线程的优雅关闭实践

平时开发中，大家更多地关注的是线程池的创建、任务的提交和执行。往往会忽略线程池的关闭，甚至忘记调用shutdown()方法，导致内存溢出。大多只知道需要调用shutdown()关闭线程池，却少研究其真正的关闭过程。

首先看源码中的一句注释：

A pool that is no longer referenced in a program and has no remaining threads will be shutdown automatically.  
如果程序中不再持有线程池的引用，并且线程池中没有线程时，线程池将会自动关闭。

线程池自动关闭有两个条件：

1. 线程池的引用不可达；
2. 线程池中没有线程；

这里对于条件2解释一下，线程池中没有线程是指线程池中的所有线程都已运行完并自动消亡。然而我们常用的FixedThreadPool的核心线程没有超时策略，所以并不会自动关闭。

展示两种不同线程池不关闭的情况：

1. FixedThreadPool示例

public static void main(String[] args) {

while(true) {

ExecutorService executorService = Executors.newFixedThreadPool(8);

executorService.execute(() -> System.out.println("running"));

executorService = null;

}

}

输出结果：

running

......

running

Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread

at java.lang.Thread.start0(Native Method)

at java.lang.Thread.start(Thread.java:714)

at java.util.concurrent.ThreadPoolExecutor.addWorker(ThreadPoolExecutor.java:950)

at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1357)

at test.PoolTest.main(PoolTest.java:29)

因为FixedThreadPool的核心线程不会自动超时关闭，使用时必须在适当的时候调用shutdown()方法。

1. CachedThreadPool示例

public static void main(String[] args) {

while(true) {

// 默认keepAliveTime为 60s

ExecutorService executorService = Executors.newCachedThreadPool();

ThreadPoolExecutor threadPoolExecutor = (ThreadPoolExecutor) executorService;

// 为了更好的模拟，动态修改为1纳秒

threadPoolExecutor.setKeepAliveTime(1, TimeUnit.NANOSECONDS);

threadPoolExecutor.execute(() -> System.out.println("running"));

}

}

输出结果：

running

running

running

running

running

......

CachedThreadPool的线程keepAliveTime默认为60s，核心线程数量为0，所以不会有核心线程存活阻止线程池自动关闭。详见《线程池之ThreadPoolExecutor构造》，为了更快的模拟，构造后将keepAliveTime修改为1纳秒，相当于线程执行完马上会消亡，所以线程池可以被回收。实际开发中，如果CachedThreadPool确实忘记关闭，在一定时间后是可以被回收的。但仍然建议显式关闭。

然而，线程池关闭的意义不仅仅在于结束线程执行、避免内存溢出，因为大多使用的场景并非上述示例那样朝生夕死。线程池一般是持续工作的全局场景，如数据库连接池。

本文更多要讨论的是当线程池调用shutdown方法后，会经历些什么？思考以下几个问题：

1. 是否可以继续接受新任务？继续提交新任务会怎样？
2. 等待队列里的任务是否还会执行？
3. 正在执行的任务是否会立即中断？

是否可以继续接受新任务？继续提交新任务会怎样？

public static void main(String[] args) {

ThreadPoolExecutor executor = new ThreadPoolExecutor(4, 4, 10, TimeUnit.SECONDS, new LinkedBlockingQueue<>());

executor.execute(() -> System.out.println("before shutdown"));

executor.shutdown();

executor.execute(() -> System.out.println("after shutdown"));

}

输出结果如下：

before shutdown

Exception in thread "main" java.util.concurrent.RejectedExecutionException: Task PoolTest$$Lambda$2/142257191@3e3abc88 rejected from java.util.concurrent.ThreadPoolExecutor@6ce253f1[Terminated, pool size = 0, active threads = 0, queued tasks = 0, completed tasks = 1]

at java.util.concurrent.ThreadPoolExecutor$AbortPolicy.rejectedExecution(ThreadPoolExecutor.java:2047)

at java.util.concurrent.ThreadPoolExecutor.reject(ThreadPoolExecutor.java:823)

at java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1369)

at PoolTest.main(PoolTest.java:12)

当线程池关闭后，继续提交新任务会抛出异常。这句话也不够准确，不一定是抛出异常，而是执行拒绝策略，默认的拒绝策略是抛出异常。可参见《线程池之ThreadPoolExecutor构造》里面自定义线程池的例子，自定义了忽略策略，但被拒绝时并没有抛出异常。

等待队列里的任务是否还会执行？

public class WaitqueueTest {

public static void main(String[] args) {

BlockingQueue<Runnable> workQueue = new LinkedBlockingQueue<>();

for(int i = 1; i <= 100 ; i++){ workQueue.add(new Task(String.valueOf(i))); }

ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 10, TimeUnit.SECONDS, workQueue);

executor.execute(new Task("0"));

executor.shutdown();

System.out.println("workQueue size = " + workQueue.size() + " after shutdown");

}

static class Task implements Runnable{

String name;

public Task(String name) { this.name = name; }

@Override

public void run() {

for(int i = 1; i <= 10; i++){ System.out.println("task " + name + " is running"); }

System.out.println("task " + name + " is over");

