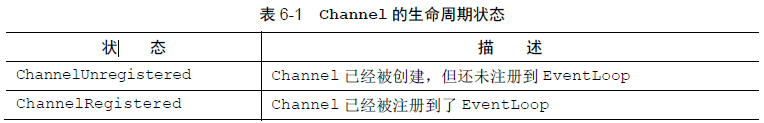
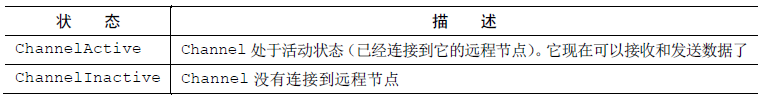
# 【第06章-ChannelHandler和ChannelPipeline】

## 6.1 ChannelHandler 家族

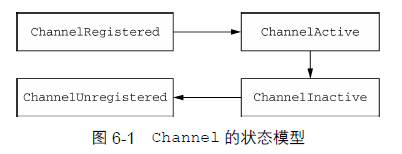
### 6.1.1 Channel 的生命周期

Interface Channel 定义了一组和ChannelInboundHandler API 密切相关的简单但功能强大的状态模型，表6-1 列出了Channel 的这4 个状态。



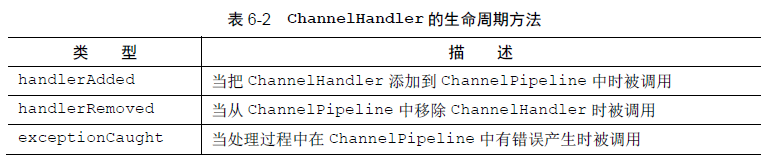


Channel 的正常生命周期如图6-1 所示。当这些状态发生改变时，将会生成对应的事件。这些事件将会被转发给ChannelPipeline 中的ChannelHandler，其可以随后对它们做出响应。



### 6.1.2 ChannelHandler 的生命周期

表6-2 中列出了interface ChannelHandler 定义的生命周期操作，在ChannelHandler被添加到ChannelPipeline 中或者被从ChannelPipeline 中移除时会调用这些操作。这些方法中的每一个都接受一个ChannelHandlerContext 参数。



Netty 定义了下面两个重要的ChannelHandler 子接口：

* ChannelInboundHandler——处理入站数据以及各种状态变化；
* ChannelOutboundHandler——处理出站数据并且允许拦截所有的操作。

在接下来的章节中，我们将详细地讨论这些子接口。

### 6.1.3 ChannelInboundHandler 接口

表6-3 列出了interface ChannelInboundHandler 的生命周期方法。这些方法将会在数据被接收时或者与其对应的Channel 状态发生改变时被调用。正如我们前面所提到的，这些方法和Channel 的生命周期密切相关。



（1）当所有可读的字节都已经从Channel 中读取之后，将会调用该回调方法；所以，可能在channelRead-Complete()被调用之前看到多次调用channelRead(...)。

当某个ChannelInboundHandler 的实现重写channelRead()方法时，它将负责显式地释放与池化的ByteBuf 实例相关的内存。Netty 为此提供了一个实用方法ReferenceCount-Util.release()，如代码清单6-1 所示。

|  |
| --- |
| *// 代码清单6-1 释放消息资源*  *// 扩展了ChannelInboundHandlerAdapter* @ChannelHandler.Sharable **public class** DiscardHandler **extends** ChannelInboundHandlerAdapter {   @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) {  *// 丢弃已接收的消息* ReferenceCountUtil.*release*(msg);  } } |

Netty 将使用WARN 级别的日志消息记录未释放的资源，使得可以非常简单地在代码中发现违规的实例。但是以这种方式管理资源可能很繁琐。一个更加简单的方式是使用SimpleChannelInboundHandler。代码清单6-2 是代码清单6-1 的一个变体，说明了这一点。

|  |
| --- |
| @Sharable **public class** SimpleDiscardHandler **extends** SimpleChannelInboundHandler<Object> {  @Override  **public void** channelRead0(ChannelHandlerContext ctx,  Object msg) {  *// 不需要任何显式的资源释放  // No need to do anything special* } } |

