

第 7 章 EventLoop 和线程模型

本章主要内容

- 线程模型概述
- 事件循环的概念和实现
- 任务调度
- 实现细节

简单地说，线程模型指定了操作系统、编程语言、框架或者应用程序的上下文中的线程管理的关键方面。显而易见地，如何以及何时创建线程将对应用程序代码的执行产生显著的影响，因此开发人员需要理解与不同模型相关的权衡。无论是他们自己选择模型，还是通过采用某种编程语言或者框架隐式地获得它，这都是真实的。

在本章中，我们将详细地探讨 Netty 的线程模型。它强大但又易用，并且和 Netty 的一贯宗旨一样，旨在简化你的应用程序代码，同时最大限度地提高性能和可维护性。我们还将讨论致使选择当前线程模型的经验。

如果你对 Java 的并发 API (`java.util.concurrent`) 有比较好的理解，那么你应该会发现在本章中的讨论都是直截了当的。如果这些概念对你来说还比较陌生，或者你需要更新自己的相关知识，那么由 Brian Goetz 等编写的《Java 并发编程实战》（Addison-Wesley Professional, 2006）这本书将是极好的资源。

7.1 线程模型概述

在这一节中，我们将介绍常见的线程模型，随后将继续讨论 Netty 过去以及当前的线程模型，并评审它们各自的优点以及局限性。

正如我们在本章开头所指出的，线程模型确定了代码的执行方式。由于我们总是必须规避并发执行可能会带来的副作用，所以理解所采用的并发模型（也有单线程的线程模型）的影响很重要。忽略这些问题，仅寄希望于最好的情况（不会引发并发问题）无疑是赌博——赔率必然会击败你。

代码清单 7-1 中说明了事件循环的基本思想，其中每个任务都是一个 Runnable 的实例（如图 7-1 所示）。

代码清单 7-1 在事件循环中执行任务

```
while (!terminated) {
    List<Runnable> readyEvents = blockUntilEventsReady();
    for (Runnable ev: readyEvents) {
        ev.run();
    }
}
```

← 阻塞，直到有事件已经就绪可被运行

← 循环遍历，并处理所有的事件

Netty 的 EventLoop 是协同设计的一部分，它采用了两个基本的 API：并发和网络编程。首先，io.netty.util.concurrent 包构建在 JDK 的 java.util.concurrent 包上，用来提供线程执行器。其次，io.netty.channel 包中的类，为了与 Channel 的事件进行交互，扩展了这些接口/类。图 7-2 展示了生成的类层次结构。

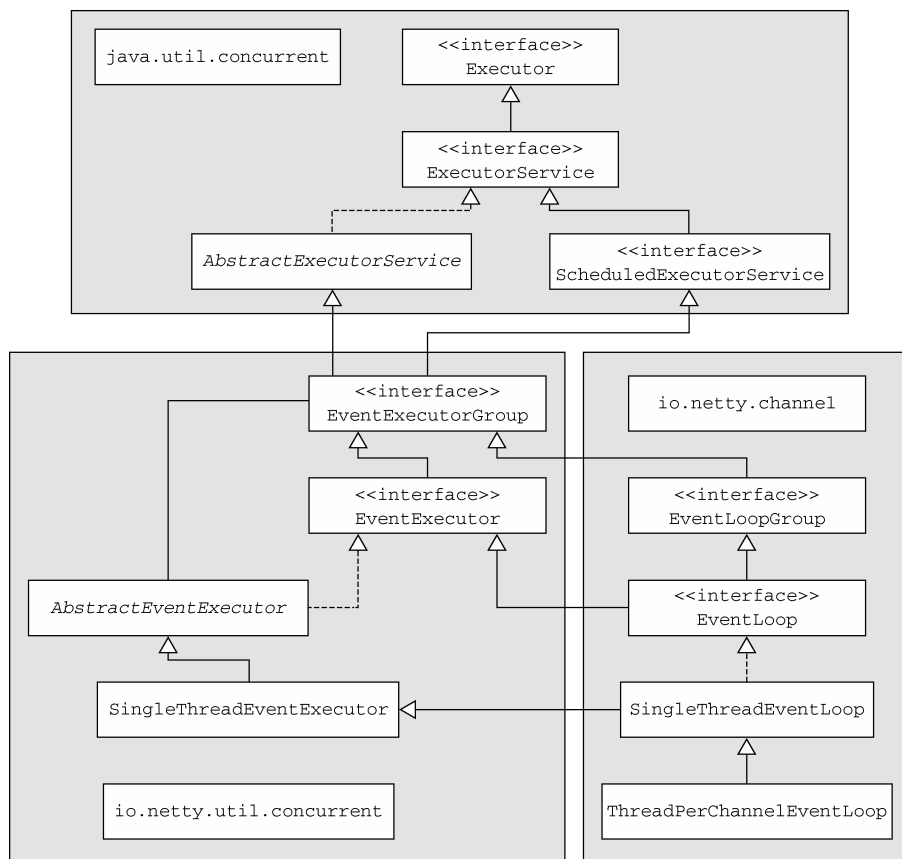


图 7-2 EventLoop 的类层次结构

在这个模型中，一个 EventLoop 将由一个永远都不会改变的 Thread 驱动，同时任务（Runnable 或者 Callable）可以直接提交给 EventLoop 实现，以立即执行或者调度执行。根据配置和可用核心的不同，可能会创建多个 EventLoop 实例用以优化资源的使用，并且单个 EventLoop 可能会被指派用于服务多个 Channel。

需要注意的是，Netty的EventLoop在继承了ScheduledExecutorService的同时，只定义了一个方法，parent()^①。这个方法，如下面的代码片断所示，用于返回到当前EventLoop实现的实例所属的EventLoopGroup的引用。

```
public interface EventLoop extends EventExecutor, EventLoopGroup {  
    @Override  
    EventLoopGroup parent();  
}
```

事件/任务的执行顺序 事件和任务是以先进先出（FIFO）的顺序执行的。这样可以通过保证字节内容总是按正确的顺序被处理，消除潜在的数据损坏的可能性。

7.2.1 Netty 4 中的 I/O 和事件处理

正如我们在第 6 章中所详细描述，由 I/O 操作触发的事件将流经安装了一个或者多个 ChannelHandler 的 ChannelPipeline。传播这些事件的方法调用可以随后被 ChannelHandler 所拦截，并且可以按需地处理事件。

事件的性质通常决定了它将被如何处理；它可能将数据从网络栈中传递到你的应用程序中，或者进行逆向操作，或者执行一些截然不同的操作。但是事件的处理逻辑必须足够的通用和灵活，以处理所有可能的用例。因此，在 Netty 4 中，所有的 I/O 操作和事件都由已经被分配给了 EventLoop 的那个 Thread 来处理^②。

这不同于 Netty 3 中所使用的模型。在下一节中，我们将讨论这个早期的模型以及它被替换的原因。

7.2.2 Netty 3 中的 I/O 操作

在以前的版本中所使用的线程模型只保证了入站（之前称为上游）事件会在所谓的 I/O 线程（对应于 Netty 4 中的 EventLoop）中执行。所有的出站（下游）事件都由调用线程处理，其可能是 I/O 线程也可能是别的线程。开始看起来这似乎是个好主意，但是已经被发现是有问题的，因为需要在 ChannelHandler 中对出站事件进行仔细的同步。简而言之，不可能保证多个线程不会在同一时刻尝试访问出站事件。例如，如果你通过在不同的线程中调用 Channel.write()

① 这个方法重写了 EventExecutor 的 EventExecutorGroup.parent() 方法。

② 这里使用的是“来处理”而不是“来触发”，其中写操作是可以从外部的任意线程触发的。——译者注

方法，针对同一个 Channel 同时触发出站的事件，就会发生这种情况。

当出站事件触发了入站事件时，将会导致另一个负面影响。当 `Channel.write()` 方法导致异常时，需要生成并触发一个 `exceptionCaught` 事件。但是在 Netty 3 的模型中，由于这是一个入站事件，需要在调用线程中执行代码，然后将事件移交给 I/O 线程去执行，然而这将带来额外的上下文切换。

Netty 4 中所采用的线程模型，通过在同一个线程中处理某个给定的 `EventLoop` 中所产生的所有事件，解决了这个问题。这提供了一个更加简单的执行体系架构，并且消除了在多个 `ChannelHandler` 中进行同步的需要（除了任何可能需要在多个 Channel 中共享的）。

现在，已经理解了 `EventLoop` 的角色，让我们来看看任务是如何被调度执行的吧。

7.3 任务调度

偶尔，你将需要调度一个任务以便稍后（延迟）执行或者周期性地执行。例如，你可能想要注册一个在客户端已经连接了 5 分钟之后触发的任务。一个常见的用例是，发送心跳消息到远程节点，以检查连接是否仍然还活着。如果没有响应，你便知道可以关闭该 Channel 了。

在接下来的几节中，我们将展示如何使用核心的 Java API 和 Netty 的 `EventLoop` 来调度任务。然后，我们将研究 Netty 的内部实现，并讨论它的优点和局限性。

