

线程池最佳实践

这篇文章篇幅虽短，但是绝对是干货。标题稍微有点夸张，嘿嘿，实际都是自己使用线程池的时候总结的一些个人感觉比较重要的点。

线程池知识回顾

开始这篇文章之前还是简单介绍一嘴线程池，之前写的[《新手也能看懂的线程池学习总结》](#)这篇文章介绍的很详细了。

为什么要使用线程池？

池化技术相比大家已经屡见不鲜了，线程池、数据库连接池、Http 连接池等等都是对这个思想的应用。池化技术的思想主要是为了减少每次获取资源的消耗，提高对资源的利用率。

线程池提供了一种限制和管理资源（包括执行一个任务）。每个**线程池**还维护一些基本统计信息，例如已完成任务的数量。

这里借用《Java 并发编程的艺术》提到的来说一下**使用线程池的好处**：

- **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。
- **提高响应速度**。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- **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。

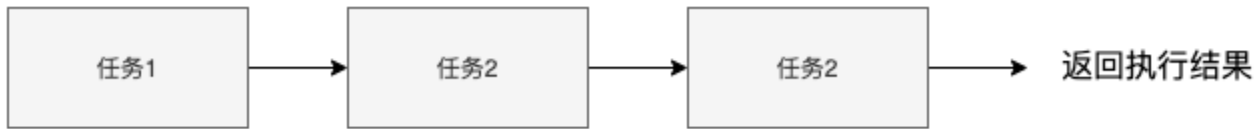
线程池在实际项目的使用场景

线程池一般用于执行多个不相关联的耗时任务，没有多线程的情况下，任务顺序执行，使用了线程池的话可让多个不相关联的任务同时执行。

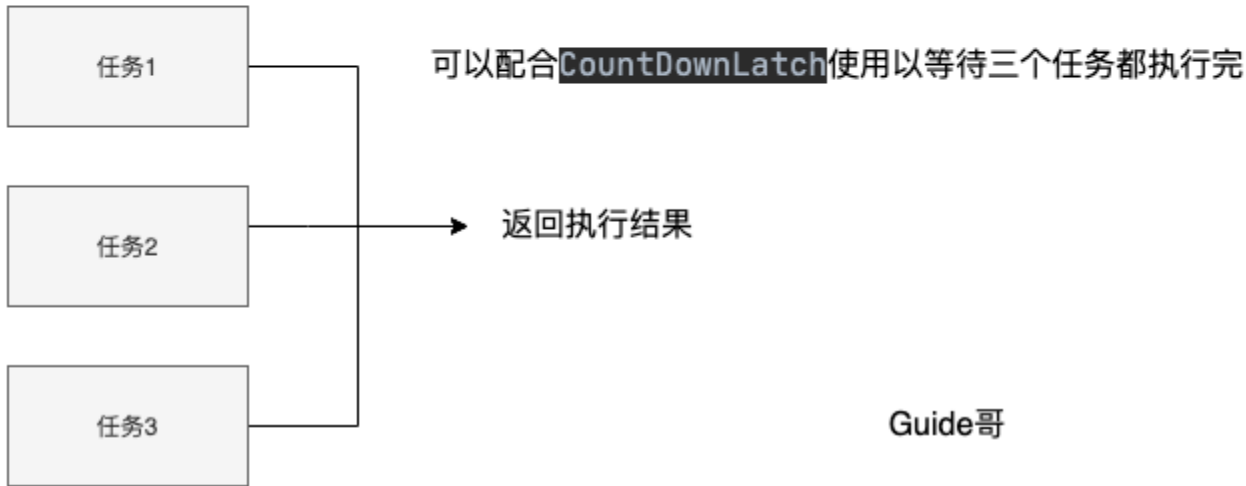
假设我们要执行三个不相关的耗时任务，Guide 画图给大家展示了使用线程池前后的区别。

注意：下面三个任务可能做的是同一件事情，也可能是不一样的事情。

使用多线程前



使用多线程后



如何使用线程池？

一般是通过 `ThreadPoolExecutor` 的构造函数来创建线程池，然后提交任务给线程池执行就可以了。

`ThreadPoolExecutor`构造函数如下：

```
/**
 * 用给定的初始参数创建一个新的ThreadPoolExecutor。
 */
public ThreadPoolExecutor(int corePoolSize, //线程池的核心线程数量
                          int maximumPoolSize, //线程池的最大线程数
                          long keepAliveTime, //当线程数大于核心线程数时，多余的空闲线程存活的最长时间
                          TimeUnit unit, //时间单位
                          BlockingQueue<Runnable> workQueue, //任务队列，用来储存等待执行任务的队列
                          ThreadFactory threadFactory, //线程工厂，用来创建线程，一般默认即可
                          RejectedExecutionHandler handler //拒绝策略，当提交的任务过多而不能及时处理时，我们可以定制策略来处理任务
                          ) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
```

```
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```

简单演示一下如何使用线程池，更详细的介绍，请看：[《新手也能看懂的线程池学习总结》](#)。

