# **四种线程池拒绝策略**



[像疯一样的女行子](https://blog.csdn.net/suifeng629" \t "https://blog.csdn.net/suifeng629/article/details/_blank) 2019-08-08 20:50:38 IMG_257 51719 IMG_258 收藏 74

文章标签： [ThreadPoolExector](https://so.csdn.net/so/search/s.do?q=ThreadPoolExector&t=blog&o=vip&s=&l=&f=&viparticle=" \t "https://blog.csdn.net/suifeng629/article/details/_blank) [abortpolicy](https://so.csdn.net/so/search/s.do?q=abortpolicy&t=blog&o=vip&s=&l=&f=&viparticle=" \t "https://blog.csdn.net/suifeng629/article/details/_blank) [discardpolicy](https://so.csdn.net/so/search/s.do?q=discardpolicy&t=blog&o=vip&s=&l=&f=&viparticle=" \t "https://blog.csdn.net/suifeng629/article/details/_blank) [discardoldestpolicy](https://so.csdn.net/so/search/s.do?q=discardoldestpolicy&t=blog&o=vip&s=&l=&f=&viparticle=" \t "https://blog.csdn.net/suifeng629/article/details/_blank) [callerrunspolicy](https://so.csdn.net/so/search/s.do?q=callerrunspolicy&t=blog&o=vip&s=&l=&f=&viparticle=" \t "https://blog.csdn.net/suifeng629/article/details/_blank)

版权

## **一、前言**

线程池，相信很多人都有用过，没用过相信的也有学习过。但是，线程池的拒绝策略，相信知道的人会少许多。

## **二、四种线程池拒绝策略**

当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize时，如果还有任务到来就会采取任务拒绝策略，通常有以下四种策略：  
ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。 ThreadPoolExecutor.DiscardPolicy：丢弃任务，但是不抛出异常。 ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新提交被拒绝的任务 ThreadPoolExecutor.CallerRunsPolicy：由调用线程（提交任务的线程）处理该任务

## **三、线程池默认的拒绝策略**

既然有四种拒绝策略可以选择，那么线程池的默认拒绝策略是什么呢？查看

java.util.concurrent.ThreadPoolExecutor类的源码，我们可以看到：

/\*\*

\* The default rejected execution handler

\*/

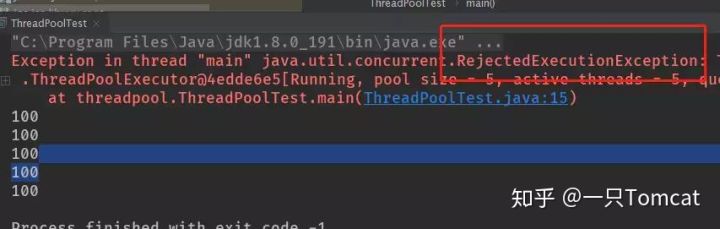
private static final RejectedExecutionHandler defaultHandler =

new AbortPolicy();

线程池的默认拒绝策略为AbortPolicy，即丢弃任务并抛出RejectedExecutionException异常。我们可以通过代码来验证这一点，现有如下代码：

public class ****ThreadPoolTest**** {  
public static void main(String[] args) {  
BlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(100);  
ThreadFactory factory = r -> new Thread(r, "test-thread-pool");  
ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 5,  
0L, TimeUnit.SECONDS, queue, factory);  
while (true) {  
executor.submit(() -> {  
try {  
System.out.println(queue.size());  
Thread.sleep(10000);  
} catch (InterruptedException e) {  
e.printStackTrace();  
}  
});  
}  
}  
}

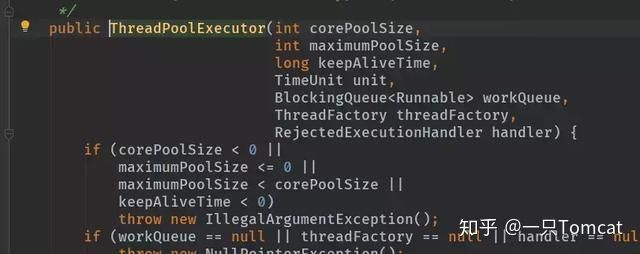
这里是一个默认的线程池，没有设置拒绝策略，设置了最大线程队列是100。运行代码：



结果是符合预期的，这也证明了线程池的默认拒绝策略是ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

## **四、设置线程池拒绝策略**

如果我们想要根据实际业务场景需要，设置其他的线程池拒绝策略，可以通过ThreadPoolExecutor重载的构造方法进行设置：



现在的开发中，相信大家都有使用spring，其实我们也可以通过spring提供的org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor构建线程池。如下：



