## 什么是线程池？

1. 降低资源的消耗
2. 提高响应速度，任务：T1创建线程时间，T2任务执行时间，T3线程销毁时间，线程池没有或者减少T1和T3
3. 提高线程的可管理性。

#### 线程池要做些什么？

1. 线程的容器和管理器
2. 工作线程
3. 任务接口
4. 任务的容器

#### 线程池的主要处理流程

1）线程池判断核心线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作

线程来执行任务。如果核心线程池里的线程都在执行任务，则进入下个流程。

2）线程池判断工作队列是否已经满。如果工作队列没有满，则将新提交的任务存储在这

个工作队列里。如果工作队列满了，则进入下个流程。

3）线程池判断线程池的线程是否都处于工作状态。如果没有，则创建一个新的工作线程

来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

## ThreadPoolExecutor执行execute()方法的示意

1）如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（注意，执行这一步骤

需要获取全局锁）。

2）如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。

3）如果无法将任务加入BlockingQueue（队列已满），则创建新的线程来处理任务（注意，执行这一步骤需要获取全局锁）。

4）如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

## 线程池的创建各个参数含义

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler)

#### corePoolSize

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于corePoolSize；

如果当前线程数为corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；

如果执行了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

#### maximumPoolSize

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于maximumPoolSize

#### keepAliveTime

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间。默认情况下，该参数只在线程数大于corePoolSize时才有用

#### TimeUnit

keepAliveTime的时间单位

#### workQueue

workQueue必须是BlockingQueue阻塞队列。当线程池中的线程数超过它的corePoolSize的时候，线程会进入阻塞队列进行阻塞等待。通过workQueue，线程池实现了阻塞功能

#### threadFactory

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名

Executors静态工厂里默认的threadFactory，线程的命名规则是“pool-数字-thread-数字”

#### RejectedExecutionHandler（饱和策略）

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

（1）AbortPolicy：直接抛出异常，默认策略；

（2）CallerRunsPolicy：用调用者所在的线程来执行任务；

（3）DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；

（4）DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

## 关闭线程池

ShutDown():interrupt方法来终止线程

shutDownNow() 尝试停止所有正在执行的线程

## 合理地配置线程池

线程数配置：

任务：计算密集型，IO密集型，混合型

计算密集型=计算机的cpu数或计算机的cpu数+1（应付页缺失）

IO密集型=计算机的cpu数\*2

混合型，拆分成计算密集型，IO密集型

Runtime.*getRuntime*().availableProcessors();当前机器中的cpu核心个数

尽量有界队列，不要使用无界队列

## Executor框架调度模型

在HotSpot VM的线程模型中，Java线程（java.lang.Thread）被一对一映射为本地操作系统线程。Java线程启动时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将它们分配给可用的CPU。

在上层，Java多线程程序通常把应用分解为若干个任务，然后使用用户级的调度器（Executor框架）将这些任务映射为固定数量的线程；在底层，操作系统内核将这些线程映射到硬件处理器上。

从图中可以看出，应用程序通过Executor框架控制上层的调度；而下层的调度由操作系统内核控制，下层的调度不受应用程序的控制。

## 三大组成部分：任务，任务的执行，异步计算的结果

#### 任务

包括被执行任务需要实现的接口：Runnable接口或Callable接口。

#### 任务的执行。

包括任务执行机制的核心接口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。

## 成员结构图

**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。

**ExecutorService**接口继承了Executor，在其上做了一些shutdown()、submit()的扩展，可以说是真正的线程池接口；

**AbstractExecutorService**抽象类实现了ExecutorService接口中的大部分方法；

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

**ScheduledExecutorService**接口继承了ExecutorService接口，提供了带"周期执行"功能ExecutorService；

**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。

## Executor框架基本使用流程

主线程首先要创建实现Runnable或者Callable接口的任务对象。

工具类Executors可以把一个Runnable对象封装为一个Callable对象（Executors.callable（Runnable task）或Executors.callable（Runnable task，Object resule））。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.submit（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）。由于FutureTask实现了Runnable，程序员也可以创建FutureTask，然后直接交给ExecutorService执行。

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。

ThreadPoolExecutor通常使用工厂类Executors来创建。Executors可以创建3种类型的ThreadPoolExecutor：SingleThreadExecutor、FixedThreadPool和CachedThreadPool。

#### FixedThreadPool详解

创建使用固定线程数的FixedThreadPool的API。**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，适用于负载比较重的服务器**。FixedThreadPool的corePoolSize和maximumPoolSize都被设置为创建FixedThreadPool时指定的参数nThreads。

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。

FixedThreadPool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）。使用无界队列作为工作队列会对线程池带来如下影响。

