迟任务的执行，从而提高了响应性。通过适当调整线程池的大小，可以创建足够多的线程以便使处理器保持忙碌状态，同时还可以防止过多线程相互竞争资源而使应用程序耗尽内存或失败。

类库提供了一个灵活的线程池以及一些有用的默认配置。可以通过调用 Executors 中的静态工厂方法之一来创建一个线程池：

newFixedThreadPool。newFixedThreadPool 将创建一个固定长度的线程池，每当提交一个任务时就创建一个线程，直到达到线程池的最大数量，这时线程池的规模将不再变化（如果某个线程由于发生了未预期的 Exception 而结束，那么线程池会补充一个新的线程）。

newCachedThreadPool。newCachedThreadPool 将创建一个可缓存的线程池，如果线程池的当前规模超过了处理需求时，那么将回收空闲的线程，而当需求增加时，则可以添加新的线程，线程池的规模不存在任何限制。

newSingleThreadExecutor。newSingleThreadExecutor 是一个单线程的 Executor，它创建单个工作者线程来执行任务，如果这个线程异常结束，会创建另一个线程来替代。newSingleThreadExecutor 能确保依照任务在队列中的顺序来串行执行（例如 FIFO、LIFO、优先级）。[⊖]

newScheduledThreadPool。newScheduledThreadPool 创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于 Timer（参见 6.2.5 节）。

newFixedThreadPool 和 newCachedThreadPool 这两个工厂方法返回通用的 ThreadPoolExecutor 实例，这些实例可以直接用来构造专门用途的 executor。我们将在第 8 章中深入讨论线程池的各个配置选项。

TaskExecutionWebServer 中的 Web 服务器使用了一个带有有界线程池的 Executor。通过 execute 方法将任务提交到工作队列中，工作线程反复地从工作队列中取出任务并执行它们。

从“为每任务分配一个线程”策略变成基于线程池的策略，将对应用程序的稳定性产生重大的影响：Web 服务器不会再在高负载情况下失败[⊜]。由于服务器不会创建数千个线程来争夺有限的 CPU 和内存资源，因此服务器的性能将平缓地降低。通过使用 Executor，可以实现各种调优、管理、监视、记录日志、错误报告和其他功能，如果不使用任务执行框架，那么要增加这些功能是非常困难的。

6.2.4 Executor 的生命周期

我们已经知道如何创建一个 Executor，但并没有讨论如何关闭它。Executor 的实现通常会创建线程来执行任务。但 JVM 只有在所有（非守护）线程全部终止后才会退出。因此，如果无法正确地关闭 Executor，那么 JVM 将无法结束。

⊖ 单线程的 Executor 还提供了大量的内部同步机制，从而确保了任务执行的任何内存写入操作对于后续任务来说都是可见的。这意味着，即使这个线程会不时地被另一个线程替代，但对象总是可以安全地封闭在“任务线程”中。

⊜ 尽管服务器不会因为创建了过多的线程而失败，但在足够长的时间内，如果任务到达的速度总是超过任务执行的速度，那么服务器仍有可能（只是更不易）耗尽内存，因为等待执行的 Runnable 队列将不断增长。可以通过使用一个有界工作队列在 Executor 框架内部解决这个问题（参见 8.3.2 节）。