# 一、Executor & ExecutorService

Executor和ExecutorService，前者是任务执行器的顶层接口，只有一个execute抽象方法，需要传入一个Runnable接口的对象。我们可以自己去实现它，但是一般情况下我们不需要自己去实现，而是直接使用丰富的已实现的API。

ExecutorService继承自Executor是一个执行器的扩展接口。另外有一个submit方法，这个方法不仅可以接受一个Runnable接口对象还可以接收一个Callable接口对象。

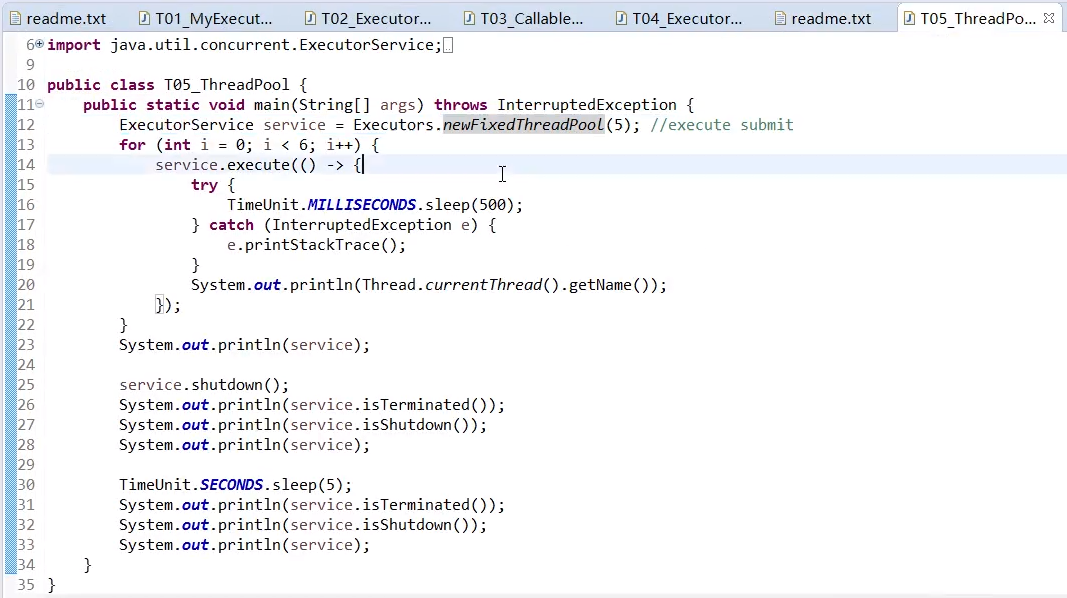
# 二、Callable接口

Callable接口与Runnable非常相似，但是Runnable接口的run方法没有返回值，并且不能抛出异常。而Callable接口中的call方法是有返回值的，且可以抛出异常。

# 三、Executors工具类

和Arrays时处理数组的工具类，Collections是处理集合的工具类一样。Executors是一个处理Executor、ExecutorService、Runnable、Callable等部分操作的工具类、工厂类。

# 四、ThreadPool线程池



解析：第一句：newFixedThreadPool(5)创建一个包含5个线程的线程池，5个线程并没有启动，只在需要的时候才启动。（回顾，ExecutorService中可以接收任务作为参数，有两个实现接收任务功能的方法：execute和submit，前者只接收实现了Runnable接口的任务，后者既可以Runnable也可以Callable）