7.3.1 JDK 的任务调度 API

在 Java 5 之前，任务调度是建立在 `java.util.Timer` 类之上的，其使用了一个后台 `Thread`，并且具有与标准线程相同的限制。随后，JDK 提供了 `java.util.concurrent` 包，它定义了 `interface ScheduledExecutorService`。表 7-1 展示了 `java.util.concurrent.Executors` 的相关工厂方法。

表 7-1 `java.util.concurrent.Executors` 类的工厂方法

方 法	描 述
<code>newScheduledThreadPool(int corePoolSize)</code>	创建一个 <code>ScheduledThreadPoolExecutorService</code> ，用于调度命令在指定延迟之后运行或者周期性地执行。它使用 <code>corePoolSize</code> 参数来计算线程数
<code>newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)</code>	
<code>newSingleThreadScheduledExecutor()</code>	创建一个 <code>ScheduledThreadPoolExecutorService</code> ，用于调度命令在指定延迟之后运行或者周期性地执行。它使用一个线程来执行被调度的任务
<code>newSingleThreadScheduledExecutor(ThreadFactory threadFactory)</code>	

虽然选择不是很多^①，但是这些预置的实现已经足以应对大多数的用例。代码清单 7-2 展示了如何使用 ScheduledExecutorService 来在 60 秒的延迟之后执行一个任务。

代码清单 7-2 使用 ScheduledExecutorService 调度任务

```
ScheduledExecutorService executor =
    Executors.newScheduledThreadPool(10);

ScheduledFuture<?> future = executor.schedule(
    new Runnable() {
        @Override
        public void run() {
            System.out.println("60 seconds later");
        }
    }, 60, TimeUnit.SECONDS);
...
executor.shutdown();
```

← 创建一个其线程池具有 10 个线程的 ScheduledExecutorService

← 创建一个 Runnable，以供调度稍后执行

← 该任务要打印的消息

← 调度任务在从现在开始
的 60 秒之后执行

← 一旦调度任务执行完成，就关闭 ScheduledExecutorService 以释放资源

虽然 ScheduledExecutorService API 是直截了当的，但是在高负载下它将带来性能上的负担。在下一节中，我们将看到 Netty 是如何以更高的效率提供相同的功能的。

7.3.2 使用 EventLoop 调度任务

ScheduledExecutorService 的实现具有局限性，例如，事实上作为线程池管理的一部分，将会有额外的线程创建。如果有大量任务被紧凑地调度，那么这将成为一个瓶颈。Netty 通过 Channel 的 EventLoop 实现任务调度解决了这一问题，如代码清单 7-3 所示。

代码清单 7-3 使用 EventLoop 调度任务

```
Channel ch = ...
ScheduledFuture<?> future = ch.eventLoop().schedule(
    new Runnable() {
        @Override
        public void run() {
            System.out.println("60 seconds later");
        }
    }, 60, TimeUnit.SECONDS);
```

← 创建一个 Runnable 以供调度稍后执行

← 要执行的代码

← 调度任务在从现在开始
的 60 秒之后执行

经过 60 秒之后，Runnable 实例将由分配给 Channel 的 EventLoop 执行。如果要调度任务以每隔 60 秒执行一次，请使用 scheduleAtFixedRate() 方法，如代码清单 7-4 所示。

^① 由 JDK 提供的这个接口的唯一具体实现是 java.util.concurrent.ScheduledThreadPoolExecutor。

代码清单 7-4 使用 `EventLoop` 调度周期性的任务

```

Channel ch = ...
ScheduledFuture<?> future = ch.eventLoop().scheduleAtFixedRate(
    new Runnable() {
        @Override
        public void run() {
            System.out.println("Run every 60 seconds");
        }
    }, 60, 60, TimeUnit.SECONDS);

```

创建一个 `Runnable`，以供调度稍后执行

这将一直运行，直到 `ScheduledFuture` 被取消

调度在 60 秒之后，并且以后每间隔 60 秒运行

如我们前面所提到的，Netty的`EventLoop`扩展了`ScheduledExecutorService`（见图7-2），所以它提供了使用JDK实现可用的所有方法，包括在前面的示例中使用到的`schedule()`和`scheduleAtFixedRate()`方法。所有操作的完整列表可以在`ScheduledExecutorService`的Javadoc中找到^①。

要想取消或者检查（被调度任务的）执行状态，可以使用每个异步操作所返回的 `ScheduledFuture`。代码清单 7-5 展示了一个简单的取消操作。

代码清单 7-5 使用 `ScheduledFuture` 取消任务

```

ScheduledFuture<?> future = ch.eventLoop().scheduleAtFixedRate(...);
// Some other code that runs...
boolean mayInterruptIfRunning = false;
future.cancel(mayInterruptIfRunning);

```

取消该任务，防止它再次运行

调度任务，并获得所返回的 `ScheduledFuture`

这些例子说明，可以利用 Netty 的任务调度功能来获得性能上的提升。反过来，这些也依赖于底层的线程模型，我们接下来将对其进行研究。

7.4 实现细节

这一节将更加详细地探讨 Netty 的线程模型和任务调度实现的主要内容。我们也将提到需要注意的局限性，以及正在不断发展中的领域。

7.4.1 线程管理

Netty线程模型的卓越性能取决于对于当前执行的Thread的身份的确定^②，也就是说，确定它是否是分配给当前Channel以及它的EventLoop的那一个线程。（回想一下EventLoop将负

① Java平台,标准版第8版API规范,java.util.concurrent,Interface ScheduledExecutorService:<http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html>。

② 通过调用 `EventLoop` 的 `inEventLoop(Thread)` 方法实现。——译者注

责处理一个 Channel 的整个生命周期内的所有事件。)

如果 (当前) 调用线程正是支撑 EventLoop 的线程, 那么所提交的代码块将会被 (直接) 执行。否则, EventLoop 将调度该任务以便稍后执行, 并将它放入到内部队列中。当 EventLoop 下次处理它的事件时, 它会执行队列中的那些任务/事件。这也就解释了任何的 Thread 是如何与 Channel 直接交互而无需在 ChannelHandler 中进行额外同步的。

注意, 每个 EventLoop 都有它自己的任务队列, 独立于任何其他 EventLoop。图 7-3 展示了 EventLoop 用于调度任务的执行逻辑。这是 Netty 线程模型的关键组成部分。

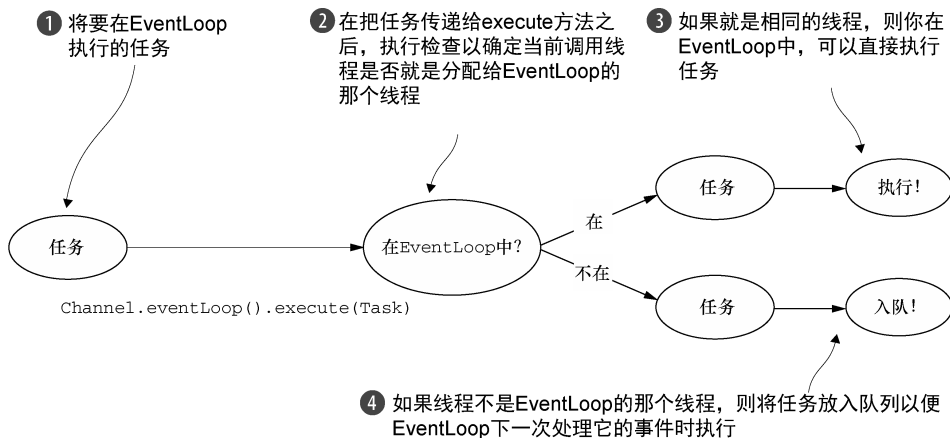


图 7-3 EventLoop 的执行逻辑

我们之前已经阐明了不要阻塞当前 I/O 线程的重要性。我们再以另一种方式重申一次: “永远不要将一个长时间运行的任务放入到执行队列中, 因为它将阻塞需要在同一线程上执行的任何其他任务。” 如果必须要进行阻塞调用或者执行长时间运行的任务, 我们建议使用一个专门的 EventExecutor。 (见 6.2.1 节的“ChannelHandler 的执行和阻塞”)。

除了这种受限的场景, 如同传输所采用的不同的事件处理实现一样, 所使用的线程模型也可以强烈地影响到排队的任务对整体系统性能的影响。(如同我们在第 4 章中所看到的, 使用 Netty 可以轻松地切换到不同的传输实现, 而不需要修改你的代码库。)

7.4.2 EventLoop/线程的分配

服务于 Channel 的 I/O 和事件的 EventLoop 包含在 EventLoopGroup 中。根据不同的传输实现, EventLoop 的创建和分配方式也不同。

1. 异步传输

异步传输实现只使用了少量的 EventLoop (以及和它们相关联的 Thread), 而且在当前的线程模型中, 它们可能会被多个 Channel 所共享。这使得可以通过尽可能少量的 Thread 来支

