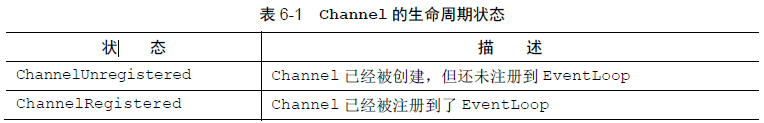
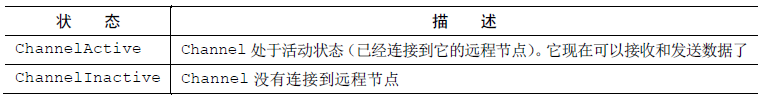
# 【第06章-ChannelHandler和ChannelPipeline】

## 6.1 ChannelHandler 家族

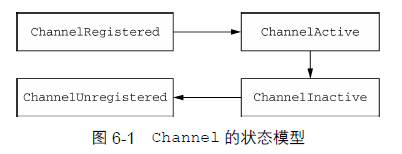
### 6.1.1 Channel 的生命周期

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。



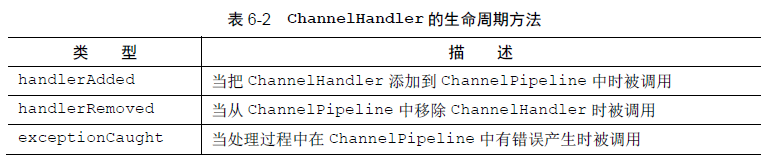


Channel 的正常生命周期如图6-1 所示。当这些状态发生改变时，将会生成对应的事件。这些事件将会被转发给ChannelPipeline 中的ChannelHandler，其可以随后对它们做出响应。



### 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* } } |

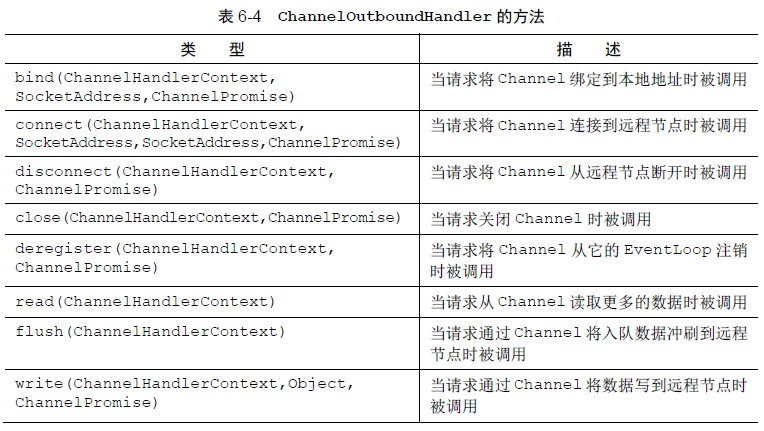
由于SimpleChannelInboundHandler 会自动释放资源，所以你不应该存储指向任何消息的引用供将来使用，因为这些引用都将会失效。

### 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。

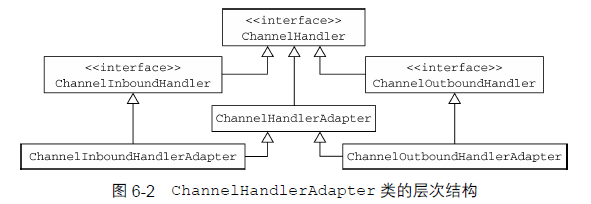
表6-4 显示了所有由ChannelOutboundHandler 本身所定义的方法（忽略了那些从ChannelHandler 继承的方法）。



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。

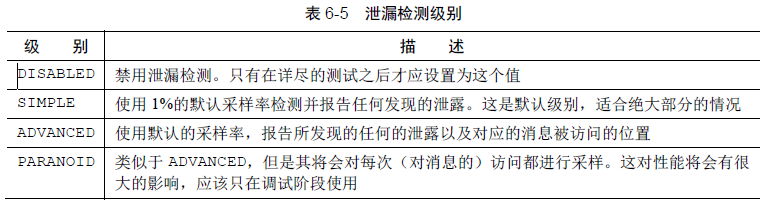
### 6.1.6 资源管理

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。

|  |
| --- |
|  |