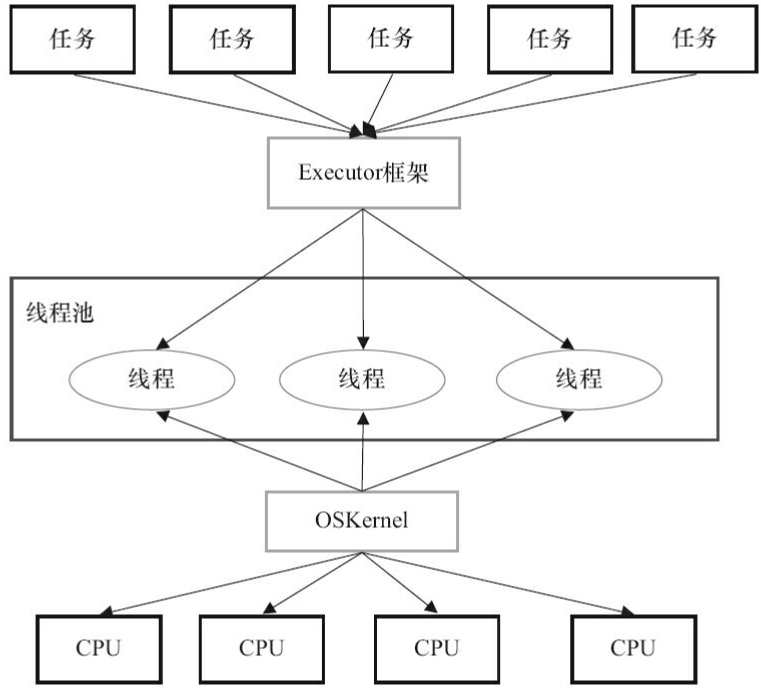
**1.Executor框架**

**1.1两级调度模型**

Java线程启动时候会创建一个本地操作系统线程，当该java线程终止时，这个操作系统线程也会被回收。操作系统会调度所有线程并分配cpu。



上层，多线程程序通常把应用分解成若干个任务，然后Executor将任务映射为固定数量的线程，底层，系统内核将线程映射到cpu处理器上。应用程序通过Executor控制上层调度，而下层通过操作系统内核控制。

**1.2Executor框架的结构及成员**

①Executor的结构

**任务**：被执行任务需要实现的接口**：Runnable接口或Callable接口**

**任务的执行**：核心接口**Executor**，以及继承自**Executor**的**ExecutorService接口**。Executor框架有两个关键类实现了ExecutorService接口（ThreadPoolExecutor和ScheduledThreadPoolExecutor）。

**异步计算的结果**。包括接口Future和实现Future接口的FutureTask类。

·**Executor是一个接口**，它是Executor框架的基础，它将**任务的提交与任务的执行分离**开来。

·ThreadPoolExecutor是**线程池的核心实现类**，用来执行**被提交的任务**。

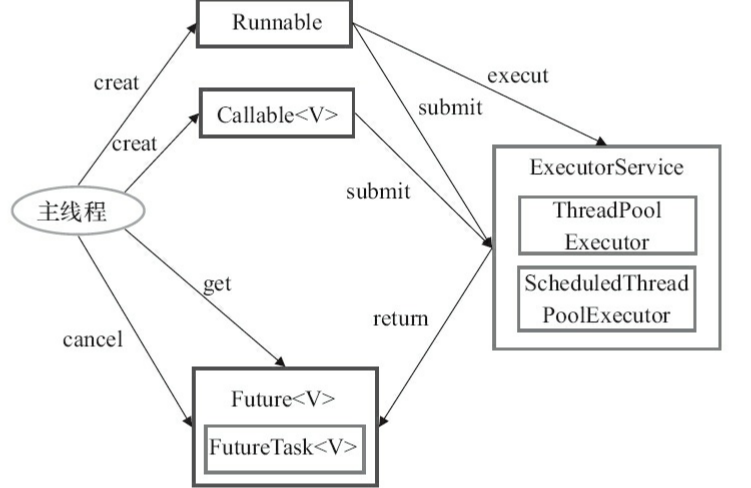
·ScheduledThreadPoolExecutor是一个实现类，可以在**给定的延迟后**运行命令，或者**定期执**