撑大量的 Channel，而不是每个 Channel 分配一个 Thread。

图 7-4 显示了一个 EventLoopGroup，它具有 3 个固定大小的 EventLoop（每个 EventLoop 都由一个 Thread 支撑）。在创建 EventLoopGroup 时就直接分配了 EventLoop（以及支撑它们的 Thread），以确保在需要时它们是可用的。

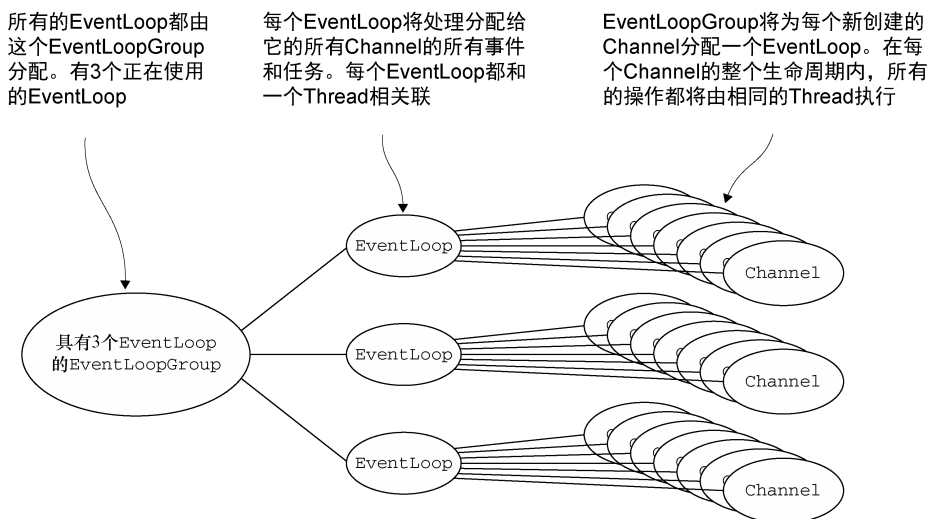


图 7-4 用于非阻塞传输（如 NIO 和 AIO）的 EventLoop 分配方式

EventLoopGroup 负责为每个新创建的 Channel 分配一个 EventLoop。在当前实现中，使用顺序循环（round-robin）的方式进行分配以获取一个均衡的分布，并且相同的 EventLoop 可能会被分配给多个 Channel。（这一点在将来的版本中可能会改变。）

一旦一个 Channel 被分配给一个 EventLoop，它将在它的整个生命周期中都使用这个 EventLoop（以及相关联的 Thread）。请牢记这一点，因为它可以使你从担忧你的 Channel-Handler 实现中的线程安全和同步问题中解脱出来。

另外，需要注意的是，EventLoop 的分配方式对 ThreadLocal 的使用的影响。因为一个 EventLoop 通常会被用于支撑多个 Channel，所以对于所有相关联的 Channel 来说，ThreadLocal 都将是一样的。这使得它对于实现状态追踪等功能来说是个糟糕的选择。然而，在一些无状态的上下文中，它仍然可以被用于在多个 Channel 之间共享一些重度的或者代价昂贵的对象，甚至是事件。

2. 阻塞传输

用于像 OIO（旧的阻塞 I/O）这样的其他传输的设计略有不同，如图 7-5 所示。

这里每一个 Channel 都将被分配给一个 EventLoop（以及它的 Thread）。如果你开发的应用程序使用过 `java.io` 包中的阻塞 I/O 实现，你可能就遇到过这种模型。

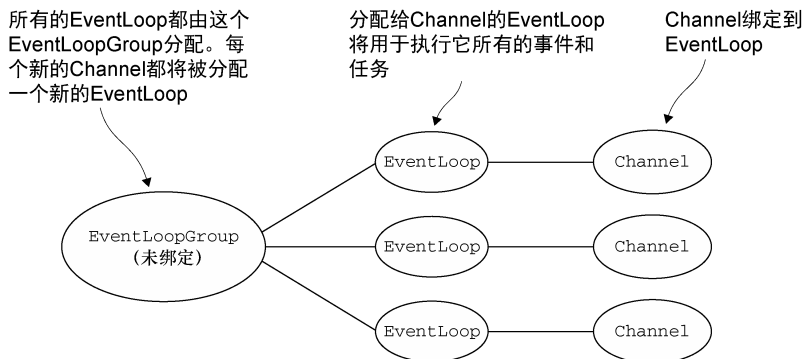


图 7-5 阻塞传输（如 OIO）的 EventLoop 分配方式

但是，正如同之前一样，得到的保证是每个 Channel 的 I/O 事件都将被只会一个 Thread（用于支撑该 Channel 的 EventLoop 的那个 Thread）处理。这也是另一个 Netty 设计一致性的例子，它（这种设计上的一致性）对 Netty 的可靠性和易用性做出了巨大贡献。

7.5 小结

在本章中，你了解了通常的线程模型，并且特别深入地学习了 Netty 所采用的线程模型，我们详细探讨了其性能以及一致性。

你看到了如何在 EventLoop（I/O Thread）中执行自己的任务，就如同 Netty 框架自身一样。你学习了如何调度任务以便推迟执行，并且我们还探讨了高负载下的伸缩性问题。你也看到了如何验证一个任务是否已被执行以及如何取消它。

通过我们对 Netty 框架的实现细节的研究所获得的这些信息，将帮助你在简化你的应用程序代码库的同时最大限度地提高它的性能。关于更多一般意义上的有关线程池和并发编程的详细信息，我们建议阅读由 Brian Goetz 编写的《Java 并发编程实战》。他的书将会带你更加深入地理解多线程处理甚至是最复杂的多线程处理用例。

我们已经到达了一个令人兴奋的时刻——在下一章中我们将讨论引导，这是一个配置以及连接所有的 Netty 组件使你的应用程序运行起来的过程。

第 8 章 引导

本章主要内容

- 引导客户端和服务端
- 从 Channel 内引导客户端
- 添加 ChannelHandler
- 使用ChannelOption和属性^①

在深入地学习了 ChannelPipeline、ChannelHandler 和 EventLoop 之后，你接下来的问题可能是：“如何将这些部分组织起来，成为一个可实际运行的应用程序呢？”

答案是？“引导”（Bootstrapping）。到目前为止，我们对这个术语的使用还比较含糊，现在已经到了精确定义它的时候了。简单来说，引导一个应用程序是指对它进行配置，并使它运行起来的过程——尽管该过程的具体细节可能并不如它的定义那样简单，尤其是对于一个网络应用程序来说。

和它对应用程序体系架构的做法^②一致，Netty处理引导的方式使你的应用程序^③和网络层相隔离，无论它是客户端还是服务器。正如同你将要看到的，所有的框架组件都将会在后台结合在一起并且启用。引导是我们一直以来都在组装的完整拼图^④中缺失的那一块。当你把它放到正确的位置上时，你的Netty应用程序就完整了。

8.1 Bootstrap 类

引导类的层次结构包括一个抽象的父类和两个具体的引导子类，如图 8-1 所示。

① Channel 继承了 AttributeMap。——译者注

② 分层抽象。——译者注

③ 应用程序的逻辑或实现。——译者注

④ “拼图”指的是 Netty 的核心概念以及组件，也包括了如何完整正确地组织并且运行一个 Netty 应用程序。——译者注

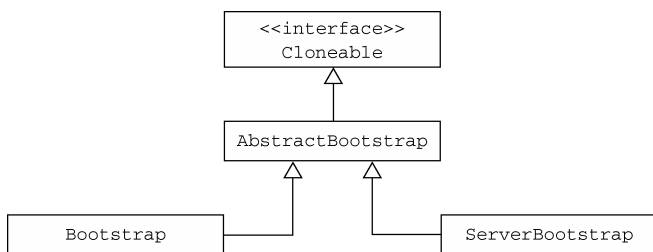


图 8-1 引导类的层次结构

相对于将具体的引导类分别看作用于服务器和客户端的引导来说，记住它们的本意是用来支撑不同的应用程序的功能的将有所裨益。也就是说，服务器致力于使用一个父 Channel 来接受来自客户端的连接，并创建子 Channel 以用于它们之间的通信；而客户端将最可能只需要一个单独的、没有父 Channel 的 Channel 来用于所有的网络交互。（正如同我们将要看到的，这也适用于无连接的传输协议，如 UDP，因为它们并不是每个连接都需要一个单独的 Channel。）

我们在前面的几章中学习的几个 Netty 组件都参与了引导的过程，而且其中一些在客户端和服务器都有用到。两种应用程序类型之间通用的引导步骤由 AbstractBootstrap 处理，而特定于客户端或者服务器的引导步骤则分别由 Bootstrap 或 ServerBootstrap 处理。

在本章中接下来的部分，我们将详细地探讨这两个类，首先从不那么复杂的 Bootstrap 类开始。

为什么引导类是 Cloneable 的

你有时可能会需要创建多个具有类似配置或者完全相同配置的 Channel。为了支持这种模式而又不需要为每个 Channel 都创建并配置一个新的引导类实例，AbstractBootstrap 被标记为了 Cloneable^①。在一个已经配置完成的引导类实例上调用 clone() 方法将返回另一个可以立即使用的引导类实例。