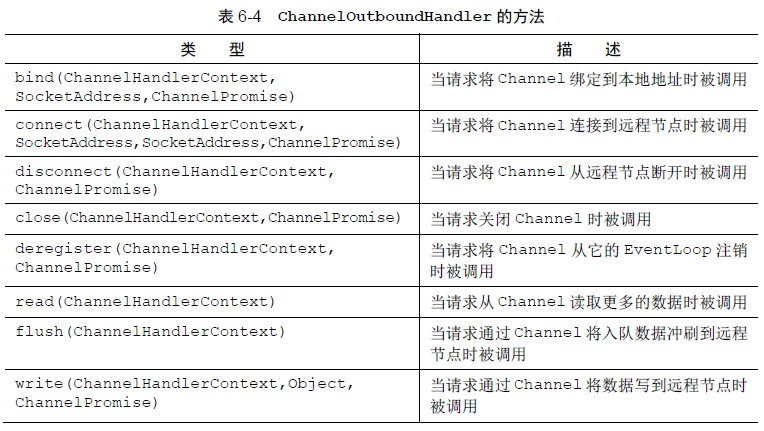
由于SimpleChannelInboundHandler 会自动释放资源，所以你不应该存储指向任何消息的引用供将来使用，因为这些引用都将会失效。

### 6.1.4 ChannelOutboundHandler 接口

出站操作和数据将由ChannelOutboundHandler 处理。它的方法将被Channel、ChannelPipeline 以及ChannelHandlerContext 调用。

ChannelOutboundHandler 的一个强大的功能是可以按需推迟操作或者事件，这使得可以通过一些复杂的方法来处理请求。例如，如果到远程节点的写入被暂停了，那么你可以推迟冲刷操作并在稍后继续。

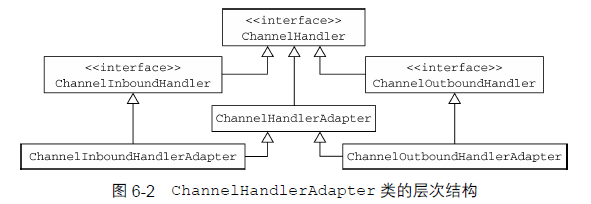
表6-4 显示了所有由ChannelOutboundHandler 本身所定义的方法（忽略了那些从ChannelHandler 继承的方法）。



ChannelPromise与ChannelFuture ChannelOutboundHandler中的大部分方法都需要一个ChannelPromise参数，以便在操作完成时得到通知。ChannelPromise是ChannelFuture的一个子类，其定义了一些可写的方法，如setSuccess()和setFailure()，从而使ChannelFuture不可变，这里借鉴的是Scala 的Promise 和Future 的设计，当一个Promise 被完成之后，其对应的Future 的值便不能再进行任何修改了。

### 6.1.5 ChannelHandler 适配器

你可以使用ChannelInboundHandlerAdapter 和ChannelOutboundHandlerAdapter类作为自己的ChannelHandler 的起始点。这两个适配器分别提供了ChannelInboundHandler和ChannelOutboundHandler 的基本实现。通过扩展抽象类ChannelHandlerAdapter，它们获得了它们共同的超接口ChannelHandler 的方法。生成的类的层次结构如图6-2 所示。



ChannelHandlerAdapter 还提供了实用方法isSharable()。如果其对应的实现被标注为Sharable，那么这个方法将返回true，表示它可以被添加到多个ChannelPipeline中。

在ChannelInboundHandlerAdapter 和ChannelOutboundHandlerAdapter 中所提供的方法体调用了其相关联的ChannelHandlerContext 上的等效方法，从而将事件转发到了ChannelPipeline 中的下一个ChannelHandler 中。

你要想在自己的ChannelHandler 中使用这些适配器类，只需要简单地扩展它们，并且重写那些你想要自定义的方法。

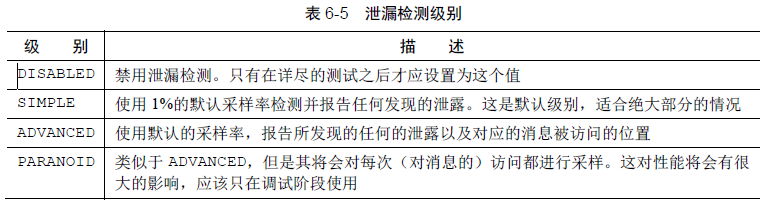
### 6.1.6 资源管理

每当通过调用ChannelInboundHandler.channelRead()或者ChannelOutboundHandler.write()方法来处理数据时，你都需要确保没有任何的资源泄漏。Netty 使用引用计数来处理池化的ByteBuf。所以在完全使用完某个ByteBuf 后，调整其引用计数是很重要的。