**行命令**。ScheduledThreadPoolExecutor比**Timer更灵活，功能更强大**。

·Future接口和实现Future接口的**FutureTask**类，代表**异步计算**的结果。

·Runnable接口和Callable接口的实现类，都可以被ThreadPoolExecutor或Scheduled-ThreadPoolExecutor执行。

主线程首先要创建实现**Runnable或者Callable接口的任务对象**。然后可以把**Runnable**对象直接交给**ExecutorService**执行（ExecutorService.**execute**（Runnablecommand））；或者也可以把**Runnable对象或Callable对象**提交给**ExecutorService**执行（Executor-Service.**submi**t（Runnable task）或ExecutorService.submit（Callable<T>task））。**如果执行ExecutorService.submit（…）**，ExecutorService将返回一个实现**Future接口的对象**（到目前为止的JDK中，返回的是**FutureTask**对象）。最后，主线程可以执行**FutureTask.get()**方法来**等待任务执行完成**。主线程也可以执行FutureTask.**cancel**（boolean mayInterruptIfRunning）来**取消此任务的执行**。



②Executor框架的主要成员

ThreadPoolExecutor、ScheduledThreadPoolExecutor、Future接口、Runnable接口、Callable接口和Executors

**（1）ThreadPoolExecutor**

ThreadPoolExecutor通常使用工厂类Executors来创建。

1. FixedThreadPool，适用于为了满足资源管理的需求，而需要限制**当前线程数量的应用场**

**景**，它适用于**负载比较重**的服务器。

2. SingleThreadExecutor。适用于需要保证**顺序地执行各个任务**；并且在任意时间点，不会有多个线程是活动的应用场景。

3. CachedThreadPool。大小无界的线程池，适用于执行**很多的短期异步任务**的小程序，或者是**负载较轻**的服务器。

**（2）ScheduledThreadPoolExecutor**

ScheduledThreadPoolExecutor通常使用工厂类Executors来创建。

1. ScheduledThreadPoolExecutor。适用于**需要多个后台线程**执行**周期任务**，同时为了**满足资源管理的需求**而**需要限制后台线程的数量**的应用场景

2. SingleThreadScheduledExecutor适用于需要单个后台线程执行周期任务，同时需要**保证顺**

**序地执行**各个任务的应用场景。

**（3）Future接口**

Future接口和实现Future接口的**FutureTask**类用来表示**异步计算**的结果。

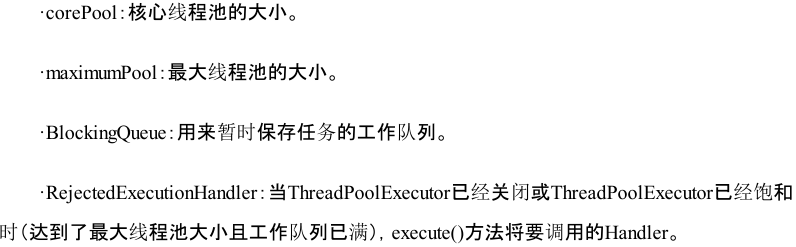
Runnable接口或Callable接口的实现类**提交（submit）**给ThreadPoolExecutor或ScheduledThreadPoolExecutor时，ThreadPoolExecutor或ScheduledThreadPoolExecutor会向我们返回一个**FutureTask对象**。



**（4）Runnable接口和Callable接口**

Runnable接口和Callable接口的实现类，都可以被ThreadPoolExecutor或Scheduled-ThreadPoolExecutor执行。它们之间的区别是Runnable**不会返回结果**，而Callable**可以返回结果。**