注意，这种方式只会创建引导类实例的 EventLoopGroup 的一个浅拷贝，所以，后者^②将在所有克隆的 Channel 实例之间共享。这是可以接受的，因为通常这些克隆的 Channel 的生命周期都很短暂，一个典型的场景是——创建一个 Channel 以进行一次 HTTP 请求。

AbstractBootstrap 类的完整声明是：

```
public abstract class AbstractBootstrap
    <B extends AbstractBootstrap<B,C>,C extends Channel>
```

在这个签名中，子类型 B 是其父类型的一个类型参数，因此可以返回到运行时实例的引用以支持方法的链式调用（也就是所谓的流式语法）。

① Java 平台，标准版第 8 版 API 规范，java.lang，Interface Cloneable：<http://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html>。

② 被浅拷贝的 EventLoopGroup。——译者注

其子类的声明如下：

```
public class Bootstrap
    extends AbstractBootstrap<Bootstrap,Channel>
```

和

```
public class ServerBootstrap
    extends AbstractBootstrap<ServerBootstrap,ServerChannel>
```

8.2 引导客户端和无连接协议

Bootstrap 类被用于客户端或者使用了无连接协议的应用程序中。表 8-1 提供了该类的一个概览，其中许多方法都继承自 AbstractBootstrap 类。

表 8-1 Bootstrap 类的 API

名 称	描 述
Bootstrap group(EventLoopGroup)	设置用于处理 Channel 所有事件的 EventLoopGroup
Bootstrap channel(Class<? extends C> Bootstrap channelFactory(ChannelFactory<? extends C>)	channel() 方法指定了 Channel 的实现类。如果该实现类没提供默认的构造函数 ^① ，可以通过调用 channelFactory() 方法来指定一个工厂类，它将会被 bind() 方法调用
Bootstrap localAddress(SocketAddress)	指定 Channel 应该绑定到的本地地址。如果没有指定，则将由操作系统创建一个随机的地址。或者，也可以通过 bind() 或者 connect() 方法指定 localAddress
<T> Bootstrap option(ChannelOption<T> option, T value)	设置 ChannelOption， <u>其将被应用到每个新创建的 Channel 的 ChannelConfig</u> 。这些选项将会通过 bind() 或者 connect() 方法设置到 Channel，不管哪个先被调用。这个方法在 Channel 已经被创建后再调用将不会有任何的效果。支持的 ChannelOption 取决于使用的 Channel 类型。 参见 8.6 节以及 ChannelConfig 的 API 文档，了解所使用的 Channel 类型
<T> Bootstrap attr(Attribute<T> key, T value)	指定新创建的 Channel 的属性值。这些属性值是通过 bind() 或者 connect() 方法设置到 Channel 的，具体取决于谁最先被调用。这个方法在 Channel 被创建后将不会有任何的效果。参见 8.6 节
Bootstrap handler(ChannelHandler)	设置将被添加到 ChannelPipeline 以接收事件通知的 ChannelHandler
Bootstrap clone()	<u>创建一个当前 Bootstrap 的克隆，其具有和原始的 Bootstrap 相同的设置信息</u>

① 这里指默认的无参构造函数，因为内部使用了反射来实现 Channel 的创建。——译者注

续表

名 称	描 述
<code>Bootstrap remoteAddress(SocketAddress)</code>	设置远程地址。或者，也可以通过 <code>connect()</code> 方法来指定它
<code>ChannelFuture connect()</code>	连接到远程节点并返回一个 <code>ChannelFuture</code> ，其将会在连接操作完成后接收到通知
<code>ChannelFuture bind()</code>	绑定 <code>Channel</code> 并返回一个 <code>ChannelFuture</code> ，其将会在绑定操作完成后接收到通知，在那之后必须调用 <code>Channel.connect()</code> 方法来建立连接

下一节将一步一步地讲解客户端的引导过程。我们也将讨论在选择可用的组件实现时保持兼容性的问题。

8.2.1 引导客户端

`Bootstrap` 类负责为客户端和使用无连接协议的应用程序创建 `Channel`，如图 8-2 所示。

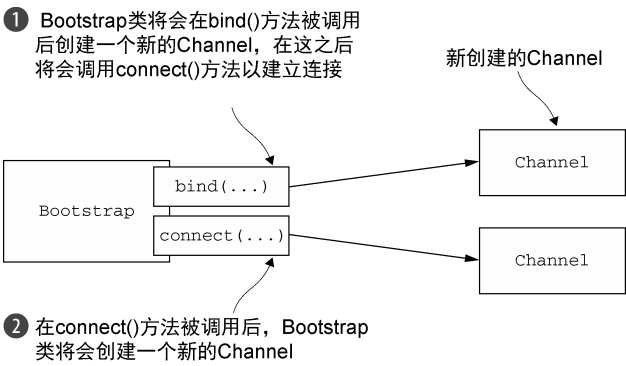


图 8-2 引导过程

代码清单 8-1 中的代码引导了一个使用 NIO TCP 传输的客户端。

代码清单 8-1 引导一个客户端

```
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap bootstrap = new Bootstrap();
bootstrap.group(group)
    .channel(NioSocketChannel.class)
    .handler(new SimpleChannelInboundHandler<ByteBuf>() {
        @Override
        protected void channelRead0(
            ChannelHandlerContext channelHandlerContext,
            ByteBuf byteBuf) throws Exception {
            System.out.println("Received data");
        }
    });
```

设置 `EventLoopGroup`，提供用于处理 `Channel` 事件的 `EventLoop`

指定要使用的 `Channel` 实现

设置用于 `Channel` 事件和数据的 `ChannelInboundHandler`

创建一个 `Bootstrap` 类的实例以创建和连接新的客户端 `Channel`

```

    }
  });
ChannelFuture future = bootstrap.connect(
    new InetSocketAddress("www.manning.com", 80));
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture)
        throws Exception {
        if (channelFuture.isSuccess()) {
            System.out.println("Connection established");
        } else {
            System.err.println("Connection attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
});
}
});

```

← 连接到远程主机

这个示例使用了前面提到的流式语法；这些方法（除了 `connect()` 方法以外）将通过每次方法调用所返回的对 Bootstrap 实例的引用链接在一起。

8.2.2 Channel 和 EventLoopGroup 的兼容性

代码清单 8-2 所示的目录清单来自 `io.netty.channel` 包。你可以从包名以及与其相对应的类名的前缀看到，对于 NIO 以及 OIO 传输两者来说，都有相关的 `EventLoopGroup` 和 `Channel` 实现。

代码清单 8-2 相互兼容的 `EventLoopGroup` 和 `Channel`

```

channel
├── nio
│   └── NioEventLoopGroup
├── oio
│   └── OioEventLoopGroup
└── socket
    ├── nio
    │   ├── NioDatagramChannel
    │   ├── NioServerSocketChannel
    │   └── NioSocketChannel
    └── oio
        ├── OioDatagramChannel
        ├── OioServerSocketChannel
        └── OioSocketChannel

```

必须保持这种兼容性，不能混用具有不同前缀的组件，如 `NioEventLoopGroup` 和 `OioSocketChannel`。代码清单 8-3 展示了试图这样做的一个例子。

代码清单 8-3 不兼容的 Channel 和 EventLoopGroup

```

EventLoopGroup group = new NioEventLoopGroup();
Bootstrap bootstrap = new Bootstrap();
bootstrap.group(group)
    .channel(OioSocketChannel.class)
    .handler(new SimpleChannelInboundHandler<ByteBuf>() {
        @Override
        protected void channelRead0(
            ChannelHandlerContext channelHandlerContext,
            ByteBuf byteBuf) throws Exception {
            System.out.println("Received data");
        }
    });
ChannelFuture future = bootstrap.connect(
    new InetSocketAddress("www.manning.com", 80));
future.syncUninterruptibly();

```

指定一个适用于 NIO 的 EventLoopGroup 实现

指定一个适用于 OIO 的 Channel 实现类

设置一个用于处理 Channel 的 I/O 事件和数据的 ChannelInboundHandler

尝试连接到远程节点

创建一个新的 Bootstrap 类的实例，以创建新的客户端 Channel

这段代码将会导致 `IllegalStateException`，因为它混用了不兼容的传输。

```

Exception in thread "main" java.lang.IllegalStateException:
incompatible event loop type: io.netty.channel.nio.NioEventLoop at
io.netty.channel.AbstractChannel$AbstractUnsafe.register(
AbstractChannel.java:571)

```

关于 `IllegalStateException` 的更多讨论

在引导的过程中，在调用 `bind()` 或者 `connect()` 方法之前，必须调用以下方法来设置所需的组件：

- `group()`;
- `channel()` 或者 `channelFactory()`;
- `handler()`。

如果不这样做，则将会导致 `IllegalStateException`。对 `handler()` 方法的调用尤其重要，因为它需要配置好 `ChannelPipeline`。