为了帮助你诊断潜在的（资源泄漏）问题，Netty提供了class ResourceLeakDetector。它将对你应用程序的缓冲区分配做大约1%的采样来检测内存泄露。相关的开销是非常小的。如果检测到了内存泄露，将会产生类似于下面的日志消息：

|  |
| --- |
| LEAK: ByteBuf.release() was not called before it's garbage-collected. Enable  advanced leak reporting to find out where the leak occurred. To enable  advanced leak reporting, specify the JVM option  '-Dio.netty.leakDetectionLevel=ADVANCED' or call  Resourc eLeakDetector.setLevel(). |

Netty 目前定义了4 种泄漏检测级别，如表6-5 所示。



泄露检测级别可以通过将下面的Java 系统属性设置为表中的一个值来定义：

java -D io.netty.leakDetectionLevel=ADVANCED

如果带着该JVM 选项重新启动你的应用程序，你将看到自己的应用程序最近被泄漏的缓冲区被访问的位置。下面是一个典型的由单元测试产生的泄漏报告：

Running io.netty.handler.codec.xml.XmlFrameDecoderTest

15:03:36.886 [main] ERROR io.netty.util.ResourceLeakDetector - LEAK:

ByteBuf.release() was not called before it's garbage-collected.

Recent access records: 1

#1: io.netty.buffer.AdvancedLeakAwareByteBuf.toString(

AdvancedLeakAwareByteBuf.java:697)

io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithXml(

XmlFrameDecoderTest.java:157)

io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithTwoMessages(

XmlFrameDecoderTest.java:133)

...

实现ChannelInboundHandler.channelRead()和ChannelOutboundHandler.write()

方法时，应该如何使用这个诊断工具来防止泄露呢？让我们看看你的channelRead()操作直接消费入站消息的情况；也就是说，它不会通过调用ChannelHandlerContext.fireChannelRead()方法将入站消息转发给下一个ChannelInboundHandler。代码清单6-3 展示了如何释放消息。

|  |
| --- |
| *// 代码清单6-3 消费并释放入站消息*  *// 扩展了ChannelInboundandlerAdapter* @ChannelHandler.Sharable **public class** DiscardInboundHandler **extends** ChannelInboundHandlerAdapter {  @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) {  *// 通过调用ReferenceCountUtil.release() 方法释放资源* ReferenceCountUtil.*release*(msg);  } } |

消费入站消息的简单方式 由于消费入站数据是一项常规任务，所以Netty 提供了一个特殊的被称为SimpleChannelInboundHandler 的ChannelInboundHandler 实现。这个实现会在消息被channelRead0()方法消费之后自动释放消息。

在出站方向这边，如果你处理了write()操作并丢弃了一个消息，那么你也应该负责释放它。代码清单6-4 展示了一个丢弃所有的写入数据的实现。

|  |
| --- |
| *// 扩展了ChannelInboundHandlerAdapter* @Sharable **public class** DiscardOutboundHandler  **extends** ChannelOutboundHandlerAdapter {  @Override  **public void** write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) {  *// 通过调用ReferenceCountUtil.release() 方法释放资源* ReferenceCountUtil.*release*(msg);  *// 通知ChannelPromise数据已经被处理了* promise.setSuccess();  } } |

重要的是，不仅要释放资源，还要通知ChannelPromise。否则可能会出现ChannelFutureListener 收不到某个消息已经被处理了的通知的情况。

总之，如果一个消息被消费或者丢弃了，并且没有传递给ChannelPipeline 中的下一个ChannelOutboundHandler，那么用户就有责任调用ReferenceCountUtil.release()。如果消息到达了实际的传输层，那么当它被写入时或者Channel 关闭时，都将被自动释放。

## 6.2 ChannelPipeline 接口

如果你认为ChannelPipeline是一个拦截流经Channel的入站和出站事件的ChannelHandler 实例链，那么就很容易看出这些ChannelHandler 之间的交互是如何组成一个应用程序数据和事件处理逻辑的核心的。