线程池会维护一个未执行的任务队列，大多数任务队列使用的是BlockingQueue；和另一个已完成任务的队列。

**线程池会接收任务并启动空闲的线程，当任务执行完毕后，线程不会销毁，而是会处于空闲（准备再次执行）的状态，当有新的任务需要执行的时候，将会复用空闲线程，不需要重新创建。**

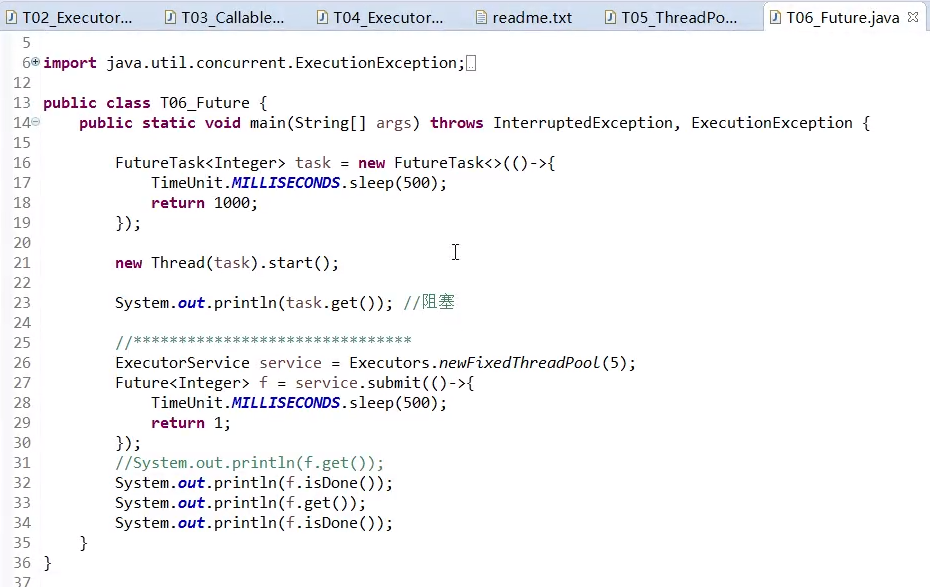
实际开发中，如果任务的数量和线程池中线程的数量控制的非常合理的情况下，不需要启动新的线程，就可以执行大量的任务，这样的效率是比较高的（启动线程和关闭线程都是需要消耗系统资源的），并发性也是非常好的。

shutdown()方法是正常关闭线程池，shutdown会等待线程池中任务全部执行完毕。另一个方法是shutdownNow()，意思是强制关闭线程池。

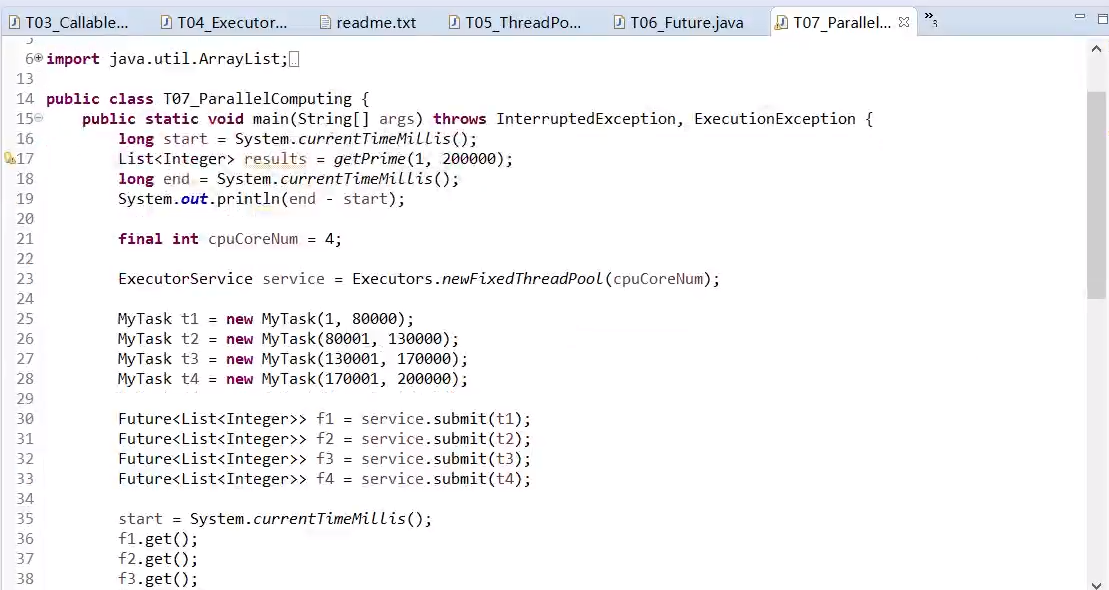
执行shutdown方法后，线程池将会从running状态变为shutting down状态，表示正在关闭，此时（执行shutdown()方法之后）调用isShutdown()将会返回true，但是这并不代表任务已经全部执行完毕，需要通过调用线程池的isTerminated()方法进一步判断。

