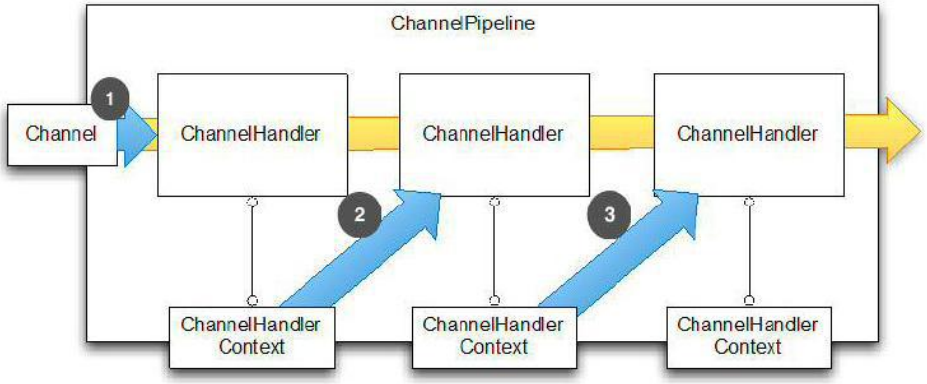
6、Outbound 事件流 :Context.fireIN_EVT() -> Connect.findContextInbound() -> nextContext.invokeIN_EVT() -> nextHandler.IN_EVT() -> nextContext.fireIN_EVT().

outbound 和 inbound 事件设计上十分相似，并且 Context 与 Handler 直接的调用关系也容易混淆，因此我们在阅读这里的源码时，需要特别的注意。

8.3 Handler 的各种姿势

8.3.1 ChannelHandlerContext

每个 ChannelHandler 被添加到 ChannelPipeline 后，都会创建一个 ChannelHandlerContext 并与之创建的 ChannelHandler 关联绑定。ChannelHandlerContext 允许 ChannelHandler 与其他的 ChannelHandler 实现进行交互。ChannelHandlerContext 不会改变添加到其中的 ChannelHandler，因此它是安全的。下图描述了 ChannelHandlerContext、ChannelHandler、ChannelPipeline 的关系：

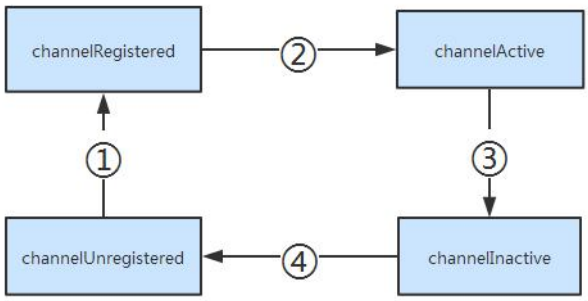


8.3.2 Channel 的生命周期

Netty 有一个简单但强大的状态模型，并完美映射到 ChannelInboundHandler 的各个方法。下面是 Channel 生命周期中四个不同的状态：

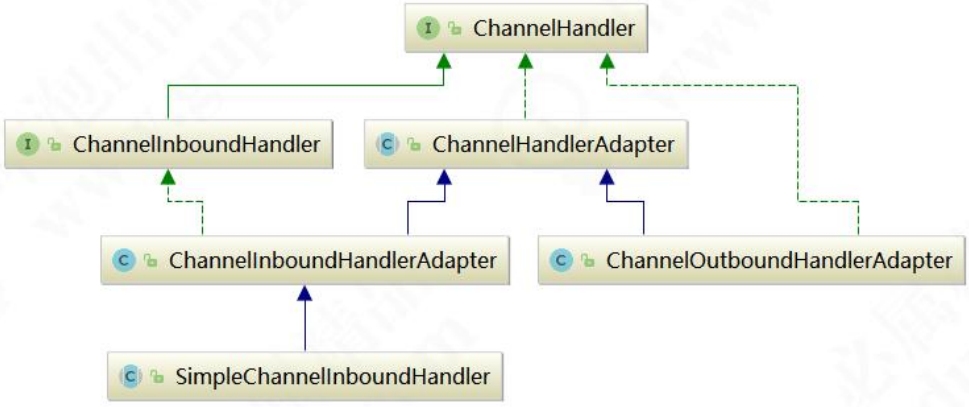
状态	描述
channelUnregistered()	Channel已创建，还未注册到一个EventLoop上
channelRegistered()	Channel已经注册到一个EventLoop上
channelActive()	Channel是活跃状态（连接到某个远端），可以收发数据
channelInactive()	Channel未连接到远端

一个 Channel 正常的生命周期如下图所示。随着状态发生变化相应的事件产生。这些事件被转发到 ChannelPipeline 中的 ChannelHandler 来触发相应的操作。



8.3.3 ChannelHandler 常用的 API

先看一下 Netty 中整个 Handler 体系的类关系图：



Netty 定义了良好的类型层次结构来表示不同的处理程序类型 ,所有的类型的父类是 ChannelHandler。ChannelHandler 提供了在其生命周期内添加或从 ChannelPipeline 中删除的方法。

状态	描述
handlerAdded()	ChannelHandler添加到实际上下文中准备处理事件
handlerRemoved()	将ChannelHandler从实际上下文中删除，不再处理事件
exceptionCaught()	处理抛出的异常

Netty 还提供了一个实现了 ChannelHandler 的抽象类 ChannelHandlerAdapter。ChannelHandlerAdapter 实现了父类的所有方法，基本上就是传递事件到 ChannelPipeline 中的下一个 ChannelHandler 直到结束。我们也可以直接继承于 ChannelHandlerAdapter，然后重写里面的方法。

8.3.5 ChannelInboundHandler

ChannelInboundHandler 提供了一些方法再接收数据或 Channel 状态改变时被调用。下面是 ChannelInboundHandler 的一些方法：

状态	描述
channelRegistered()	ChannelHandlerContext的Channel被注册到EventLoop
channelUnregistered()	ChannelHandlerContext的Channel从EventLoop中注销
channelActive()	ChannelHandlerContext的Channel已激活

channelInactive	ChannelHanderContxt的Channel结束生命周期
channelRead	从当前Channel的对端读取消息
channelReadComplete	消息读取完成后执行
userEventTriggered	一个用户事件被触发
channelWritabilityChanged	改变通道的可写状态，可以使用Channel.isWritable()检查
exceptionCaught	重写父类ChannelHandler的方法，处理异常

Netty 提供了一个实现了 ChannelInboundHandler 接口并继承 ChannelHandlerAdapter 的类：ChannelInboundHandlerAdapter。ChannelInboundHandlerAdapter 实现了 ChannelInboundHandler 的所有方法，作用就是处理消息并将消息转发到 ChannelPipeline 中的下一个 ChannelHandler。ChannelInboundHandlerAdapter 的 channelRead() 方法处理完消息后不会自动释放消息，若想自动释放收到的消息，可以使用 SimpleChannelInboundHandler，看下面的代码：

```
public class UnreleaseHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        //手动释放消息
        ReferenceCountUtil.release(msg);
    }
}
```

SimpleChannelInboundHandler 会自动释放消息：

```
public class ReleaseHandler extends SimpleChannelInboundHandler<Object> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws Exception {
        //不需要手动释放
    }
}
```

ChannelInitializer 用来初始化 ChannelHandler，将自定义的各种 ChannelHandler 添加到 ChannelPipeline 中。