

## 34 | Worker Thread模式：如何避免重复创建线程？

2019-05-16 王宝令

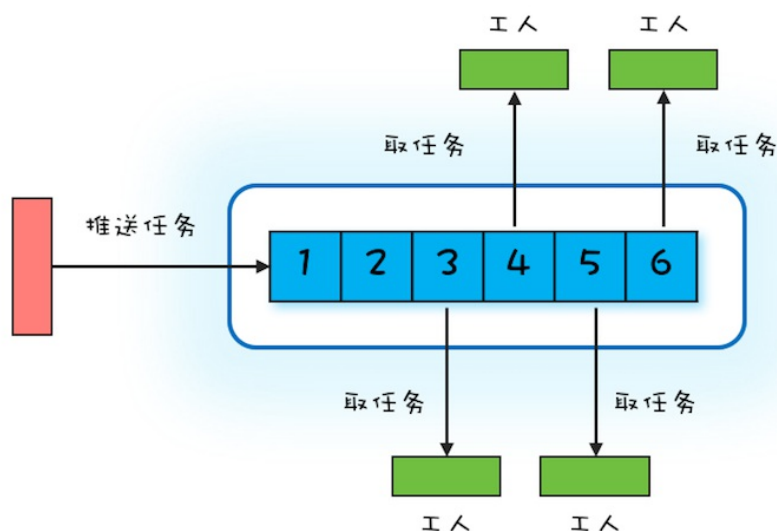


在[上一篇文章](#)中，我们介绍了一种最简单的分工模式——Thread-Per-Message模式，对应到现实世界，其实就是委托代办。这种分工模式如果用Java Thread实现，频繁地创建、销毁线程非常影响性能，同时无限制地创建线程还可能导致OOM，所以在Java领域使用场景就受限了。

要想有效避免线程的频繁创建、销毁以及OOM问题，就不得不提今天我们要细聊的，也是Java领域使用最多的Worker Thread模式。

### Worker Thread模式及其实现

Worker Thread模式可以类比现实世界里车间的工作模式：车间里的工人，有活儿了，大家一起干，没活儿了就聊聊天等着。你可以参考下面的示意图来理解，Worker Thread模式中Worker Thread对应到现实世界里，其实指的就是车间里的工人。不过这里需要注意的是，车间里的工人数量往往是确定的。



车间工作示意图

那在编程领域该如何模拟车间的这种工作模式呢？或者说如何去实现**Worker Thread**模式呢？通过上面的图，你很容易就能想到用阻塞队列做任务池，然后创建固定数量的线程消费阻塞队列中的任务。其实你仔细想会发现，这个方案就是**Java**语言提供的线程池。

线程池有很多优点，例如能够避免重复创建、销毁线程，同时能够限制创建线程的上限等等。学习完上一篇文章后你已经知道，用**Java**的**Thread**实现**Thread-Per-Message**模式难以应对高并发场景，原因就在于频繁创建、销毁**Java**线程的成本有点高，而且无限制地创建线程还可能导致应用**OOM**。线程池，则恰好能解决这些问题。

那我们还是以**echo**程序为例，看看如何用线程池来实现。

下面的示例代码是用线程池实现的**echo**服务端，相比于**Thread-Per-Message**模式的实现，改动非常少，仅仅是创建了一个最多线程数为**500**的线程池**es**，然后通过**es.execute()**方法将请求处理的任务提交给线程池处理。

```

ExecutorService es = Executors
    .newFixedThreadPool(500);
final ServerSocketChannel ssc =
    ServerSocketChannel.open().bind(
        new InetSocketAddress(8080));
//处理请求
try {
    while (true) {
        // 接收请求
        SocketChannel sc = ssc.accept();
        // 将请求处理任务提交给线程池
        es.execute()->{
            try {
                // 读Socket
                ByteBuffer rb = ByteBuffer
                    .allocateDirect(1024);
                sc.read(rb);
                //模拟处理请求
                Thread.sleep(2000);
                // 写Socket
                ByteBuffer wb =
                    (ByteBuffer)rb.flip();
                sc.write(wb);
                // 关闭Socket
                sc.close();
            }catch(Exception e){
                throw new UncheckedIOException(e);
            }
        };
    }
} finally {
    ssc.close();
    es.shutdown();
}