8.3 引导服务器

我们将从 `ServerBootstrap` API 的概要视图开始我们对服务器引导过程的概述。然后，我们将会探讨引导服务器过程中所涉及的几个步骤，以及几个相关的主题，包含从一个 `ServerChannel` 的子 Channel 中引导一个客户端这样的特殊情况。

8.3.1 `ServerBootstrap` 类

表 8-2 列出了 `ServerBootstrap` 类的方法。

表 8-2 ServerBootstrap 类的方法

名 称	描 述
group	设置 ServerBootstrap 要用的 EventLoopGroup。这个 EventLoopGroup 将用于 ServerChannel 和被接受的子 Channel 的 I/O 处理
channel	设置将要被实例化的 ServerChannel 类
channelFactory	如果不能通过默认的构造函数 ^① 创建 Channel，那么可以提供 ChannelFactory
localAddress	指定 ServerChannel 应该绑定到的本地地址。如果没有指定，则将由操作系统使用一个随机地址。或者，可以通过 bind() 方法来指定该 localAddress
option	指定要应用到新创建的 ServerChannel 的 ChannelConfig 的 ChannelOption。这些选项将会通过 bind() 方法设置到 Channel。在 bind() 方法被调用之后，设置或者改变 ChannelOption 都不会有任何的效果。所支持的 ChannelOption 取决于所使用的 Channel 类型。参见正在使用的 ChannelConfig 的 API 文档
childOption	指定当子 Channel 被接受时，应用到子 Channel 的 ChannelConfig 的 ChannelOption。所支持的 ChannelOption 取决于所使用的 Channel 的类型。参见正在使用的 ChannelConfig 的 API 文档
attr	指定 ServerChannel 上的属性，属性将会通过 bind() 方法设置给 Channel。在调用 bind() 方法之后改变它们将不会有任何的效果
childAttr	将属性设置给已经被接受的子 Channel。接下来的调用将不会有任何的效果
handler	设置被添加到 ServerChannel 的 ChannelPipeline 中的 ChannelHandler。更加常用的方法参见 childHandler()
childHandler	设置将被添加到已被接受的子 Channel 的 ChannelPipeline 中的 ChannelHandler。handler() 方法和 childHandler() 方法之间的区别是：前者所添加的 ChannelHandler 由接受子 Channel 的 ServerChannel 处理，而 childHandler() 方法所添加的 ChannelHandler 将由已被接受的子 Channel 处理，其代表一个绑定到远程节点的套接字
clone	克隆一个设置和原始的 ServerBootstrap 相同的 ServerBootstrap
bind	绑定 ServerChannel 并且返回一个 ChannelFuture，其将会在绑定操作完成后收到通知（带着成功或者失败的结果）

下一节将介绍服务器引导的详细过程。

8.3.2 引导服务器

你可能已经注意到了，表 8-2 中列出了一些在表 8-1 中不存在的方法：childHandler()、childAttr() 和 childOption()。这些调用支持特别用于服务器应用程序的操作。具体来说，ServerChannel 的实现负责创建子 Channel，这些子 Channel 代表了已被接受的连接。因

^① 这里指无参数的构造函数。——译者注

此,负责引导 `ServerChannel` 的 `ServerBootstrap` 提供了这些方法,以简化将设置应用到已被接受的子 `Channel` 的 `ChannelConfig` 的任务。

图 8-3 展示了 `ServerBootstrap` 在 `bind()` 方法被调用时创建了一个 `ServerChannel`,并且该 `ServerChannel` 管理了多个子 `Channel`。

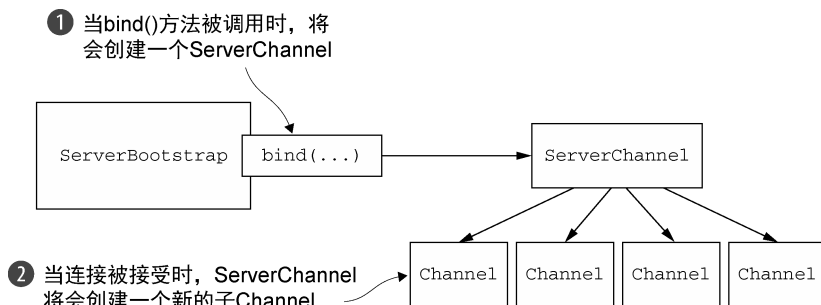


图 8-3 `ServerBootstrap` 和 `ServerChannel`

代码清单 8-4 中的代码实现了图 8-3 中所展示的服务器的引导过程。

代码清单 8-4 引导服务器

```

创建 ServerBootstrap
NioEventLoopGroup group = new NioEventLoopGroup();
ServerBootstrap bootstrap = new ServerBootstrap();
bootstrap.group(group);

bootstrap.channel(NioServerSocketChannel.class)
        .childHandler(new SimpleChannelInboundHandler<ByteBuf>() {
            @Override
            protected void channelRead0(ChannelHandlerContext ctx,
            ByteBuf byteBuf) throws Exception {
                System.out.println("Received data");
            }
        });

ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture)
        throws Exception {
        if (channelFuture.isSuccess()) {
            System.out.println("Server bound");
        } else {
            System.err.println("Bound attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
});
  
```

指定要使用的 Channel 实现

设置 `EventLoopGroup`, 其提供了用于处理 Channel 事件的 `EventLoop`

设置用于处理已被接受的子 Channel 的 I/O 及数据的 `ChannelInboundHandler`

通过配置好的 `ServerBootstrap` 的实例绑定该 Channel

8.4 从 Channel 引导客户端

假设你的服务器正在处理一个客户端的请求，这个请求需要它充当第三方系统的客户端。当一个应用程序（如一个代理服务器）必须要和组织现有的系统（如 Web 服务或者数据库）集成时，就可能发生这种情况。在这种情况下，**将需要从已经被接受的子 Channel 中引导一个客户端 Channel。**

你可以按照 8.2.1 节中所描述的方式创建新的 Bootstrap 实例，但是这并不是最高效的解决方案，因为它将要求你为每个新创建的客户端 Channel 定义另一个 EventLoop。这会产生额外的线程，以及在**已被接受的子 Channel 和客户端 Channel 之间**交换数据时不可避免的上下文切换。

一个更好的解决方案是：**通过将被接受的子 Channel 的 EventLoop 传递给 Bootstrap 的 group() 方法来共享该 EventLoop。**因为分配给 EventLoop 的所有 Channel 都使用同一个线程，所以这避免了额外的线程创建，**以及前面所提到的相关的上下文切换。**这个共享的解决方案如图 8-4 所示。

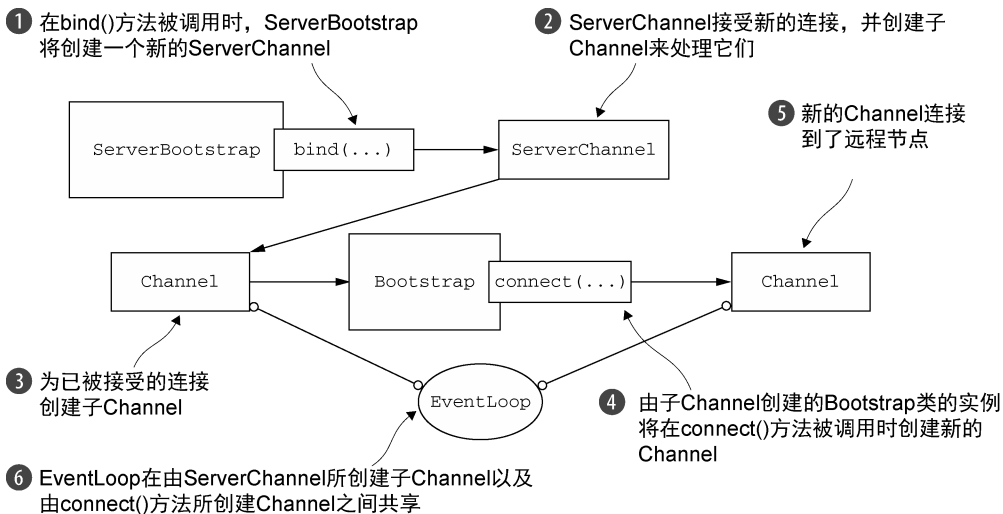


图 8-4 在两个 Channel 之间共享 EventLoop

实现 EventLoop 共享涉及通过调用 group() 方法来设置 EventLoop，如代码清单 8-5 所示。

代码清单 8-5 引导服务器

```

设置 EventLoopGroup，其将提供用
以处理 Channel 事件的 EventLoop
ServerBootstrap bootstrap = new ServerBootstrap();
bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup())