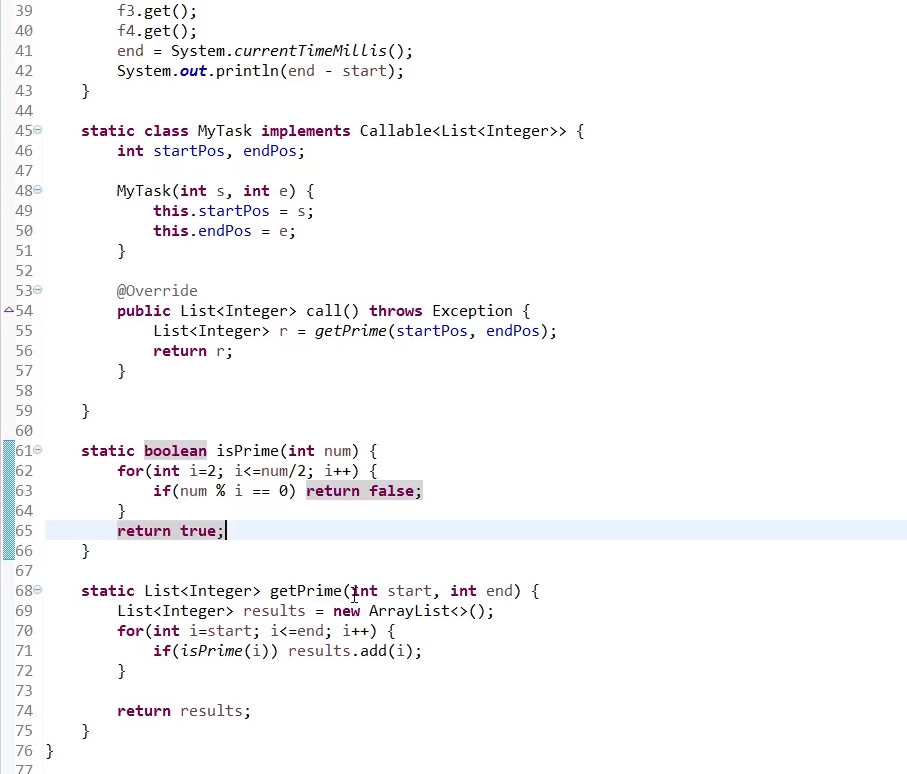
当线程池中任务执行完毕后，线程池的状态就会由Shutting down变为Terminated，此时表示任务一全部完毕。