```

## 正确地创建线程池

Java的线程池既能够避免无限制地**创建线程**导致OOM，也能避免无限制地**接收任务**导致OOM。只不过后者经常容易被我们忽略，例如在上面的实现中，就被我们忽略了。所以强烈建议你用**创建有界的队列**来接收任务。

当请求量大于有界队列的容量时，就需要合理地拒绝请求。如何合理地拒绝呢？这需要你结合具体的业务场景来制定，即便线程池默认的拒绝策略能够满足你的需求，也同样建议你在**创建线程池时，清晰地指明拒绝策略**。

同时，为了便于调试和诊断问题，我也强烈建议你在**实际工作中给线程赋予一个业务相关的名字**。

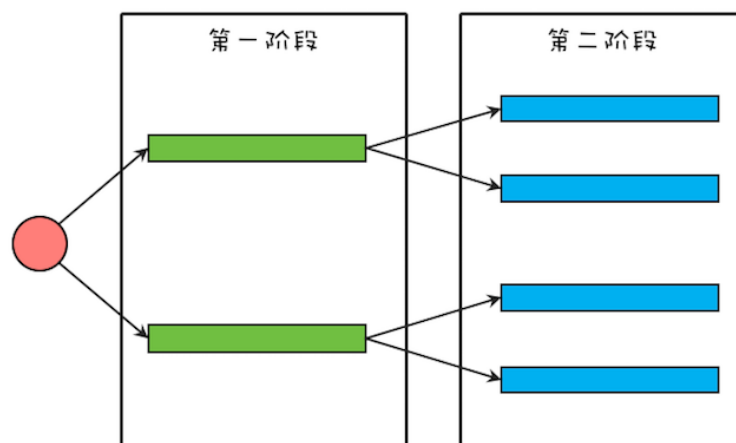
综合以上这三点建议，**echo**程序中创建线程可以使用下面的示例代码。

```
ExecutorService es = new ThreadPoolExecutor(
    50, 500,
    60L, TimeUnit.SECONDS,
    //注意要创建有界队列
    new LinkedBlockingQueue<Runnable>(2000),
    //建议根据业务需求实现ThreadFactory
    r->{
        return new Thread(r, "echo-"+ r.hashCode());
    },
    //建议根据业务需求实现RejectedExecutionHandler
    new ThreadPoolExecutor.CallerRunsPolicy());
```

## 避免线程死锁

使用线程池过程中，还要注意一种**线程死锁**的场景。如果提交到相同线程池的任务不是相互独立的，而是有依赖关系的，那么就有可能导致线程死锁。实际工作中，我就亲历过这种线程死锁的场景。具体现象是应用每运行一段时间偶尔就会处于无响应的状态，监控数据看上去一切都正常，但是实际上已经不能正常工作了。

这个出问题的应用，相关的逻辑精简之后，如下图所示，该应用将一个大型的计算任务分成两个阶段，第一个阶段的任务会等待第二阶段的子任务完成。在这个应用里，每一个阶段都使用了线程池，而且两个阶段使用的还是同一个线程池。



应用业务逻辑示意图

我们可以用下面的示例代码来模拟该应用，如果你执行下面的这段代码，会发现它永远执行不到最后一行。执行过程中没有任何异常，但是应用已经停止响应了。

```

//L1、L2阶段共用的线程池
ExecutorService es = Executors.
    newFixedThreadPool(2);
//L1阶段的闭锁
CountDownLatch l1=new CountDownLatch(2);
for (int i=0; i<2; i++){
    System.out.println("L1");
    //执行L1阶段任务
    es.execute()->{
        //L2阶段的闭锁
        CountDownLatch l2=new CountDownLatch(2);
        //执行L2阶段子任务
        for (int j=0; j<2; j++){
            es.execute()->{
                System.out.println("L2");
                l2.countDown();
            };
        }
        //等待L2阶段任务执行完
        l2.await();
        l1.countDown();
    };
}
//等着L1阶段任务执行完
l1.await();
System.out.println("end");