```

创建 ServerBootstrap 以创建
ServerSocketChannel，并绑定它

```

        .channel(NioServerSocketChannel.class)
        .childHandler(
            new SimpleChannelInboundHandler<ByteBuf>() {
                ChannelFuture connectFuture;
                @Override
                public void channelActive(ChannelHandlerContext ctx)
                    throws Exception {
                    Bootstrap bootstrap = new Bootstrap();
                    bootstrap.channel(NioSocketChannel.class).handler(
                        new SimpleChannelInboundHandler<ByteBuf>() {
                            @Override
                            protected void channelRead0(
                                ChannelHandlerContext ctx, ByteBuf in)
                                throws Exception {
                                System.out.println("Received data");
                            }
                        }
                    );
                    bootstrap.group(ctx.channel().eventLoop());
                    connectFuture = bootstrap.connect(
                        new InetSocketAddress("www.manning.com", 80));
                }
                @Override
                protected void channelRead0(
                    ChannelHandlerContext channelHandlerContext,
                    ByteBuf byteBuf) throws Exception {
                    if (connectFuture.isDone()) {
                        // do something with the data
                    }
                }
            }
        );
        ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));
        future.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture channelFuture)
                throws Exception {
                if (channelFuture.isSuccess()) {
                    System.out.println("Server bound");
                } else {
                    System.err.println("Bind attempt failed");
                    channelFuture.cause().printStackTrace();
                }
            }
        });
    }
}

```

指定要使用的 Channel 实现

创建一个 Bootstrap 类的实例以连接到远程主机

指定 Channel 的实现

为入站 I/O 设置 ChannelInboundHandler

使用与分配给已被接受的子 Channel 相同的 EventLoop

连接到远程节点

当连接完成时, 执行一些数据操作 (如代理)

通过配置好的 ServerBootstrap 绑定该 ServerSocketChannel

我们在这一节中所讨论的主题以及所提出的解决方案都反映了编写 Netty 应用程序的一个一般准则: **尽可能地重用 EventLoop, 以减少线程创建所带来的开销。**

8.5 在引导过程中添加多个 ChannelHandler

在所有我们展示过的代码示例中, 我们都在引导的过程中调用了 handler() 或者 childHandler() 方法来添加单个的 ChannelHandler。这对于简单的应用程序来说可能已经足够

了,但是它不能满足更加复杂的需求。例如,一个必须要支持多种协议的应用程序将会有很多的 ChannelHandler,而不会是一个庞大而又笨重的类。

正如你经常所看到的一样,你可以根据需要,通过在 ChannelPipeline 中将它们链接在一起来部署尽可能多的 ChannelHandler。但是,如果在引导的过程中你只能设置一个 ChannelHandler,那么你应该怎么做到这一点呢?

正是针对于这个用例,Netty 提供了一个特殊的 ChannelInboundHandlerAdapter 子类:

```
public abstract class ChannelInitializer<C extends Channel>
    extends ChannelInboundHandlerAdapter
```

它定义了下面的方法:

```
protected abstract void initChannel(C ch) throws Exception;
```

这个方法提供了一种将多个 ChannelHandler 添加到一个 ChannelPipeline 中的简便方法。你只需要简单地向 Bootstrap 或 ServerBootstrap 的实例提供你的 ChannelInitializer 实现即可,并且一旦 Channel 被注册到了它的 EventLoop 之后,就会调用你的 initChannel() 版本。^①在该方法返回之后,ChannelInitializer 的实例将会从 ChannelPipeline 中移除它自己。

代码清单 8-6 定义了 ChannelInitializerImpl 类,并通过 ServerBootstrap 的 childHandler() 方法注册它^②。你可以看到,这个看似复杂的操作实际上是相当简单直接的。

代码清单 8-6 引导和使用 ChannelInitializer

```

    设置 EventLoopGroup, 其将提供用
    以处理 Channel 事件的 EventLoop
    ServerBootstrap bootstrap = new ServerBootstrap();
    > bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup()) <— 创建 ServerBootstrap 以创
    建和绑定新的 Channel
    绑定 .channel(NioServerSocketChannel.class) <— 指定 Channel 的
    到地址 .childHandler(new ChannelInitializerImpl()); <— 实现
    > ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));
    future.sync();
    将所需的 ChannelHandler
    添加到 ChannelPipeline
    final class ChannelInitializerImpl extends ChannelInitializer<Channel> {② <—
        @Override
        > protected void initChannel(Channel ch) throws Exception {
            ChannelPipeline pipeline = ch.pipeline();
            pipeline.addLast(new HttpClientCodec());
            }
        }
    }
    注册一个 ChannelInitializerImpl 的
    实例来设置 ChannelPipeline
    用以设置 ChannelPipeline 的自
    定义 ChannelInitializerImpl 实现

```

① 注册到 ServerChannel 的子 Channel 的 ChannelPipeline。——译者注

② 在大部分的场景下,如果你不需要使用只存在于 SocketChannel 上的方法,使用 ChannelInitializer<Channel>就可以了,否则你可以使用 ChannelInitializer<SocketChannel>,其中 SocketChannel 扩展了 Channel。——译者注

```

        pipeline.addLast(new HttpObjectAggregator(Integer.MAX_VALUE));
    }
}

```

如果你的应用程序使用了多个 `ChannelHandler`，请定义你自己的 `ChannelInitializer` 实现来将它们安装到 `ChannelPipeline` 中。

8.6 使用 Netty 的 `ChannelOption` 和属性

在每个 `Channel` 创建时都手动配置它可能会变得相当乏味。幸运的是，你不必这样做。相反，你可以使用 `option()` 方法来将 `ChannelOption` 应用到引导。你所提供的值将会被自动应用到引导所创建的所有 `Channel`。可用的 `ChannelOption` 包括了底层连接的详细信息，如 `keep-alive` 或者超时属性以及缓冲区设置。

Netty 应用程序通常与组织的专有软件集成在一起，而像 `Channel` 这样的组件可能甚至会在正常的 Netty 生命周期之外被使用。在某些常用的属性和数据不可用时，Netty 提供了 `AttributeMap` 抽象（一个由 `Channel` 和引导类提供的集合）以及 `AttributeKey<T>`（一个用于插入和获取属性值的泛型类）。使用这些工具，便可以安全地将任何类型的数据项与客户端和服务器 `Channel`（包含 `ServerChannel` 的子 `Channel`）相关联了。

例如，考虑一个用于跟踪用户和 `Channel` 之间的关系的服务端应用程序。这可以通过将用户的 ID 存储为 `Channel` 的一个属性来完成。类似的技术可以被用来基于用户的 ID 将消息路由给用户，或者关闭活动较少的 `Channel`。

代码清单 8-7 展示了可以如何使用 `ChannelOption` 来配置 `Channel`，以及如果使用属性来存储整型值。

代码清单 8-7 使用属性值

创建一个 `Bootstrap` 类的实例以
创建客户端 `Channel` 并连接它们

```

final AttributeKey<Integer> id = new AttributeKey<Integer>("ID");
Bootstrap bootstrap = new Bootstrap();
bootstrap.group(new NioEventLoopGroup())
    .channel(NioSocketChannel.class)
    .handler(
        new SimpleChannelInboundHandler<ByteBuf>() {
            @Override
            public void channelRegistered(ChannelHandlerContext ctx)
                throws Exception {
                Integer idValue = ctx.channel().attr(id).get();
                // do something with the idValue
            }

            @Override
            protected void channelRead0(

```

创建一个 `AttributeKey`
以标识该属性

设置 `EventLoopGroup`，其
提供了用以处理 `Channel`
事件的 `EventLoop`

设置用以处理 `Channel` 的
I/O 以及数据的 `Channel-
InboundHandler`

使用 `AttributeKey` 检索
属性以及它的值

指定
`Channel`
的实现

```

        ChannelHandlerContext channelHandlerContext,
        ByteBuf byteBuf) throws Exception {
            System.out.println("Received data");
        }
    }
    // 存储该 id 属性
    bootstrap.option(ChannelOption.SO_KEEPALIVE, true)
        .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 5000);
    bootstrap.attr(id, 123456);
    ChannelFuture future = bootstrap.connect(
        new InetSocketAddress("www.manning.com", 80));
    future.syncUninterruptibly();

```

设置 ChannelOption, 其将在 connect() 或者 bind() 方法被调用时被设置到已经创建的 Channel 上

使用配置好的 Bootstrap 实例连接到远程主机

8.7 引导 DatagramChannel

前面的引导代码示例使用的都是基于 TCP 协议的 SocketChannel, 但是 Bootstrap 类也可以被用于无连接的协议。为此, Netty 提供了各种 DatagramChannel 的实现。唯一区别就是, 不再调用 connect() 方法, 而是只调用 bind() 方法, 如代码清单 8-8 所示。

代码清单 8-8 使用 Bootstrap 和 DatagramChannel

```

// 设置 EventLoopGroup, 其提供了用以处理 Channel 事件的 EventLoop
Bootstrap bootstrap = new Bootstrap();
bootstrap.group(new OioEventLoopGroup()).channel(
    // 指定 Channel 的实现
    OioDatagramChannel.class).handler(
        new SimpleChannelInboundHandler<DatagramPacket>() {
            @Override
            public void channelRead0(ChannelHandlerContext ctx,
                DatagramPacket msg) throws Exception {
                // Do something with the packet
            }
        });
ChannelFuture future = bootstrap.bind(new InetSocketAddress(0));
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture)
        throws Exception {
        if (channelFuture.isSuccess()) {
            System.out.println("Channel bound");
        } else {
            System.err.println("Bind attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
});