# Future、newFixedThreadPool()



# 并行计算案例分析





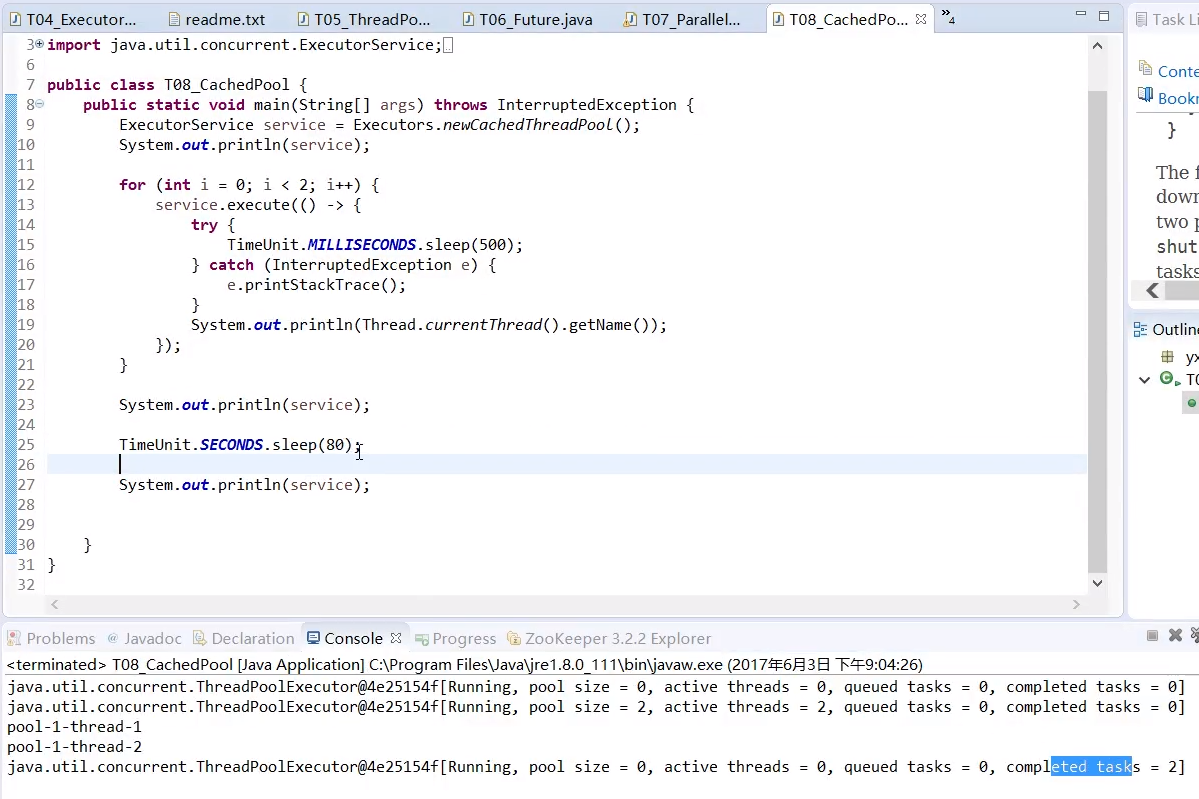
解析：这种分区域求值的算法模型曾经应用于美国NASA航天局。

该程序分析了串行计算时间和并行计算时间的比较，从而展示了一种线程池的简单应用。

cpuCorNum指的是CPU的核心数量，比如8核CPU那这个值就是8。一般在实际开发中线程池的数量往往根据服务器的CPU核数来设置，尽量不低于CPU的核数，这样每个内核可以有机会执行到一个线程，可以充分利用CPU多核的性能。

在每个MyTask中传入了一个区间范围，这个区间范围合起来就是从1 到 200000，但是四个线程并不是平均分割这个范围，这是因为后面的数会越来越大，计算质数（只能被1和自己整除的数）的时间将会加长，所以，分割区间的长度会依次递减。

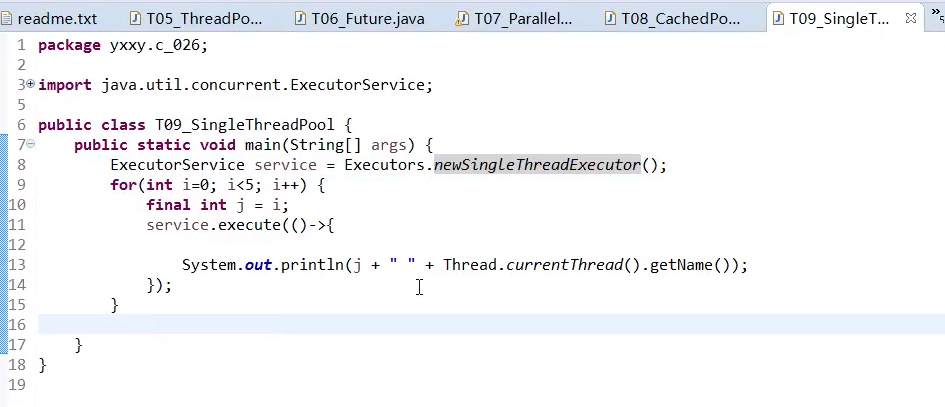
# newCachedThreadPool()



