# 【第07章-EventLoop和线程模型】

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。

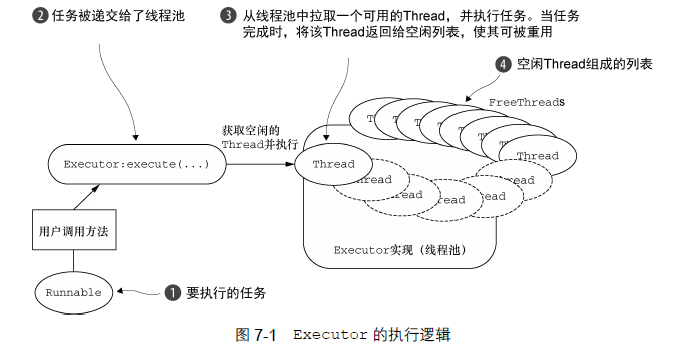
## 7.1 线程模型概述

因为具有多核心或多个CPU 的计算机现在已经司空见惯，大多数的现代应用程序都利用了复杂的多线程处理技术以有效地利用系统资源。相比之下，在早期的Java 语言中，我们使用多线程处理的主要方式无非是按需创建和启动新的Thread 来执行并发的任务单元——一种在高负载下工作得很差的原始方式。Java 5 随后引入了Executor API，其线程池通过缓存和重用Thread 极大地提高了性能。

基本的线程池化模式可以描述为：

* 从池的空闲线程列表中选择一个Thread，并且指派它去运行一个已提交的任务（一个Runnable 的实现）；
* 当任务完成时，将该Thread 返回给该列表，使其可被重用。

图7-1 说明了这个模式。



虽然池化和重用线程相对于简单地为每个任务都创建和销毁线程是一种进步，但是它并不能消除由上下文切换所带来的开销，其将随着线程数量的增加很快变得明显，并且在高负载下愈演愈烈。此外，仅仅由于应用程序的整体复杂性或者并发需求，在项目的生命周期内也可能会出现其他和线程相关的问题。

简而言之，多线程处理是很复杂的。在接下来的章节中，我们将会看到Netty 是如何帮助简化它的。

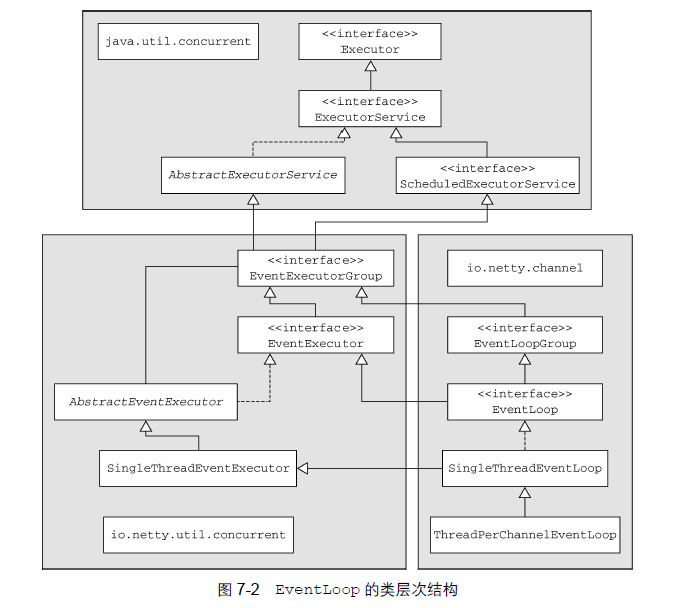
## 7.2 EventLoop 接口

**运行任务来处理在连接的生命周期内发生的事件是任何网络框架的基本功能。与之相应的编程上的构造通常被称为事件循环**—一个Netty 使用了interface io.netty.channel.EventLoop 来适配的术语。

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**  **public static void** executeTaskInEventLoop() {  **boolean** terminated = **true**;  *//...* **while** (!terminated) {  *// 阻塞，直到有事件已经就绪可被运行* List<Runnable> readyEvents = *blockUntilEventsReady*();  **for** (Runnable ev: readyEvents) {  *// 循环遍历，并处理所有的事件* ev.run();  }  } } |

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。



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

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，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。。

### 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()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。

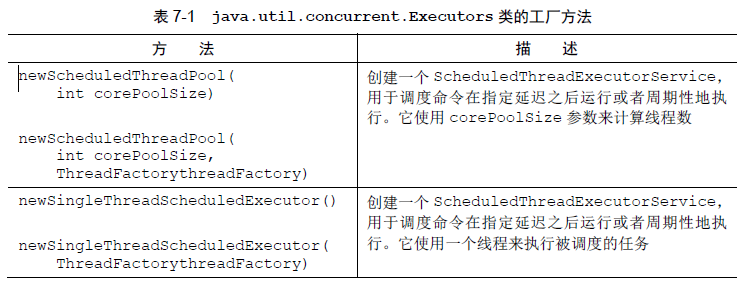
## 7.3 任务调度

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。



虽然选择不是很多，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**  **public static void** schedule() {  *// 创建一个其线程池具有10 个线程的ScheduledExecutorService* ScheduledExecutorService executor = Executors.*newScheduledThreadPool*(**10**);   ScheduledFuture<?> future = executor.schedule(**new** Runnable() { *// 创建一个 Runnable，以供调度稍后执行* @Override  **public void** run() {  *// 该任务要打印的消息* System.***out***.println(**"Now it is 60 seconds later"**);  }  }, **60**, TimeUnit.***SECONDS***); *// 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* executor.shutdown(); } |