```

创建一个 Bootstrap 的实例以创建和绑定新的数据报 Channel

设置用以处理 Channel 的 I/O 以及数据的 Channel-InboundHandler

调用 bind() 方法, 因为该协议是无连接的

8.8 关闭

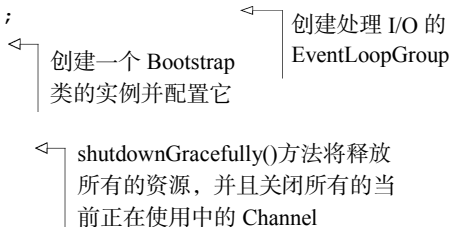
引导使你的应用程序启动并且运行起来,但是迟早你都需要优雅地将它关闭。当然,你也可以让 JVM 在退出时处理好一切,但是这不符合优雅的定义,优雅是指干净地释放资源。关闭 Netty 应用程序并没有太多的魔法,但是还是有些事情需要记在心上。

最重要的是,你需要关闭 `EventLoopGroup`,它将处理任何挂起的事件和任务,并且随后释放所有活动的线程。这就是调用 `EventLoopGroup.shutdownGracefully()` 方法的作用。这个方法调用将会返回一个 `Future`,这个 `Future` 将在关闭完成时接收到通知。需要注意的是,`shutdownGracefully()` 方法也是一个异步的操作,所以你需要阻塞等待直到它完成,或者向所返回的 `Future` 注册一个监听器以在关闭完成时获得通知。

代码清单 8-9 符合优雅关闭的定义。

代码清单 8-9 优雅关闭

```
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap bootstrap = new Bootstrap();
bootstrap.group(group)
    .channel(NioSocketChannel.class);
...
Future<?> future = group.shutdownGracefully();
// block until the group has shutdown
future.syncUninterruptibly();
```



← 创建一个 Bootstrap 类的实例并配置它

← 创建处理 I/O 的 EventLoopGroup

← shutdownGracefully()方法将释放所有的资源,并且关闭所有的当前正在使用中的 Channel

或者,你也可以在调用 `EventLoopGroup.shutdownGracefully()` 方法之前,显式地在所有活动的 `Channel` 上调用 `Channel.close()` 方法。但是在任何情况下,都请记住关闭 `EventLoopGroup` 本身。

8.9 小结

在本章中,你学习了如何引导 Netty 服务器和客户端应用程序,包括那些使用无连接协议的应用程序。我们也涵盖了一些特殊情况,包括在服务器应用程序中引导客户端 `Channel`,以及使用 `ChannelInitializer` 来处理引导过程中的多个 `ChannelHandler` 的安装。你看到了如何设置 `Channel` 的配置选项,以及如何使用属性来将信息附加到 `Channel`。最后,你学习了如何优雅地关闭应用程序,以有序地释放所有的资源。

在下一章中,我们将研究 Netty 提供的帮助你测试你的 `ChannelHandler` 实现的工具。

第 9 章 单元测试

本章主要内容

- 单元测试
- EmbeddedChannel 概述
- 使用 EmbeddedChannel 测试 ChannelHandler

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。

表 9-1 特殊的 EmbeddedChannel 方法

名 称	职 责
writeInbound(Object... msgs)	将入站消息写到 EmbeddedChannel 中。如果可以通过 readInbound() 方法从 EmbeddedChannel 中读取数据，则返回 true
readInbound()	从 EmbeddedChannel 中读取一个入站消息。任何返回的东西都穿越了整个 ChannelPipeline。如果没有任何可供读取的，则返回 null
writeOutbound(Object... msgs)	将出站消息写到 EmbeddedChannel 中。如果现在可以通过 readOutbound() 方法从 EmbeddedChannel 中读取到什么东西，则返回 true
readOutbound()	从 EmbeddedChannel 中读取一个出站消息。任何返回的东西都穿越了整个 ChannelPipeline。如果没有任何可供读取的，则返回 null
finish()	将 EmbeddedChannel 标记为完成，并且如果有可被读取的入站数据或者出站数据，则返回 true。这个方法还将会调用 EmbeddedChannel 上的 close() 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。

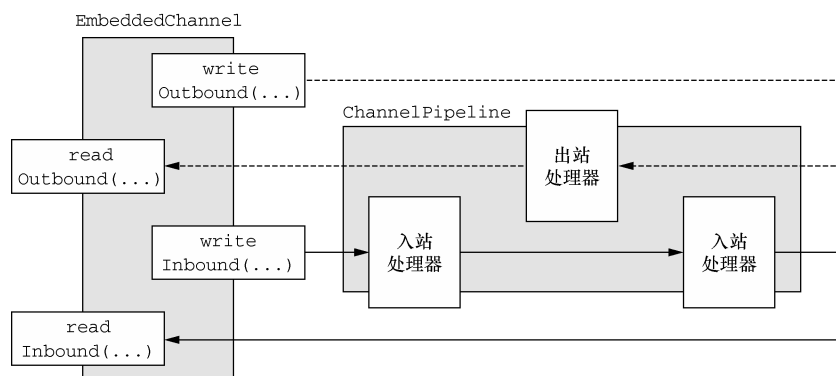


图 9-1 EmbeddedChannel 的数据流

接下来让我们仔细看看这两种场景，以及它们是如何应用于测试你的应用程序逻辑的吧。

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。

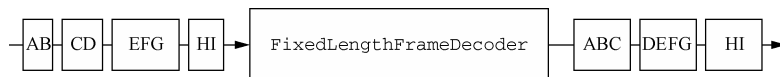


图 9-2 通过 FixedLengthFrameDecoder 解码

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

```

指定要生成                                扩展 ByteToMessageDecoder 以处理入
的帧的长度                                站字节，并将它们解码为消息
public class FixedLengthFrameDecoder extends ByteToMessageDecoder {
    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 {
        while (in.readableBytes() >= frameLength) {
            ByteBuf buf = in.readBytes(frameLength);
            out.add(buf);
        }
    }
}

```

检查是否有足够的字节可以被读取，以生成下一个帧

从 ByteBuf 中读取一个新帧

将该帧添加到已被解码的消息列表中

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。

代码清单 9-2 展示了一个使用 EmbeddedChannel 的对于前面代码的测试。

代码清单 9-2 测试 FixedLengthFrameDecoder

```

public class FixedLengthFrameDecoderTest {
    @Test
    public void testFramesDecoded() {
        ByteBuf buf = Unpooled.buffer();
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();
        EmbeddedChannel channel = new EmbeddedChannel(
            new FixedLengthFrameDecoder(3));
        // write bytes
        assertTrue(channel.writeInbound(input.retain()));
    }
}

```

使用了注解@Test 标注，因此 JUnit 将会执行该方法

第一个测试方法：testFramesDecoded()

创建一个 ByteBuf，并存储 9 字节

创建一个 EmbeddedChannel，并添加一个 FixedLengthFrameDecoder，其将以 3 字节的帧长度被测试

将数据写入 EmbeddedChannel

```

assertTrue(channel.finish());

// read messages
ByteBuf read = (ByteBuf) channel.readInbound();
assertEquals(buf.readSlice(3), read);
read.release();

read = (ByteBuf) channel.readInbound();
assertEquals(buf.readSlice(3), read);
read.release();

read = (ByteBuf) channel.readInbound();
assertEquals(buf.readSlice(3), read);
read.release();

assertNull(channel.readInbound());
buf.release();
}

@Test
public void testFramesDecoded2() {
    ByteBuf buf = Unpooled.buffer();
    for (int i = 0; i < 9; i++) {
        buf.writeByte(i);
    }
    ByteBuf input = buf.duplicate();

    EmbeddedChannel channel = new EmbeddedChannel(
        new FixedLengthFrameDecoder(3));
    assertFalse(channel.writeInbound(input.readBytes(2)));
    assertTrue(channel.writeInbound(input.readBytes(7)));

    assertTrue(channel.finish());
    ByteBuf read = (ByteBuf) channel.readInbound();
    assertEquals(buf.readSlice(3), read);
    read.release();

    read = (ByteBuf) channel.readInbound();
    assertEquals(buf.readSlice(3), read);
    read.release();

    read = (ByteBuf) channel.readInbound();
    assertEquals(buf.readSlice(3), read);
    read.release();

    assertNull(channel.readInbound());
    buf.release();
}

```

标记 Channel 为已完成状态

读取所生成的消息，并且验证是否有 3 帧（切片），其中每帧（切片）都为 3 字节

第二个测试方法：
testFramesDecoded2()

返回 false，因为没有一个完整的可供读取的帧

该 `testFramesDecoded()` 方法验证了：一个包含 9 个可读字节的 `ByteBuf` 被解码为 3 个 `ByteBuf`，每个都包含了 3 字节。需要注意的是，仅通过一次对 `writeInbound()` 方法的调用，`ByteBuf` 是如何被填充了 9 个可读字节的。在此之后，通过执行 `finish()` 方法，将 `EmbeddedChannel` 标记为了已完成状态。最后，通过调用 `readInbound()` 方法，从 `EmbeddedChannel` 中正好读取了 3 个帧和一个 `null`。