解析：这种方法生成的线程池，会默认空闲线程存活60秒（可以设置）。

虽然这种线程池没有线程数量限制，但是一般不会超过CPU所能支撑的最大线程数（一般在10000个左右）

# newSingleThreadExecutor()

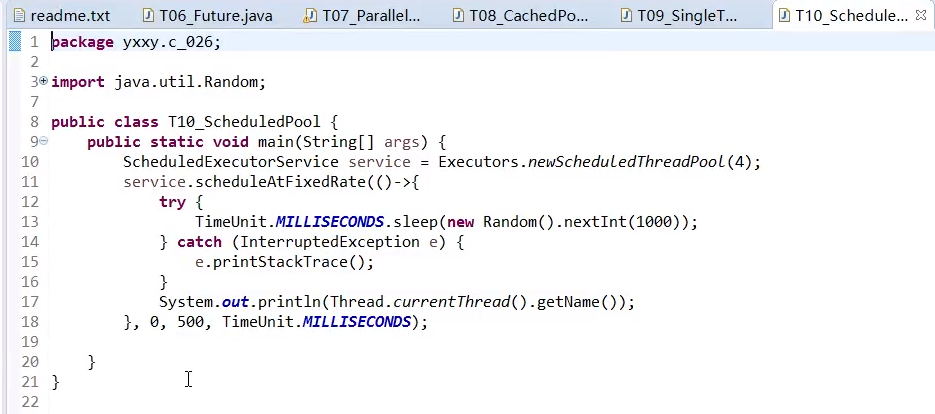


执行结果：



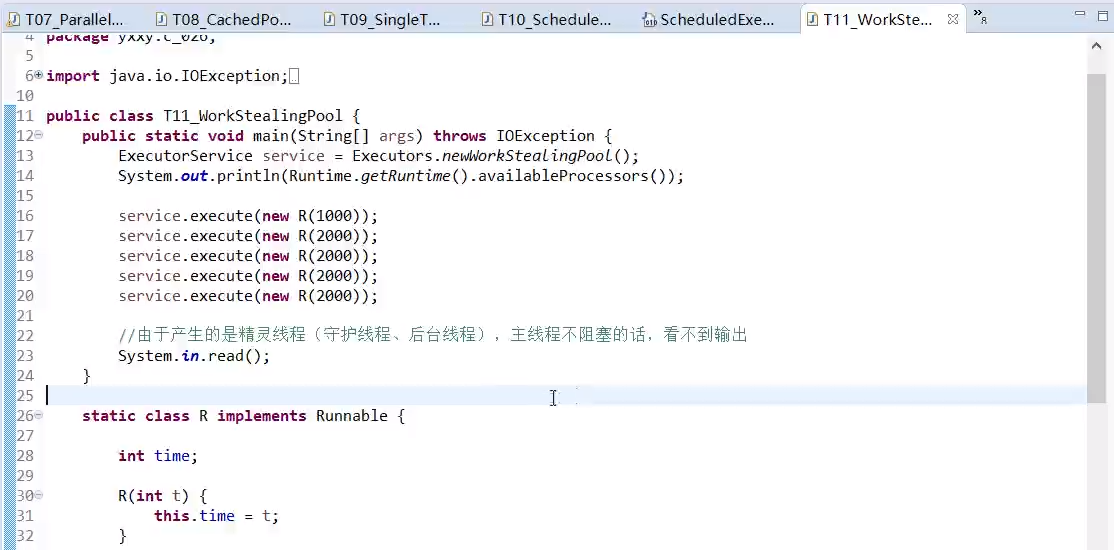
解析：先放入的任务一定先执行完。这种线程池的用途是可以保证线程之间一定是顺序执行的，这在某些面试题中也会问道，如何保证多个线程之间顺序执行。

