JAVA线程池的分析和使用

## 1. 引言

合理利用线程池能够带来三个好处。第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

## 2. 线程池的使用

**线程池的创建**

我们可以通过 ThreadPoolExecutor 来创建一个线程池。

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

|  |  |
| --- | --- |
|  |  |

创建一个线程池需要输入几个参数：

* corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。
* runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。 可以选择以下几个阻塞队列。
  + ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列。
  + LinkedBlockingQueue：一个基于链表结构的阻塞队列。
  + SynchronousQueue：一个不存储元素的阻塞队列。
  + PriorityBlockingQueue：一个具有优先级的无限阻塞队列。
* maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。
* ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字(设置线程名称用以区分任务)、线程优先级、设置非守护线程相关。
* RejectedExecutionHandler（拒绝策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是 AbortPolicy，表示无法处理新任务时抛出异常。以下是 JDK1.5 提供的四种策略。
  + AbortPolicy：直接抛出异常。
  + CallerRunsPolicy：只用调用者所在线程来运行任务。
  + DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
  + DiscardPolicy：不处理，丢弃掉。
  + 正常情况需要判断任务是否必须被执行，如果需要的话，可以通过实现 RejectedExecutionHandler 接口自定义策略。如将任务持久化到磁盘，可以专门启动一个线程，等线程池的工作负载降低了，可以慢慢的从磁盘读取之前持久化的任务，重新提交。
* keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。
* TimeUnit（线程活动保持时间的单位）：

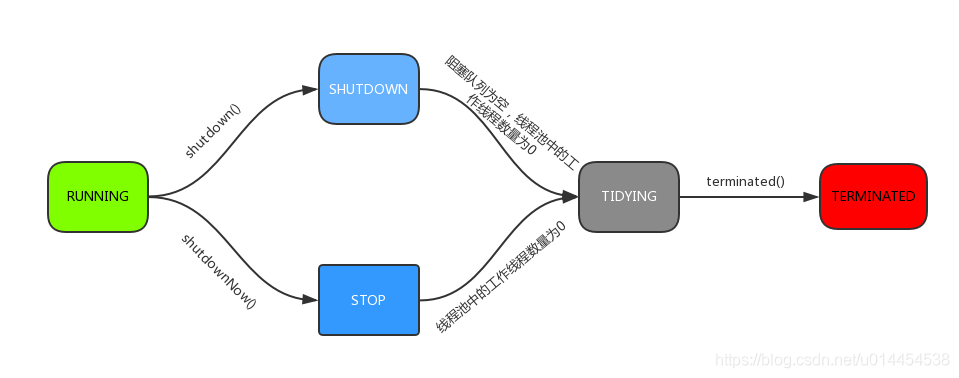
**向线程池提交任务**

我们可以使用 execute 提交的任务，但是 execute 方法没有返回值，所以无法判断任务是否被线程池执行成功以及无法捕获抛出的异常。我们也可以使用 submit 方法来提交任务，它会返回一个 future, 那么我们可以通过这个 future 来判断任务是否执行成功和异常相关信息，通过 future 的 get 方法来获取返回值，get 方法会阻塞住直到任务完成，而使用 get(long timeout, TimeUnit unit) 方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。

**线程池的关闭**

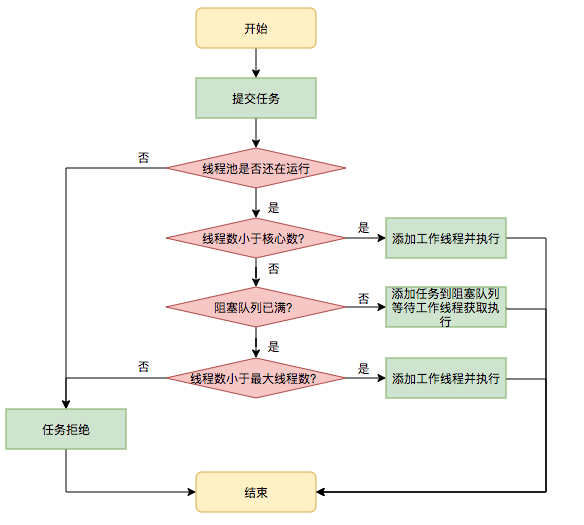
我们可以通过调用线程池的 shutdown 或 shutdownNow 方法来关闭线程池，它们的原理是遍历线程池中的工作线程，然后逐个调用线程的 interrupt 方法来中断线程，所以无法响应中断的任务可能永远无法终止。但是它们存在一定的区别，shutdownNow 首先将线程池的状态设置成 STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而 shutdown 只是将线程池的状态设置成 SHUTDOWN 状态，然后中断所有没有正在执行任务的线程。

只要调用了这两个关闭方法的其中一个，isShutdown 方法就会返回 true。当所有的任务都已关闭后, 才表示线程池关闭成功，这时调用 isTerminaed 方法会返回 true。至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用 shutdown 来关闭线程池，如果任务不一定要执行完，则可以调用 shutdownNow。



## 3. 线程池的分析

流程分析：线程池的主要工作流程如下图：



从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

1. 首先线程池判断**基本线程池**是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。
2. 其次线程池判断**工作队列**是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。
3. 最后线程池判断**整个线程池**是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

java.util.concurrent.ThreadPoolExecutor#execute

## 4. 合理的配置线程池

要想合理的配置线程池，就必须首先分析任务特性，主要分为CPU 密集型任务、IO 密集型任务，混合型任务。任务性质不同的任务可以用不同规模的线程池分开处理。

CPU 密集型任务配置尽可能小的线程，如配置 Ncpu+1 个线程的线程池。

IO 密集型任务由于可能依赖数据库连接池的任务，因为线程提交 SQL 后需要等待数据库返回结果，如果等待的时间越长 CPU 空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用 CPU，如 2\*Ncpu。

混合型任务，如果可以拆分，则将其拆分成一个 CPU 密集型任务和一个 IO 密集型任务。只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率；如果这两个任务执行时间相差太大，则没必要进行分解。

我们可以通过 Runtime.getRuntime().availableProcessors() 方法获得当前设备的 CPU 个数。

。

## 5. 线程池的监控

通过线程池提供的参数进行监控。线程池里有一些属性在监控线程池的时候可以使用

* taskCount：线程池需要执行的任务数量。
* completedTaskCount：线程池在运行过程中已完成的任务数量。小于或等于 taskCount。
* largestPoolSize：线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小，则表示线程池曾经满了。
* getPoolSize: 线程池的线程数量。如果线程池不销毁的话，池里的线程不会自动销毁，所以这个大小只增不 + getActiveCount：获取活动的线程数。

通过扩展线程池进行监控。通过继承线程池并重写线程池的 **beforeExecute**，**afterExecute** 和 terminated 方法，我们可以在任务执行前，执行后和线程池关闭前干一些事情。如监控任务的平均执行时间，最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。

## 6. 线程隔离

通常情况下会根据任务的类型来进行分组，如果一个任务的线程池资源耗尽，也不会影响另外一个任务的线程池。Hystrix 实现

https://blog.csdn.net/qq\_29777207/article/details/88829188

## 7. 面试问题

1. 远程服务异常的情况下，使用无界阻塞队列，是否会导致内存异常飙升

可以使用submit方法提交，并通过future的限时获取 异步非阻塞方式

2.如果线上机器突然宕机，线程池阻塞队列中的请求怎么办

持久化到数据库，启动一个后台线程去扫描此表，重新提交到线程池

3.线程池保持核心线程数的原理 getTask方法

阻塞队列的take方法会阻塞该线程 是否允许核心线程数超时字段