**2. ThreadPoolExecutor详解**



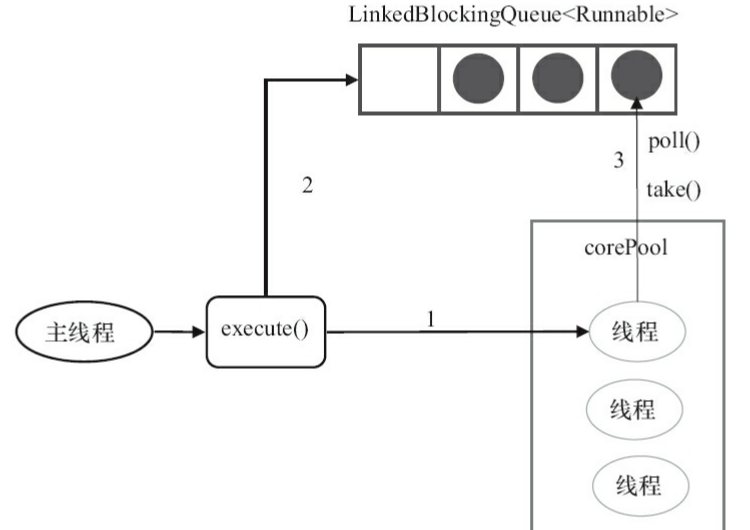
**2.1FixedThreadPool详解**

FixedThreadPool被称为可重用固定线程数的线程池。

FixedThreadPool的corePoolSize和maximumPoolSize都被设置为创建FixedThreadPool时指定的参数nThreads，指定的容量。

当线程池中的**线程数大于corePoolSize**时，keepAliveTime为多余的**空闲线程等待新任务的**

**最长时间**，超过这个时间后**多余的线程将被终止**。这里把keepAliveTime设置为0L，意味着**多余的空闲线程会被立即终止**。



1）如果当前运行的线程数少于corePoolSize，则创建新线程来执行任务。

2）在线程池完成预热之后（**当前运行的线程数等于corePoolSize**），将任务加入LinkedBlockingQueue。

3）线程执行完1中的任务后，会在循环中**反复从LinkedBlockingQueue获取任务**来执行。

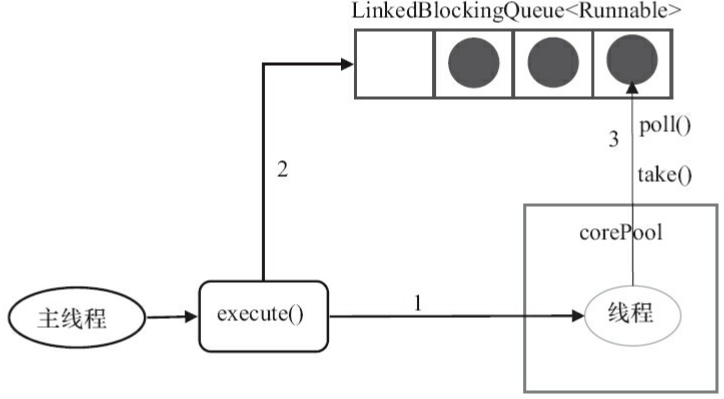
FixedThreadPool使用无界队列LinkedBlockingQueue作为线程池的工作队列：

缺点：达到corepoolsize后，新任务在**无界队列中等待**，线程池中的线程数**不会超过corepoolsize**。maximumPoolSize将是一个**无效参数**。FixedThreadPool**不会进行饱和策略**。

**2.2SingleThreadExecutor详解**

使用单个worker线程的Executor。

SingleThreadExecutor的corePoolSize和maximumPoolSize被设置为1。其他参数与FixedThreadPool相同。SingleThreadExecutor使用无界队列LinkedBlockingQueue作为线程池的工作队列（队列的容量为Integer.MAX\_VALUE）。