```
private static final int CORE_POOL_SIZE = 5;
private static final int MAX_POOL_SIZE = 10;
private static final int QUEUE_CAPACITY = 100;
private static final Long KEEP_ALIVE_TIME = 1L;

public static void main(String[] args) {

    //使用阿里巴巴推荐的创建线程池的方式
    //通过ThreadPoolExecutor构造函数自定义参数创建
    ThreadPoolExecutor executor = new ThreadPoolExecutor(
        CORE_POOL_SIZE,
        MAX_POOL_SIZE,
        KEEP_ALIVE_TIME,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<>(QUEUE_CAPACITY),
        new ThreadPoolExecutor.CallerRunsPolicy());

    for (int i = 0; i < 10; i++) {
        executor.execute(() -> {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("CurrentThread name:" +
                Thread.currentThread().getName() + "date: " + Instant.now());
        });
    }
    //终止线程池
    executor.shutdown();
    try {
        executor.awaitTermination(5, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Finished all threads");
}
```

控制台输出：

```
CurrentThread name:pool-1-thread-5date: 2020-06-06T11:45:31.639Z
CurrentThread name:pool-1-thread-3date: 2020-06-06T11:45:31.639Z
CurrentThread name:pool-1-thread-1date: 2020-06-06T11:45:31.636Z
CurrentThread name:pool-1-thread-4date: 2020-06-06T11:45:31.639Z
CurrentThread name:pool-1-thread-2date: 2020-06-06T11:45:31.639Z
CurrentThread name:pool-1-thread-2date: 2020-06-06T11:45:33.656Z
CurrentThread name:pool-1-thread-4date: 2020-06-06T11:45:33.656Z
CurrentThread name:pool-1-thread-1date: 2020-06-06T11:45:33.656Z
CurrentThread name:pool-1-thread-3date: 2020-06-06T11:45:33.656Z
CurrentThread name:pool-1-thread-5date: 2020-06-06T11:45:33.656Z
Finished all threads
```

线程池最佳实践

简单总结一下我了解的使用线程池的时候应该注意的东西，网上似乎还没有专门写这方面的文章。

因为Guide还比较菜，有补充和完善的地方，可以在评论区告知或者在微信上与我交流。

1. 使用 `ThreadPoolExecutor` 的构造函数声明线程池

1. 线程池必须手动通过 `ThreadPoolExecutor` 的构造函数来声明，避免使用 `Executors` 类的 `newFixedThreadPool` 和 `newCachedThreadPool`，因为可能会有 OOM 的风险。

`Executors` 返回线程池对象的弊端如下：

- `FixedThreadPool` 和 `SingleThreadExecutor`：允许请求的队列长度为 `Integer.MAX_VALUE`，可能堆积大量的请求，从而导致 OOM。
- `CachedThreadPool` 和 `ScheduledThreadPool`：允许创建的线程数量为 `Integer.MAX_VALUE`，可能会创建大量线程，从而导致 OOM。

说白了就是：**使用有界队列，控制线程创建数量。**

除了避免 OOM 的原因之外，不推荐使用 `Executors` 提供的两种快捷的线程池的原因还有：

1. 实际使用中需要根据自己机器的性能、业务场景来手动配置线程池的参数比如核心线程数、使用的任务队列、饱和策略等等。
2. 我们应该显示地给我们的线程池命名，这样有助于我们定位问题。

2. 监测线程池运行状态

你可以通过一些手段来检测线程池的运行状态比如 SpringBoot 中的 Actuator 组件。

除此之外，我们还可以利用 `ThreadPoolExecutor` 的相关 API 做一个简陋的监控。从下图可以看出，`ThreadPoolExecutor` 提供了获取线程池当前的线程数和活跃线程数、已经执行完成的任务数、正在排队中的任务数等等。

