

## 03 | 线程池：业务代码最常用也最容易犯错的组件

2020-03-12 朱晔

Java 业务开发常见错误 100 例

[进入课程 >](#)



讲述：王少泽

时长 22:11 大小 20.33M



你好，我是朱晔。今天，我来讲讲使用线程池需要注意的一些问题。

在程序中，我们会用各种池化技术来缓存创建昂贵的对象，比如线程池、连接池、内存池。一般是预先创建一些对象放入池中，使用的时候直接取出使用，用完归还以便复用，还会通过一定的策略调整池中缓存对象的数量，实现池的动态伸缩。

由于线程的创建比较昂贵，随意、没有控制地创建大量线程会造成性能问题，因此短平快的任务一般考虑使用线程池来处理，而不是直接创建线程。



今天，我们就针对线程池这个话题展开讨论，通过三个生产事故，来看看使用线程池应该注意些什么。

## 线程池的声明需要手动进行

Java 中的 `Executors` 类定义了一些快捷的工具方法，来帮助我们快速创建线程池。《阿里巴巴 Java 开发手册》中提到，禁止使用这些方法来创建线程池，而应该手动 `new ThreadPoolExecutor` 来创建线程池。这一条规则的背后，是大量血淋淋的生产事故，最典型的的就是 `newFixedThreadPool` 和 `newCachedThreadPool`，可能因为资源耗尽导致 OOM 问题。

首先，我们来看一下 `newFixedThreadPool` 为什么可能会出现 OOM 的问题。

我们写一段测试代码，来初始化一个单线程的 `FixedThreadPool`，循环 1 亿次向线程池提交任务，每个任务都会创建一个比较大的字符串然后休眠一小时：

 复制代码

```
1 @GetMapping("oom1")
2 public void oom1() throws InterruptedException {
3
4     ThreadPoolExecutor threadPool = (ThreadPoolExecutor) Executors.newFixedThreadPool(1);
5     //打印线程池的信息，稍后我会解释这段代码
6     printStats(threadPool);
7     for (int i = 0; i < 100000000; i++) {
8         threadPool.execute(() -> {
9             String payload = IntStream.rangeClosed(1, 1000000)
10                 .mapToObj(__ -> "a")
11                 .collect(Collectors.joining("")) + UUID.randomUUID().toString();
12             try {
13                 TimeUnit.HOURS.sleep(1);
14             } catch (InterruptedException e) {
15                 // ...
16             }
17             log.info(payload);
18         });
19     }
20     threadPool.shutdown();
21     threadPool.awaitTermination(1, TimeUnit.HOURS);
22 }
```

执行程序后不久，日志中就出现了如下 OOM：

 复制代码

```
1 Exception in thread "http-nio-45678-ClientPoller" java.lang.OutOfMemoryError: (
```

翻看 `newFixedThreadPool` 方法的源码不难发现，线程池的工作队列直接 `new` 了一个 `LinkedBlockingQueue`，而默认构造方法的 `LinkedBlockingQueue` 是一个 `Integer.MAX_VALUE` 长度的队列，可以认为是无界的：

 复制代码

```
1 public static ExecutorService newFixedThreadPool(int nThreads) {
2     return new ThreadPoolExecutor(nThreads, nThreads,
3                                   0L, TimeUnit.MILLISECONDS,
4                                   new LinkedBlockingQueue<Runnable>());
5 }
6
7 public class LinkedBlockingQueue<E> extends AbstractQueue<E>
8     implements BlockingQueue<E>, java.io.Serializable {
9     ...
10
11
12     /**
13      * Creates a {@code LinkedBlockingQueue} with a capacity of
14      * {@code Integer#MAX_VALUE}.
15      */
16     public LinkedBlockingQueue() {
17         this(Integer.MAX_VALUE);
18     }
19     ...
20 }
```

虽然使用 `newFixedThreadPool` 可以把工作线程控制在固定的数量上，但任务队列是无界的。如果任务较多并且执行较慢的话，队列可能会快速积压，撑爆内存导致 OOM。

我们再把刚才的例子稍微改一下，改为使用 `newCachedThreadPool` 方法来获得线程池。程序运行不久后，同样看到了如下 OOM 异常：


 复制代码

```
1 [11:30:30.487] [http-nio-45678-exec-1] [ERROR] [.a.c.c.C.[.[]].[dispatcherSe
2 java.lang.OutOfMemoryError: unable to create new native thread
```

从日志中可以看到，这次 OOM 的原因是无法创建线程，翻看 `newCachedThreadPool` 的源码可以看到，这种线程池的最大线程数是 `Integer.MAX_VALUE`，可以认为是没有上限的，而其工作队列 `SynchronousQueue` 是一个没有存储空间的阻塞队列。这意味着，只