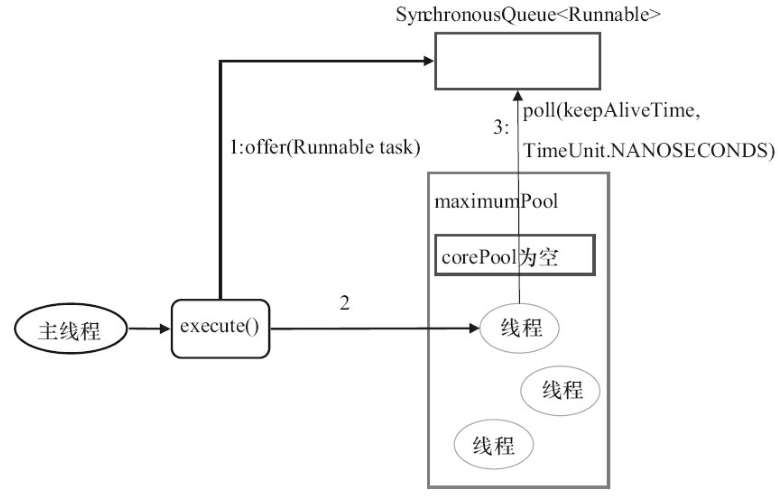


当前运行线程少于corePoolSize，创建一个新线程执行任务，如果已经有一个任务在执行，将任务加入无界队列，执行完任务后，无限循环反复从无界队列获取任务

**2.3CachedThreadPool详解**

**CachedThreadPool**的**corePoolSize**被设置为0，即**corePool为空**；maximumPoolSize被设置为Integer.MAX\_VALUE，即**maximumPool是无界的**。这里把keepAliveTime设置为60L，意味着CachedThreadPool中的**空闲线程等待新任务的最长时间为60秒**，空闲线程超过**60秒后将会被终止**。

CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列，但CachedThreadPool的maximumPool是无界的。这意味着，**如果主线程提交任务的速度高于maximumPool中线程处理任务的速度时，CachedThreadPool会不断创建新线程。极端情况下，CachedThreadPool会因为创建过多线程而耗尽CPU和内存资源**



1）首先执行**SynchronousQueue.offer**。如果当前maximumPool中有空闲线程正在执行**SynchronousQueue.poll**，那么主线程执行offer操作与空闲线程执行的poll操作**配对成功**，主线程把**任务交给空闲线程执行，execute()方法执行完成**；否则执行下面的步骤2）。

2）**没有线程执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS）**。这种情况下，步骤1）将失败。此时CachedThreadPool会**创建一个新线程执行任务，execute()方法执行完成。**

3）在步骤2）中新创建的线程将**任务执行完**后，会执行**SynchronousQueue.poll**。这个poll操作会让空闲线程最多在**SynchronousQueue中等待60秒钟**。如果60秒钟内主线程提交了**一个新任务**（主线程**执行步骤1**）），那么这个**空闲线程将执行主线程提交的新任务**；否则，这个**空闲线程将终止**。由于空闲60秒的空闲线程会被终止，因此长时间保持空闲的CachedThreadPool不会使用任何资源。

**3. ScheduledThreadPoolExecutor详解**

**-------------------以后整理**

**4. FutureTask详解**

Future接口和实现Future接口的FutureTask类，代表异步计算的结果

**4.1FutureTask简介**

FutureTask除了实现Future接口外，还实现了Runnable接口。因此，FutureTask可以交给Executor执行，也可以由调用线程直接执行（FutureTask.run()）