`testFramesDecoded2()` 方法也是类似的，只有一处不同：入站 `ByteBuf` 是通过两个步骤写入的。当 `writeInbound(input.readBytes(2))` 被调用时，返回了 `false`。为什么呢？正如同表 9-1 中所描述的，如果对 `readInbound()` 的后续调用将会返回数据，那么 `writeInbound()` 方法将会返回 `true`。但是只有当有 3 个或者更多的字节可供读取时，`FixedLengthFrameDecoder` 才会产生输出。该测试剩下的部分和 `testFramesDecoded()` 是相同的。

9.2.2 测试出站消息

测试出站消息的处理过程和刚才所看到的类似。在下面的例子中，我们将会展示如何使用 `EmbeddedChannel` 来测试一个编码器形式的 `ChannelOutboundHandler`，编码器是一种将一种消息格式转换为另一种的组件。你将在下一章中非常详细地学习编码器和解码器，所以现在只需要简单地提及我们正在测试的处理器——`AbsIntegerEncoder`，它是 `Netty` 的 `MessageToMessageEncoder` 的一个特殊化的实现，用于将负值整数转换为绝对值。

该示例将会按照下列方式工作：

- 持有 `AbsIntegerEncoder` 的 `EmbeddedChannel` 将会以 4 字节的负整数的形式写出站数据；
- 编码器将从传入的 `ByteBuf` 中读取每个负整数，并将会调用 `Math.abs()` 方法来获取其绝对值；
- 编码器将会把每个负整数的绝对值写到 `ChannelPipeline` 中。

图 9-3 展示了该逻辑。

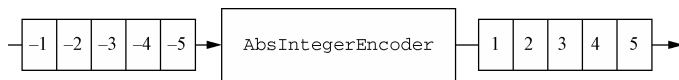


图 9-3 通过 `AbsIntegerEncoder` 编码

代码清单 9-3 实现了这个逻辑，如图 9-3 所示。`encode()` 方法将把产生的值写到一个 `List` 中。

代码清单 9-3 `AbsIntegerEncoder`

```
public class AbsIntegerEncoder extends
    MessageToMessageEncoder<ByteBuf> {
    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext,
```



扩展 `MessageToMessageEncoder` 以
将一个消息编码为另外一种格式

```

    ByteBuf in, List<Object> out) throws Exception {
        while (in.readableBytes() >= 4) {
            int value = Math.abs(in.readInt());
            out.add(value);
        }
    }
}

```

检查是否有足够的字节用来编码

将该整数写入到编码消息的 List 中

从输入的 ByteBuf 中读取下一个整数，并且计算其绝对值

代码清单 9-4 使用了 EmbeddedChannel 来测试代码。

代码清单 9-4 测试 AbsIntegerEncoder

```

public class AbsIntegerEncoderTest {
    @Test
    public void testEncoded() {
        ByteBuf buf = Unpooled.buffer();
        for (int i = 1; i < 10; i++) {
            buf.writeInt(i * -1);
        }

        EmbeddedChannel channel = new EmbeddedChannel(
            new AbsIntegerEncoder());
        assertTrue(channel.writeOutbound(buf));
        assertTrue(channel.finish());

        // read bytes
        for (int i = 1; i < 10; i++) {
            assertEquals(i, channel.readOutbound());
        }
        assertNull(channel.readOutbound());
    }
}

```

1 创建一个 ByteBuf，并且写入 9 个负整数

2 创建一个 EmbeddedChannel，并安装一个要测试的 AbsIntegerEncoder

3 写入 ByteBuf，并断言调用 readOutbound() 方法将会产生数据

4 将该 Channel 标记为已完成状态

5 读取所产生的消息，并断言它们包含了对应的绝对值

下面是代码中执行的步骤。

- ❶ 将 4 字节的负整数写到一个新的 ByteBuf 中。
- ❷ 创建一个 EmbeddedChannel，并为它分配一个 AbsIntegerEncoder。
- ❸ 调用 EmbeddedChannel 上的 writeOutbound() 方法来写入该 ByteBuf。
- ❹ 标记该 Channel 为已完成状态。
- ❺ 从 EmbeddedChannel 的出站端读取所有的整数，并验证是否只产生了绝对值。

9.3 测试异常处理

应用程序通常需要执行比转换数据更加复杂的任务。例如，你可能需要处理格式不正确的输入或者过量的数据。在下一个示例中，如果所读取的字节数超出了某个特定的限制，我们将会抛出一个 TooLongFrameException。这是一种经常用来防范资源被耗尽的方法。

在图 9-4 中，最大的帧大小已经被设置为 3 字节。如果一个帧的大小超出了该限制，那么程序将会丢弃它的字节，并抛出一个 `TooLongFrameException`。位于 `ChannelPipeline` 中的其他 `ChannelHandler` 可以选择在 `exceptionCaught()` 方法中处理该异常或者忽略它。

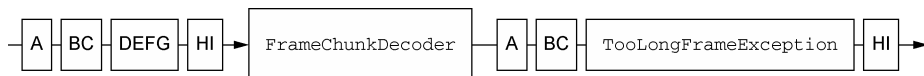


图 9-4 通过 `FrameChunkDecoder` 解码

其实现如代码清单 9-5 所示。

代码清单 9-5 `FrameChunkDecoder`

```

public class FrameChunkDecoder extends ByteToMessageDecoder {
    private final int maxFrameSize;

    public FrameChunkDecoder(int maxFrameSize) {
        this.maxFrameSize = maxFrameSize;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
        List<Object> out) throws Exception {
        int readableBytes = in.readableBytes();
        if (readableBytes > maxFrameSize) {
            // discard the bytes
            in.clear();
            throw new TooLongFrameException();
        }
        ByteBuf buf = in.readBytes(readableBytes);
        out.add(buf);
    }
}

```

扩展 `ByteToMessageDecoder` 以将入站字节解码为消息

指定将要产生的帧的最大允许大小

如果该帧太大，则丢弃它并抛出一个 `TooLongFrameException`……

……否则，从 `ByteBuf` 中读取一个新的帧

将该帧添加到解码消息的 `List` 中

我们再使用 `EmbeddedChannel` 来测试一次这段代码，如代码清单 9-6 所示。

代码清单 9-6 测试 `FrameChunkDecoder`

```

public class FrameChunkDecoderTest {
    @Test
    public void testFramesDecoded() {
        ByteBuf buf = Unpooled.buffer();
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();

        EmbeddedChannel channel = new EmbeddedChannel(
            new FrameChunkDecoder(3));
    }
}

```

创建一个 `ByteBuf`，并向它写入 9 字节

创建一个 `EmbeddedChannel`，并向其安装一个帧大小为 3 字节的 `FixedLengthFrameDecoder`


```

    assertTrue(channel.writeInbound(input.readBytes(2)));
    try {
        channel.writeInbound(input.readBytes(4));
        Assert.fail();
    } catch (TooLongFrameException e) {
        // expected exception
    }

    assertTrue(channel.writeInbound(input.readBytes(3)));
    assertTrue(channel.finish());

    // Read frames
    ByteBuf read = (ByteBuf) channel.readInbound();
    assertEquals(buf.readSlice(2), read);
    read.release();

    read = (ByteBuf) channel.readInbound();
    assertEquals(buf.skipBytes(4).readSlice(3), read);
    read.release();
    buf.release();
}

```

如果上面没有抛出异常，那么就会到达这个断言，并且测试失败

向它写入 2 字节，并断言它们将会产生一个新帧

写入一个 4 字节大小的帧，并捕获预期的 TooLongFrameException

写入剩余的 2 字节，并断言将会产生一个有效帧

将该 Channel 标记为已完成状态

读取产生的消息，并且验证值

乍一看，这看起来非常类似于代码清单 9-2 中的测试，但是它有一个有趣的转折点，即对 TooLongFrameException 的处理。这里使用的 try/catch 块是 EmbeddedChannel 的一个特殊功能。如果其中一个 write* 方法产生了一个受检查的 Exception，那么它将会被包装在一个 RuntimeException 中并抛出^①。这使得可以容易地测试出一个 Exception 是否在处理数据的过程中已经被处理了。

这里介绍的测试方法可以用于任何能抛出 Exception 的 ChannelHandler 实现。

9.4 小结

使用 JUnit 这样的测试工具来进行单元测试是一种非常行之有效的方式，它能保证你的代码的正确性并提高它的可维护性。在本章中，你学习了如何使用 Netty 提供的测试工具来测试你自定义的 ChannelHandler。

在接下来的章节中，我们将专注于使用 Netty 编写真实世界的应用程序。我们不会再提供任何进一步的测试代码示例了，所以我们希望你将这里所展示的测试方法的重要性牢记于心。

^① 需要注意的是，如果该类实现了 exceptionCaught() 方法并处理了该异常，那么它将不会被 catch 块所捕获。