要有请求到来，就必须找到一条工作线程来处理，如果当前没有空闲的线程就再创建一条新的。

由于我们的任务需要 1 小时才能执行完成，大量的任务进来后会创建大量的线程。我们知道线程是需要分配一定的内存空间作为线程栈的，比如 1MB，因此无限制创建线程必然会导致 OOM：

 复制代码

```
1 public static ExecutorService newCachedThreadPool() {  
2     return new ThreadPoolExecutor(0, Integer.MAX_VALUE,  
3                                   60L, TimeUnit.SECONDS,  
4                                   new SynchronousQueue<Runnable>());  
}
```

其实，大部分 Java 开发同学知道这两种线程池的特性，只是抱有侥幸心理，觉得只是使用线程池做一些轻量级的任务，不可能造成队列积压或开启大量线程。

但，现实往往是残酷的。我之前就遇到过这么一个事故：用户注册后，我们调用一个外部服务去发送短信，发送短信接口正常时可以在 100 毫秒内响应，TPS 100 的注册量，CachedThreadPool 能稳定在占用 10 个左右线程的情况下满足需求。在某个时间点，外部短信服务不可用了，我们调用这个服务的超时又特别长，比如 1 分钟，1 分钟可能就进来了 6000 用户，产生 6000 个发送短信的任务，需要 6000 个线程，没多久就因为无法创建线程导致了 OOM，整个应用程序崩溃。

因此，**我同样不建议使用 Executors 提供的两种快捷的线程池，原因如下：**

我们需要根据自己的场景、并发情况来评估线程池的几个核心参数，包括核心线程数、最大线程数、线程回收策略、工作队列的类型，以及拒绝策略，确保线程池的工作行为符合需求，一般都需要设置有界的工作队列和可控的线程数。


任何时候，都应该为自定义线程池指定有意义的名称，以方便排查问题。当出现线程数量暴增、线程死锁、线程占用大量 CPU、线程执行出现异常等问题时，我们往往会抓取线程栈。此时，有意义的线程名称，就可以方便我们定位问题。

除了建议手动声明线程池以外，我还建议**用一些监控手段来观察线程池的状态**。线程池这个组件往往会表现得任劳任怨、默默无闻，除非是出现了拒绝策略，否则压力再大都不会抛出

一个异常。如果我们能提前观察到线程池队列的积压，或者线程数量的快速膨胀，往往可以提早发现并解决问题。

## 线程池线程管理策略详解

在之前的 Demo 中，我们用一个 `printStats` 方法实现了最简陋的监控，每秒输出一次线程池的基本内部信息，包括线程数、活跃线程数、完成了多少任务，以及队列中还有多少积压任务等信息：

 复制代码

```
1 private void printStats(ThreadPoolExecutor threadPool) {
2     Executors.newSingleThreadScheduledExecutor().scheduleAtFixedRate(() -> {
3         log.info("=====");
4         log.info("Pool Size: {}", threadPool.getPoolSize());
5         log.info("Active Threads: {}", threadPool.getActiveCount());
6         log.info("Number of Tasks Completed: {}", threadPool.getCompletedTaskC
7         log.info("Number of Tasks in Queue: {}", threadPool.getQueue().size())
8
9         log.info("=====");
10    }, 0, 1, TimeUnit.SECONDS);
11 }
```

接下来，我们就利用这个方法来自观察一下线程池的基本特性吧。

首先，自定义一个线程池。这个线程池具有 2 个核心线程、5 个最大线程、使用容量为 10 的 `ArrayBlockingQueue` 阻塞队列作为工作队列，使用默认的 `AbortPolicy` 拒绝策略，也就是任务添加到线程池失败会抛出 `RejectedExecutionException`。此外，我们借助了 Jodd 类库的 `ThreadFactoryBuilder` 方法来构造一个线程工厂，实现线程池线程的自定义命名。

然后，我们写一段测试代码来自观察线程池管理线程的策略。测试代码的逻辑为，每次间隔 1 秒向线程池提交任务，循环 20 次，每个任务需要 10 秒才能执行完成，代码如下：