1）当线程池中的线程数达到corePoolSize后，新任务将在无界队列中等待，因此线程池中的线程数不会超过corePoolSize。

2）由于1，使用无界队列时maximumPoolSize将是一个无效参数。

3）由于1和2，使用无界队列时keepAliveTime将是一个无效参数。

4）由于使用无界队列，运行中的FixedThreadPool（未执行方法shutdown()或

shutdownNow()）不会拒绝任务（不会调用RejectedExecutionHandler.rejectedExecution方法）。

#### SingleThreadExecutor详解

创建使用单个线程的SingleThread-Executor的API，**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。**

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）。

#### CachedThreadPool详解

创建一个会根据需要创建新线程的CachedThreadPool的API。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。**

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。

FixedThreadPool和SingleThreadExecutor使用无界队列LinkedBlockingQueue作为线程池的工作队列。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。

#### WorkStealingPool

利用所有运行的处理器数目来创建一个工作窃取的线程池，使用forkjoin实现。

## ScheduledThreadPoolExecutor详解

使用工厂类Executors来创建。Executors可以创建2种类

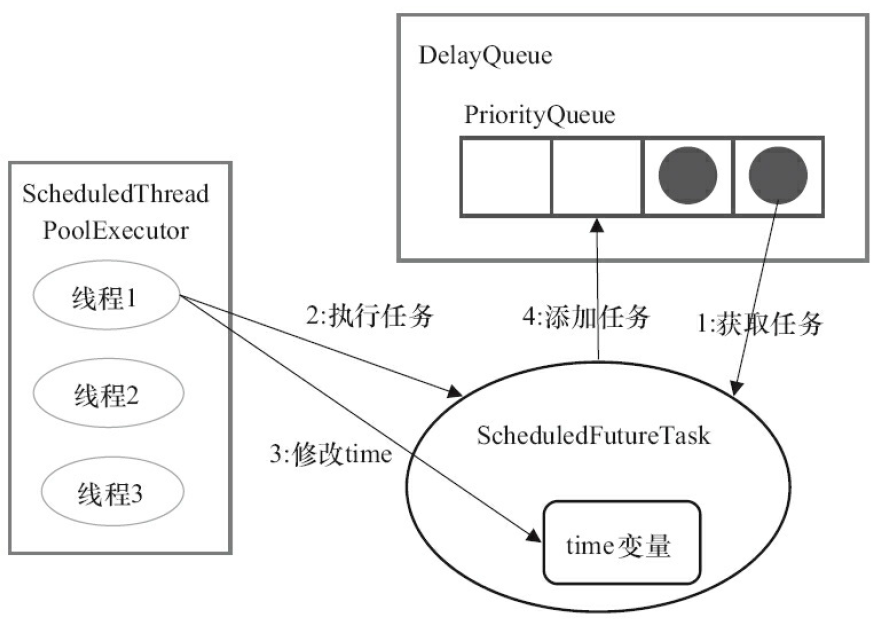
型的ScheduledThreadPoolExecutor，如下。

·**ScheduledThreadPoolExecutor**。包含若干个线程的ScheduledThreadPoolExecutor。

·**SingleThreadScheduledExecutor**。只包含一个线程的ScheduledThreadPoolExecutor。

**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。**

**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。**



对这4个步骤的说明。

1）线程1从DelayQueue中获取已到期的ScheduledFutureTask（DelayQueue.take()）。到期任务是指ScheduledFutureTask的time大于等于当前时间。

2）线程1执行这个ScheduledFutureTask。

3）线程1修改ScheduledFutureTask的time变量为下次将要被执行的时间。

4）线程1把这个修改time之后的ScheduledFutureTask放回DelayQueue中（Delay-

Queue.add()）。

有关提交定时任务的四个方法：

//向定时任务线程池提交一个延时Runnable任务（仅执行一次）

public ScheduledFuture<?> **schedule**(Runnable command, long delay, TimeUnit unit)

//向定时任务线程池提交一个延时的Callable任务（仅执行一次）

public <V> ScheduledFuture<V> **schedule**(Callable<V> callable, long delay, TimeUnit unit);

//向定时任务线程池提交一个固定时间间隔执行的任务

public ScheduledFuture<?> **scheduleAtFixedRate**(Runnable command, long initialDelay,

                                                  long period, TimeUnit unit)

**//向定时任务线程池提交一个固定延时间隔执行的任务**

public ScheduledFuture<?> **scheduleWithFixedDelay**(Runnable command, long initialDelay,

                                                      long delay, TimeUnit unit);

固定时间间隔的任务不论每次任务花费多少时间，下次任务开始执行时间是确定的，当然执行任务的时间不能超过执行周期。

固定延时间隔的任务是指每次执行完任务以后都延时一个固定的时间。由于操作系统调度以及每次任务执行的语句可能不同，所以每次任务执行所花费的时间是不确定的，也就导致了每次任务的执行周期存在一定的波动。