# ScheduledPool



解析：这种线程池一般可以替代Timer

# WorkStealingPool



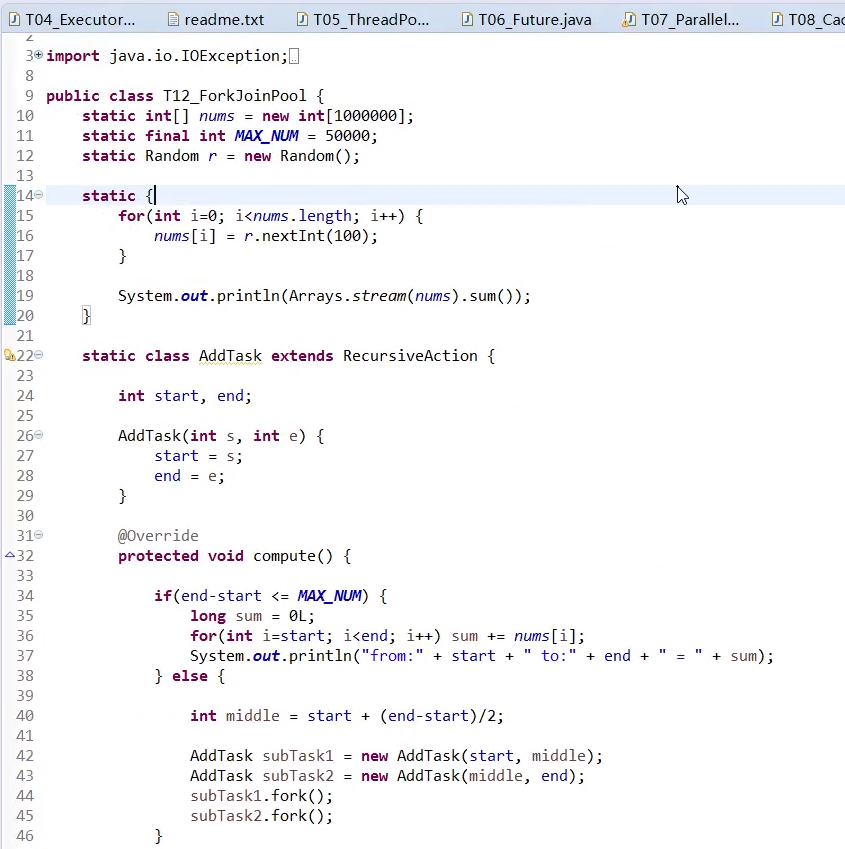


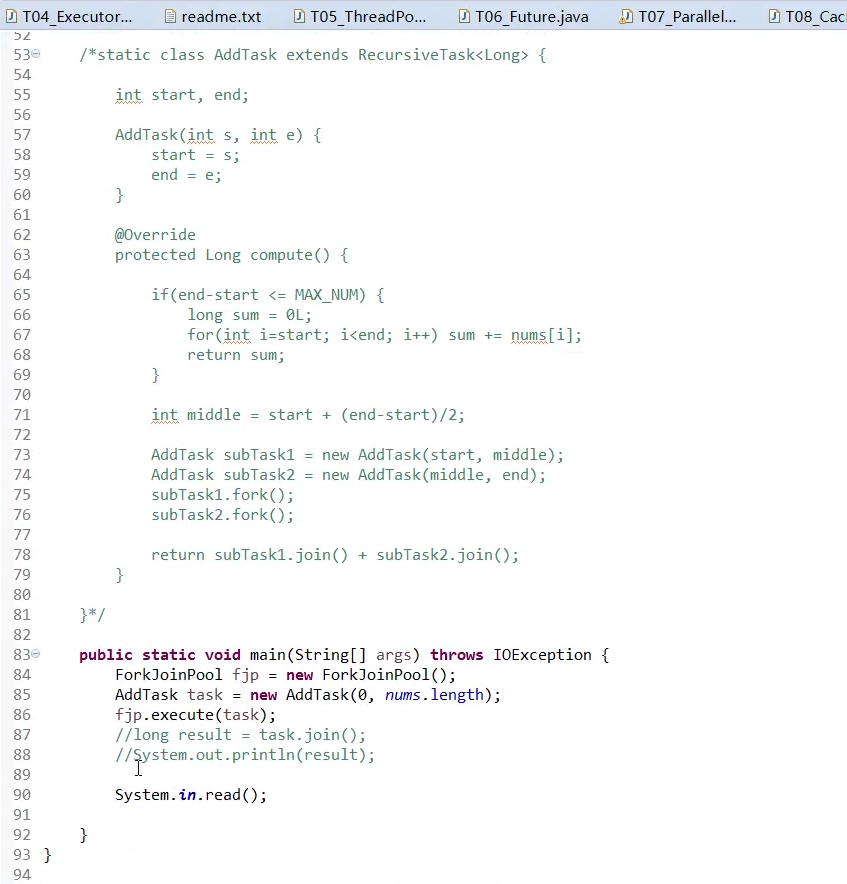
解析：每个线程都维护一个自己的任务队列，workStealingPool线程池中某个线程执行完自己线程任务队列中的任务后，便会去其他线程任务列表中取得未执行的任务继续执行。是一种线程主动执行任务的一种线程池实现。它本身是通过ForkJoinPool来实现的。

上图代码中，availableProcessors()执行结果是获得CPU核心数。而workStealingPool会根据CPU核心数默认创建相同数量的线程。

WorkStealingPool是一种守护线程（或叫精灵线程）池，daemon Thread Pool，精灵线程会在后台不断运行，直到虚拟机停止。而其他线程池中都属于普通的线程。

# ForkJoinPool





解析：ForkJoinPool会将大的任务分配给多个子任务去执行，最后将计算结果合并。

由于子任务还可以继续分配子任务，所以其内部是采用了一种递归的方式来实现这个线程池的。回看区间：87:47-104

# ThreadPoolExecutor线程池实现

FixedThreadPool、CachedThreadPool、ScheduledPool、SingleThreadPool，这四种线程池其底层都是使用ThreadPoolExecutor来实现的。

# （补充知识）ParallelStream API