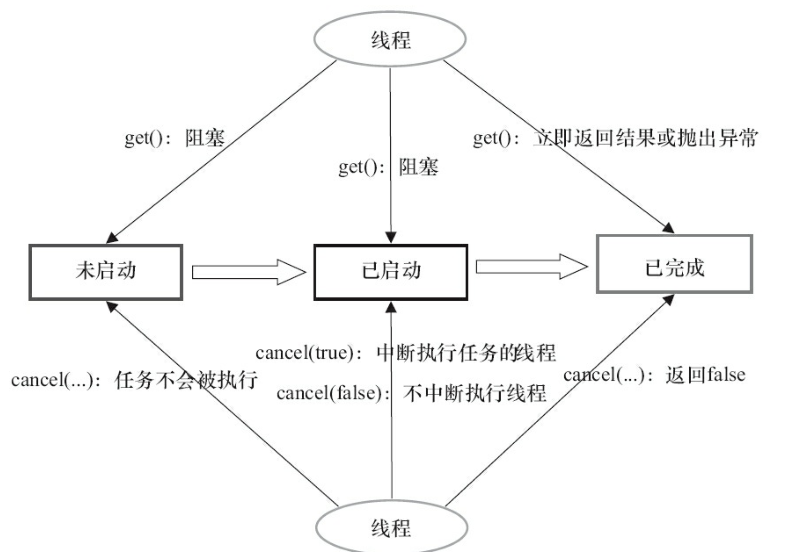
**FutureTask的3种状态**

1）未启动。FutureTask.run()方法还没有**被执行之前**，FutureTask处于未启动状态。

2）已启动。FutureTask.run()方法**被执行的过程中**，FutureTask处于**已启动状态**。

3）已完成。FutureTask.run()方法执行完后**正常结束**，或**被取消**（FutureTask.cancel（…）），或**执行FutureTask.run()方法时抛出异常而异常结束**，FutureTask处于**已完成状态**。

当FutureTask处于**未启动或已启动**状态时，执行FutureTask.get()方法将导致调用**线程阻塞**；当FutureTask处于已完成状态时，执行FutureTask.get()方法将导致调用线程**立即返回结果或抛出异常。**



4.2FutureTask的使用

可以把**FutureTask交给Executor执行**；也可以通过**ExecutorService.submit（…）方法返回一个FutureTask**，然后执行**FutureTask.get()方法或FutureTask.cancel（…）方法**。当一个线程需要等待另一个线程把某个任务执行完后它才能继续执行，此时可以使用FutureTask。假设有**多个线程执行若干任务，每个任务最多只能被执行一次**。**当多个线程试图同时执行同一个任务时，只允许一个线程执行任务，其他线程需要等待这个任务执行完后才能继续执行。**

4.3FutureTask的实现

基于AbstractQueuedSynchronizer（以下简称为AQS）它提供通用机制来原子性管理同步状态、阻塞和唤醒线程，以及维护被阻塞线程的队列。

每一个基于AQS实现的同步器都会包含两种类型的操作

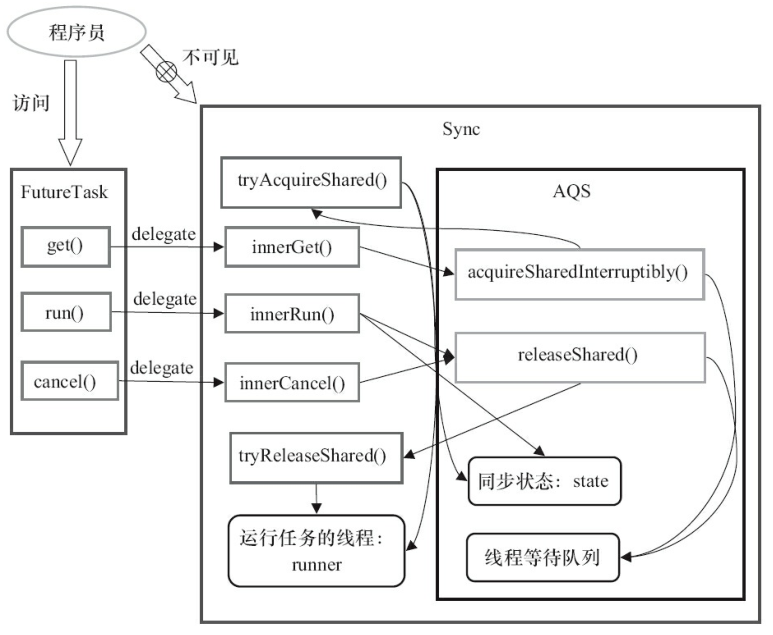
·至少一个acquire操作。这个操作阻塞调用线程，除非/直到AQS的状态允许这个线程继续