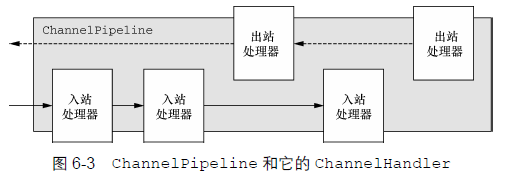
每一个新创建的Channel 都将会被分配一个新的ChannelPipeline。这项关联是永久性的；Channel 既不能附加另外一个ChannelPipeline，也不能分离其当前的。在Netty 组件的生命周期中，这是一项固定的操作，不需要开发人员的任何干预。

根据事件的起源，事件将会被ChannelInboundHandler 或者ChannelOutboundHandler处理。随后，通过调用ChannelHandlerContext 实现，它将被转发给同一超类型的下一个ChannelHandler。

**ChannelHandlerContext**

ChannelHandlerContext使得ChannelHandler能够和它的ChannelPipeline以及其他的ChannelHandler 交互。ChannelHandler 可以通知其所属的ChannelPipeline 中的下一个ChannelHandler，甚至可以动态修改它所属的ChannelPipeline，这里指修改ChannelPipeline 中的ChannelHandler 的编排。ChannelHandlerContext 具有丰富的用于处理事件和执行I/O 操作的API。

图6-3 展示了一个典型的同时具有入站和出站ChannelHandler 的ChannelPipeline 的布局，并且印证了我们之前的关于ChannelPipeline 主要由一系列的ChannelHandler 所组成的说法。ChannelPipeline 还提供了通过ChannelPipeline 本身传播事件的方法。如果一个入站事件被触发，它将被从ChannelPipeline 的头部开始一直被传播到Channel Pipeline 的尾端。在图6-3 中，一个出站I/O 事件将从ChannelPipeline 的最右边开始，然后向左传播。



**ChannelPipeline 相对论**

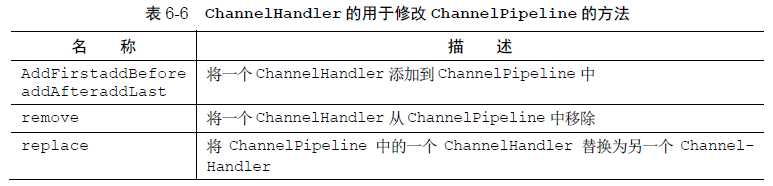
你可能会说，从事件途经ChannelPipeline 的角度来看，ChannelPipeline 的头部和尾端取决于该事件是入站的还是出站的。然而Netty 总是将ChannelPipeline 的入站口（图6-3 中的左侧）作为头部，而将出站口（该图的右侧）作为尾端。

当你完成了通过调用ChannelPipeline.add\*()方法将入站处理器（ChannelInboundHandler）和出站处理器（ ChannelOutboundHandler ） 混合添加到ChannelPipeline 之后， 每一个ChannelHandler 从头部到尾端的顺序位置正如同我们方才所定义它们的一样。因此，如果你将图6-3 中的处理器（ChannelHandler）从左到右进行编号，那么第一个被入站事件看到的ChannelHandler 将是1，而第一个被出站事件看到的ChannelHandler 将是5。

在ChannelPipeline 传播事件时，它会测试ChannelPipeline 中的下一个ChannelHandler 的类型是否和事件的运动方向相匹配。如果不匹配，ChannelPipeline 将跳过该ChannelHandler 并前进到下一个，直到它找到和该事件所期望的方向相匹配的为止。（当然，ChannelHandler 也可以同时实现ChannelInboundHandler 接口和ChannelOutboundHandler 接口。）

### 6.2.1 修改ChannelPipeline

ChannelHandler 可以通过添加、删除或者替换其他的ChannelHandler 来实时地修改ChannelPipeline 的布局。（它也可以将它自己从ChannelPipeline 中移除。）这是ChannelHandler 最重要的能力之一，所以我们将仔细地来看看它是如何做到的。表6-6 列出了相关的方法。