 复制代码

```
1 @GetMapping("right")
2 public int right() throws InterruptedException {
3     //使用一个计数器跟踪完成的任务数
4     AtomicInteger atomicInteger = new AtomicInteger();
5     //创建一个具有2个核心线程、5个最大线程，使用容量为10的ArrayBlockingQueue阻塞队列作为
6     ThreadPoolExecutor threadPool = new ThreadPoolExecutor(
```



```

7         2, 5,
8         5, TimeUnit.SECONDS,
9         new ArrayBlockingQueue<>(10),
10        new ThreadFactoryBuilder().setNameFormat("demo-threadpool-%d").get
11        new ThreadPoolExecutor.AbortPolicy());
12
13    printStats(threadPool);
14    //每隔1秒提交一次，一共提交20次任务
15    IntStream.rangeClosed(1, 20).forEach(i -> {
16        try {
17            TimeUnit.SECONDS.sleep(1);
18        } catch (InterruptedException e) {
19            e.printStackTrace();
20        }
21        int id = atomicInteger.incrementAndGet();
22        try {
23            threadPool.submit(() -> {
24                log.info("{} started", id);
25                //每个任务耗时10秒
26                try {
27                    TimeUnit.SECONDS.sleep(10);
28                } catch (InterruptedException e) {
29                }
30                log.info("{} finished", id);
31            });
32        } catch (Exception ex) {
33            //提交出现异常的话，打印出错信息并为计数器减一
34            log.error("error submitting task {}", id, ex);
35            atomicInteger.decrementAndGet();
36        }
37    });
38
39    TimeUnit.SECONDS.sleep(60);
40    return atomicInteger.intValue();
41 }

```

60 秒后页面输出了 17，有 3 次提交失败了：

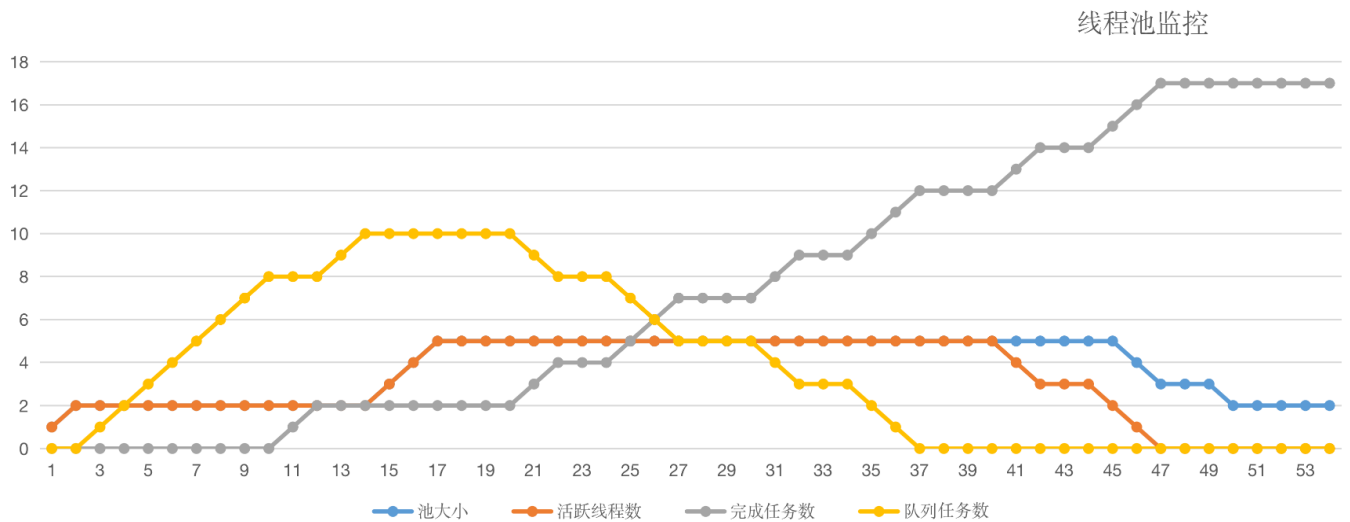
← → ↻ 🏠 ⓘ localhost:45678/threadpoolloom/right

17

并且日志中也出现了 3 次类似的错误信息：

```
1 [14:24:52.879] [http-nio-45678-exec-1] [ERROR] [.t.c.t.demo1.ThreadPoolOOMCont
2 java.util.concurrent.RejectedExecutionException: Task java.util.concurrent.Futi
```

我们把 printStats 方法打印出的日志绘制成图表，得出如下曲线：



至此，我们可以总结出线程池默认的工作行为：

不会初始化 corePoolSize 个线程，有任务来了才创建工作线程；

当核心线程满了之后不会立即扩容线程池，而是把任务堆积到工作队列中；

当工作队列满了后扩容线程池，一直到线程个数达到 maximumPoolSize 为止；

如果队列已满且达到了最大线程后还有任务进来，按照拒绝策略处理；

当线程数大于核心线程数时，线程等待 keepAliveTime 后还是没有任务需要处理的话，收缩线程到核心线程数。

了解这个策略，有助于我们根据实际的容量规划需求，为线程池设置合适的初始化参数。当然，我们也可以通过一些手段来改变这些默认工作行为，比如：

声明线程池后立即调用 prestartAllCoreThreads 方法，来启动所有核心线程；

传入 true 给 allowCoreThreadTimeOut 方法，来让线程池在空闲的时候同样回收核心线程。

不知道你有没有想过：Java 线程池是先用工作队列来存放来不及处理的任务，满了之后再扩容线程池。当我们的工作队列设置得很大时，最大线程数这个参数显得没有意义，因为队列很难满，或者到满的时候再去扩容线程池已经于事无补了。