```

m  getActiveCount(): int
m  getCompletedTaskCount(): long
m  getCorePoolSize(): int
m  getKeepAliveTime(TimeUnit): long
m  getLargestPoolSize(): int
m  getMaximumPoolSize(): int
m  getPoolSize(): int
m  getQueue(): BlockingQueue<Runnable>
m  getRejectedExecutionHandler(): RejectedExecutionHandler
m  getTask(): Runnable
m  getTaskCount(): long
m  getThreadFactory(): ThreadFactory

```

下面是一个简单的 Demo。`printThreadPoolStatus()` 会每隔一秒打印出线程池的线程数、活跃线程数、完成的任务数、以及队列中的任务数。

```

/**
 * 打印线程池的状态
 *
 * @param threadPool 线程池对象
 */
public static void printThreadPoolStatus(ThreadPoolExecutor threadPool) {
    ScheduledExecutorService scheduledExecutorService = new
    ScheduledThreadPoolExecutor(1, createThreadFactory("print-images/thread-pool-
    status", false));
    scheduledExecutorService.scheduleAtFixedRate(() -> {
        log.info("=====");
        log.info("ThreadPool Size: {}", threadPool.getPoolSize());
        log.info("Active Threads: {}", threadPool.getActiveCount());
        log.info("Number of Tasks : {}", threadPool.getCompletedTaskCount());
        log.info("Number of Tasks in Queue: {}",
        threadPool.getQueue().size());
        log.info("=====");
    }, 0, 1, TimeUnit.SECONDS);
}

```

3. 建议不同类别的业务用不同的线程池

很多人在实际项目中都会有类似这样的问题：**我的项目中多个业务需要用到线程池，是为每个线程池都定义一个还是说定义一个公共的线程池呢？**

一般建议是不同的业务使用不同的线程池，配置线程池的时候根据当前业务的情况对当前线程池进行配置，因为不同的业务的并发以及对资源的使用情况都不同，重心优化系统性能瓶颈相关的业务。

我们再来看一个真实的故事案例！（本案例来源自：[《线程池运用不当的一次线上事故》](#)，很精彩的一个案例）

```
/**
 * 执行扣费任务
 */
public Result<Integer> executeDeduct(ChargeInputDTO chargeInput) {
    ChargeTask chargeTask = new ChargeTask(chargeInput);
    bizThreadPool.execute(() -> chargeTaskBll.execute(chargeTask ));
    return Result.success();
}

/**
 * 扣费任务的具体业务逻辑
 */
public class ChargeTaskBll implements Runnable {

    public void execute(ChargeTask chargeTask) {
        // 第一步：参数校验
        verifyInputParam(chargeTask);
        // 第二步：执行反作弊子任务
        executeUserSpam(SpamHelper.userConfigs);
        // 第三步：执行扣费
        handlePay(chargeTask);
        // 其他步骤：点击埋点等
        ...
    }
}

/**
 * 执行反作弊子任务
 */
public void executeUserSpam(List<SpamUserConfigDO> configs) {
    if (CollectionUtils.isEmpty(configs)) {
        return;
    }

