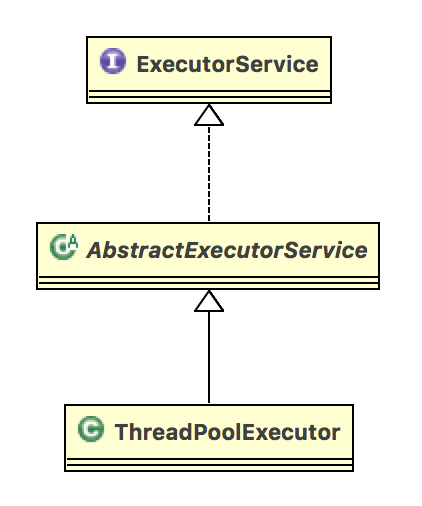
线程池之ThreadPoolExecutor概述



在Java源码里对ThreadPoolExecutor的解析注释如下：

ExecutorService（ThreadPoolExecutor的顶层接口）使用线程池中的线程执行每个提交的任务，通常我们使用Executors的工厂方法来创建ExecutorService。

线程池解决了两个不同的问题：

1. 提升性能

它们通常在执行大量异步任务时，由于减少了每个任务的调用开销，并且它们提供了一种限制和管理资源（包括线程）的方法，使得性能提升明显；

1. 统计信息

每个ThreadPoolExecutor保持一些基本的统计信息，例如完成的任务数量。

为了在广泛的上下文中有用，此类提供了许多可调参数和可扩展性钩子。但是，在常见场景中，我们预配置了几种线程池，我们敦促程序员使用更方便的Executors的工厂方法直接使用。

Executors.newCachedThreadPool（无界线程池，自动线程回收）

Executors.newFixedThreadPool（固定大小的线程池）

Executors.newSingleThreadExecutor（单一后台线程）

在自定义线程池时，请参考以下指南：

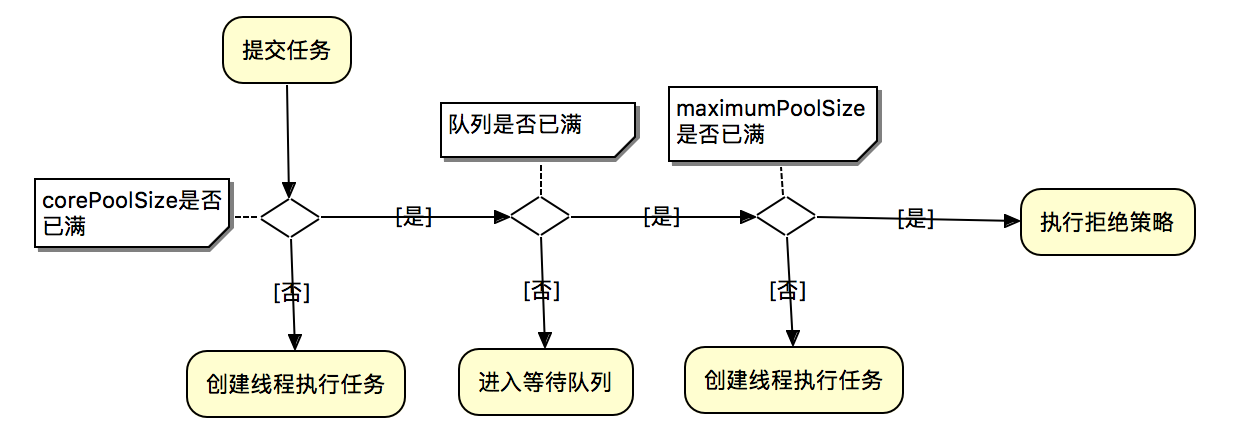
1. Core and maximum pool sizes核心和最大线程池数量

|  |  |
| --- | --- |
| **参数** | **翻译** |
| corePoolSize | 核心线程池数量 |
| maximumPoolSize | 最大线程池数量 |

线程池执行器将会根据corePoolSize和maximumPoolSize自动调整线程池大小。

当在execute（Runnable）方法中提交新任务并且少于corePoolSize线程正在运行时，即使其他工作线程处于空闲状态，也会创建一个新线程来处理该请求。如果有多于corePoolSize但小于maximumPoolSize线程正在运行，则仅当队列已满时才会创建新线程。通过设置corePoolSize和maximumPoolSize相同，您可以创建一个固定大小的线程池。通过将maximumPoolSize设置为基本上无界的值，例如Integer.MAX\_VALUE，您可以允许容纳任意数量的并发任务。通常，核心和最大池大小仅在构建时设置，但也可以使用setCorePoolSize和setMaximumPoolSize进行动态更改。

这段话详细描述了线程池对任务的处理流程，这里用个图总结一下



1. prestartCoreThread核心线程预启动

在默认情况下，只有当新任务到达时，才开始创建和启动核心线程，但是我们可以使用prestartCoreThread()和prestartAllCoreThreads()方法动态调整。

如果使用非空队列构建池，则可能需要预先启动线程。

|  |  |
| --- | --- |
| 方法 | 作用 |
| prestartCoreThread() | 创建一个空闲任务线程等待任务的到达 |
| prestartAllCoreThreads() | 创建核心线程池数量的空闲任务线程等待任务的到达 |

1. ThreadFactory线程工厂

新线程使用ThreadFactory创建。如果未另行指定，则使用Executors.defaultThreadFactory默认工厂，使其全部位于同一个ThreadGroup中，并且具有相同的NORM\_PRIORITY优先级和非守护进程状态。