那么，**我们有没有办法让线程池更激进一点，优先开启更多的线程，而把队列当成一个后备方案呢？**比如我们这个例子，任务执行得很慢，需要 10 秒，如果线程池可以优先扩容到 5 个最大线程，那么这些任务最终都可以完成，而不会因为线程池扩容过晚导致慢任务来不及处理。

限于篇幅，这里我只给你一个大致思路：

1. 由于线程池在工作队列满了无法入队的情况下会扩容线程池，那么我们可以是否可以重写队列的 `offer` 方法，造成这个队列已满的假象呢？
2. 由于我们 Hack 了队列，在达到了最大线程后势必会触发拒绝策略，那么能否实现一个自定义的拒绝策略处理程序，这个时候再把任务真正插入队列呢？

接下来，就请你动手试试看如何实现这样一个“弹性”线程池吧。Tomcat 线程池也实现了类似的效果，可供你借鉴。

## 务必确认清楚线程池本身是不是复用的

不久之前我遇到了这样一个事故：某项目生产环境时不时有报警提示线程数过多，超过 2000 个，收到报警后查看监控发现，瞬时线程数比较多但过一会儿又会降下来，线程数抖动很厉害，而应用的访问量变化不大。

为了定位问题，我们在线程数比较高的时候进行线程栈抓取，抓取后发现内存中有 1000 多个自定义线程池。一般而言，线程池肯定是复用的，有 5 个以内的线程池都可以认为正常，而 1000 多个线程池肯定不正常。

在项目代码里，我们没有搜到声明线程池的地方，搜索 `execute` 关键字后定位到，原来是业务代码调用了一个类库来获得线程池，类似如下的业务代码：调用 `ThreadPoolHelper` 的 `getThreadPool` 方法来获得线程池，然后提交数个任务到线程池处理，看不出什么异常。



```
1 @GetMapping("wrong")
2 public String wrong() throws InterruptedException {
3     ThreadPoolExecutor threadPool = ThreadPoolHelper.getThreadPool();
4     IntStream.rangeClosed(1, 10).forEach(i -> {
5         threadPool.execute(() -> {
6             ...
7             try {
8                 TimeUnit.SECONDS.sleep(1);
9             } catch (InterruptedException e) {
10            }
11        });
12    });
13    return "OK";
14 }
```

[复制代码](#)

但是，来到 `ThreadPoolHelper` 的实现让人大跌眼镜，`getThreadPool` 方法居然是每次都使用 `Executors.newCachedThreadPool` 来创建一个线程池。

```
1 class ThreadPoolHelper {
2     public static ThreadPoolExecutor getThreadPool() {
3         //线程池没有复用
4         return (ThreadPoolExecutor) Executors.newCachedThreadPool();
5     }
6 }
```

[复制代码](#)

通过上一小节的学习，我们可以想到 `newCachedThreadPool` 会在需要时创建必要多的线程，业务代码的一次业务操作会向线程池提交多个慢任务，这样执行一次业务操作就会开启多个线程。如果业务操作并发量较大的话，的确有可能一下子开启几千个线程。

那，为什么我们能在监控中看到线程数量会下降，而不会撑爆内存呢？

回到 `newCachedThreadPool` 的定义就会发现，它的核心线程数是 0，而 `keepAliveTime` 是 60 秒，也就是在 60 秒之后所有的线程都是可以回收的。好吧，就因为这个特性，我们的业务程序死得没太难看。

要修复这个 Bug 也很简单，使用一个静态字段来存放线程池的引用，返回线程池的代码直接返回这个静态字段即可。这里一定要记得我们的最佳实践，手动创建线程池。修复后的 `ThreadPoolHelper` 类如下：

```
1 class ThreadPoolHelper {
2     private static ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecuto
3         10, 50,
4         2, TimeUnit.SECONDS,
5         new ArrayBlockingQueue<>(1000),
6         new ThreadFactoryBuilder().setNameFormat("demo-threadpool-%d").get());
7     public static ThreadPoolExecutor getRightThreadPool() {
8         return threadPoolExecutor;
9     }
10 }
```

## 需要仔细斟酌线程池的混用策略

线程池的意义在于复用，那这是不是意味着程序应该始终使用一个线程池呢？

当然不是。通过第一小节的学习我们知道，**要根据任务的“轻重缓急”来指定线程池的核心参数，包括线程数、回收策略和任务队列：**

对于执行比较慢、数量不大的 IO 任务，或许要考虑更多的线程数，而不需要太大的队列。

而对于吞吐量较大的计算型任务，线程数量不宜过多，可以是 CPU 核数或核数 \*2（理由是，线程一定调度到某个 CPU 进行执行，如果任务本身是 CPU 绑定的任务，那么过多的线程只会增加线程切换的开销，并不能提升吞吐量），但可能需要较长的队列来做缓冲。