```

当应用出现类似问题时，首选的诊断方法是查看线程栈。下图是上面示例代码停止响应后的线程栈，你会发现线程池中的两个线程全部都阻塞在 `l2.await();` 这行代码上了，也就是说，线程池里所有的线程都在等待L2阶段的任务执行完，那L2阶段的子任务什么时候能够执行完呢？永远都没那一天了，为什么呢？因为线程池里的线程都阻塞了，没有空闲的线程执行L2阶段的任务了。

```

"pool-1-thread-2" #12 prio=5 os_prio=0 tid=0x000000001e382000 nid=0x5610 waiting on condition [0x000000001edbe000]
java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x0000000076c9ccd48> (a java.util.concurrent.CountDownLatch$Sync)
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:836)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireSharedInterruptibly(AbstractQueuedSynchronizer.java:997)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(AbstractQueuedSynchronizer.java:1304)
    at java.util.concurrent.CountDownLatch.await(CountDownLatch.java:231)
    at org.i7.cp.lesson.one.WorkerThreadTest.lambda$main$1(WorkerThreadTest.java:24)
    at org.i7.cp.lesson.one.WorkerThreadTest$$Lambda$1/1831932724.run(Unknown Source)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
    at java.lang.Thread.run(Thread.java:748)

Locked ownable synchronizers:
    - <0x0000000076c96f0c8> (a java.util.concurrent.ThreadPoolExecutor$Worker)

"pool-1-thread-1" #11 prio=5 os_prio=0 tid=0x000000001e37f000 nid=0x3850 waiting on condition [0x000000001ecbe000]
java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x0000000076cb0f688> (a java.util.concurrent.CountDownLatch$Sync)
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:836)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireSharedInterruptibly(AbstractQueuedSynchronizer.java:997)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(AbstractQueuedSynchronizer.java:1304)
    at java.util.concurrent.CountDownLatch.await(CountDownLatch.java:231)
    at org.i7.cp.lesson.one.WorkerThreadTest.lambda$main$1(WorkerThreadTest.java:24)
    at org.i7.cp.lesson.one.WorkerThreadTest$$Lambda$1/1831932724.run(Unknown Source)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
    at java.lang.Thread.run(Thread.java:748)

```

原因找到了，那如何解决就简单了，最简单粗暴的办法就是将线程池的最大线程数调大，如果能够确定任务的数量不是非常多的话，这个办法也是可行的，否则这个办法就行不通了。其实这种问题通用的解决方案是为不同的任务创建不同的线程池。对于上面的这个应用，L1阶段的任务和L2阶段的任务如果各自都有自己的线程池，就不会出现这种问题了。

最后再次强调一下：提交到相同线程池中的任务一定是相互独立的，否则就一定要慎重。

## 总结

我们曾经说过，解决并发编程里的分工问题，最好的办法是和现实世界做对比。对比现实世界构建编程领域的模型，能够让模型更容易理解。上一篇我们介绍的Thread-Per-Message模式，类似于现实世界里的委托他人办理，而今天介绍的Worker Thread模式则类似于车间里工人的工作模式。如果你在设计阶段，发现对业务模型建模之后，模型非常类似于车间的工作模式，那基本上就能确定可以在实现阶段采用Worker Thread模式来实现。

Worker Thread模式和Thread-Per-Message模式的区别有哪些呢？从现实世界的角度看，你委托代办人做事，往往是和代办人直接沟通的；对应到编程领域，其实现也是主线程直接创建了一个子线程，主子线程之间是可以直接通信的。而车间工人的工作方式则是完全围绕任务展开的，一个具体的任务被哪个工人执行，预先是无法知道的；对应到编程领域，则是主线程提交任务到线程池，但主线程并不关心任务被哪个线程执行。

**Worker Thread**模式能避免线程频繁创建、销毁的问题，而且能够限制线程的最大数量。**Java**语言里可以直接使用线程池来实现**Worker Thread**模式，线程池是一个非常基础和优秀的工具类，甚至有些大厂的编码规范都不允许用**new Thread()**来创建线程的，必须使用线程池。

不过使用线程池还是需要格外谨慎的，除了今天重点讲到的如何正确创建线程池、如何避免线程死锁问题，还需要注意前面我们曾经提到的**ThreadLocal**内存泄露问题。同时对于提交到线程池的任务，还要做好异常处理，避免异常的任务从眼前溜走，从业务的角度看，有时没有发现异常的任务后果往往都很严重。