代码清单6-5 展示了这些方法的使用。

|  |
| --- |
| **// 代码清单6-5 修改ChannelPipeline**  **public class** ModifyChannelPipeline {  **private static final** ChannelPipeline ***CHANNEL\_PIPELINE\_FROM\_SOMEWHERE*** = ***DUMMY\_INSTANCE***;   */\*\*  \* Listing 6.5 Modify the ChannelPipeline  \*/* **public static void** modifyPipeline() {  ChannelPipeline pipeline = ***CHANNEL\_PIPELINE\_FROM\_SOMEWHERE***; *// get reference to pipeline;* FirstHandler firstHandler = **new** FirstHandler();  *// 将该实例作为"handler1" 添加到ChannelPipeline 中* pipeline.addLast(**"handler1"**, firstHandler);  *// 将一个SecondHandler的实例作为"handler2"添加到ChannelPipeline的第一个槽中。  // 这意味着它将被放置在已有的"handler1"之前* pipeline.addFirst(**"handler2"**, **new** SecondHandler());  *// 将一个ThirdHandler 的实例作为"handler3"添加到ChannelPipeline 的最后一个槽中* pipeline.addLast(**"handler3"**, **new** ThirdHandler());  *//...  // 通过名称移除"handler3"* pipeline.remove(**"handler3"**);  *// 通过引 用移除FirstHandler（它是唯一的，所以不需要它的名称）* pipeline.remove(firstHandler);  *// 将SecondHandler("handler2")替换为FourthHandler:"handler4"* pipeline.replace(**"handler2"**, **"handler4"**, **new** FourthHandler());   }   **private static final class** FirstHandler **extends** ChannelHandlerAdapter {  }   **private static final class** SecondHandler **extends** ChannelHandlerAdapter {   }   **private static final class** ThirdHandler **extends** ChannelHandlerAdapter {  }   **private static final class** FourthHandler **extends** ChannelHandlerAdapter {  } } |

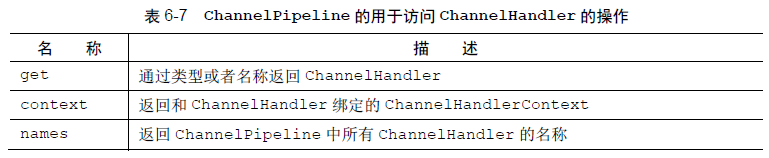
稍后，你将看到，重组ChannelHandler 的这种能力使我们可以用它来轻松地实现极其灵活的逻辑。

**ChannelHandler 的执行和阻塞**

通常ChannelPipeline 中的每一个ChannelHandler 都是通过它的EventLoop（I/O 线程）来处理传递给它的事件的。所以至关重要的是不要阻塞这个线程，因为这会对整体的I/O 处理产生负面的影响。

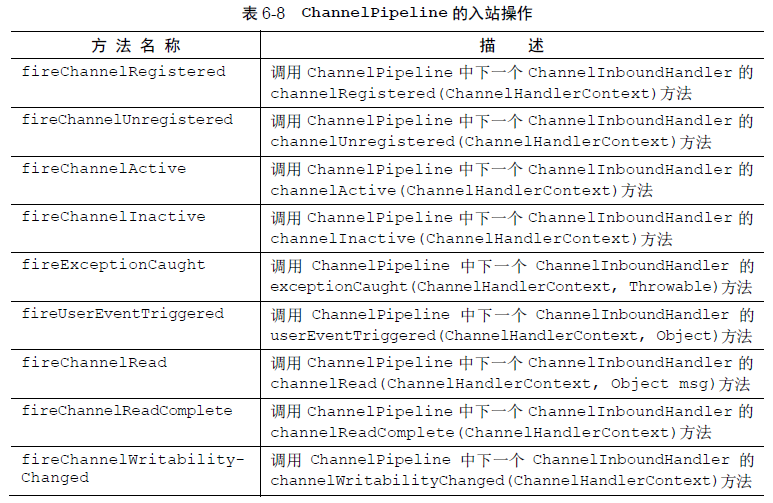
但有时可能需要与那些使用阻塞API 的遗留代码进行交互。对于这种情况，ChannelPipeline 有一些接受一个EventExecutorGroup 的add()方法。如果一个事件被传递给一个自定义的ventExecutorGroup，它将被包含在这个EventExecutorGroup 中的某个EventExecutor 所处理，从而被从该Channel 本身的EventLoop 中移除。对于这种用例，Netty 提供了一个叫DefaultEventExecutorGroup 的默认实现。

除了这些操作，还有别的通过类型或者名称来访问ChannelHandler 的方法。这些方法都列在了表6-7 中。



### 6.2.2 触发事件

ChannelPipeline 的API 公开了用于调用入站和出站操作的附加方法。表6-8 列出了入站操作，用于通知ChannelInboundHandler 在ChannelPipeline 中所发生的事件。