之前我也遇到过这么一个问题，业务代码使用了线程池异步处理一些内存中的数据，但通过监控发现处理得非常慢，整个处理过程都是内存中的计算不涉及 IO 操作，也需要数秒的处理时间，应用程序 CPU 占用也不是特别高，有点不可思议。

经排查发现，业务代码使用的线程池，还被一个后台的文件批处理任务用到了。

或许是够用就好的原则，这个线程池只有 2 个核心线程，最大线程也是 2，使用了容量为 100 的 ArrayBlockingQueue 作为工作队列，使用了 CallerRunsPolicy 拒绝策略：


```
1 private static ThreadPoolExecutor threadPool = new ThreadPoolExecutor(
2     2, 2,
```

```

3      1, TimeUnit.HOURS,
4      new ArrayBlockingQueue<>(100),
5      new ThreadFactoryBuilder().setNameFormat("batchfileprocess-threadpool-");
6      new ThreadPoolExecutor.CallerRunsPolicy());

```

这里，我们模拟一下文件批处理的代码，在程序启动后通过一个线程开启死循环逻辑，不断向线程池提交任务，任务的逻辑是向一个文件中写入大量的数据：

 复制代码

```

1  @PostConstruct
2  public void init() {
3      printStats(threadPool);
4
5      new Thread(() -> {
6          //模拟需要写入的大量数据
7          String payload = IntStream.rangeClosed(1, 1_000_000)
8              .mapToObj(__ -> "a")
9              .collect(Collectors.joining(""));
10         while (true) {
11             threadPool.execute(() -> {
12                 try {
13                     //每次都是创建并写入相同的数据到相同的文件
14                     Files.write(Paths.get("demo.txt"), Collections.singletonList(payload));
15                 } catch (IOException e) {
16                     e.printStackTrace();
17                 }
18                 log.info("batch file processing done");
19             });
20         }
21     }).start();
22 }

```

可以想象到，这个线程池中的 2 个线程任务是相当重的。通过 printStats 方法打印出的日志，我们观察下线程池的负担：

```


[16:10:32.062] [batchfileprocess-threadpool-0] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:86] - batch file processing done
[16:10:32.064] [batchfileprocess-threadpool-1] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:86] - batch file processing done
[16:10:32.066] [batchfileprocess-threadpool-0] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:86] - batch file processing done
[16:10:32.066] [Thread-4] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:86] - batch file processing done
[16:10:32.069] [batchfileprocess-threadpool-1] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:86] - batch file processing done
[16:10:32.069] [pool-4-thread-1] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:44] - =====
[16:10:32.069] [pool-4-thread-1] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:45] - Pool Size: 2
[16:10:32.070] [pool-4-thread-1] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:46] - Active Threads: 2
[16:10:32.070] [pool-4-thread-1] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:47] - Number of Tasks Completed: 1540
[16:10:32.070] [pool-4-thread-1] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:48] - Number of Tasks in Queue: 99
[16:10:32.070] [pool-4-thread-1] [INFO] [g.t.c.t.d.ThreadPoolMixuseController:50] - =====

```

可以看到，**线程池的 2 个线程始终处于活跃状态，队列也基本处于打满状态**。因为开启了 CallerRunsPolicy 拒绝处理策略，所以当线程满载队列也满的情况下，任务会在提交任务的线程，或者说调用 execute 方法的线程执行，也就是说不能认为提交到线程池的任务就一定是异步处理的。如果使用了 CallerRunsPolicy 策略，那么有可能异步任务变为同步执行。从日志的第四行也可以看到这点。这也是这个拒绝策略比较特别的原因。

不知道写代码的同学为什么设置这个策略，或许是测试时发现线程池因为任务处理不过来出现了异常，而又不希望线程池丢弃任务，所以最终选择了这样的拒绝策略。不管怎样，这些日志足以说明线程池是饱和状态。

可以想象到，业务代码复用这样的线程池来做内存计算，命运一定是悲惨的。我们写一段代码测试下，向线程池提交一个简单的任务，这个任务只是休眠 10 毫秒没有其他逻辑：

 复制代码

```
1 private Callable<Integer> calcTask() {
2     return () -> {
3         TimeUnit.MILLISECONDS.sleep(10);
4         return 1;
5     };
6 }
7
8 @GetMapping("wrong")
9 public int wrong() throws ExecutionException, InterruptedException {
10     return threadPool.submit(calcTask()).get();
11 }
```

