java线程池与五种常用线程池策略使用与解析

一.线程池

关于为什么要使用线程池久不赘述了，首先看一下java中作为线程池Executor底层实现类的ThredPoolExecutor的构造函数

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {

...

}

其中各个参数含义如下：

corePoolSize- 池中所保存的线程数，包括空闲线程。需要注意的是在初创建线程池时线程不会立即启动，直到有任务提交才开始启动线程并逐渐时线程数目达到corePoolSize。若想一开始就创建所有核心线程需调用prestartAllCoreThreads方法。

maximumPoolSize-池中允许的最大线程数。需要注意的是当核心线程满且阻塞队列也满时才会判断当前线程数是否小于最大线程数，并决定是否创建新线程。

keepAliveTime - 当线程数大于核心时，多于的空闲线程最多存活时间

unit - keepAliveTime 参数的时间单位。

workQueue - 当线程数目超过核心线程数时用于保存任务的队列。主要有3种类型的BlockingQueue可供选择：无界队列，有界队列和同步移交。将在下文中详细阐述。从参数中可以看到，此队列仅保存实现Runnable接口的任务。

threadFactory - 执行程序创建新线程时使用的工厂。

handler - 阻塞队列已满且线程数达到最大值时所采取的饱和策略。java默认提供了4种饱和策略的实现方式：中止、抛弃、抛弃最旧的、调用者运行。将在下文中详细阐述。

二.可选择的阻塞队列BlockingQueue详解

首先看一下新任务进入时线程池的执行策略：

如果运行的线程少于corePoolSize，则 Executor始终首选添加新的线程，而不进行排队。（如果当前运行的线程小于corePoolSize，则任务根本不会存入queue中，而是直接运行）

如果运行的线程大于等于 corePoolSize，则 Executor始终首选将请求加入队列，而不添加新的线程。

如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。

主要有3种类型的BlockingQueue：

2.1 无界队列

队列大小无限制，常用的为无界的LinkedBlockingQueue，使用该队列做为阻塞队列时要尤其当心，当任务耗时较长时可能会导致大量新任务在队列中堆积最终导致OOM。最近工作中就遇到因为采用LinkedBlockingQueue作为阻塞队列，部分任务耗时80s＋且不停有新任务进来，导致cpu和内存飙升服务器挂掉。

2.2 有界队列

常用的有两类，一类是遵循FIFO原则的队列如ArrayBlockingQueue与有界的LinkedBlockingQueue，另一类是优先级队列如PriorityBlockingQueue。PriorityBlockingQueue中的优先级由任务的Comparator决定。

使用有界队列时队列大小需和线程池大小互相配合，线程池较小有界队列较大时可减少内存消耗，降低cpu使用率和上下文切换，但是可能会限制系统吞吐量。

2.3 同步移交

如果不希望任务在队列中等待而是希望将任务直接移交给工作线程，可使用SynchronousQueue作为等待队列。SynchronousQueue不是一个真正的队列，而是一种线程之间移交的机制。要将一个元素放入SynchronousQueue中，必须有另一个线程正在等待接收这个元素。只有在使用无界线程池或者有饱和策略时才建议使用该队列。

2.4 几种BlockingQueue的具体实现原理

关于上述几种BlockingQueue的具体实现原理与分析将在下篇博文中详细阐述。

三.可选择的饱和策略RejectedExecutionHandler详解

JDK主要提供了4种饱和策略供选择。4种策略都做为静态内部类在ThreadPoolExcutor中进行实现。

3.1 AbortPolicy中止策略

该策略是默认饱和策略。

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

throw new RejectedExecutionException("Task " + r.toString() +

" rejected from " +

e.toString());

}

使用该策略时在饱和时会抛出RejectedExecutionException（继承自RuntimeException），调用者可捕获该异常自行处理。

3.2 DiscardPolicy抛弃策略

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

}

1

2

如代码所示，不做任何处理直接抛弃任务

3.3 DiscardOldestPolicy抛弃旧任务策略

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

if (!e.isShutdown()) {

e.getQueue().poll();

e.execute(r);

}

}

如代码，先将阻塞队列中的头元素出队抛弃，再尝试提交任务。如果此时阻塞队列使用PriorityBlockingQueue优先级队列，将会导致优先级最高的任务被抛弃，因此不建议将该种策略配合优先级队列使用。

3.4 CallerRunsPolicy调用者运行

public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {

if (!e.isShutdown()) {

r.run();

}

}

既不抛弃任务也不抛出异常，直接运行任务的run方法，换言之将任务回退给调用者来直接运行。使用该策略时线程池饱和后将由调用线程池的主线程自己来执行任务，因此在执行任务的这段时间里主线程无法再提交新任务，从而使线程池中工作线程有时间将正在处理的任务处理完成。