    try {
        CountDownLatch latch = new CountDownLatch(configs.size());
        for (SpamUserConfigDO config : configs) {
            UserSpamTask task = new UserSpamTask(config, latch);
            bizThreadPool.execute(task);
        }
        latch.await();
    } catch (Exception ex) {
        logger.error("", ex);
    }
}
```

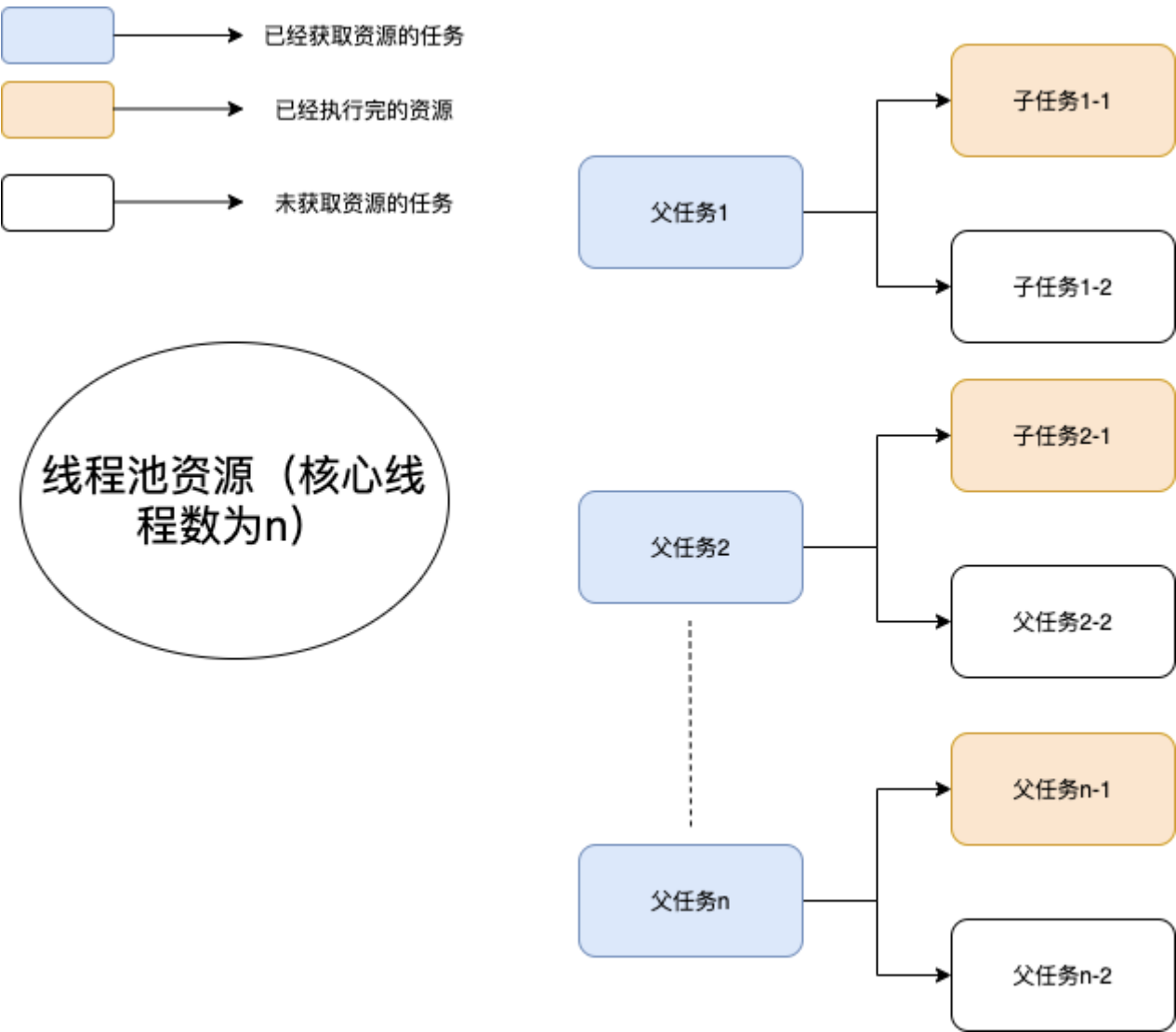


父任务和子任务使用了同一个线程池

上面的代码可能会存在死锁的情况，为什么呢？画个图给大家捋一捋。

试想这样一种极端情况：

假如我们线程池的核心线程数为 n ，父任务（扣费任务）数量为 n ，父任务下面有两个子任务（扣费任务下的子任务），其中一个已经执行完成，另外一个被放在了任务队列中。由于父任务把线程池核心线程资源用完，所以子任务因为无法获取到线程资源无法正常执行，一直被阻塞在队列中。父任务等待子任务执行完成，而子任务等待父任务释放线程池资源，这也就造成了“死锁”。



解决方法也很简单，就是新增加一个用于执行子任务的线程池专门为其服务。

4.别忘记给线程池命名

初始化线程池的时候需要显示命名（设置线程池名称前缀），有利于定位问题。

默认情况下创建的线程名字类似 pool-1-thread-n 这样的，没有业务含义，不利于我们定位问题。

给线程池里的线程命名通常有下面两种方式：

****1.利用 guava 的 ThreadFactoryBuilder ****

```
ThreadFactory threadFactory = new ThreadFactoryBuilder()
    .setNameFormat(threadNamePrefix + "-%d")
    .setDaemon(true).build();
```

```
ExecutorService threadPool = new ThreadPoolExecutor(corePoolSize, maximumPoolSize,
keepAliveTime, TimeUnit.MINUTES, workQueue, threadFactory)
```

2.自己实现 ThreadFactory。

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.atomic.AtomicInteger;
/**
 * 线程工厂，它设置线程名称，有利于我们定位问题。
 */
public final class NamingThreadFactory implements ThreadFactory {

    private final AtomicInteger threadNum = new AtomicInteger();
    private final ThreadFactory delegate;
    private final String name;

    /**
     * 创建一个带名字的线程池生产工厂
     */
    public NamingThreadFactory(ThreadFactory delegate, String name) {
        this.delegate = delegate;
        this.name = name; // TODO consider uniquifying this
    }

    @Override
    public Thread newThread(Runnable r) {
        Thread t = delegate.newThread(r);
        t.setName(name + " [# " + threadNum.incrementAndGet() + "]");
        return t;
    }
}
```