在出站这边，处理事件将会导致底层的套接字上发生一系列的动作。表6-9 列出了ChannelPipeline API 的出站操作。



总结一下：

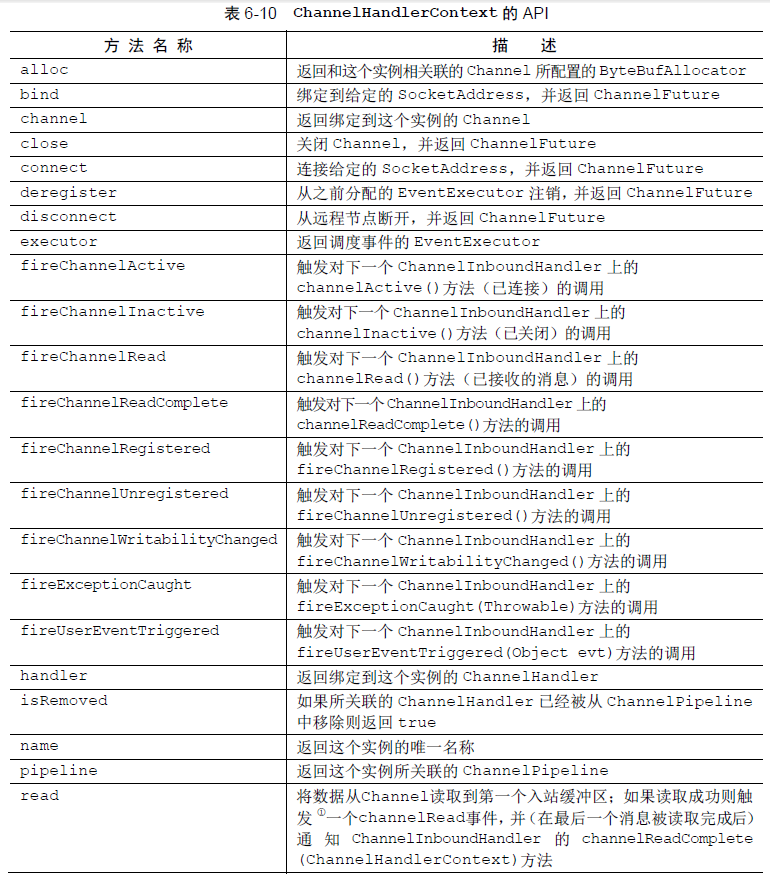
* ChannelPipeline 保存了与Channel 相关联的ChannelHandler；
* ChannelPipeline 可以根据需要，通过添加或者删除ChannelHandler 来动态地修改；
* ChannelPipeline 有着丰富的API 用以被调用，以响应入站和出站事件。

## 6.3 ChannelHandlerContext 接口

ChannelHandlerContext 代表了ChannelHandler 和ChannelPipeline 之间的关联，每当有ChannelHandler 添加到ChannelPipeline 中时，都会创建ChannelHandlerContext。ChannelHandlerContext 的主要功能是管理它所关联的ChannelHandler 和在同一个ChannelPipeline 中的其他ChannelHandler 之间的交互。

ChannelHandlerContext 有很多的方法，其中一些方法也存在于Channel 和ChannelPipeline 本身上，但是有一点重要的不同。如果调用Channel 或者ChannelPipeline 上的这些方法，它们将沿着整个ChannelPipeline 进行传播。而调用位于ChannelHandlerContext上的相同方法，则将从当前所关联的ChannelHandler 开始，并且只会传播给位于该ChannelPipeline 中的下一个能够处理该事件的ChannelHandler。

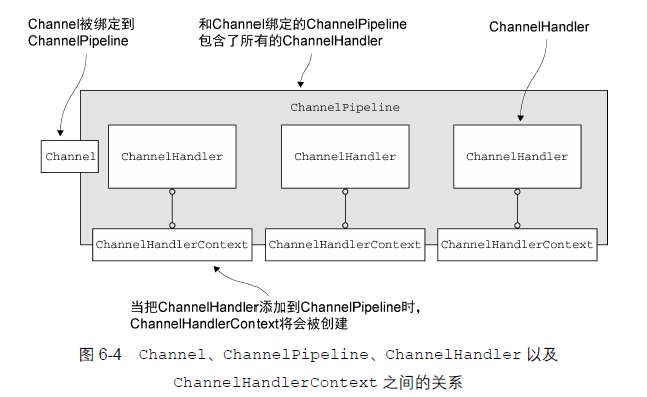
表6-10 对ChannelHandlerContext API 进行了总结。



通过配合ChannelConfig.setAutoRead(boolean autoRead)方法，可以实现反应式系统的特性之一回压（back-pressure）