注意：定时或延时任务中所涉及到时间、周期不能保证实时性及准确性，实际运行中会有一定的误差。

ScheduleThreadPoolExecutor与Timer相比的优势。

（1）Timer是基于绝对时间的延时执行或周期执行，当系统时间改变，则任务的执行会受到的影响。而ScheduleThreadPoolExecutore中，任务时基于相对时间进行周期或延时操作。

（2）Timer也可以提交多个TimeTask任务，但只有一个线程来执行所有的TimeTask，这样并发性受到影响。而ScheduleThreadPoolExecutore可以设定池中线程的数量。

（3）Timer不会捕获TimerTask的异常，只是简单地停止，这样势必会影响其他TimeTask的执行。而ScheduleThreadPoolExecutore中，如果一个线程因某些原因停止，线程池可以自动创建新的线程来维护池中线程的数量。

#### scheduleAtFixedRate定时任务超时问题

若任务处理时长超出设置的定时频率时长,本次任务执行完才开始下次任务,下次任务已经处于超时状态,会马上开始执行.

若任务处理时长小于定时频率时长,任务执行完后,定时器等待,下次任务会在定时器等待频率时长后执行

如下例子:

设置定时任务每60s执行一次

若第一次任务时长80s,第二次任务时长20ms,第三次任务时长50ms

第一次任务第0s开始,第80s结束;

第二次任务第80s开始,第110s结束;(上次任务已超时,本次不会再等待60s,会马上开始),

第三次任务第150s开始,第200s结束.

第四次任务第210s开始.....

## Callable、Future和FutureTask详解

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

当我们把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。

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。

除了可以自己创建实现Callable接口的对象外，还可以使用工厂类Executors来把一个Runnable包装成一个Callable。

Executors提供的，把一个Runnable包装成一个Callable的API。

public static Callable<Object> callable(Runnable task) // 假设返回对象Callable1

Executors提供的，把一个Runnable和一个待返回的结果包装成一个Callable的API。

public static <T> Callable<T> callable(Runnable task, T result) // 假设返回对象Callable2

当任务成功完成后FutureTask.get()将返回该任务的结果。例如，如果提交的是对象Callable1，FutureTask.get()方法将返回null；如果提交的是对象Callable2，FutureTask.get()方法将返回result对象。

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()）。

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。

当FutureTask处于未启动状态时，执行FutureTask.cancel()方法将导致此任务永远不会被执行；当FutureTask处于已启动状态时，执行FutureTask.cancel（true）方法将以中断执行此任务线程的方式来试图停止任务；当FutureTask处于已启动状态时，执行FutureTask.cancel（false）方法将不会对正在执行此任务的线程产生影响（让正在执行的任务运行完成）；当FutureTask处于已完成状态时，执行FutureTask.cancel（…）方法将返回false。

## CompletionService详解

CompletionService实际上可以看做是Executor和BlockingQueue的结合体。CompletionService在接收到要执行的任务时，通过类似BlockingQueue的put和take获得任务执行的结果。CompletionService的一个实现是ExecutorCompletionService，ExecutorCompletionService把具体的计算任务交给Executor完成。在实现上，ExecutorCompletionService在构造函数中会创建一个BlockingQueue（使用的基于链表的无界队列LinkedBlockingQueue），该BlockingQueue的作用是保存Executor执行的结果。当计算完成时，调用FutureTask的done方法。当提交一个任务到ExecutorCompletionService时，首先将任务包装成QueueingFuture，它是FutureTask的一个子类，然后改写FutureTask的done方法，之后把Executor执行的计算结果放入BlockingQueue中。

与ExecutorService最主要的区别在于submit的task不一定是按照加入时的顺序完成的。CompletionService对ExecutorService进行了包装，内部维护一个保存Future对象的BlockingQueue。只有当这个Future对象状态是结束的时候，才会加入到这个Queue中，take()方法其实就是Producer-Consumer中的Consumer。它会从Queue中取出Future对象，如果Queue是空的，就会阻塞在那里，直到有完成的Future对象加入到Queue中。所以，先完成的必定先被取出。这样就减少了不必要的等待时间。

总结：

使用方法一，自己创建一个集合来保存Future存根并循环调用其返回结果的时候，主线程并不能保证首先获得的是最先完成任务的线程返回值。它只是按加入线程池的顺序返回。因为take方法是阻塞方法，后面的任务完成了，前面的任务却没有完成，主程序就那样等待在那儿，只到前面的完成了，它才知道原来后面的也完成了。

使用方法二，使用CompletionService来维护处理线程不的返回结果时，主线程总是能够拿到最先完成的任务的返回值，而不管它们加入线程池的顺序。