5.正确配置线程池参数

说到如何给线程池配置参数，美团的骚操作至今让我难忘（后面会提到）！

我们先来看一下各种书籍和博客上一般推荐的配置线程池参数的方式，可以作为参考！

常规操作

很多人甚至可能都会觉得把线程池配置过大一点比较好！我觉得这明显是有问题的。就拿我们生活中非常常见的一例子来说：**并不是人多就能把事情做好，增加了沟通交流成本。你本来一件事情只需要 3 个人做，你硬是拉来了 6 个人，会提升做事效率嘛？我想并不会。** 线程数量过多的影响也是和我们分配多少人做事情一样，对于多线程这个场景来说主要是增加了上下文切换成本。不清楚什么是上下文切换的话，可以看我下面的介绍。

上下文切换：

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。**任务从保存到再加载的过程就是一次上下文切换。**

上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

类比于实现世界中的人类通过合作做某件事情，我们可以肯定的一点是线程池大小设置过大或者过小都会有问题，合适的才是最好。

如果我们设置的线程池数量太小的话，如果同一时间有大量任务/请求需要处理，可能会导致大量的请求/任务在任务队列中排队等待执行，甚至会出现任务队列满了之后任务/请求无法处理的情况，或者大量任务堆积在任务队列导致 OOM。这样很明显是有问题的！CPU 根本没有得到充分利用。

但是，如果我们设置线程数量太大，大量线程可能会同时在争取 CPU 资源，这样会导致大量的上下文切换，从而增加线程的执行时间，影响了整体执行效率。

有一个简单并且适用面比较广的公式：

- **CPU 密集型任务($N+1$)**：这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N （CPU 核心数）+1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停，CPU 就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。
- **I/O 密集型任务($2N$)**：这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交出给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法是 $2N$ 。

如何判断是 CPU 密集任务还是 IO 密集任务？

CPU 密集型简单理解就是利用 CPU 计算能力的任务比如你在内存中对大量数据进行排序。但凡涉及到网络读取，文件读取这类都是 IO 密集型，这类任务的特点是 CPU 计算耗费时间相比于等待 IO 操作完成的时间来说很少，大部分时间都花在了等待 IO 操作完成上。

美团的骚操作

美团技术团队在《[Java 线程池实现原理及其在美团业务中的实践](#)》这篇文章中介绍到对线程池参数实现可自定义配置的思路和方法。

美团技术团队的思路是主要对线程池的核心参数实现自定义可配置。这三个核心参数是：








- **corePoolSize**：核心线程数线程数定义了最小可以同时运行的线程数量。
- **maximumPoolSize**：当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。

- **workQueue**: 当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，信任就会被存放在队列中。

为什么是这三个参数？

我在这篇《[新手也能看懂的线程池学习总结](#)》中就说过这三个参数是 `ThreadPoolExecutor` 最重要的参数，它们基本决定了线程池对于任务的处理策略。

如何支持参数动态配置？且看 `ThreadPoolExecutor` 提供的下面这些方法。

```
▼  ThreadPoolExecutor has any value set  
   setCorePoolSize(int): void  
   setKeepAliveTime(long, TimeUnit): void  
   setMaximumPoolSize(int): void  
   setRejectedExecutionHandler(RejectedExecutionHandler): void  
   setThreadFactory(ThreadFactory): void  
   workers: HashSet<Worker> = new HashSet<Worker>()
```

格外需要注意的是 `corePoolSize`，程序运行期间的时候，我们调用 `setCorePoolSize()` 这个方法的话，线程池会首先判断当前工作线程数是否大于 `corePoolSize`，如果大于的话就会回收工作线程。

另外，你也看到了上面并没有动态指定队列长度的方法，美团的方式是自定义了一个叫做 `ResizableCapacityLinkedBlockingQueue` 的队列（主要就是把 `LinkedBlockingQueue` 的 `capacity` 字段的 `final` 关键字修饰给去掉了，让它变为可变的）。

最终实现的可动态修改线程池参数效果如下。👏👏👏

修改线程池参数

应用名

线程池名

核心数

最大值

队列类型

队列长度

是否告警

容量告警

活跃度告警

3

5

SynchronousQueue

队列长度

ON

队列容量告警阈值

80

取消

保存

还没看够？推荐 why神的[《如何设置线程池参数？美团给出了一个让面试官虎躯一震的回答。》](#)这篇文章，深度剖析，很不错哦！