### 1、线程池的好处

#### 1.1 线程池的重用

线程的创建和销毁的开销是巨大的，而通过线程池的重用大大减少了这些不必要的开销，当然既然少了这么多消费内存的开销，其线程执行速度也是突飞猛进的提升。

#### 1.2 控制线程池的并发数

万万记着并发可跟并行不一样。

并发：在某个时间段内，多个程序都处在执行和执行完毕之间；但在一个时间点上只有一个程序在运行。头脑风暴：老鹰妈妈喂小雏鹰食物，小雏鹰很多，而老鹰只有一张嘴，她需要一个个喂过去，到最后每个小雏鹰都可以吃到，但是在一个时间点里只能有一个小雏鹰可以吃到美味的食物。

并行：在某个时间段里，每个程序按照自己独立异步的速度执行，程序之间互不干扰。头脑风暴：这就好似老鹰妈妈决定这样喂食太费劲于是为每个小雏鹰请了个保姆，这样子在一个时间点里，每个小雏鹰都可以同时吃到食物，而且互相不干扰。

回到线程池，控制线程池的并发数可以有效的避免大量的线程池争夺CPU资源而造成堵塞。头脑风暴：还是拿老鹰的例子来讲，妈妈只有一个，要这么一个个喂下去，一些饿坏的小雏鹰等不下去了就要破坏规则，抢在靠前喂食的雏鹰面前，而前面的雏鹰也不是吃软饭的，于是打起来了，场面混乱。老鹰生气了，这么不懂事，谁也别吃了，于是造成了最后谁也没食吃的局面。

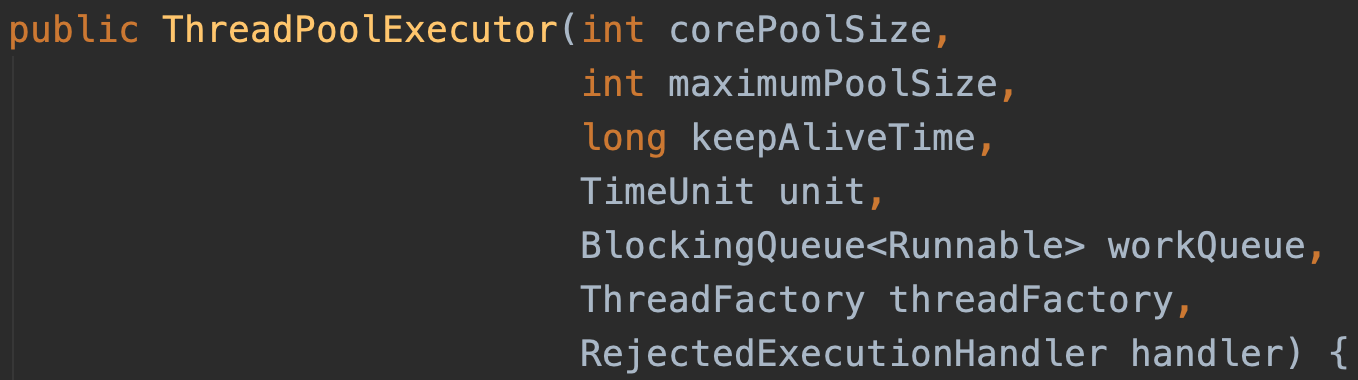
#### 1.3 线程池可以对线程进行管理

线程池可以提供定时、定期、单线程、并发数控制等功能。比如通过ScheduledThreadPool线程池来执行S秒后，每隔N秒执行一次的任务。

### 2、线程池详解

<http://blog.csdn.net/seu_calvin/article/details/52415337>

#### 2.1 ThreadPoolExecutor



这里七个参数，开发中最多的是用5个参数的构造方法：

* corePoolSize 线程池中核心线程的数量
* maximumPoolSize 线程池中最大线程数量
* keepAliveTime 非核心线程的超时时长，当系统中非核心线程闲置时间超过keepAliveTime之后，则会被回收。如果ThreadPoolExecutor的allowCoreThreadTimeOut属性设置为true，则该参数也表示核心线程的超时时长
* unit 第三个参数的单位，有纳秒、微秒、毫秒、秒、分、时、天等
* workQueue 线程池中的任务队列，该队列主要用来存储已经被提交但是尚未执行的任务。存储在这里的任务是由ThreadPoolExecutor的execute方法提交来的。
* threadFactory 为线程池提供创建新线程的功能，这个我们一般使用默认即可
* handler 拒绝策略，当线程无法执行新任务时（一般是由于线程池中的线程数量已经达到最大数或者线程池关闭导致的），默认情况下，当线程池无法处理新线程时，会抛出一个RejectedExecutionException。

maximumPoolSize(最大线程数) = corePoolSize(核心线程数) + noCorePoolSize(非核心线程数)；

（1）当currentSize<corePoolSize时，没什么好说的，直接启动一个核心线

程并执行任务。

（2）当currentSize>=corePoolSize、并且workQueue未满时，添加进来的任务会被安排到workQueue中等待执行。

（3）当workQueue已满，但是currentSize<maximumPoolSize时，会立即开启一个非核心线程来执行任务。

（4）当currentSize>=corePoolSize、workQueue已满、并且currentSize>maximumPoolSize时，调用handler默认抛出RejectExecutionExpection异常。