}

}

}

这个demo解释一下，我们用LinkedBlockingQueue构造了一个线程池，在线程池启动前，我们先将工作队列填充100个任务，然后执行task0后立即shutdown()线程池，来验证线程池关闭队列的任务运行状态。

输出结果如下：

......

task 0 is running

task 0 is over

workQueue size = 100 after shutdown //表示线程池关闭后，队列任然有100个任务

task 1 is running

......

task 100 is running

task 100 is over

从结果中我们可以看到，线程池虽然关闭，但是队列中的任务仍然继续执行，所以用shutdown()方式关闭线程池时需要考虑是否是你想要的效果。

如果你希望线程池中的等待队列中的任务不继续执行，可以使用shutdownNow()方法，将上述代码进行调整，如下：

public class WaitqueueTest {

public static void main(String[] args) {

BlockingQueue<Runnable> workQueue = new LinkedBlockingQueue<>();

for(int i = 1; i <= 100 ; i++){ workQueue.add(new Task(String.valueOf(i))); }

ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 10, TimeUnit.SECONDS, workQueue);

executor.execute(new Task("0"));

List<Runnable> dropList = executor.shutdownNow(); //shutdownNow有返回值，返回被抛弃的任务list

System.out.println("workQueue size = " + workQueue.size() + " after shutdown");

System.out.println("dropList size = " + dropList.size());

}

static class Task implements Runnable{

String name;

public Task(String name) { this.name = name; }

@Override

public void run() {

for(int i = 1; i <= 10; i++){ System.out.println("task " + name + " is running"); }

System.out.println("task " + name + " is over");

}

}

}

输出结果如下：

task 0 is running

workQueue size = 0 after shutdown

task 0 is running

task 0 is running

task 0 is running

task 0 is running

task 0 is running

task 0 is running

task 0 is running

task 0 is running

task 0 is running

dropList size = 100

task 0 is over

从上述输出可以看到，只有任务0执行完毕，其他任务都被drop掉了，dropList的size为100。通过dropList我们可以对未处理的任务进行进一步的处理，如log记录，转发等；

正在执行的任务是否会立即中断？

要验证这个问题，需要对线程的interrupt方法有一定了解。推荐阅读《线程中断机制》

关于interrupt方法：

首先，一个线程不应该由其他线程来强制中断或停止，而是应该由线程自己自行停止。所以，Thread.stop、Thread.suspend、Thread.resume都已经被废弃了。

Thread.interrupt的作用其实也不是中断线程，而是“通知线程应该中断了”，具体到底中断还是继续运行，应该由被通知的线程自己处理。具体来说，当一个线程调用interrupt()时：

1. 如果线程处于被阻塞状态（例如处于sleep、wait、join等状态），那么线程将立即退出被阻塞状态，并抛出一个InterruptedException异常。仅此而已。
2. 如果线程处于正常活动状态，那么会将该线程的中断标志设置为true，仅此而已。被设置中断标志的线程将继续正常运行，不受影响。interrupt()并不能真正的中断线程，需要被调用的线程自己进行配合才行。也就是说，一个线程如果有被中断的需求，那么就可以这样做。：
3. 在正常运行任务时，经常检查本线程的中断标志位，如果被设置了中断标志就自行停止线程。
4. 在调用阻塞方法时正确处理InterruptedException异常。（例如，catch异常后就结束线程）

public class InteruptTest {

public static void main(String[] args) throws InterruptedException {

ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 1, 10, TimeUnit.SECONDS, new LinkedBlockingQueue<>());

executor.execute(new Task("0"));

Thread.sleep(1);

executor.shutdown();

System.out.println("executor has been shutdown");

}

static class Task implements Runnable {

String name;

public Task(String name) { this.name = name; }

@Override

public void run() {

for (int i = 1; i <= 100 && !Thread.interrupted(); i++) {

Thread.yield();

System.out.println("task " + name + " is running, round " + i);

}

}

}

}

输出结果如下：

task 0 is running, round 1

task 0 is running, round 2

task 0 is running, round 3

......

task 0 is running, round 28

executor has been shutdown

......

task 0 is running, round 99

task 0 is running, round 100

为了体现在任务执行中打断，在主线程进行短暂sleep，task中调用Thread.yield()，出让时间片。从结果中可以看到，线程池被关闭后，正在运行的任务没有被interrupt。说明shutdown()方法不会interrupt运行中线程。再将其修改为shutdownNow()后输出结果如下：

task 0 is running, round 1

task 0 is running, round 2

......

task 0 is running, round 56

task 0 is running, round 57

task 0 is running, round 58

task 0 is running, round 59

executor has been shutdown

修改为shutdownNow()后，task任务没有执行完，执行到中间的时候就被interrupt后没有继续执行了。

总结，想要正确的关闭线程池，并不是简单的调用shutdown方法那么简单，要考虑到应用场景的需求，如果拒绝新来的请求任务？如果处理等待队列中的任务？如果处理正在执行的任务？想好这几个问题，再确定如何优雅而正确地关闭线程池。

PS：线程被interrupt后，需要在run方法中单独处理interrupted状态，interrupt更类似一个标志位，不会直接打断线程的执行。