@Configuration  
public class ****TaskExecutorConfig**** implements ****AsyncConfigurer**** {  
*/\*\*  
\* Set the ThreadPoolExecutor's core pool size.  
\*/*  
private static final int CORE\_POOL\_SIZE = 5;  
*/\*\*  
\* Set the ThreadPoolExecutor's maximum pool size.  
\*/*  
private static final int MAX\_POOL\_SIZE = 5;  
*/\*\*  
\* Set the capacity for the ThreadPoolExecutor's BlockingQueue.  
\*/*  
private static final int QUEUE\_CAPACITY = 1000;  
*/\*\*  
\* 通过重写getAsyncExecutor方法，制定默认的任务执行由该方法产生  
\* <p>  
\* 配置类实现AsyncConfigurer接口并重写getAsyncExcutor方法，并返回一个ThreadPoolTaskExevutor  
\* 这样我们就获得了一个基于线程池的TaskExecutor  
\*/*  
@Override  
public Executor getAsyncExecutor() {  
ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor();  
taskExecutor.setCorePoolSize(CORE\_POOL\_SIZE);  
taskExecutor.setMaxPoolSize(MAX\_POOL\_SIZE);  
taskExecutor.setQueueCapacity(QUEUE\_CAPACITY);  
taskExecutor.initialize();  
taskExecutor.setRejectedExecutionHandler(new ThreadPoolExecutor.AbortPolicy());  
return taskExecutor;  
}  
}

## **五、拒绝策略场景分析**

****（1）AbortPolicy****  
AbortPolicy

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

A handler for rejected tasks that throws a {@code RejectedExecutionException}.

这是线程池默认的拒绝策略，在任务不能再提交的时候，抛出异常，及时反馈程序运行状态。如果是比较关键的业务，推荐使用此拒绝策略，这样子在系统不能承载更大的并发量的时候，能够及时的通过异常发现。

****（2）DiscardPolicy****

ThreadPoolExecutor.DiscardPolicy：丢弃任务，但是不抛出异常。如果线程队列已满，则后续提交的任务都会被丢弃，且是静默丢弃。

A handler for rejected tasks that silently discards therejected task.

使用此策略，可能会使我们无法发现系统的异常状态。建议是一些无关紧要的业务采用此策略。例如，本人的博客网站统计阅读量就是采用的这种拒绝策略。

****（3）DiscardOldestPolicy****

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新提交被拒绝的任务。

A handler for rejected tasks that discards the oldest unhandled request and then retries {@code execute}, unless the executor is shut down, in which case the task is discarded.

此拒绝策略，是一种喜新厌旧的拒绝策略。是否要采用此种拒绝策略，还得根据实际业务是否允许丢弃老任务来认真衡量。

****（4）CallerRunsPolicy****

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

A handler for rejected tasks that runs the rejected task directly in the calling thread of the {@code execute} method, unless the executor has been shut down, in which case the task is discarded.

如果任务被拒绝了，则由调用线程（提交任务的线程）直接执行此任务，我们可以通过代码来验证这一点：

把之前的代码修改如下：

public static void main(String[] args) {

BlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(10);

ThreadFactory factory = r -> new Thread(r, "test-thread-pool");

ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 5,

0L, TimeUnit.SECONDS, queue, factory, new ThreadPoolExecutor.CallerRunsPolicy());

for (int i = 0; i < 1000; i++) {

executor.submit(() -> {

try {

System.out.println(Thread.currentThread().getName() + ":执行任务");

Thread.sleep(1000);

} catch (InterruptedException e) {

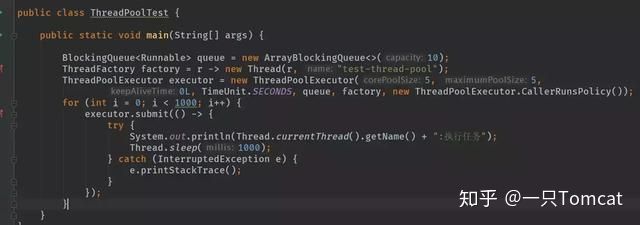
e.printStackTrace();

}

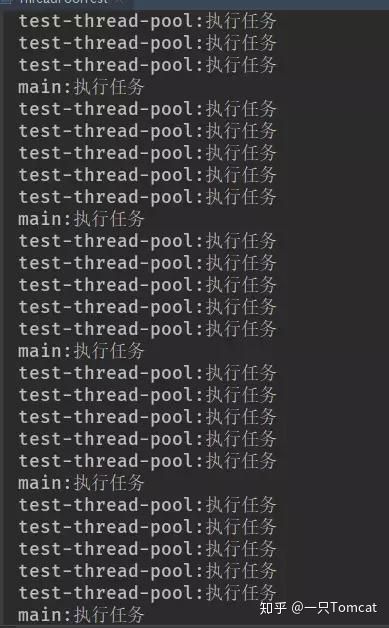
});

}

}



把队列最大值改为10，打印输出线程的名称。执行结果如下：



通过结果可以看到，主线程main也执行了任务，这正说明了此拒绝策略由调用线程（提交任务的线程）直接执行被丢弃的任务的。

## **六、总结**

本文介绍和演示了四种线程池拒绝策略，具体使用哪种策略，还得根据实际业务场景才能做出抉择。