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**  **public static void** scheduleViaEventLoop() {  Channel ch = ***CHANNEL\_FROM\_SOMEWHERE***; *// get reference from somewhere* ScheduledFuture<?> future = ch.eventLoop().schedule(  **new** Runnable() { *// 创建一个 Runnable，以供调度稍后执行* @Override  **public void** run() {  *// 要执行的代码* System.***out***.println(**"60 seconds later"**);  }  }, **60**, TimeUnit.***SECONDS***); *// 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。

|  |
| --- |
| **// 代码清单7-4 使用EventLoop 调度周期性的任务**  **public static void** scheduleFixedViaEventLoop() {  Channel ch = ***CHANNEL\_FROM\_SOMEWHERE***; *// get reference from somewhere  // 创建一个 Runnable，以供调度稍后执行* ScheduledFuture<?> future = ch.eventLoop().scheduleAtFixedRate(**new** Runnable() {  @Override  **public void** run() {  *// 这将一直运行，直到ScheduledFuture 被取消* System.***out***.println(**"Run every 60 seconds"**);  }  }, **60**, **60**, TimeUnit.***SECONDS***); *// 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。

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。

|  |
| --- |
| **//**  **public static void** cancelingTaskUsingScheduledFuture() {  Channel ch = ***CHANNEL\_FROM\_SOMEWHERE***; *// get reference from somewhere  // 调度任务，并获得所返回的ScheduledFuture* ScheduledFuture<?> future = ch.eventLoop().scheduleAtFixedRate(**new** Runnable() {  @Override  **public void** run() {  System.***out***.println(**"Run every 60 seconds"**);  }  }, **60**, **60**, TimeUnit.***SECONDS***);  *// Some other code that runs...* **boolean** mayInterruptIfRunning = **false**;  *// 取消该任务，防止它再次运行* future.cancel(mayInterruptIfRunning); } |

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

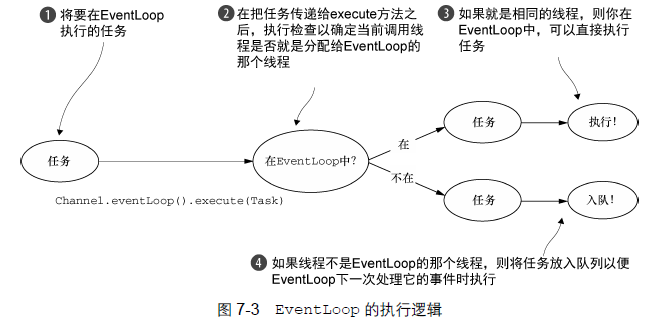
这一节将更加详细地探讨Netty 的线程模型和任务调度实现的主要内容。我们也将会提到需要注意的局限性，以及正在不断发展中的领域。

### 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，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，通过调用EventLoop 的inEventLoop(Thread)方法实现。（回想一下EventLoop将负责处理一个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 都有它自已的任务队列，独立于任何其他的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。



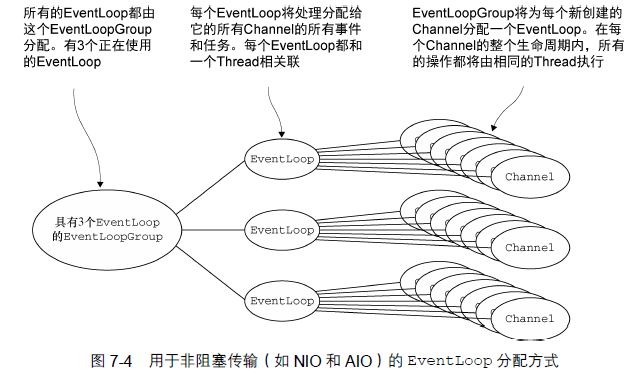
我们之前已经阐明了不要阻塞当前I/O 线程的重要性。我们再以另一种方式重申一次：**“永远不要将一个长时间运行的任务放入到执行队列中，因为它将阻塞需要在同一线程上执行的任何其他任务。”如果必须要进行阻塞调用或者执行长时间运行的任务，我们建议使用一个专门的EventExecutor。**

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。，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。

**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。



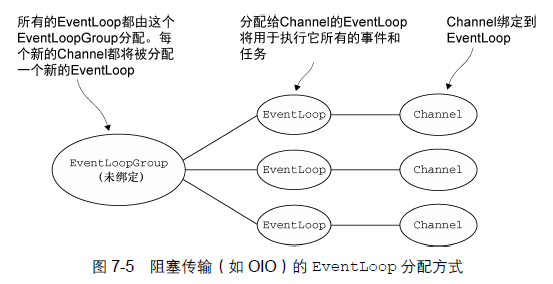
EventLoopGroup 负责为每个新创建的Channel 分配一个EventLoop。在当前实现中，使用顺序循环（round-robin）的方式进行分配以获取一个均衡的分布，并且相同的EventLoop可能会被分配给多个Channel。（这一点在将来的版本中可能会改变。）

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ChannelHandler 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。



但是，正如同之前一样，得到的保证是每个Channel 的I/O 事件都将只会被一个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。