## 课后思考

小灰同学写了如下的代码，本义是异步地打印字符串“QQ”，请问他的实现是否有问题呢？

```
ExecutorService pool = Executors
    .newSingleThreadExecutor();
pool.submit(() -> {
    try {
        String qq=pool.submit()->"QQ").get();
        System.out.println(qq);
    } catch (Exception e) {
    }
});
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。



# Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



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。

## 精选留言



vector

16

工厂里只有一个工人，他的工作就是同步的等待工厂里其他人给他提供东西，然而并没有其他人，他将等到天荒地老，海枯石烂~

2019-05-16

作者回复

比喻很形象

2019-05-16



曾轶麟

2

EagerThreadPool 老师这个线程池可以避免死锁的情况，死锁的时候会自动撑大

2019-05-23

作者回复

||

2019-05-23



zero

2

感觉这程序会调用栈内存溢出，这段代码相当于无限的递归调用啊。不知道理解的对不对，请老师指点。

2019-05-18

作者回复

不是递归，但会死锁

2019-05-18



linqw

1

**newSingleThreadExecutor**线程池只有单个线程，先将外部线程提交给线程池，外部线程等待内部线程执行完成，但由于线程池只有单线程，导致内部线程一直没有执行的机会，相当于内部线程需要线程池的资源，外部线程需要内部线程的结果，导致死锁。

2019-05-26



佑儿

1

原始的**workerThread**模式包含三种角色：工人、传送带、产品，传送带中维护一个**productionsQueue**以及最大的产品数量（为了防止产品无限积压），在传送带初始化时，创建了若干个**worker**（线程），**worker**不断从传送带取产品进行加工，当传送带中无产品时，**worker**线程被挂起等待唤醒，当有新的产品加入到传送带中时，挂起的**worker**会被唤醒，取产品加工。  
当上游线程**Thread**往传送带中加入产品时，如果**productionsQueue**到达最大产品数量时，**Thread**会被挂起。  
当有**worker**线程取出产品后，会唤醒阻塞的线程**Thread**(当然这里也有可能唤醒**worker**)  
线程池只是**workerThread**的一种实现，那么线程池中创建的**Thread**就是工人，线程池本身就是传送带，产品就是提交到线程池中的**Runnable**，  
而在线程池中的阻塞队列就相当于**productionsQueue**，请问老师，我这样理解是否正确？

2019-05-17



nonohony

0

外部线程会由于内部线程**submit.get**而阻塞，占有**single**线程池的唯一**worker**资源，从而导致内部线程永远无法执行，形成活锁。解法可以拆分为两个线程池。

2019-05-22



扬~

0

可以出个线程池异常处理的方案吗

2019-05-18



木刻

0

希望老师能开一栏专门讲一讲**Linux**下多线程并发情况下程序性能的排查和调优。谢谢老师

2019-05-17

作者回复

好累

2019-05-18



佑儿

0

有问题，**singlepool**中只有一个线程池，**future.get**方法阻塞当前线程，导致打印**qq**的线程没有机会执行，会根据丢弃策略进行不同的操作。

2019-05-16



ack

👍 0

老师，请教个问题，线程死锁那个代码，是活锁吗，思考题我也认为是活锁

2019-05-16

作者回复

我觉得是死锁，活锁有释放再获取的过程

2019-05-17



峰

👍 0

线程池只有一个线程，在任务执行的时候不能有再多的线程去处理提交的任务。

2019-05-16



晓杰

👍 0

线程池里面的最大线程数只有一个，无法做到异步

2019-05-16



周治慧

👍 0

两个线程共用一个线程池，当线程池中只有一个线程时，第二个线程是拿不到线程的

2019-05-16



孙志强

👍 0

死锁

2019-05-16



密码123456

👍 0

跟今天的例子好像。一个线程池，却提交2个任务，其中一个线程等待另外一个线程

2019-05-16



智超

👍 0

线程池只有一个线程就悲剧了

2019-05-16



QQ怪

👍 0

老师，有个疑问，想问下线程池该什么时候销毁？

2019-05-16

作者回复

最多的情况是重启的时候

2019-05-17



Liam

👍 0

A任务等待B任务返回QQ字符串，但是线程池只有一个线程，B任务没有多余的线程执行，导致线程池瘫痪

2019-05-16



张三

👍 0



为什么思考提会有两次**submit**方法，猜这里应该就是问题了。打卡！

2019-05-16