执行。**FutureTask的acquire操作为get()/get（long timeout，TimeUnit unit）方法调用。**

·至少一个release操作。**这个操作改变AQS的状态，改变后的状态可允许一个或多个阻塞**

**线程被解除阻塞**。**FutureTask的release操作包括run()方法和cancel（…）方法。**

FutureTask包含内部类sync，sync实现了AQS的tryAquireShared（int）方法和tryReleaseShared（int）方法，Sync通过这两个方法来检查和更新同步状态。



FutureTask.**get()**方法会调用**AQS.acquireSharedInterruptibly（int arg）**方法：

1）调用**AQS.acquireSharedInterruptibly（int arg）方法**，这个方法首先会回调在子类Sync中实现的**tryAcquireShared()方法**来判断**acquire操作是否可以成功**。acquire操作可以成功的条件为：**state为执行完成状态RAN或已取消状态CANCELLED**，**且runner不为nul**l。

2）**如果成功则get()方法立即返回**。如果失败则到**线程等待队列中去等待其他线程执行release操作**。

3）当其他线程执行release操作（**比如FutureTask.run()或FutureTask.cancel（…）**）唤醒当前线程后，当前线程**再次执行tryAcquireShared()将返回正值1**，当前线程将**离开线程等待队列并唤醒它的后继线程**（这里会产生级联唤醒的效果，后面会介绍）。

4）最后返回**计算的结果或抛出异常**。

**FutureTask.run()的执行过程如下**。

1）执行在构造函数中指定的任务（Callable.call()）。

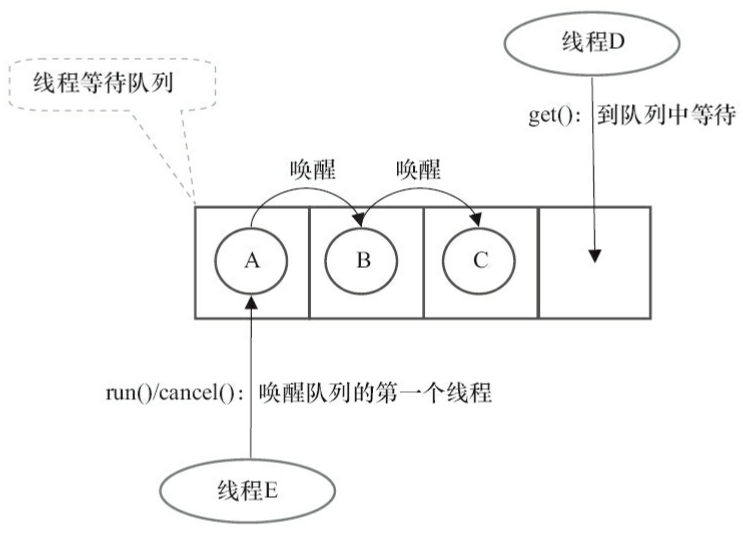
2）以原子方式来更新同步状态（**调用AQS.compareAndSetState（int expect，int update），设置state为执行完成状态RAN**）。如果这个原子**操作成功**，就设置代表计算结果的**变量result的值为Callable.call()的返回值**，然后调用**AQS.releaseShared（int arg）**。

3）AQS.releaseShared（int arg）首先会回调在子类**Sync中实现的tryReleaseShared（arg）来执行release操作**（设置运行任务的线程runner为null，然会返回true）；AQS.releaseShared（int arg），然后**唤醒线程等待队列中的第一个线程**。

4）调用FutureTask.cancel()。

当执行FutureTask.get()方法时，如果FutureTask不是处于执行完成状态RAN或已取消状态

CANCELLED，当前执行线程将到AQS的线程等待队列中等待（见下图的线程A、B、C和D）。当某个线程执行FutureTask.run()方法或FutureTask.cancel（...）方法时，会唤醒线程等待队列的第一个线程（见图10-16所示的线程E唤醒线程A）。



当线程E执行run()方法时，会**唤醒队列中的第一个线程A**。线程**A被唤醒后，首先把自己从队列中删除，然后唤醒它的后继线程B**，最后线程**A从get()方法返回**。线程B、C和D重复A线程的处理流程。最终，在队列中等待的所有线程都被级联唤醒并从get()方法返回。