我们使用 wrk 工具对这个接口进行一个简单的压测，可以看到 TPS 为 75，性能的确非常差。

```
➔ ~ wrk -t10 -c100 -d 10s http://localhost:45678/threadpoolmixuse/wrong
Running 10s test @ http://localhost:45678/threadpoolmixuse/wrong
10 threads and 100 connections
Thread Stats   Avg      Stdev     Max    +/-  Stdev
Latency       1.22s   530.46ms  1.96s   80.79%
Req/Sec       11.10    14.08   110.00   91.82%
755 requests in 10.07s, 92.40KB read
Requests/sec: 74.99
Transfer/sec: 9.18KB
```

细想一下，问题其实没有这么简单。因为原来执行 IO 任务的线程池使用的是 `CallerRunsPolicy` 策略，所以直接使用这个线程池进行异步计算的话，**当线程池饱和的时候，计算任务会在执行 Web 请求的 Tomcat 线程执行，这时就会进一步影响到其他同步处理的线程，甚至造成整个应用程序崩溃。**

解决方案很简单，使用独立的线程池来做这样的“计算任务”即可。计算任务打了双引号，是因为我们的模拟代码执行的是休眠操作，并不属于 CPU 绑定的操作，更类似 IO 绑定的操作，如果线程池线程数设置太小会限制吞吐能力：

 复制代码

```
1 private static ThreadPoolExecutor asyncCalcThreadPool = new ThreadPoolExecutor
2     200, 200,
3     1, TimeUnit.HOURS,
4     new ArrayBlockingQueue<>(1000),
5     new ThreadFactoryBuilder().setNameFormat("asynccalc-threadpool-%d").get();
6
7
8 @GetMapping("right")
9 public int right() throws ExecutionException, InterruptedException {
10     return asyncCalcThreadPool.submit(calcTask()).get();
11 }
```

使用单独的线程池改造代码后再来测试一下性能，TPS 提高到了 1727：

```
➔ ~ wrk -t10 -c100 -d 10s http://localhost:45678/threadpoolmixuse/right
Running 10s test @ http://localhost:45678/threadpoolmixuse/right
10 threads and 100 connections
Thread Stats   Avg      Stdev     Max    +/-  Stdev
  Latency    57.59ms   12.11ms  108.71ms   88.33%
  Req/Sec   173.68    21.89   242.00    69.80%
17371 requests in 10.05s, 2.07MB read
Requests/sec:  1727.68
Transfer/sec:   211.08KB
```

可以看到，盲目复用线程池混用线程的问题在于，别人定义的线程池属性不一定适合你的任务，而且混用会相互干扰。这就好比，我们往往会用虚拟化技术来实现资源的隔离，而不是让所有应用程序都直接使用物理机。

就线程池混用问题，我想再和你补充一个坑：**Java 8 的 parallel stream 功能，可以让我们很方便地并行处理集合中的元素，其背后是共享同一个 ForkJoinPool，默认并行度是**



**CPU 核数 -1**。对于 CPU 绑定的任务来说，使用这样的配置比较合适，但如果集合操作涉及同步 IO 操作的话（比如数据库操作、外部服务调用等），建议自定义一个 ForkJoinPool（或普通线程池）。你可以参考 [🔗 第一讲](#) 的相关 Demo。

## 重点回顾

线程池管理着线程，线程又属于宝贵的资源，有许多应用程序的性能问题都来自线程池的配置和使用不当。在今天的学习中，我通过三个和线程池相关的生产事故，和你分享了使用线程池的几个最佳实践。

第一，Executors 类提供一些快捷声明线程池的方法虽然简单，但隐藏了线程池的参数细节。因此，使用线程池时，我们一定要根据场景和需求配置合理的线程数、任务队列、拒绝策略、线程回收策略，并对线程进行明确的命名方便排查问题。

第二，既然使用了线程池就需要确保线程池是在复用的，每次 new 一个线程池出来可能比不用线程池还糟糕。如果你没有直接声明线程池而是使用其他同学提供的类库来获得一个线程池，请务必查看源码，以确认线程池的实例化方式和配置是符合预期的。

第三，复用线程池不代表应用程序始终使用同一个线程池，我们应该根据任务的性质来选用不同的线程池。特别注意 IO 绑定的任务和 CPU 绑定的任务对于线程池属性的偏好，如果希望减少任务间的相互干扰，考虑按需使用隔离的线程池。

最后我想强调的是，**线程池作为应用程序内部的核心组件往往缺乏监控**（如果你使用类似 RabbitMQ 这样的 MQ 中间件，运维同学一般会帮我们做好中间件监控），往往到程序崩溃后才发现线程池的问题，很被动。在设计篇中我们会重新谈及这个问题及其解决方案。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [🔗 这个链接](#) 查看。