### 6.3.1 使用ChannelHandlerContext

在这一节中我们将讨论ChannelHandlerContext 的用法，以及存在于ChannelHandler-Context、Channel 和ChannelPipeline 上的方法的行为。图6-4 展示了它们之间的关系。



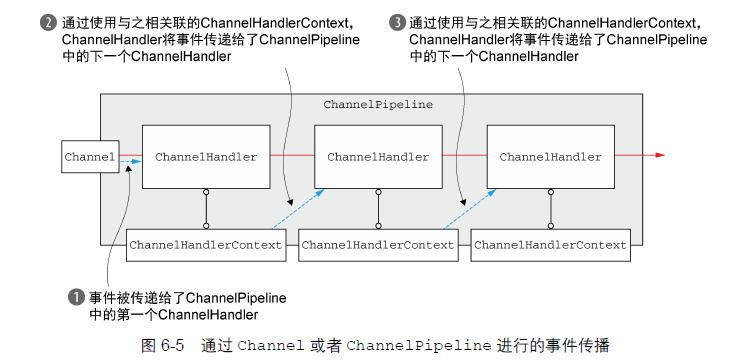
在代码清单6-6 中，将通过ChannelHandlerContext 获取到Channel 的引用。调用Channel 上的write()方法将会导致写入事件从尾端到头部地流经ChannelPipeline。

|  |
| --- |
| **// 代码清单6-6 从ChannelHandlerContext 访问Channel**  **public static void** writeViaChannel() {  ChannelHandlerContext ctx = ***CHANNEL\_HANDLER\_CONTEXT\_FROM\_SOMEWHERE***; *//get reference form somewhere  // 获取到与ChannelHandlerContext相关联的Channel 的引用* Channel channel = ctx.channel();  channel.write(Unpooled.*copiedBuffer*(**"Netty in Action"**, CharsetUtil.***UTF\_8***)); } |

代码清单6-7 展示了一个类似的例子，但是这一次是写入ChannelPipeline。我们再次看到，（到ChannelPipline 的）引用是通过ChannelHandlerContext 获取的。

|  |
| --- |
| **// 代码清单6-7 通过ChannelHandlerContext 访问ChannelPipeline**  **public static void** writeViaChannelPipeline() {  ChannelHandlerContext ctx = ***CHANNEL\_HANDLER\_CONTEXT\_FROM\_SOMEWHERE***; *//get reference form somewhere  // 获取到与ChannelHandlerContext相关联的ChannelPipeline的引用* ChannelPipeline pipeline = ctx.pipeline(); *//get reference form somewhere  // 通过Channel 写入缓冲区* pipeline.write(Unpooled.*copiedBuffer*(**"Netty in Action"**, CharsetUtil.***UTF\_8***)); } |

如同在图6-5 中所能够看到的一样，代码清单6-6 和代码清单6-7 中的事件流是一样的。重要的是要注意到，虽然被调用的Channel 或ChannelPipeline 上的write()方法将一直传播事件通过整个ChannelPipeline，但是在ChannelHandler 的级别上，事件从一个ChannelHandler到下一个ChannelHandler 的移动是由ChannelHandlerContext 上的调用完成的。



为什么会想要从ChannelPipeline 中的某个特定点开始传播事件呢？

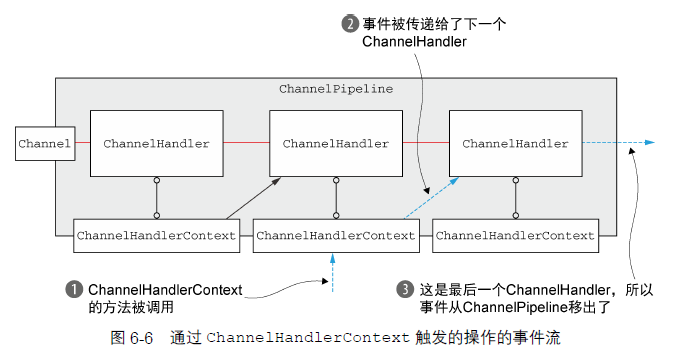
* 为了减少将事件传经对它不感兴趣的ChannelHandler 所带来的开销。
* 为了避免将事件传经那些可能会对它感兴趣的ChannelHandler。

