# 【第09章-单元测试】

ChannelHandler 是Netty 应用程序的关键元素，所以彻底地测试它们应该是你的开发过程的一个标准部分。最佳实践要求你的测试不仅要能够证明你的实现是正确的，而且还要能够很容易地隔离那些因修改代码而突然出现的问题。这种类型的测试叫作单元测试。

虽然单元测试没有统一的定义，但是大多数的从业者都有基本的共识。其基本思想是，以尽可能小的区块测试你的代码，并且尽可能地和其他的代码模块以及运行时的依赖（如数据库和网络）相隔离。如果你的应用程序能通过测试验证每个单元本身都能够正常地工作，那么在出了问题时将可以更加容易地找出根本原因。

在本章中，我们将学习一种特殊的Channel 实现——EmbeddedChannel，它是Netty 专门为改进针对ChannelHandler 的单元测试而提供的。

因为正在被测试的代码模块或者单元将在它正常的运行时环境之外被执行，所以你需要一个框架或者脚手架以便在其中运行它。在我们的例子中，我们将使用JUnit 4 作为我们的测试框架，所以你需要对它的用法有一个基本的了解。如果它对你来说比较陌生，不要害怕；虽然它功能强大，但却很简单，你可以在JUnit 的官方网站（www.junit.org）上找到你所需要的所有信息。

你可能会发现回顾前面关于ChannelHandler 的章节很有用，因为这将为我们的示例提供素材。

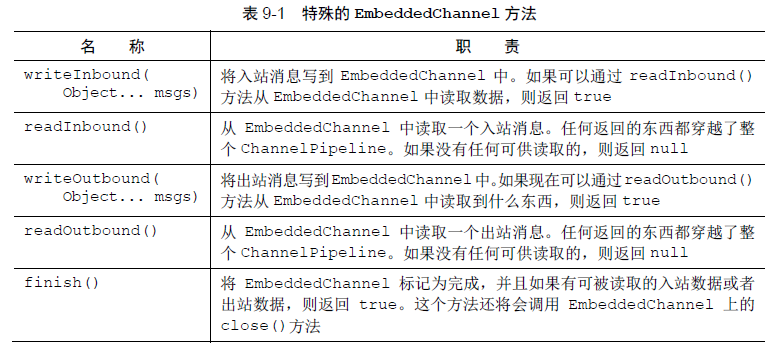
## 9.1 EmbeddedChannel 概述

你已经知道，可以将ChannelPipeline 中的ChannelHandler 实现链接在一起，以构建你的应用程序的业务逻辑。我们已经在前面解释过，这种设计支持将任何潜在的复杂处理过程分解为小的可重用的组件，每个组件都将处理一个明确定义的任务或者步骤。在本章中，我们还将展示它是如何简化测试的。

Netty 提供了它所谓的Embedded 传输，用于测试ChannelHandler。这个传输是一种特殊的Channel 实现—EmbeddedChannel—的功能，这个实现提供了通过ChannelPipeline传播事件的简便方法。

这个想法是直截了当的：将入站数据或者出站数据写入到EmbeddedChannel 中，然后检查是否有任何东西到达了ChannelPipeline 的尾端。以这种方式，你便可以确定消息是否已经被编码或者被解码过了，以及是否触发了任何的ChannelHandler 动作。

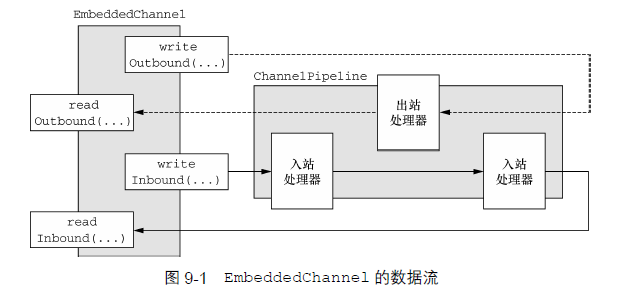
表9-1 中列出了EmbeddedChannel 的相关方法。



入站数据由ChannelInboundHandler 处理，代表从远程节点读取的数据。出站数据由ChannelOutboundHandler 处理，代表将要写到远程节点的数据。根据你要测试的ChannelHandler，你将使用\*Inbound()或者\*Outbound()方法对，或者兼而有之。

图9-1 展示了使用EmbeddedChannel 的方法，数据是如何流经ChannelPipeline 的。你可以使用writeOutbound()方法将消息写到Channel 中，并通过ChannelPipeline 沿着出站的方向传递。随后，你可以使用readOutbound()方法来读取已被处理过的消息，以确定结果是否和预期一样。 类似地，对于入站数据，你需要使用writeInbound()和readInbound()方法。

在每种情况下，消息都将会传递过ChannelPipeline，并且被相关的ChannelInboundHandler 或者ChannelOutboundHandler 处理。如果消息没有被消费，那么你可以使用readInbound()或者readOutbound()方法来在处理过了这些消息之后，酌情把它们从Channel中读出来。



## 9.2 使用EmbeddedChannel 测试ChannelHandler

在这一节中，我们将讲解如何使用EmbeddedChannel 来测试ChannelHandler。

**JUnit 断言**

org.junit.Assert 类提供了很多用于测试的静态方法。失败的断言将导致一个异常被抛出，并将终止当前正在执行中的测试。导入这些断言的最高效的方式是通过一个import static 语句来实现：

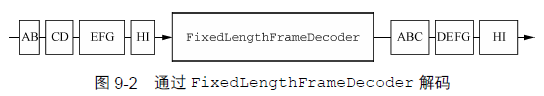
import static org.junit.Assert.\*;

一旦这样做了，就可以直接调用Assert 方法了：

assertEquals(buf.readSlice(3), read);

### 9.2.1 测试入站消息

图9-2 展示了一个简单的ByteToMessageDecoder 实现。给定足够的数据，这个实现将产生固定大小的帧。如果没有足够的数据可供读取，它将等待下一个数据块的到来，并将再次检查是否能够产生一个新的帧。



正如可以从图9-2 右侧的帧看到的那样，这个特定的解码器将产生固定为3 字节大小的帧。因此，它可能会需要多个事件来提供足够的字节数以产生一个帧。

最终，每个帧都会被传递给ChannelPipeline 中的下一个ChannelHandler。该解码器的实现，如代码清单9-1 所示。

|  |
| --- |
| *// 代码清单9-1 FixedLengthFrameDecoder*  *// 扩展ByteToMessageDecoder 以处理入站字节，并将它们解码为消息* **public class** FixedLengthFrameDecoder **extends** ByteToMessageDecoder {  *// 指定要生成的帧的长度* **private final int frameLength**;   **public** FixedLengthFrameDecoder(**int** frameLength) {  **if** (frameLength <= **0**) {  **throw new** IllegalArgumentException(**"frameLength must be a positive integer: "** + frameLength);  }  **this**.**frameLength** = frameLength;  }   @Override  **protected void** decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) **throws** Exception {  *// 检查是否有足够的字节可以被读取，以生成下一个帧* **while** (in.readableBytes() >= **frameLength**) {  *// 从ByteBuf 中读取一个新帧* ByteBuf buf = in.readBytes(**frameLength**);  *// 将该帧添加到已被解码的消息列表中* out.add(buf);  }  } } |

现在，让我们创建一个单元测试，以确保这段代码将按照预期执行。正如我们前面所指出的，即使是在简单的代码中，单元测试也能帮助我们防止在将来代码重构时可能会导致的问题，并且能在问题发生时帮助我们诊断它们。