通过提供不同的ThreadFactory，您可以更改线程的名称、线程组、优先级、守护进程状态等。如果ThreadFactory在通过从new Thread返回null询问时未能创建线程，则执行程序将继续，但可能无法执行任务任务。

线程应该有modifyThread权限。如果工作线程或使用该池的其他线程不具备此权限，则服务可能会降级：配置更改可能无法及时生效，并且关闭池可能会保持可终止但尚未完成的状态。

1. Keep-alive times线程存活时间

如果线程池当前拥有超过corePoolSize的线程，那么多余的线程在空闲时间超过keepAliveTime时会被终止（请参阅getKeepAliveTime(TimeUnit)）。这提供了一种在不积极使用线程池时减少资源消耗的方法。如果池在以后变得更加活跃，则应构建新线程。也可以使用方法setKeepAliveTime(long, TimeUnit)进行动态调整。

防止空闲线程在关闭之前终止，可以使用如下方法：

setKeepAliveTime(Long.MAX\_VALUE，TimeUnit.NANOSECONDS);

默认情况下，keep-alive策略仅适用于存在超时corePoolSize线程的情况。但是，只要keepAliveTime值不为零，方法allowCoreThreadTimeOut( Boolean )也可用于将此超时策略应用于核心线程。

1. Queuing队列

BlockingQueue用于存放提交的任务，队列的实际容量与线程池大小相关联。

* 如果当前线程池任务数量小于核心线程池数量，执行器总是优先创建一个任务线程，而不是从线程队列中取一个空闲线程。
* 如果当前线程池任务线程数量大于核心线程池数量，执行器总是优先从线程队列中取一个空闲线程，而不是创建一个任务线程。
* 如果当前线程池任务线程数量大于核心线程池数量，且队列中无空闲任务线程，将会创建一个任务线程，直到超出maximumPoolSize，如果超过maximumPoolSize，则任务将会被拒绝。

主要有三种队列策略：

1. Direct handoffs直接握手队列

Direct handoffs的一个很好的默认选择时SynchronousQueue，它将任务交给线程而不需要保留。这里，如果没有线程立即可用来运行它，那么排队任务的尝试将失败，因此将构建新的线程。

此策略在处理可能具有内部依赖关系的请求集时避免锁定。Direct handoffs通常需要无限制的maximumPoolSizes来避免拒绝新提交的任务。但得注意，当任务持续以平均提交速度大于平均处理速度时，会导致线程数量会无限增长问题。

1. Unbounded queues无界队列

当所有corePoolSize线程繁忙时，使用无界队列（例如，没有预定义容量的LinkedBlockingQueue）将导致新任务在队列中等待，从而导致maximumPoolSize的值没有任何作用。当每个任务互不影响，完全独立于其他任务时，这可能是合适的；例如，在网页服务器中，这种队列方式可以用于平滑瞬时大量请求。但得注意，当任务持续以平均提交速度大于平均处理速度时，会导致队列无限增长问题。

1. Bounded queues有界队列

一个有界的队列（例如，一个ArrayBlockingQueue）和有限的maximumPoolSizes配置有助于防止资源耗尽，但是难以控制。队列大小和maximumPoolSizes需要相互权衡：

* 使用大队列和较小的maximumPoolSizes可以最大限度地减少CPU使用率，操作系统资源和上下文切换开销，但会导致人为的低吞吐量。如果任务经常被阻塞（比如I/O限制），那么系统可以调度比我们允许的更多的线程。
* 使用小队列通常需要较大的maximumPoolSizes，这会使CPU更繁忙，但可能会遇到不可接受的调度开销，这也会降低吞吐量。

1. Rejected tasks拒绝任务

拒绝任务有两种情况：

1. 线程池已经被关闭；
2. 任务队列已满且maximumPoolSize已满；

无论哪种情况，都会调用RejectedExecutionHandler的rejectedExecution方法。预定义了四种处理策略：

1. AbortPolicy

默认策略，抛出RejectedExecutionException运行时异常;

1. CallerRunsPolicy

这提供了一个简单的反馈控制机制，可以减慢提交新任务的速度；

1. DiscardPolicy

直接丢弃新提交的任务；

1. DiscardOldestPolicy

如果执行器没有关闭，队列头的任务将会被丢弃，然后执行器重新尝试执行任务（如果失败，则重复这一过程）；

我们可以自己定义RejectedExecutionHandler，以适应特殊的容量和队列策略场景中。

1. Hook methods钩子方法

ThreadPoolExecutor为每个任务执行前后提供了钩子方法，重写beforeExecute(Thread, Runnable)和afterExecute(Runnable, Throwable)方法来操纵执行环境；例如，重新初始化ThreadLocals，收集统计信息或记录日志等。此外，terminated()在Executor完全终止后会被调用，可以重写此方法，以执行特殊处理。

注意：如果hook或回调方法抛出异常，内部的任务线程将会失败并结束。

1. Queue maintenance维护队列

getQueue()方法可以访问任务队列，一般用于监控和调试。绝不建议将这个方法用于其他目的。当在大量的队列任务被取消时，remove()和purge()方法可用于回收空间。

1. Finalization关闭

如果程序中不再持有线程池的引用，并且线程池中没有线程时，线程池将会自动关闭。如果您希望确保即使用户忘记调用shutdown()方法也可以回收未引用的线程池，使未使用的线程最终死亡。那么必须通过设置适当的keep-alive times并设置allowCoreThreadTimeOut( boolean )或者是corePoolSize下限为0.

一般情况下，线程池启动后建议手动调用shutdown()关闭。

总结，通过解读ThreadPoolExecutor的注释，我们对ThreadPoolExecutor应该有了比较全面的了解，其实现方式，后续章节详解。