要想调用从某个特定的ChannelHandler 开始的处理过程，必须获取到在（ChannelPipeline）该ChannelHandler 之前的ChannelHandler 所关联的ChannelHandlerContext。这个ChannelHandlerContext 将调用和它所关联的ChannelHandler 之后的ChannelHandler。

代码清单6-8 和图6-6 说明了这种用法。

|  |
| --- |
| **//代码清单6-8 调用ChannelHandlerContext 的write()方法**  **public static void** writeViaChannelHandlerContext() {  *// 获取到ChannelHandlerContext的引用* ChannelHandlerContext ctx = ***CHANNEL\_HANDLER\_CONTEXT\_FROM\_SOMEWHERE***; *//get reference form somewhere;  // write()方法将把缓冲区数据发送到下一个ChannelHandler* ctx.write(Unpooled.*copiedBuffer*(**"Netty in Action"**, CharsetUtil.***UTF\_8***)); } |

如图6-6 所示，消息将从下一个ChannelHandler 开始流经ChannelPipeline，绕过了所有前面的ChannelHandler。



我们刚才所描述的用例是常见的，对于调用特定的ChannelHandler 实例上的操作尤其有用。

### 6.3.2 ChannelHandler 和ChannelHandlerContext 的高级用法

正如我们在代码清单6-6 中所看到的，你可以通过调用ChannelHandlerContext 上的pipeline()方法来获得被封闭的ChannelPipeline 的引用。这使得运行时得以操作ChannelPipeline 的ChannelHandler，我们可以利用这一点来实现一些复杂的设计。例如，你可以通过将ChannelHandler 添加到ChannelPipeline 中来实现动态的协议切换。

另一种高级的用法是缓存到ChannelHandlerContext 的引用以供稍后使用，这可能会发生在任何的ChannelHandler 方法之外，甚至来自于不同的线程。代码清单6-9 展示了用这种模式来触发事件。

|  |
| --- |
| **//代码清单 6-9 缓存到 ChannelHandlerContext 的引用**  **public class** WriteHandler **extends** ChannelHandlerAdapter {  **private** ChannelHandlerContext **ctx**;   @Override  **public void** handlerAdded(ChannelHandlerContext ctx) {  *// 存储到 ChannelHandlerContext的引用以供稍后使用* **this**.**ctx** = ctx;  }   **public void** send(String msg) {  *// 使用之前存储的到ChannelHandlerContext的引用来发送消息* **ctx**.writeAndFlush(msg);  } } |

因为一个ChannelHandler 可以从属于多个ChannelPipeline，所以它也可以绑定到多个ChannelHandlerContext 实例。对于这种用法指在多个ChannelPipeline 中共享同一个ChannelHandler，对应的ChannelHandler 必须要使用@Sharable 注解标注；否则，试图将它添加到多个ChannelPipeline 时将会触发异常。显而易见，为了安全地被用于多个并发的Channel（即连接），这样的ChannelHandler 必须是线程安全的。

代码清单6-10 展示了这种模式的一个正确实现。

|  |
| --- |
| *// 代码清单 6-10 可共享的 ChannelHandler*  *// 使用注解@Sharable标注* @Sharable **public class** SharableHandler **extends** ChannelInboundHandlerAdapter {  @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) {  System.***out***.println(**"channel read message "** + msg);  ctx.fireChannelRead(msg);  } } |

前面的 ChannelHandler 实现符合所有的将其加入到多个ChannelPipeline 的需求，即它使用了注解@Sharable 标注，并且也不持有任何的状态。相反，代码清单6-11 中的实现将会导致问题。

|  |
| --- |
| *//代码清单 6-11 @Sharable 的错误用法*  *// 使用注解@Sharable标注* @Sharable **public class** UnsharableHandler **extends** ChannelInboundHandlerAdapter {  **private int count**;   @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) {  *// 将 count 字段的值加1* **count**++;  *// 记录方法调用，并转发给下一个ChannelHandler* System.***out***.println(**"inboundBufferUpdated(...) called the "** + **count** + **" time"**);  ctx.fireChannelRead(msg);  } } |