## 思考与讨论

1. 在第一节中我们提到，或许一个激进创建线程的弹性线程池更符合我们的需求，你能给出相关的实现吗？实现后再测试一下，是否所有的任务都可以正常处理完成呢？
2. 在第二节中，我们改进了 ThreadPoolHelper 使其能够返回复用的线程池。如果我们不小心每次都创建了这样一个自定义的线程池（10 核心线程，50 最大线程，2 秒回收的），反复执行测试接口线程，最终可以被回收吗？会出现 OOM 问题吗？

你还遇到过线程池相关的其他坑吗？我是朱晔，欢迎在评论区与我留言分享你的想法，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

## 进入朱晔老师「读者群」带你攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 代码加锁：不要让“锁”事成为烦心事

下一篇 04 | 连接池：别让连接池帮了倒忙

### 精选留言 (14)

 写留言




**Darren** 置顶

2020-03-12

第一个问题的来了，请老师指点：

<https://github.com/y645194203/geektime-java-100/blob/master/ExtremeThreadPoolExecutor.java>

里面自定义了一个extremeOffer方法，因为不是BlockQueue接口的方法，所以在执行...  
展开 

作者回复：直接用put即可，可以参考这里的回复：

<https://stackoverflow.com/questions/19528304/how-to-get-the-threadpoolexecutor-to-i>

ncrease-threads-to-max-before-queueing

不过要考虑选择丢数据还是阻塞

其实，实现自己的RejectedExecutionHandler耦合自己的Queue也无可厚非。Tomcat也是这样的，其实现参考这里：<https://github.com/apache/tomcat/blob/a801409b37294c3f3dd5590453fb9580d7e33af2/java/org/apache/tomcat/util/threads/ThreadPoolExecutor.java>

4

3



小美 置顶

2020-03-14

第二个问题大家都说核心线程数不会被回收，但是方法执行完线程池的引用已经引用不到了吧，线程池对象会被垃圾回收吧，垃圾回收时核心线程怎么办呢

展开

作者回复: ThreadPoolExecutor回收不了，可以看看其源码，工作线程Worker是内部类，只要它活着，换句话说线程在跑，就会阻止ThreadPoolExecutor回收，所以其实ThreadPoolExecutor是无法回收的，并不能认为ThreadPoolExecutor没有引用就能回收

...

3



陈天柱

2020-03-12

首先赞一下老师的排查问题的思路!!! 然后针对第二个问题，我觉得不会被回收且很快就会OOM了，因为每次请求都新建线程池，每个线程池的核心数都是10，虽然自定义线程池设置2秒回收，但是没超过线程池核心数10是不会被回收的，不间断的请求过来导致创建大量线程，最终OOM

展开

作者回复: 3

...

2



G小调

2020-03-12

文章非常棒 通过真实案例讲解 透过现象看本质

展开

...

2



**Darren**

2020-03-12

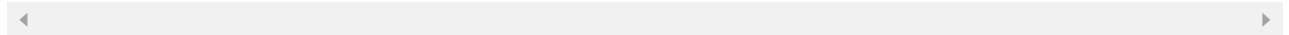
先回答第二个问题吧，第一个等天亮了，试一试

不会被回收，会OOM，即使是自定义线程池，核心线程是不会回收的，每次需要10个线程，刚好是核心线程数，因此每次请求都会创建10个核心线程数的线程池，请求次数多了后，很快就回OOM。

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

展开 ∨

作者回复: ㊦



👍 2



**mgs2002**

2020-03-14

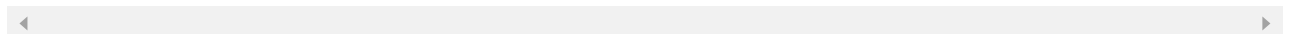
老师，我还有个问题，我在本机做了一下默认线程池（CompletableFuture）和自定义线程池的调用时间对比，测试了好多次每次都是默认线程池的时间快，任务越多自定义线程池的时间就越慢，默认的相对稳定，不懂怎么回事了。。

这是我的测试结果，default是默认的，custom是自定义的

When 1 tasks => future default: 11,future custom: 4...