四.java提供的四种常用线程池解析

在JDK帮助文档中，有如此一段话：

强烈建议程序员使用较为方便的Executors工厂方法Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）、Executors.newFixedThreadPool(int)（固定大小线程池）Executors.newSingleThreadExecutor()（单个后台线程）它们均为大多数使用场景预定义了设置。

详细介绍一下上述四种线程池。

4.1 newCachedThreadPool

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,

60L, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>());

}

在newCachedThreadPool中如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

初看该构造函数时我有这样的疑惑：核心线程池为0，那按照前面所讲的线程池策略新任务来临时无法进入核心线程池，只能进入 SynchronousQueue中进行等待，而SynchronousQueue的大小为1，那岂不是第一个任务到达时只能等待在队列中，直到第二个任务到达发现无法进入队列才能创建第一个线程？

这个问题的答案在上面讲SynchronousQueue时其实已经给出了，要将一个元素放入SynchronousQueue中，必须有另一个线程正在等待接收这个元素。因此即便SynchronousQueue一开始为空且大小为1，第一个任务也无法放入其中，因为没有线程在等待从SynchronousQueue中取走元素。因此第一个任务到达时便会创建一个新线程执行该任务。

这里引申出一个小技巧：有时我们可能希望线程池在没有任务的情况下销毁所有的线程，既设置线程池核心大小为0，但又不想使用SynchronousQueue而是想使用有界的等待队列。显然，不进行任何特殊设置的话这样的用法会发生奇怪的行为：直到等待队列被填满才会有新线程被创建，任务才开始执行。这并不是我们希望看到的，此时可通过allowCoreThreadTimeOut使等待队列中的元素出队被调用执行，详细原理和使用将会在后续博客中阐述。

4.2 newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

public static ExecutorService newFixedThreadPool(int nThreads) {

return new ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<Runnable>());

}

看代码一目了然了，使用固定大小的线程池并使用无限大的队列

4.3 newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {

return new ScheduledThreadPoolExecutor(corePoolSize);

}

在来看看ScheduledThreadPoolExecutor（）的构造函数

public ScheduledThreadPoolExecutor(int corePoolSize) {

super(corePoolSize, Integer.MAX\_VALUE, 0, NANOSECONDS,

new DelayedWorkQueue());

}

ScheduledThreadPoolExecutor的父类即ThreadPoolExecutor，因此这里各参数含义和上面一样。值得关心的是DelayedWorkQueue这个阻塞对列，在上面没有介绍，它作为静态内部类就在ScheduledThreadPoolExecutor中进行了实现。具体分析讲会在后续博客中给出，在这里只进行简单说明：DelayedWorkQueue是一个无界队列，它能按一定的顺序对工作队列中的元素进行排列。因此这里设置的最大线程数 Integer.MAX\_VALUE没有任何意义。关于ScheduledThreadPoolExecutor的具体使用将会在后续quartz的周期性任务实现原理中进行进一步分析。

4.4 newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

public static ScheduledExecutorService newSingleThreadScheduledExecutor() {

return new DelegatedScheduledExecutorService

(new ScheduledThreadPoolExecutor(1));

}

首先new了一个线程数目为1的ScheduledThreadPoolExecutor，再把该对象传入DelegatedScheduledExecutorService中，看看DelegatedScheduledExecutorService的实现代码：

DelegatedScheduledExecutorService(ScheduledExecutorService executor) {

super(executor);

e = executor;

}

在看看它的父类

DelegatedExecutorService(ExecutorService executor) { e = executor; }

1

其实就是使用装饰模式增强了ScheduledExecutorService（1）的功能，不仅确保只有一个线程顺序执行任务，也保证线程意外终止后会重新创建一个线程继续执行任务。具体实现原理会在后续博客中讲解。

4.5 newWorkStealingPool创建一个拥有多个任务队列（以便减少连接数）的线程池。

这是jdk1.8中新增加的一种线程池实现，先看一下它的无参实现

public static ExecutorService newWorkStealingPool() {

return new ForkJoinPool

(Runtime.getRuntime().availableProcessors(),

ForkJoinPool.defaultForkJoinWorkerThreadFactory,

null, true);

}

返回的ForkJoinPool从jdk1.7开始引进，个人感觉类似于mapreduce的思想。这个线程池较为特殊，将在后续博客中给出详细的使用说明和原理。

---------------------

作者：Melody4Java

来源：CSDN

原文：https://blog.csdn.net/u011479540/article/details/51867886

版权声明：本文为博主原创文章，转载请附上博文链接！