什么currentSize,corePoolSize,maximumPoolSize,workQueue比来比去的都比

迷糊了，哈哈，那我举个烧烤店的例子来想必大家理解起来更快。

夏天了，很热，所以很多烧烤店都会在外面也布置座位，分为室内、室外两个地方可以吃烧烤。（室内有空调电视，而且室内比室外烧烤更加优惠，而且外面下着瓢泼大雨所以顾客会首先选择室内）

corePoolSize(烧烤店室内座位)，cuurentPoolSize（目前到烧烤店的顾客数量），maximumPoolSize（烧烤店室内+室外+侯厅室所有座位），workQueue(烧烤店为顾客专门设置的侯厅室)

第（1）种，烧烤店人数不多的时候，室内位置很多，大家都其乐融融，开心的坐在室内吃着烧烤，看着世界杯。

第（2）种，生意不错，室内烧烤店坐无空席，大家都不愿意去外面吃，于是在侯厅室里呆着，侯厅室位置没坐满。

第（3）种，生意兴隆，室内、侯厅室都坐无空席，但是顾客太饿了，剩下的人没办法只好淋着雨吃烧烤，哈哈，好可怜。

第（4）种，生意爆棚，室内、室外、侯厅室都坐无空席，在有顾客过来直接赶走。

#### 2.2 其他线程池的记法

剩下的那四种主要的线程池大概思路，用法在推荐博客：<http://blog.csdn.net/u012702547/article/details/52259529>。

（1）FixedThreadPool:

Fixed中文解释为固定。结合在一起解释固定的线程池，说的更全面点就是，有固定数量线程的线程池。其corePoolSize=maximumPoolSize，且keepAliveTime为0，适合线程稳定的场所。

（2）SingleThreadPool:

Single中文解释为单一。结合在一起解释单一的线程池，说的更全面点就是，有固定数量线程的线程池，且数量为一，从数学的角度来看SingleThreadPool应该属于FixedThreadPool的子集。其corePoolSize=maximumPoolSize=1,且keepAliveTime为0，适合线程同步操作的场所。

（3）CachedThreadPool:

Cached中文解释为储存。结合在一起解释储存的线程池，说的更通俗易懂，既然要储存，其容量肯定是很大，所以他的corePoolSize=0，maximumPoolSize=Integer.MAX\_VALUE(2^32-1一个很大的数字)

（4）ScheduledThreadPool:

Scheduled中文解释为计划。结合在一起解释计划的线程池，顾名思义既然涉及到计划，必然会涉及到时间。所以ScheduledThreadPool是一个具有定时定期执行任务功能的线程池。

### 3、线程池的单例

#### 3.1 单例

单例模式（Singleton Pattern）。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

注意事项：

单例类只能有一个实例。

单例类必须自己创建自己的唯一实例。

单例类必须给所有其他对象提供这一实例。

#### 3.2 线程池的单例

为什么要将线程池设计成单例模式呢。一般情况下，整个系统中只需要单种线程池，多个线程公用一个线程池，不会是每创一个线程就要创建一个线程池，那样子你还不如不用线程池呢。

言归正传，咱们来看看如何将线程池设计成单例模式。废话少说上代码

首先在ThreadPool类里面实现线程池的创建，我们这里创建的是FixedThreadPool线程池（记住构造方法要私有，保证不被其他类实例化）

再然后对ThreadPool内部类，在类里面对他实例化，实现单例：

public class ThreadPoolManager {

private static ThreadPoolConfig *threadPool*;  
  
  
 */\*\*  
 \* 获取单例的线程池对象  
 \*/* public static ThreadPoolConfig getThreadPool() {  
 if (*threadPool* == null) {  
 synchronized (ThreadPoolManager.class) {  
 if (*threadPool* == null) {  
 // 获取处理器数量  
 int cpuNum = Runtime.*getRuntime*().availableProcessors();  
 // 根据cpu数量,计算出合理的线程并发数  
 int threadNum = cpuNum \* 2 + 1;  
 System.*out*.println("cpu num:" + cpuNum);  
 *threadPool* = new ThreadPoolConfig(threadNum, threadNum, 0L);  
 }  
 }  
 }  
 return *threadPool*;  
 }  
  
 */\*\*  
 \* 内部类  
 \*/* static class ThreadPoolConfig {  
  
 private volatile int corePoolSize;  
 private volatile int maximumPoolSize;  
 private volatile long keepAliveTime;  
  
 private ThreadPoolExecutor executor;  
  
 private ThreadPoolConfig(int corePoolSize, int maximumPoolSize, long keepAliveTime) {  
 this.corePoolSize = corePoolSize;  
 this.maximumPoolSize = maximumPoolSize;  
 this.keepAliveTime = keepAliveTime;  
 }  
  
 public void executor(Runnable runnable) {  
 if (runnable == null) {  
 return;  
 }  
 if (executor == null) {  
 executor = new ThreadPoolExecutor(corePoolSize,  
 maximumPoolSize,  
 keepAliveTime, //闲置线程存活时间  
 TimeUnit.*MILLISECONDS*,  
 new LinkedBlockingDeque<Runnable>(), //线程队列  
 Executors.*defaultThreadFactory*(), //线程工厂  
 new ThreadPoolExecutor.AbortPolicy() //队列已满，而且当前线程数已经超过最大线程数时的异常处理策略  
 );  
 }  
 executor.execute(runnable);  
 }  
 }  
}