展开 ∨

作者回复: 这段代码不是很能理解意思，testCompletableFutureDefaultExecutor是什么？最好给出源码链接



**汝林外史**

2020-03-14

1. 既然选择先扩容线程池再加入队列，那为什么不干脆把核心线程数设置大一些，然后核心线程数可回收这种策略呢？

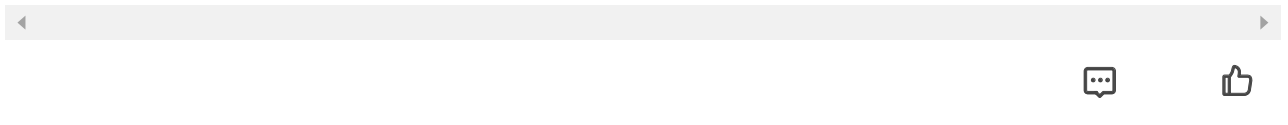
2. 复用线程池，任务很慢，主线程get结果的时候不会导致主线程卡死的状态吗？不是也提倡不同的任务用不同的线程池，那复用与不复用的边界在哪里呢？是要根据也无需求自己评估吗？

展开 ∨

作者回复: 1. 你说的这种策略，此文也有提到：<https://stackoverflow.com/questions/19528304/how-to-get-the-threadpool-executor-to-increase-threads-to-max-before-queueing>  
其实，我们希望的是尽量确保有足够多的线程能处理任务，但是又不闲置过多线程，或临时创建过多线程，换句话说让线程的创建和回收不要太频繁。选择哪个策略要根据任务的性质和压力的

流量形态来决定。

2. 这里我说的复用线程池是指不每次都创建线程池，线程池必须复用而不是按需创建，但是不推荐一味混用一个线程池。对于选择是否混用线程池，至少对于频+快的任务和少+慢的任务应该分开，还是要根据实际任务的性质来选择。



**张少华**

2020-03-13

挺厉害

展开 ▾



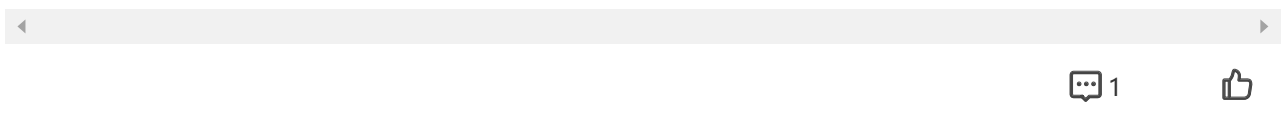
**mgs2002**

2020-03-13

学到了，我现在代码还是用的CompletableFuture默认线程池。。。有个问题请教老师，我有两个项目，里面有很多数据统计的地方使用到了线程池，这种是属于CPU绑定类型的吧（通过数据库来查询统计），还有我想分别给两个项目设置不同的线程池可行吗

展开 ▾

作者回复: 通过数据库来查询统计就是IO操作，既然项目都不同了，那么就不是一个进程了，我们说的线程池隔离是针对一个进程的



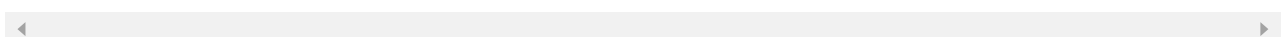
**HW**

2020-03-13

```
mysqld: Table 'mysql.plugin' doesn't exist
[Warning] Insecure configuration for --pid-file: Location '/var/run/mysqld' in the path is accessible to all OS users. Consider choosing a secure directory.
[Warning] Failed to open optimizer cost constant tables
[ERROR] Fatal error: Can't open and lock privilege tables: Table 'mysql.user' does not exist
```

展开 ▾

作者回复: 你是什么os呢？我是macos没有遇到过这个问题，尝试先重启docker service，或OS。docker-compose是方便大家快速启动依赖，不是必须的，实在不行你可以自己手动安装mysql，都是一样的





**汝林外史**

2020-03-12

老师的文章真的是看得很爽。问题如下：

- 1.激进的都适应什么场合呢？先扩容线程池再加入任务队列，也可能队列满了还来任务，还是要再拒绝。
  - 2.因为核心线程不会回收，所以会OOM。可以设置allowCoreThreadTimeOut参数让核心线程也可以回收。另外文中的ThreadPoolHelper是用来复用线程池的，但是提交的都是...
- 展开 ∨

作者回复: 1、可以想想tomcat为什么觉得这样激进的线程更适合。其实.NET的线程池就是这样的弹性线程池，只不过创建新的线程还会有一定的思考时间，延迟新线程的创建，更智能。

2、其实和任务慢慢慢没有太大关系，即使任务不慢，不复用也是有问题的。

**pedro**

2020-03-12

总结一下今天学到的；1、线程池的OOM问题，可能是队列满造成的，也可能是线程太多造成的，至于后面的那个2000个线程池太多，大概是这辈子都不会遇到这种错误吧😅。2、线程池的策略问题，优先根据场景来选择合适的参数来新建线程池，若还是无法满足，可自定义线程池，总之一切以实际为准。

展开 ∨



2020-03-12

OOM，全称“Out Of Memory”，中文意思是“内存用尽”。当JVM因为没有足够的内存来为对象分配空间，并且垃圾回收器也没有空间可回收时，就会抛出这个Error。

**JavaGuide**

2020-03-12

很棒！

展开 ∨



